

The Art of Assembly Language

(Full Contents)

“The Art of Assembly Language Programming” is going *hard*. In early 2003 this text will be available in published form from “No Starch Press” (<http://www.nostarchpress.com>). Please check out their website for more details.

1.1 Foreword to the HLA Version of “The Art of Assembly...”	3
1.2 Intended Audience	6
1.3 Teaching From This Text	6
1.4 Copyright Notice	7
1.5 How to Get a Hard Copy of This Text	8
1.6 Obtaining Program Source Listings and Other Materials in This Text	8
1.7 Where to Get Help	8
1.8 Other Materials You Will Need (Windows Version)	8
1.9 Other Materials You Will Need (Linux Version)	9
2.1 Chapter Overview	11
2.2 Installing the HLA Distribution Package	11
2.2.1 Installation Under Windows	12
2.2.2 Installation Under Linux	15
2.2.3 Installing “Art of Assembly” Related Files	18
2.3 The Anatomy of an HLA Program	19
2.4 Some Basic HLA Data Declarations	21
2.5 Boolean Values	23
2.6 Character Values	23
2.7 An Introduction to the Intel 80x86 CPU Family	23
2.8 Some Basic Machine Instructions	26
2.9 Some Basic HLA Control Structures	29
2.9.1 Boolean Expressions in HLA Statements	30
2.9.2 The HLA IF..THEN..ELSEIF..ELSE..ENDIF Statement	32
2.9.3 The WHILE..ENDWHILE Statement	33
2.9.4 The FOR..ENDFOR Statement	34
2.9.5 The REPEAT..UNTIL Statement	35
2.9.6 The BREAK and BREAKIF Statements	36
2.9.7 The FOREVER..ENDFOR Statement	36
2.9.8 The TRY..EXCEPTION..ENDTRY Statement	37
2.10 Introduction to the HLA Standard Library	38
2.10.1 Predefined Constants in the STDIO Module	40
2.10.2 Standard In and Standard Out	40
2.10.3 The stdout.newIn Routine	41

2.10.4	The stdout.putiX Routines	41
2.10.5	The stdout.putiXSize Routines	41
2.10.6	The stdout.put Routine	42
2.10.7	The stdin.getc Routine.	43
2.10.8	The stdin.getiX Routines	44
2.10.9	The stdin.readLn and stdin.flushInput Routines	46
2.10.10	The stdin.get Macro	46
2.11	Putting It All Together	47
2.12	Sample Programs	47
2.12.1	Powers of Two Table Generation	47
2.12.2	Checkerboard Program	48
2.12.3	Fibonacci Number Generation	50
3.1	Chapter Overview	53
3.2	Numbering Systems	53
3.2.1	A Review of the Decimal System	53
3.2.2	The Binary Numbering System	54
3.2.3	Binary Formats	55
3.3	Data Organization	56
3.3.1	Bits	56
3.3.2	Nibbles	56
3.3.3	Bytes	57
3.3.4	Words	58
3.3.5	Double Words	59
3.4	The Hexadecimal Numbering System	60
3.5	Arithmetic Operations on Binary and Hexadecimal Numbers	62
3.6	A Note About Numbers vs. Representation	63
3.7	Logical Operations on Bits	65
3.8	Logical Operations on Binary Numbers and Bit Strings	68
3.9	Signed and Unsigned Numbers	69
3.10	Sign Extension, Zero Extension, Contraction, and Saturation	73
3.11	Shifts and Rotates	76
3.12	Bit Fields and Packed Data	81
3.13	Putting It All Together	85
4.1	Chapter Overview	87
4.2	An Introduction to Floating Point Arithmetic	87
4.2.1	IEEE Floating Point Formats	90
4.2.2	HLA Support for Floating Point Values	93
4.3	Binary Coded Decimal (BCD) Representation	95
4.4	Characters	96
4.4.1	The ASCII Character Encoding	97
4.4.2	HLA Support for ASCII Characters	100
4.4.3	The ASCII Character Set	104
4.5	The UNICODE Character Set	108
4.6	Other Data Representations	109

4.6.1	Representing Colors on a Video Display	109
4.6.2	Representing Audio Information	111
4.6.3	Representing Musical Information	114
4.6.4	Representing Video Information	115
4.6.5	Where to Get More Information About Data Types	115
4.7	Putting It All Together	116
5.1	Questions	119
5.2	Programming Projects for Chapter Two	124
5.3	Programming Projects for Chapter Three	124
5.4	Programming Projects for Chapter Four	125
5.5	Laboratory Exercises for Chapter Two	126
5.5.1	A Short Note on Laboratory Exercises and Lab Reports	126
5.5.2	Compiling Your First Program	127
5.5.3	Compiling Other Programs Appearing in this Chapter	128
5.5.4	Creating and Modifying HLA Programs	129
5.5.5	Writing a New Program	129
5.5.6	Correcting Errors in an HLA Program	130
5.5.7	Write Your Own Sample Program	131
5.6	Laboratory Exercises for Chapter Three and Chapter Four	132
5.6.1	Data Conversion Exercises	132
5.6.2	Logical Operations Exercises	133
5.6.3	Sign and Zero Extension Exercises	133
5.6.4	Packed Data Exercises	134
5.6.5	Running this Chapter's Sample Programs	134
5.6.6	Write Your Own Sample Program	134
1.1	Chapter Overview	137
1.2	The Basic System Components	137
1.2.1	The System Bus	138
1.2.1.1	The Data Bus	138
1.2.1.2	The Address Bus	139
1.2.1.3	The Control Bus	139
1.2.2	The Memory Subsystem	140
1.2.3	The I/O Subsystem	146
1.3	HLA Support for Data Alignment	146
1.4	System Timing	149
1.4.1	The System Clock	149
1.4.2	Memory Access and the System Clock	150
1.4.3	Wait States	151
1.4.4	Cache Memory	153
1.5	Putting It All Together	156
2.1	Chapter Overview	157
2.2	The 80x86 Addressing Modes	157
2.2.1	80x86 Register Addressing Modes	157
2.2.2	80x86 32-bit Memory Addressing Modes	158
2.2.2.1	The Displacement Only Addressing Mode	158
2.2.2.2	The Register Indirect Addressing Modes	159
2.2.2.3	Indexed Addressing Modes	160

2.2.2.4	Variations on the Indexed Addressing Mode	161
2.2.2.5	Scaled Indexed Addressing Modes	163
2.2.2.6	Addressing Mode Wrap-up	164
2.3	Run-Time Memory Organization	164
2.3.1	The Code Section	165
2.3.2	The Static Sections	167
2.3.3	The Read-Only Data Section	167
2.3.4	The Storage Section	168
2.3.5	The @NOSTORAGE Attribute	169
2.3.6	The Var Section	169
2.3.7	Organization of Declaration Sections Within Your Programs	170
2.4	Address Expressions	171
2.5	Type Coercion	173
2.6	Register Type Coercion	175
2.7	The Stack Segment and the Push and Pop Instructions	176
2.7.1	The Basic PUSH Instruction	176
2.7.2	The Basic POP Instruction	177
2.7.3	Preserving Registers With the PUSH and POP Instructions	179
2.7.4	The Stack is a LIFO Data Structure	180
2.7.5	Other PUSH and POP Instructions	183
2.7.6	Removing Data From the Stack Without Popping It	184
2.7.7	Accessing Data You've Pushed on the Stack Without Popping It	186
2.8	Dynamic Memory Allocation and the Heap Segment	187
2.9	The INC and DEC Instructions	190
2.10	Obtaining the Address of a Memory Object	191
2.11	Bonus Section: The HLA Standard Library CONSOLE Module	192
2.11.1	Clearing the Screen	192
2.11.2	Positioning the Cursor	193
2.11.3	Locating the Cursor	194
2.11.4	Text Attributes	195
2.11.5	Filling a Rectangular Section of the Screen	197
2.11.6	Console Direct String Output	199
2.11.7	Other Console Module Routines	200
2.12	Putting It All Together	201
3.1	Boolean Algebra	203
3.2	Boolean Functions and Truth Tables	205
3.3	Algebraic Manipulation of Boolean Expressions	208
3.4	Canonical Forms	209
3.5	Simplification of Boolean Functions	214
3.6	What Does This Have To Do With Computers, Anyway?	221
3.6.1	Correspondence Between Electronic Circuits and Boolean Functions	221
3.6.2	Combinatorial Circuits	223
3.6.3	Sequential and Clocked Logic	228
3.7	Okay, What Does It Have To Do With Programming, Then?	232
3.8	Putting It All Together	233

4.1 Chapter Overview	234
4.2 The History of the 80x86 CPU Family	234
4.3 A History of Software Development for the x86	241
4.4 Basic CPU Design	245
4.5 Decoding and Executing Instructions: Random Logic Versus Microcode	247
4.6 RISC vs. CISC vs. VLIW	248
4.7 Instruction Execution, Step-By-Step	250
4.8 Parallelism – the Key to Faster Processors	253
4.8.1 The Prefetch Queue – Using Unused Bus Cycles	255
4.8.2 Pipelining – Overlapping the Execution of Multiple Instructions	259
4.8.2.1 A Typical Pipeline	259
4.8.2.2 Stalls in a Pipeline	261
4.8.3 Instruction Caches – Providing Multiple Paths to Memory	262
4.8.4 Hazards	263
4.8.5 Superscalar Operation– Executing Instructions in Parallel	265
4.8.6 Out of Order Execution	266
4.8.7 Register Renaming	266
4.8.8 Very Long Instruction Word Architecture (VLIW)	267
4.8.9 Parallel Processing	268
4.8.10 Multiprocessing	268
4.9 Putting It All Together	269
5.1 Chapter Overview	270
5.2 The Importance of the Design of the Instruction Set	270
5.3 Basic Instruction Design Goals	271
5.3.1 Addressing Modes on the Y86	278
5.3.2 Encoding Y86 Instructions	279
5.3.3 Hand Encoding Instructions	282
5.3.4 Using an Assembler to Encode Instructions	286
5.3.5 Extending the Y86 Instruction Set	287
5.4 Encoding 80x86 Instructions	288
5.4.1 Encoding Instruction Operands	290
5.4.2 Encoding the ADD Instruction: Some Examples	296
5.4.3 Encoding Immediate Operands	300
5.4.4 Encoding Eight, Sixteen, and Thirty-Two Bit Operands	301
5.4.5 Alternate Encodings for Instructions	301
5.5 Putting It All Together	302
6.1 Chapter Overview	303
6.2 The Memory Hierarchy	303
6.3 How the Memory Hierarchy Operates	305
6.4 Relative Performance of Memory Subsystems	306
6.5 Cache Architecture	308
6.6 Virtual Memory, Protection, and Paging	312
6.7 Thrashing	314
6.8 NUMA and Peripheral Devices	315

6.9 Segmentation	316
6.10 Segments and HLA	316
6.10.1 Renaming Segments Under Windows	317
6.11 User Defined Segments in HLA (Windows Only)	319
6.12 Controlling the Placement and Attributes of Segments in Memory (Windows Only)	321
6.13 Putting it All Together	325
7.1 Chapter Overview	327
7.2 Connecting a CPU to the Outside World	327
7.3 Read-Only, Write-Only, Read/Write, and Dual I/O Ports	329
7.4 I/O (Input/Output) Mechanisms	331
7.4.1 Memory Mapped Input/Output	331
7.4.2 I/O Mapped Input/Output	332
7.4.3 Direct Memory Access	333
7.5 I/O Speed Hierarchy	333
7.6 System Busses and Data Transfer Rates	334
7.7 The AGP Bus	336
7.8 Handshaking	337
7.9 Time-outs on an I/O Port	340
7.10 Interrupts and Polled I/O	342
7.11 Using a Circular Queue to Buffer Input Data from an ISR	343
7.12 Using a Circular Queue to Buffer Output Data for an ISR	349
7.13 I/O and the Cache	352
7.14 Protected Mode Operation	352
7.15 Device Drivers	353
7.16 Putting It All Together	354
8.1 Questions	355
8.2 Programming Projects	361
8.3 Chapters One and Two Laboratory Exercises	363
8.3.1 Memory Organization Exercises	363
8.3.2 Data Alignment Exercises	364
8.3.3 Readonly Segment Exercises	367
8.3.4 Type Coercion Exercises	367
8.3.5 Dynamic Memory Allocation Exercises	368
8.4 Chapter Three Laboratory Exercises	369
8.4.1 Truth Tables and Logic Equations Exercises	370
8.4.2 Canonical Logic Equations Exercises	371
8.4.3 Optimization Exercises	372
8.4.4 Logic Evaluation Exercises	372
8.5 Laboratory Exercises for Chapters Four, Five, Six, and Seven	377
8.5.1 The SIMY86 Program – Some Simple Y86 Programs	377
8.5.2 Simple I/O-Mapped Input/Output Operations	380
8.5.3 Memory Mapped I/O	381
8.5.4 DMA Exercises	382

8.5.5	Interrupt Driven I/O Exercises	383
8.5.6	Machine Language Programming & Instruction Encoding Exercises	384
8.5.7	Self Modifying Code Exercises	386
8.5.8	Virtual Memory Exercise	388
1.1	Chapter Overview	393
1.2	Some Additional Instructions: INTMUL, BOUND, INTO	393
1.3	The QWORD and TBYTE Data Types	397
1.4	HLA Constant and Value Declarations	397
1.4.1	Constant Types	400
1.4.2	String and Character Literal Constants	401
1.4.3	String and Text Constants in the CONST Section	402
1.4.4	Constant Expressions	403
1.4.5	Multiple CONST Sections and Their Order in an HLA Program	405
1.4.6	The HLA VAL Section	406
1.4.7	Modifying VAL Objects at Arbitrary Points in Your Programs	406
1.5	The HLA TYPE Section	407
1.6	ENUM and HLA Enumerated Data Types	408
1.7	Pointer Data Types	409
1.7.1	Using Pointers in Assembly Language	410
1.7.2	Declaring Pointers in HLA	411
1.7.3	Pointer Constants and Pointer Constant Expressions	411
1.7.4	Pointer Variables and Dynamic Memory Allocation	412
1.7.5	Common Pointer Problems	413
1.8	Putting It All Together	417
2.1	Chapter Overview	419
2.2	Composite Data Types	419
2.3	Character Strings	419
2.4	HLA Strings	421
2.5	Accessing the Characters Within a String	426
2.6	The HLA String Module and Other String-Related Routines	428
2.7	In-Memory Conversions	437
2.8	Putting It All Together	438
3.1	Chapter Overview	439
3.2	The HLA Standard Library CHAR.S.HHF Module	439
3.3	Character Sets	441
3.4	Character Set Implementation in HLA	442
3.5	HLA Character Set Constants and Character Set Expressions	443
3.6	The IN Operator in HLA HLL Boolean Expressions	444
3.7	Character Set Support in the HLA Standard Library	445
3.8	Using Character Sets in Your HLA Programs	447
3.9	Low-level Implementation of Set Operations	449
3.9.1	Character Set Functions That Build Sets	449
3.9.2	Traditional Set Operations	455

3.9.3 Testing Character Sets	458
3.10 Putting It All Together	461
4.1 Chapter Overview	463
4.2 Arrays	463
4.3 Declaring Arrays in Your HLA Programs	464
4.4 HLA Array Constants	464
4.5 Accessing Elements of a Single Dimension Array	465
4.5.1 Sorting an Array of Values	467
4.6 Multidimensional Arrays	468
4.6.1 Row Major Ordering	469
4.6.2 Column Major Ordering	473
4.7 Allocating Storage for Multidimensional Arrays	474
4.8 Accessing Multidimensional Array Elements in Assembly Language ...	475
4.9 Large Arrays and MASM	476
4.10 Dynamic Arrays in Assembly Language	477
4.11 HLA Standard Library Array Support	479
4.12 Putting It All Together	481
5.1 Chapter Overview	483
5.2 Records	483
5.3 Record Constants	485
5.4 Arrays of Records	486
5.5 Arrays/Records as Record Fields	487
5.6 Controlling Field Offsets Within a Record	489
5.7 Aligning Fields Within a Record	490
5.8 Pointers to Records	491
5.9 Unions	492
5.10 Anonymous Unions	494
5.11 Variant Types	495
5.12 Namespaces	496
5.13 Putting It All Together	498
6.1 Chapter Overview	501
6.2 Dates	501
6.3 A Brief History of the Calendar	502
6.4 HLA Date Functions	505
6.4.1 date.IsValid and date.validate	505
6.4.2 Checking for Leap Years	507
6.4.3 Obtaining the System Date	509
6.4.4 Date to String Conversions and Date Output	510
6.4.5 date.unpack and data.pack	511
6.4.6 date.Julian, date.fromJulian	512
6.4.7 date.datePlusDays, date.datePlusMonths, and date.daysBetween ..	512
6.4.8 date.dayNumber, date.daysLeft, and date.dayOfWeek	513

6.5	Times	514
6.5.1	time.curTime	514
6.5.2	time.hmsToSecs and time.secstoHMS	515
6.5.3	Time Input/Output	515
6.6	Putting It All Together	516
7.1	Chapter Overview	517
7.2	File Organization	517
7.2.1	Files as Lists of Records	517
7.2.2	Binary vs. Text Files	518
7.3	Sequential Files	520
7.4	Random Access Files	527
7.5	ISAM (Indexed Sequential Access Method) Files	530
7.6	Truncating a File	533
7.7	File Utility Routines	534
7.7.1	Copying, Moving, and Renaming Files	534
7.7.2	Computing the File Size	536
7.7.3	Deleting Files	538
7.8	Directory Operations	538
7.9	Putting It All Together	539
8.1	Chapter Overview	541
8.2	Procedures	541
8.3	Saving the State of the Machine	543
8.4	Prematurely Returning from a Procedure	546
8.5	Local Variables	547
8.6	Other Local and Global Symbol Types	551
8.7	Parameters	552
8.7.1	Pass by Value	552
8.7.2	Pass by Reference	555
8.8	Functions and Function Results	557
8.8.1	Returning Function Results	558
8.8.2	Instruction Composition in HLA	558
8.8.3	The HLA RETURNS Option in Procedures	560
8.9	Side Effects	562
8.10	Recursion	563
8.11	Forward Procedures	567
8.12	Putting It All Together	567
9.1	Chapter Overview	569
9.2	Managing Large Programs	569
9.3	The #INCLUDE Directive	570
9.4	Ignoring Duplicate Include Operations	571
9.5	UNITs and the EXTERNAL Directive	572
9.5.1	Behavior of the EXTERNAL Directive	575

9.5.2 Header Files in HLA	576
9.6 Make Files	578
9.7 Code Reuse	580
9.8 Creating and Managing Libraries	581
9.9 Name Space Pollution	583
9.10 Putting It All Together	585
10.1 Chapter Overview	587
10.2 80x86 Integer Arithmetic Instructions	587
10.2.1 The MUL and IMUL Instructions	587
10.2.2 The DIV and IDIV Instructions	589
10.2.3 The CMP Instruction	592
10.2.4 The SETcc Instructions	593
10.2.5 The TEST Instruction	596
10.3 Arithmetic Expressions	597
10.3.1 Simple Assignments	597
10.3.2 Simple Expressions	598
10.3.3 Complex Expressions	600
10.3.4 Commutative Operators	603
10.4 Logical (Boolean) Expressions	604
10.5 Machine and Arithmetic Idioms	606
10.5.1 Multiplying without MUL, IMUL, or INTMUL	606
10.5.2 Division Without DIV or IDIV	607
10.5.3 Implementing Modulo-N Counters with AND	608
10.5.4 Careless Use of Machine Idioms	608
10.6 The HLA (Pseudo) Random Number Unit	608
10.7 Putting It All Together	610
11.1 Chapter Overview	611
11.2 Floating Point Arithmetic	611
11.2.1 FPU Registers	611
11.2.1.1 FPU Data Registers	612
11.2.1.2 The FPU Control Register	612
11.2.1.3 The FPU Status Register	615
11.2.2 FPU Data Types	619
11.2.3 The FPU Instruction Set	621
11.2.4 FPU Data Movement Instructions	621
11.2.4.1 The FLD Instruction	621
11.2.4.2 The FST and FSTP Instructions	622
11.2.4.3 The FXCH Instruction	622
11.2.5 Conversions	623
11.2.5.1 The FILD Instruction	623
11.2.5.2 The FIST and FISTP Instructions	623
11.2.5.3 The FBLD and FBSTP Instructions	624
11.2.6 Arithmetic Instructions	624
11.2.6.1 The FADD and FADDP Instructions	625
11.2.6.2 The FSUB, FSUBP, FSUBR, and FSUBRP Instructions	625
11.2.6.3 The FMUL and FMULP Instructions	626
11.2.6.4 The FDIV, FDIVP, FDIVR, and FDIVRP Instructions	626

11.2.6.5	The FSQRT Instruction	627
11.2.6.6	The FPREM and FPREM1 Instructions	628
11.2.6.7	The FRNDINT Instruction	628
11.2.6.8	The FABS Instruction	628
11.2.6.9	The FCHS Instruction	629
11.2.7	Comparison Instructions	629
11.2.7.1	The FCOM, FCOMP, and FCOMPP Instructions	629
11.2.7.2	The FTST Instruction	630
11.2.8	Constant Instructions	631
11.2.9	Transcendental Instructions	631
11.2.9.1	The F2XM1 Instruction	631
11.2.9.2	The FSIN, FCOS, and FSINCOS Instructions	631
11.2.9.3	The FPTAN Instruction	632
11.2.9.4	The FPATAN Instruction	632
11.2.9.5	The FYL2X Instruction	632
11.2.9.6	The FYL2XP1 Instruction	632
11.2.10	Miscellaneous instructions	633
11.2.10.1	The FINIT and FNINIT Instructions	633
11.2.10.2	The FLDCW and FSTCW Instructions	633
11.2.10.3	The FCLEX and FNCLEX Instructions	633
11.2.10.4	The FSTSW and FNSTSW Instructions	633
11.2.11	Integer Operations	634
11.3	Converting Floating Point Expressions to Assembly Language	634
11.3.1	Converting Arithmetic Expressions to Postfix Notation	635
11.3.2	Converting Postfix Notation to Assembly Language	637
11.3.3	Mixed Integer and Floating Point Arithmetic	638
11.4	HLA Standard Library Support for Floating Point Arithmetic	638
11.4.1	The stdin.getf and fileio.getf Functions	639
11.4.2	Trigonometric Functions in the HLA Math Library	639
11.4.3	Exponential and Logarithmic Functions in the HLA Math Library	640
11.5	Sample Program	640
11.6	Putting It All Together	646
12.1	Chapter Overview	647
12.2	Tables	647
12.2.1	Function Computation via Table Look-up	647
12.2.2	Domain Conditioning	650
12.2.3	Generating Tables	651
12.3	High Performance Implementation of cs.rangeChar	655
13.1	Questions	663
13.2	Programming Projects	670
13.3	Laboratory Exercises	677
13.3.1	Using the BOUND Instruction to Check Array Indices	677
13.3.2	Using TEXT Constants in Your Programs	680
13.3.3	Constant Expressions Lab Exercise	682
13.3.4	Pointers and Pointer Constants Exercises	684
13.3.5	String Exercises	685
13.3.6	String and Character Set Exercises	687
13.3.7	Console Array Exercise	691

13.3.8	Multidimensional Array Exercises	693
13.3.9	Console Attributes Laboratory Exercise	696
13.3.10	Records, Arrays, and Pointers Laboratory Exercise	698
13.3.11	Separate Compilation Exercises	704
13.3.12	The HLA (Pseudo) Random Number Unit	710
13.3.13	File I/O in HLA	711
13.3.14	Timing Various Arithmetic Instructions	712
13.3.15	Using the RDTSC Instruction to Time a Code Sequence	715
13.3.16	Timing Floating Point Instructions	719
13.3.17	Table Lookup Exercise	722
1.1	Chapter Overview	727
1.2	Conjunction, Disjunction, and Negation in Boolean Expressions	727
1.3	TRY..ENDTRY	729
1.3.1	Nesting TRY..ENDTRY Statements	730
1.3.2	The UNPROTECTED Clause in a TRY..ENDTRY Statement	732
1.3.3	The ANYEXCEPTION Clause in a TRY..ENDTRY Statement	735
1.3.4	Raising User-Defined Exceptions	735
1.3.5	Reraising Exceptions in a TRY..ENDTRY Statement	737
1.3.6	A List of the Predefined HLA Exceptions	737
1.3.7	How to Handle Exceptions in Your Programs	737
1.3.8	Registers and the TRY..ENDTRY Statement	739
1.4	BEGIN..EXIT..EXITIF..END	740
1.5	CONTINUE..CONTINUEIF	745
1.6	SWITCH..CASE..DEFAULT..ENDSWITCH	747
1.7	Putting It All Together	749
2.1	Chapter Overview	751
2.2	Low Level Control Structures	751
2.3	Statement Labels	751
2.4	Unconditional Transfer of Control (JMP)	753
2.5	The Conditional Jump Instructions	755
2.6	“Medium-Level” Control Structures: JT and JF	759
2.7	Implementing Common Control Structures in Assembly Language	759
2.8	Introduction to Decisions	760
2.8.1	IF..THEN..ELSE Sequences	761
2.8.2	Translating HLA IF Statements into Pure Assembly Language	764
2.8.3	Implementing Complex IF Statements Using Complete Boolean Evaluation	768
2.8.4	Short Circuit Boolean Evaluation	769
2.8.5	Short Circuit vs. Complete Boolean Evaluation	770
2.8.6	Efficient Implementation of IF Statements in Assembly Language	772
2.8.7	SWITCH/CASE Statements	776
2.9	State Machines and Indirect Jumps	784
2.10	Spaghetti Code	786
2.11	Loops	787
2.11.1	While Loops	787
2.11.2	Repeat..Until Loops	788
2.11.3	FOREVER..ENDFOR Loops	789

2.11.4	FOR Loops	790
2.11.5	The BREAK and CONTINUE Statements	791
2.11.6	Register Usage and Loops	795
2.12	Performance Improvements	796
2.12.1	Moving the Termination Condition to the End of a Loop	796
2.12.2	Executing the Loop Backwards	798
2.12.3	Loop Invariant Computations	799
2.12.4	Unraveling Loops	800
2.12.5	Induction Variables	801
2.13	Hybrid Control Structures in HLA	802
2.14	Putting It All Together	804
3.1	Chapter Overview	805
3.2	Procedures and the CALL Instruction	805
3.3	Procedures and the Stack	807
3.4	Activation Records	810
3.5	The Standard Entry Sequence	813
3.6	The Standard Exit Sequence	814
3.7	HLA Local Variables	815
3.8	Parameters	816
3.8.1	Pass by Value	817
3.8.2	Pass by Reference	817
3.8.3	Passing Parameters in Registers	818
3.8.4	Passing Parameters in the Code Stream	820
3.8.5	Passing Parameters on the Stack	822
3.8.5.1	Accessing Value Parameters on the Stack	824
3.8.5.2	Passing Value Parameters on the Stack	825
3.8.5.3	Accessing Reference Parameters on the Stack	831
3.8.5.4	Passing Reference Parameters on the Stack	834
3.8.5.5	Passing Formal Parameters as Actual Parameters	836
3.8.5.6	HLA Hybrid Parameter Passing Facilities	838
3.8.5.7	Mixing Register and Stack Based Parameters	839
3.9	Procedure Pointers	839
3.10	Procedural Parameters	842
3.11	Untyped Reference Parameters	843
3.12	Iterators and the FOREACH Loop	843
3.13	Sample Programs	846
3.13.1	Generating the Fibonacci Sequence Using an Iterator	846
3.13.2	Outer Product Computation with Procedural Parameters	848
3.14	Putting It All Together	851
4.1	Chapter Overview	853
4.2	Multiprecision Operations	853
4.2.1	Multiprecision Addition Operations	853
4.2.2	Multiprecision Subtraction Operations	856
4.2.3	Extended Precision Comparisons	857
4.2.4	Extended Precision Multiplication	860

4.2.5	Extended Precision Division	864
4.2.6	Extended Precision NEG Operations	872
4.2.7	Extended Precision AND Operations	873
4.2.8	Extended Precision OR Operations	874
4.2.9	Extended Precision XOR Operations	874
4.2.10	Extended Precision NOT Operations	874
4.2.11	Extended Precision Shift Operations	875
4.2.12	Extended Precision Rotate Operations	878
4.2.13	Extended Precision I/O	878
4.2.13.1	Extended Precision Hexadecimal Output	879
4.2.13.2	Extended Precision Unsigned Decimal Output	879
4.2.13.3	Extended Precision Signed Decimal Output	882
4.2.13.4	Extended Precision Formatted I/O	883
4.2.13.5	Extended Precision Input Routines	884
4.2.13.6	Extended Precision Hexadecimal Input	887
4.2.13.7	Extended Precision Unsigned Decimal Input	891
4.2.13.8	Extended Precision Signed Decimal Input	895
4.3	Operating on Different Sized Operands	895
4.4	Decimal Arithmetic	897
4.4.1	Literal BCD Constants	898
4.4.2	The 80x86 DAA and DAS Instructions	898
4.4.3	The 80x86 AAA, AAS, AAM, and AAD Instructions	900
4.4.4	Packed Decimal Arithmetic Using the FPU	901
4.5	Sample Program	903
4.6	Putting It All Together	906
5.1	Chapter Overview	909
5.2	What is Bit Data, Anyway?	909
5.3	Instructions That Manipulate Bits	910
5.4	The Carry Flag as a Bit Accumulator	916
5.5	Packing and Unpacking Bit Strings	917
5.6	Coalescing Bit Sets and Distributing Bit Strings	920
5.7	Packed Arrays of Bit Strings	922
5.8	Searching for a Bit	923
5.9	Counting Bits	925
5.10	Reversing a Bit String	927
5.11	Merging Bit Strings	929
5.12	Extracting Bit Strings	930
5.13	Searching for a Bit Pattern	931
5.14	The HLA Standard Library Bits Module	932
5.15	Putting It All Together	933
6.1	Chapter Overview	935
6.2	The 80x86 String Instructions	935
6.2.1	How the String Instructions Operate	936
6.2.2	The REP/REPE/REPZ and REPNZ/REPNE Prefixes	936
6.2.3	The Direction Flag	937

6.2.4	The MOVS Instruction	938
6.2.5	The CMPS Instruction	943
6.2.6	The SCAS Instruction	946
6.2.7	The STOS Instruction	946
6.2.8	The LODS Instruction	947
6.2.9	Building Complex String Functions from LODS and STOS	947
6.3	Putting It All Together	948
7.1	Chapter Overview	949
7.2	Introduction to the Compile-Time Language (CTL)	949
7.3	The #PRINT and #ERROR Statements	951
7.4	Compile-Time Constants and Variables	952
7.5	Compile-Time Expressions and Operators	953
7.6	Compile-Time Functions	956
7.6.1	Type Conversion Compile-time Functions	957
7.6.2	Numeric Compile-Time Functions	957
7.6.3	Character Classification Compile-Time Functions	958
7.6.4	Compile-Time String Functions	958
7.6.5	Compile-Time Pattern Matching Functions	958
7.6.6	Compile-Time Symbol Information	959
7.6.7	Compile-Time Expression Classification Functions	960
7.6.8	Miscellaneous Compile-Time Functions	961
7.6.9	Predefined Compile-Time Variables	961
7.6.10	Compile-Time Type Conversions of TEXT Objects	961
7.7	Conditional Compilation (Compile-Time Decisions)	962
7.8	Repetitive Compilation (Compile-Time Loops)	966
7.9	Putting It All Together	968
8.1	Chapter Overview	969
8.2	Macros (Compile-Time Procedures)	969
8.2.1	Standard Macros	969
8.2.2	Macro Parameters	971
8.2.2.1	Standard Macro Parameter Expansion	971
8.2.2.2	Macros with a Variable Number of Parameters	974
8.2.2.3	Required Versus Optional Macro Parameters	975
8.2.2.4	The "#(" and ")#" Macro Parameter Brackets	976
8.2.2.5	Eager vs. Deferred Macro Parameter Evaluation	977
8.2.3	Local Symbols in a Macro	981
8.2.4	Macros as Compile-Time Procedures	985
8.2.5	Multi-part (Context-Free) Macros	985
8.2.6	Simulating Function Overloading with Macros	990
8.3	Writing Compile-Time "Programs"	995
8.3.1	Constructing Data Tables at Compile Time	996
8.3.2	Unrolling Loops	999
8.4	Using Macros in Different Source Files	1001
8.5	Putting It All Together	1001
9.1	Chapter Overview	1003
9.2	Introduction to DSELS in HLA	1003

9.2.1	Implementing the Standard HLA Control Structures	1003
9.2.1.1	The FOREVER Loop	1004
9.2.1.2	The WHILE Loop	1007
9.2.1.3	The IF Statement	1009
9.2.2	The HLA SWITCH/CASE Statement	1015
9.2.3	A Modified WHILE Loop	1026
9.2.4	A Modified IF..ELSE..ENDIF Statement	1030
9.3	Sample Program: A Simple Expression Compiler	1036
9.4	Putting It All Together	1057
10.1	Chapter Overview	1059
10.2	General Principles	1059
10.3	Classes in HLA	1061
10.4	Objects	1063
10.5	Inheritance	1064
10.6	Overriding	1065
10.7	Virtual Methods vs. Static Procedures	1066
10.8	Writing Class Methods, Iterators, and Procedures	1067
10.9	Object Implementation	1071
10.9.1	Virtual Method Tables	1073
10.9.2	Object Representation with Inheritance	1075
10.10	Constructors and Object Initialization	1079
10.10.1	Dynamic Object Allocation Within the Constructor	1081
10.10.2	Constructors and Inheritance	1082
10.10.3	Constructor Parameters and Procedure Overloading	1085
10.11	Destructors	1086
10.12	HLA's "_initialize_" and "_finalize_" Strings	1087
10.13	Abstract Methods	1091
10.14	Run-time Type Information (RTTI)	1094
10.15	Calling Base Class Methods	1095
10.16	Sample Program	1096
10.17	Putting It All Together	1112
11.1	Chapter Overview	1113
11.2	Determining if a CPU Supports the MMX Instruction Set	1113
11.3	The MMX Programming Environment	1114
11.3.1	The MMX Registers	1114
11.3.2	The MMX Data Types	1116
11.4	The Purpose of the MMX Instruction Set	1117
11.5	Saturation Arithmetic and Wraparound Mode	1118
11.6	MMX Instruction Operands	1118
11.7	MMX Technology Instructions	1123
11.7.1	MMX Data Transfer Instructions	1123
11.7.2	MMX Conversion Instructions	1123
11.7.3	MMX Packed Arithmetic Instructions	1131

11.7.4	MMX Logic Instructions	1133
11.7.5	MMX Comparison Instructions	1134
11.7.6	MMX Shift Instructions	1138
11.8	The EMMS Instruction	1139
11.9	The MMX Programming Paradigm	1140
11.10	Putting It All Together	1148
12.1	Chapter Overview	1151
12.2	Mixing HLA and MASM/Gas Code in the Same Program	1151
12.2.1	In-Line (MASM/Gas) Assembly Code in Your HLA Programs	1151
12.2.2	Linking MASM/Gas-Assembled Modules with HLA Modules	1154
12.3	Programming in Delphi/Kylix and HLA	1157
12.3.1	Linking HLA Modules With Delphi Programs	1158
12.3.2	Register Preservation	1161
12.3.3	Function Results	1161
12.3.4	Calling Conventions	1167
12.3.5	Pass by Value, Reference, CONST, and OUT in Delphi	1172
12.3.6	Scalar Data Type Correspondence Between Delphi and HLA	1173
12.3.7	Passing String Data Between Delphi and HLA Code	1175
12.3.8	Passing Record Data Between HLA and Delphi	1177
12.3.9	Passing Set Data Between Delphi and HLA	1181
12.3.10	Passing Array Data Between HLA and Delphi	1181
12.3.11	Delphi Limitations When Linking with (Non-TASM) Assembly Code	1182
12.3.12	Referencing Delphi Objects from HLA Code	1182
12.4	Programming in C/C++ and HLA	1182
12.4.1	Linking HLA Modules With C/C++ Programs	1183
12.4.2	Register Preservation	1186
12.4.3	Function Results	1186
12.4.4	Calling Conventions	1186
12.4.5	Pass by Value and Reference in C/C++	1189
12.4.6	Scalar Data Type Correspondence Between C/C++ and HLA	1189
12.4.7	Passing String Data Between C/C++ and HLA Code	1191
12.4.8	Passing Record/Structure Data Between HLA and C/C++	1191
12.4.9	Passing Array Data Between HLA and C/C++	1192
12.5	Putting It All Together	1193
13.1	Questions	1195
13.2	Programming Problems	1203
13.3	Laboratory Exercises	1212
13.3.1	Dynamically Nested TRY..ENDTRY Statements	1213
13.3.2	The TRY..ENDTRY Unprotected Section	1214
13.3.3	Performance of SWITCH Statement	1215
13.3.4	Complete Versus Short Circuit Boolean Evaluation	1219
13.3.5	Conversion of High Level Language Statements to Pure Assembly	1222
13.3.6	Activation Record Exercises	1222
13.3.6.1	Automatic Activation Record Generation and Access	1222
13.3.6.2	The <code>_vars_</code> and <code>_parms_</code> Constants	1224
13.3.6.3	Manually Constructing an Activation Record	1226
13.3.7	Reference Parameter Exercise	1228
13.3.8	Procedural Parameter Exercise	1231

13.3.9	Iterator Exercises	1234
13.3.10	Performance of Multiprecision Multiplication and Division Operations	1237
13.3.11	Performance of the Extended Precision NEG Operation	1237
13.3.12	Testing the Extended Precision Input Routines	1238
13.3.13	Illegal Decimal Operations	1238
13.3.14	MOVS Performance Exercise #1	1238
13.3.15	MOVS Performance Exercise #2	1240
13.3.16	Memory Performance Exercise	1242
13.3.17	The Performance of Length-Prefixed vs. Zero-Terminated Strings	1243
13.3.18	Introduction to Compile-Time Programs	1249
13.3.19	Conditional Compilation and Debug Code	1250
13.3.20	The Assert Macro	1252
13.3.21	Demonstration of Compile-Time Loops (#while)	1254
13.3.22	Writing a Trace Macro	1256
13.3.23	Overloading	1258
13.3.24	Multi-part Macros and RatASM (Rational Assembly)	1261
13.3.25	Virtual Methods vs. Static Procedures in a Class	1264
13.3.26	Using the <code>_initialize_</code> and <code>_finalize_</code> Strings in a Program	1267
13.3.27	Using RTTI in a Program	1270
1.1	Chapter Overview	1279
1.2	First Class Objects	1279
1.3	Thunks	1281
1.4	Initializing Thunks	1282
1.5	Manipulating Thunks	1283
1.5.1	Assigning Thunks	1283
1.5.2	Comparing Thunks	1284
1.5.3	Passing Thunks as Parameters	1285
1.5.4	Returning Thunks as Function Results	1286
1.6	Activation Record Lifetimes and Thunks	1288
1.7	Comparing Thunks and Objects	1289
1.8	An Example of a Thunk Using the Fibonacci Function	1289
1.9	Thunks and Artificial Intelligence Code	1294
1.10	Thunks as Triggers	1296
1.11	Jumping Out of a Thunk	1299
1.12	Handling Exceptions with Thunks	1302
1.13	Using Thunks in an Appropriate Manner	1302
1.14	Putting It All Together	1303
2.1	Chapter Overview	1305
2.2	Review of Iterators	1305
2.2.1	Implementing Iterators Using In-Line Expansion	1307
2.2.2	Implementing Iterators with Resume Frames	1308
2.3	Other Possible Iterator Implementations	1314
2.4	Breaking Out of a FOREACH Loop	1316
2.5	An Iterator Implementation of the Fibonacci Number Generator	1317
2.6	Iterators and Recursion	1323

2.7	Calling Other Procedures Within an Iterator	1327
2.8	Iterators Within Classes	1327
2.9	Putting It Altogether	1327
3.1	Chapter Overview	1329
3.2	Coroutines	1329
3.3	Parameters and Register Values in Coroutine Calls	1334
3.4	Recursion, Reentrancy, and Variables	1335
3.5	Generators	1337
3.6	Exceptions and Coroutines	1340
3.7	Putting It All Together	1340
4.1	Chapter Overview	1341
4.2	Parameters	1341
4.3	Where You Can Pass Parameters	1341
4.3.1	Passing Parameters in (Integer) Registers	1342
4.3.2	Passing Parameters in FPU and MMX Registers	1345
4.3.3	Passing Parameters in Global Variables	1346
4.3.4	Passing Parameters on the Stack	1347
4.3.5	Passing Parameters in the Code Stream	1351
4.3.6	Passing Parameters via a Parameter Block	1353
4.4	How You Can Pass Parameters	1354
4.4.1	Pass by Value-Result	1354
4.4.2	Pass by Result	1359
4.4.3	Pass by Name	1360
4.4.4	Pass by Lazy-Evaluation	1362
4.5	Passing Parameters as Parameters to Another Procedure	1363
4.5.1	Passing Reference Parameters to Other Procedures	1363
4.5.2	Passing Value-Result and Result Parameters as Parameters	1365
4.5.3	Passing Name Parameters to Other Procedures	1365
4.5.4	Passing Lazy Evaluation Parameters as Parameters	1366
4.5.5	Parameter Passing Summary	1366
4.6	Variable Parameter Lists	1368
4.7	Function Results	1370
4.7.1	Returning Function Results in a Register	1370
4.7.2	Returning Function Results on the Stack	1371
4.7.3	Returning Function Results in Memory Locations	1371
4.7.4	Returning Large Function Results	1372
4.8	Putting It All Together	1372
5.1	Chapter Overview	1375
5.2	Lexical Nesting, Static Links, and Displays	1375
5.2.1	Scope	1375
5.2.2	Unit Activation, Address Binding, and Variable Lifetime	1376
5.2.3	Static Links	1377
5.2.4	Accessing Non-Local Variables Using Static Links	1382
5.2.5	Nesting Procedures in HLA	1384
5.2.6	The Display	1387

5.2.7 The 80x86 ENTER and LEAVE Instructions	1391
5.3 Passing Variables at Different Lex Levels as Parameters.	1394
5.3.1 Passing Parameters by Value	1394
5.3.2 Passing Parameters by Reference, Result, and Value-Result ...	1395
5.3.3 Passing Parameters by Name and Lazy-Evaluation in a Block Structured Language	1395
5.4 Passing Procedures as Parameters	1396
5.5 Faking Intermediate Variable Access	1396
5.6 Putting It All Together	1397
6.1 Questions	1399
6.2 Programming Problems	1402
C.1 Introduction	1411
C.1.1 Intended Audience	1411
C.1.2 Readability Metrics	1411
C.1.3 How to Achieve Readability	1412
C.1.4 How This Document is Organized	1413
C.1.5 Guidelines, Rules, Enforced Rules, and Exceptions	1413
C.1.6 Source Language Concerns	1414
C.2 Program Organization	1414
C.2.1 Library Functions	1414
C.2.2 Common Object Modules	1415
C.2.3 Local Modules	1415
C.2.4 Program Make Files	1416
C.3 Module Organization	1417
C.3.1 Module Attributes	1417
C.3.1.1 Module Cohesion	1417
C.3.1.2 Module Coupling	1418
C.3.1.3 Physical Organization of Modules	1418
C.3.1.4 Module Interface	1419
C.4 Program Unit Organization	1420
C.4.1 Routine Cohesion	1420
C.4.2 Routine Coupling	1421
C.4.3 Routine Size	1421
C.5 Statement Organization	1422
C.5.1 Writing “Pure” Assembly Code	1422
C.5.2 Using HLA’s High Level Control Statements	1424
C.6 Comments	1430
C.6.1 What is a Bad Comment?	1430
C.6.2 What is a Good Comment?	1431
C.6.3 Endline vs. Standalone Comments	1432
C.6.4 Unfinished Code	1433
C.6.5 Cross References in Code to Other Documents	1434
C.7 Names, Instructions, Operators, and Operands	1435
C.7.1 Names	1435
C.7.1.1 Naming Conventions	1437
C.7.1.2 Alphabetic Case Considerations	1437

C.7.1.3 Abbreviations	1438
C.7.1.4 The Position of Components Within an Identifier	1439
C.7.1.5 Names to Avoid	1440
C.7.1.6 Special Identifiers	1441
C.7.2 Instructions, Directives, and Pseudo-Opcodes	1442
C.7.2.1 Choosing the Best Instruction Sequence	1442
C.7.2.2 Control Structures	1443
C.7.2.3 Instruction Synonyms	1446
C.8 Data Types	1447
C.8.1 Declaring Structures in Assembly Language	1447
H.1 Conversion Functions	1493
H.2 Numeric Functions	1495
H.3 Date/Time Functions	1498
H.4 Classification Functions	1498
H.5 String and Character Set Functions	1500
H.6 Pattern Matching Functions	1504
H.6.1 String/Cset Pattern Matching Functions	1505
H.6.2 String/Character Pattern Matching Functions	1510
H.6.3 String/Case Insensitive Character Pattern Matching Functions	1514
H.6.4 String/String Pattern Matching Functions	1516
H.6.5 String/Misc Pattern Matching Functions	1517
H.7 HLA Information and Symbol Table Functions	1521
H.8 Compile-Time Variables	1527
H.9 Miscellaneous Compile-Time Functions	1528
J.1 The @TRACE Pseudo-Variable	1533
J.2 The Assert Macro	1536
L.1 The HLA Standard Library	1541
L.2 Compiling to MASM Code -- The Final Word	1542
L.3 The HLA if..then..endif Statement, Part I	1547
L.4 Boolean Expressions in HLA Control Structures	1548
L.5 The JT/JF Pseudo-Instructions	1554
L.6 The HLA if..then..elseif..else..endif Statement, Part II	1554
L.7 The While Statement	1558
L.8 repeat..until	1559
L.9 for..endfor	1559
L.10 forever..endfor	1559
L.11 break, breakif	1559
L.12 continue, continueif	1559
L.13 begin..end, exit, exitif	1559
L.14 foreach..endfor	1559
L.15 try..unprotect..exception..anyexception..endtry, raise	1559

The Art of Assembly Language Programming (Short Contents)

“The Art of Assembly Language Programming” is going *hard*. In early 2003 this text will be available in published form from “No Starch Press” (<http://www.nostarchpress.com>). Please check out their website for more details.

The Art of Assembly Language	1
Chapter Two Volume One:Data Representation	1
Chapter One Foreword	3
Chapter Two Hello, World of Assembly Language	11
Chapter Three Data Representation	53
Chapter Four More Data Representation	87
Chapter Five	119
Chapter Five Questions, Projects, and Lab Exercises	119
Volume Two:	135
Machine Architecture	135
Chapter One System Organization	137
Chapter Two Memory Access and Organization	157
Chapter Three Introduction to Digital Design	203
Chapter Four CPU Architecture	234
Chapter Five Instruction Set Architecture	270
Chapter Six Memory Architecture	303
Chapter Seven The I/O Subsystem	327
Chapter Eight Questions, Projects, and Labs	355
Volume Three:	391
Basic Assembly Language	391
Chapter One Constants, Variables, and Data Types	393
Chapter Two Introduction to Character Strings	419
Chapter Three Characters and Character Sets	439
Chapter Four Arrays	463
Chapter Five Records, Unions, and Name Spaces	483
Chapter Six Dates and Times	501
Chapter Seven Files	517
Chapter Eight Introduction to Procedures	541
Chapter Nine Managing Large Programs	569
Chapter Ten Integer Arithmetic	587

Chapter Eleven Real Arithmetic	611
Chapter Twelve Calculation Via Table Lookups	647
Chapter Thirteen Questions, Projects, and Labs	663
Volume Four:	725
Intermediate Assembly Language	725
Chapter One Advanced High Level Control Structures	727
Chapter Two Low-Level Control Structures	751
Chapter Three Intermediate Procedures	805
Chapter Four Advanced Arithmetic	853
Chapter Five Bit Manipulation	909
Chapter Six The String Instructions	935
Chapter Seven The HLA Compile-Time Language	949
Chapter Eight Macros	969
Chapter Nine Domain Specific Embedded Languages	1003
Chapter Ten Classes and Objects	1059
Chapter Eleven The MMX Instruction Set	1113
Chapter Twelve Mixed Language Programming	1151
Chapter Thirteen Questions, Projects, and Labs	1195
Volume Five:	1277
Advanced Procedures	1277
Chapter One Thunks	1279
Chapter Two Iterators	1305
Chapter Three Coroutines and Generators	1329
Chapter Four Advanced Parameter Implementation	1341
Chapter Five Lexical Nesting	1375
Chapter Six Questions, Projects, and Labs	1399
Appendix A Answers to Selected Exercises	1405
Appendix B Console Graphic Characters	1407
Appendix D The 80x86 Instruction Set	1449
Appendix E The HLA Language Reference	1483
Appendix F The HLA Standard Library Reference	1485
Appendix G HLA Exceptions	1487
Appendix H HLA Compile-Time Functions	1493
Appendix I Installing HLA on Your System	1531
Appendix J Debugging HLA Programs	1533

Appendix K Comparing HLA and MASM 1539
Appendix L HLA Code Generation for HLL Statements 1541
Index 1561

Index

(numeric character constant prefix) 100
 # character literal constants 402
 #(and)# (macro parameter brackets) 976
 #{ (Hybrid parameter passing syntax) 838
 #asm 1151
 #emit 1151
 #endasm 1151
 #ERROR statement 951
 #include 20
 #includeonce directive 571
 #KEYWORD section in a macro 989
 #PRINT statement 951
 #TERMINATOR section in a macro 988
 ? Compile-time operator 407
 @c 30
 @ELEMENTS compile-time function 974
 @Elements compile-time function 479
 @EVAL function 980
 @global operator (in name spaces) 497
 @nc 30
 @no 30
 @NOALIGNSTACK option 805
 @NODISPLAY option 805
 @NOFRAME option 805
 @NOSTORAGE 169
 @ns 30
 @nz 30
 @o 30
 @offset compile-time function 1153
 @s 30
 @size compile time function 487
 @Size compile-time function 479
 @size function 187
 @size function (applied to UNIONS) 493
 @StaticName function 1152
 @STRING
 name operator 975
 @TYPENAME function 992
 @USE procedure option 830, 836
 @z 30
 finalize strings 1087
 initialize strings 1087
 pVMT 1080, 1084
 vars constant 816
 VMT 1075

16-bit registers 24
 32016 microprocessor 235
 32-bit registers 24

4004 microprocessor 234
 4040 microprocessor 234
 64-bit constant expressions 1119
 6502 microprocessor 234
 6800 microprocessor 234
 68000 microprocessor 235
 8008 microprocessor 234
 8080 microprocessor 234
 8085 microprocessor 234
 8088 microprocessor 235
 8-bit registers 24

A

AAA instruction 900
 AAD instruction 900
 AAM instruction 900
 AAS instruction 900
 Absolute value (floating point) 628
 Abstract base classes 1091
 Abstract data types 1060
 ABSTRACT keyword 1093
 Abstract method declarations 1091
 Accessing a word in byte addressable memory 141
 Accessing an element of a single dimension array 464
 Accessing data on the stack. 186
 Accessing data via a pointer 410
 Accessing data with a 16-bit bus 144
 Accessing double words in memory 145
 Accessing elements of 3 & 4 dimensional arrays 471
 Accessing elements of a two-dimensional array 470
 Accessing elements of an array 465
 Accessing elements of multidimensional arrays 475
 Accessing fields of a structure 484
 Accessing local variables 815
 Accessing names outside a namespace 497
 Accessing reference parameters 831
 Accessing the characters within a string 426
 Accessing the fields of a class 1069
 Accessing the fields of a UNION 492
 Accessing value parameters 824
 Accessing words at odd addresses 145
 Accessor methods 1060
 Activation records 810
 Active high logic 225
 Active low logic 225
 Actual parameters 836
 ADC instruction 855
 ADD 28
 Adders 223
 Addition (extended precision) 853
 Addition table 848
 Address binding 1375, 1376
 Address bus 139
 Address expressions 171
 Address of operator 160
 Address spaces 140

- Addressable memory 139
- Addressing modes 157
- Addressing modes (Y86) 278
- Address-of operator 191
- AGP Bus 336
- AH 24
- AL 24
- Algorithm 541
- Aliases 430, 494, 557
- Aligning fields within a record 490
- Allocating objects dynamically 1081
- Allocating storage for arrays 474
- AND 605
- AND instruction 68, 910
- AND operation 65, 204
- Anonymous unions 494
- Anonymous variables 160
- Anyexception (try..endtry) 735
- Arbitrary text as macro parameters 976
- Arc cosecant 639
- Arc cosine 639
- Arc cotangent 639
- Arc secant 639
- Arc sin 639
- Architecture 137
- Arctangent 632
- arg.c function 640
- arg.v function 640
- Arithmetic expressions 597, 600
- Arithmetic idioms 606
- Arithmetic logical systems 605
- Arithmetic operators within a constant expression 404
- Arithmetic shift right 78
- Arity 478, 479
- Array access 464
- Array variables 464
- array.cpy function 481
- array.daAlloc function 480
- array.daFree function 480
- array.dArray declaration 479
- array.index function 480
- Arrays 463
- Arrays as structure fields 487
- Arrays of arrays 471
- Arrays of records 486
- Arrays of two or more dimensions 468
- ASCII character set 58, 97, 104
- Assembly language newsgroups 8
- Assert Macro 1252
- Assigning a constant to a variable 597
- Assigning one variable to another 597
- Assignment by reference 428
- Assignments 597
- Associativity 203, 600, 601
- Audio and video data 1117
- Automatic storage allocation 551
- Automatic variables 169
- AX 24

B

- Background colors on the text display 195
- backspace 40
- Base address (of an array) 463
- Base classes 1075
- Based indexed addressing mode 163
- Basic System Components 137
- BCD 87
- BCD arithmetic 897
- BCD numbers 56
- BCD values 397
- BEGIN..END statement 740
- bell character 40
- BH 24
- Biased (excess) exponents 91
- Big-endian data format 928
- binary 53
- Binary Coded Decimal 87
- Binary coded decimal arithmetic 897
- Binary coded decimal numbers 56
- binary data types 56
- Binary Formats 55
- Binary Numbering System 54
- Binary operator 203
- Binding an address to a variable 1376
- Bit data 909
- Bit fields and packed data 81
- Bit masks 910
- Bit offset 909
- Bit runs 909
- Bit scanning 923
- Bit sets 909
- Bit strings 909
- Bits 56
- bits.cnt function 932
- bits.coalesce function 932
- bits.distribute function 932
- bits.extract function 932
- bits.merge8, bits.merge16, and bits.merge32 functions 933
- bits.nibbles8, bits.nibbles16, and bits.nibbles32 functions 933
- bits.reverse8, bits.reverse16, and bits.reverse32 functions 933
- Bitwise logical operators within a constant expression 404
- Bitwise operations 68
- BL 24
- Boolean Algebra 203
- Boolean algebra theorems 204
- Boolean expression canonical form 209
- Boolean expressions 30, 604
- Boolean function equivalence to electronic circuits 221
- Boolean function names 207
- Boolean function numbers 208
- Boolean function simplification 214
- Boolean functions 205
- Boolean functions of n variables 207

- Boolean logical systems 605
 - Boolean map simplification 214
 - Boolean term 209
 - Boolean values 56
 - BOUND instruction 393
 - BP 24
 - Branch out of range 758
 - BREAK 791
 - BREAK statement 36
 - BREAKIF statement 36
 - bs 40
 - BSF and BSR instructions 924
 - BSWAP instruction 928
 - BT, BTC, BTR, and BTS instructions 915
 - Buffering data to improve I/O performance 336
 - Bus contention 261
 - Bus interface unit 256
 - BX 24
 - Byte 57
 - Byte addressable memory array 143
 - Byte enable lines 140, 145
 - Byte strings 935
 - Bytes 56
- C**
- Cache and I/O devices 352
 - Cache Architecture 308
 - Cache associativity 308
 - Cache coherency 269
 - Cache hit 154
 - Cache hit ratio 154
 - Cache line replacement policy 310
 - Cache memory 153
 - Cache miss 154
 - Cache write policies 311
 - Cache, two level 155
 - CALL Instruction 805
 - CALL instruction 541, 805
 - Callee register preservation 544
 - Caller register preservation 544
 - Calling Base Class Methods 1095
 - Canonical form of boolean expressions 208
 - Canonical forms 209
 - Carriage return 40
 - carry 26
 - Carry flag 26, 916
 - carry flag 592
 - Case insensitive string comparisons 436
 - Case labels (non-contiguous) 782
 - Case neutral 19
 - CASE statement 747
 - Case Statement 776
 - Case statement 761
 - cbw instruction 74
 - cdq instruction 74
 - CD-Quality recording 113
 - Celeron microprocessor 239
 - Central Processing Unit 137
 - Central processing unit 24
 - CH 24
 - Change sign (floating point) 629
 - Changing the value of a VAL constant at different points in your program 406
 - Char data type 101
 - Character classification compile-time functions 958
 - Character constants 401
 - Character literal constants 100
 - Character strings 419
 - Characters 96
 - Circular queues 343
 - Circular queues and output transfers 349
 - CL 24
 - CL register in rotate operations 80
 - CL register in SHL instruction 77
 - Class Methods, Iterators, and Procedures 1067
 - Classes 1061
 - clc instruction 84
 - cld instruction 84
 - Clearing the FPU exception bits 633
 - Clearing the Screen 192
 - CLI instruction 348
 - cli instruction 84
 - Clipping (saturation) 76
 - Clock 149
 - Clock frequency 150
 - Clock period 150
 - Clocked logic 228
 - Closure 203
 - Closure of an operator 203
 - cls (clear screen) routine 192
 - CMC instruction 917
 - cmc instruction 84
 - CMPS 935, 943
 - Coalescing bit sets 920
 - Coarse-grained parallelism 269
 - Code reuse 580
 - Code stream parameters 1341
 - Coercion 173
 - Color depth 109
 - Colors on a video display 109
 - Column major ordering 469, 473
 - Combinatorial circuits 223
 - Command line arguments 640
 - Command line parameters 641
 - Comments 22
 - Commutative operators 603
 - Commutativity 203
 - comp.lang.asm.x86 newsgroup 8
 - Compare strings 935
 - Comparing dates 84
 - Comparing floating point numbers 89
 - Comparing registers with signed integer values 175
 - Comparing two strings 436
 - Comparison Instructions (MMX) 1134

- Comparison operators in a constant expression 404
- Compile-Time Constants and Variables 952
- Compile-Time Expressions and Operators 953
- Compile-Time Functions 956
- Compile-Time Language 949
- Compile-time loops 966
- Compile-Time Pattern Matching Functions 958
- Compile-time procedures 969, 985
- Compile-time programs 995
- Compile-time string functions 958
- Compile-time symbol information 959
- Compile-time variables 961
- Complete boolean evaluation 768
- Complex arithmetic expressions 600
- Complex string functions 947
- Composite data types 419
- Computer Architecture 137
- Computing 2**x 631
- Computing MOD using an AND instruction 345
- Concatenating two string literals 401
- Concatenation 433
- Condition codes 26
- Condition jump instructions (opposite conditions). 758
- Conditional compilation 962
- Conditional jump aliases 758
- Conditional Jump Instructions 755
- Conditional jumps (x86) 282
- Conditional statements 761
- Console application 20
- CONSOLE Module (Standard Library) 192
- console.cls routine 192
- console.fillRect routine (standard library) 197
- console.getX 194
- console.getY 194
- console.gotoxy routine 193
- console.puts routine (standard library) 199
- console.putsx routine (standard library) 199
- console.setOutputAttr routine (standard library) 196
- CONST declarations 397
- Constant expressions 172, 403
- Constructing a truth map 215
- Constructing data tables at compile time 996
- Constructing logic functions using only NAND operations 222
- Constructing truth tables from the canonical form 210
- Constructors 1079, 1081
- Constructors and Inheritance 1082
- Contention (for the bus) 261
- Context-free macros 985
- CONTINUE 791
- CONTINUE and CONTINUEIF statements 745
- Control bus 139
- Control characters 98
- Control characters within string constants 402
- Controlling field offsets within a record 489
- conv.strToFt function 640
- Converting Arithmetic Expressions to Postfix Notation 635
- Converting BCD to floating point 624
- Converting between canonical forms 213
- Converting between HLA time format and seconds 515
- Converting binary to hex 61
- Converting Floating Point Expressions to Assembly Language 634
- Converting hex to binary 61
- Converting IF statements to assembly language 761
- Converting Postfix Notation to Assembly Language 637
- Converting UNICODE to ASCII 1124
- Copy by reference 430
- cosecant 639
- Cosine 631
- cot function 639
- Cotangent 639
- Count (string elements) 936
- Counters 231
- Counting bits 925
- CPI (clocks per instruction) 265
- CPU 24, 137
- CPUID instruction 1113
- cr 40
- Create procedure for an object 1081
- Creating libraries 581
- Creating lookup tables 651
- cs.difference function 1134, 1141
- CTL (compile time language) 949
- Current string length 420
- Cursor location on the screen (standard library) 194
- Cursor positioning 193
- cwd instruction 74
- cwde instruction 74
- CX 24
- CYMK color space 110

D

- D (data) flip-flop 229
- DAA instruction 898
- Dangling pointers 414
- DAS instruction 898
- Data bus 138
- Data Transfer Rates 334
- data.pack function 511
- Date arithmetic 512
- Date comparison 84
- Date to string conversions 510
- date.a_toString function 510
- date.datePlusDays function 512
- date.datePlusMonths function 512
- date.dayNumber function 513
- date.dayOfWeek function 513
- date.daysBetween function 512
- date.daysLeft function 513
- date.fromJulian function 512
- date.IsValid function 505
- date.Julian function 512

- date.OutputFormat values 510
- date.Print function 510
- date.today function 509
- date.toString function 510
- date.unpack function 511
- date.validate function 505
- Deadlock 349
- DEC instruction 190
- decimal 53
- Decimal arithmetic 96, 897
- Decisions 760
- Declarations
 - static 21
- Declaring arrays 464
- Decoder circuits 224
- Decoding instruction opcodes 225
- DEFAULT section of a SWITCH statement 748
- Deferring macro parameter text expansion 977
- delete memory deallocation operator (C++) 187
- DeMorgan's Theorems 205
- Denormalized exception (FPU) 614
- Denormalized floating point values 621
- Denormalized values 92
- Destination index 936
- Destroy procedure in a class 1086
- Destructors 1086
- Destructuring 774
- Device Drivers 353
- DH 24
- DI 24
- Digital video 115
- Direct addressing mode 158
- Direct jump instructions 753
- Direct mapped caches 308
- Direct Memory Access 333
- Direct memory access 331
- Direction flag 936, 937
- Dirty bits 312
- Disassembly 379
- Displacement only addressing mode 158
- Display (in an activation record) 806
- Display (lexical nesting data structure) 1375
- dispose memory deallocation operator (Pascal) 187
- Distributed Shared Memory 304
- Distributing bit strings 920
- Distributive law 204
- div (within a constant expression) 404
- DIV simulation 607
- DL 24
- DMA 331
- Domain conditioning 650
- Domain of a function 649
- Dope vector 478
- Dot operator 484
- Double precision floating point format 91
- Double word storage in byte addressable memory 141
- Double word strings 935
- Double words 59

- double words 56
- Downloading MASM 8
- Dual I/O ports 329
- Duality 205
- DUP operator 464
- dwords 56
- DX 24
- Dynamic Arrays 477
- Dynamic link 1308, 1378
- Dynamic memory allocation 187, 412
- Dynamic nesting of control structures 731
- Dynamic Object Allocation 1081
- Dynamic type systems 495

E

- Eager macro parameter text expansion 977
- Eager vs. deferred macro parameter evaluation 977
- EAX 24
- EBP 24
- EBX 24
- ECX 24
- EDI 24
- EDX 24
- Effective addresses 160
- EFLAGS 25
- EFLAGS register 85
- EFLAGS register 184
- EIP register 26
- EISA bus 334
- Electronic circuit equivalence to boolean functions 221
- ELSE 30, 32, 761
- ELSEIF 30, 32
- Embedding control characters in string constants 402
- EMMS Instruction 1139
- EMMS instruction 1115
- Encoding instructions 271
- ENDFOR 34, 36
- ENDIF 30, 32
- ENDTRY 37
- ENDWHILE 30, 33
- ENUM 408
- Enumerated data types 408
- eoln 40
- Errors when using pointers 189
- Escape character sequences 401
- ESI 24
- ESP 24
- ex.DivisionError exception 590
- ex.FileOpenFailure exception 522
- ex.IntoInstr exception 590
- ex.InvalidDate exception 505
- ex.MemoryAllocationFailure exception 188
- ex.StringIndexError exception 428
- ex.StringOverflow exception 423, 434
- EXCEPTION 37
- Exception flags (FPU) 616

- Exception handling 37
- Exception masks (FPU) 614
- Exception values 735
- Exclusive-or 65
- Exclusive-OR operation 207
- Exclusive-or operation 67
- Executing a loop backwards 798
- Execution units 265
- EXIT 740
- EXIT and EXITF statements 546
- EXITIF 740
- exp function 640
- Exponent 88
- Expression classification functions 960
- Expressions 600
- Expressions and temporary values 603
- Extended precision (80 bit) floating point values 397
- Extended precision addition 853
- Extended precision AND 873
- Extended precision comparisons 857
- Extended precision division 864
- Extended precision floating point format 91
- Extended precision formatted I/O 883
- Extended precision I/O 878
- Extended precision input routines 884
- Extended precision multiplication 860
- Extended precision NEG 872
- Extended precision NOT 874
- Extended precision OR 874
- Extended Precision Rotates 878
- Extended precision shifts 875
- Extended precision XOR 874
- EXTERNAL directive 572, 575
- Extracting bit sets 920
- Extracting bit strings 919, 930

F

- F2XM1 instruction 631
- FABS instruction 628
- FADD/FADDP instructions 625
- Falling edge of a clock 150
- False (representation) 604
- FBLD instruction 624, 901
- FBLD/FBSTP instructions 624
- FBSTP Instruction 624
- FBSTP instruction 901
- FCHS instruction 629
- FCLEX/FNCLEX instructions 633
- FCOM, FCOMP, and FCOMPP instructions 629
- FCOM/FCOMP/FCOMPP instructions 629
- FCOMI and FCOMIP instructions 629
- FCOS instruction 631
- FDIV/FDIVP/FDIVR/FDIVRP instructions 626
- FIADD instruction 634
- Fibonacci sequence 846
- FICOM instruction 634

- FICOMP instruction 634
- FIDIV instruction 634
- FIDIVR instruction 634
- Field alignment within a record 490
- Field Offsets Within a Record 489
- Field width 41
- FILD Instruction 623
- File handles 521
- File Storage (in the memory hierarchy) 304
- fileio.getf function 639
- Filling a Rectangular Section of the Screen 197
- fillRect routine (standard library) 197
- FIMUL instruction 634
- Fine-grained parallelism 268
- FINIT/FNINIT instructions 633
- First-in, First-out (FIFO) cache replacement policy 311
- FIST instruction 623
- FISTP Instruction 623
- FISUB instruction 634
- FISUBR instruction 634
- FLAG register 85
- Flags 25
- Flags (and CMP) 592
- FLD Instruction 621
- FLD1 instruction (load 1.0) 631
- FLDCW instruction 633
- FLDL2E instruction (load lg(e)) 631
- FLDL2T instruction (load lg(10)) 631
- FLDLG2 instruction (load log(2)) 631
- FLDLN2 instruction (load ln(2)) 631
- FLDPI instruction (load pi) 631
- FLDZ instruction (load 0.0) 631
- Flip-flops 229
- Floating point arithmetic 611
- Floating point comparisons 89, 629
- Floating point data types 619
- Floating point registers as procedure parameters 1341
- Floating point unit 237
- Floating point values 60
- Flushing the pipeline 261
- FMUL/FMULP instructions 626
- for 923
- For loops 790
- FOR statement 34
- Forcing bits to one 68
- Forcing bits to zero 68
- Forcing bits to zero (MMX) 1134
- Forcing selected bits to one. 911
- FOREACH..ENDFOR 843
- Foreground colors on the text display 195
- FOREVER loops 787
- FOREVER statement 36
- Formal parameters 836
- FORWARD (variable and type declarations) 1089
- Forward procedure declarations 567
- Four-way set associative caches 310
- FPATAN instruction 632
- FPREM/FPREM1 instructions 628

FPTAN instruction 632
 FPU busy bit 618
 FPU condition code bits 616
 FPU Control Register 612
 FPU control word 633
 FPU exception bits 633
 FPU exception flags 616
 FPU exception masks 614
 FPU interrupt enable mask 615
 FPU precision control 614
 FPU Registers 611
 FPU rounding control 613
 FPU stack fault flag 616
 FPU Status Register 615
 FPU Status register 633
 FPU top of stack pointer 618
 free 187
 Free function 413
 FRNDINT instruction 628
 FSIN instruction 631
 FSINCOS instruction 631
 FSQRT instruction 627
 FST instruction 622
 FSTCW instruction 633
 FSTP Instruction 622
 FSTSW instruction 615, 629
 FSTSW/FNSTSW instructions 633
 FSUB/FSUBP/FSUBR/FSUBRP instructions 625
 FTST instruction 630
 Full adders 223
 Function Computation via Table Look-up 647
 Function instance 1376
 Function numbers 208
 Function overloading 990
 Function results 557, 1370
 Functional units 255
 FXCH Instruction 622
 FYL2X instruction 632
 FYL2XP1 instruction 632

G

General protection fault 165
 General purpose registers 24
 Generating a unique label in an HLA program 984
 Get routine 46
 Getc routine 43
 Getting an integer value 44
 getY routine (standard library) 194
 Global memory locations as parameters 1341
 gotoxy routine (standard library) 193
 Guard digits/bits 88

H

H.O. 55
 Half adder 223

Handshaking 337
 Hard Copy storage (in the memory hierarchy) 305
 Harvard architecture 262
 Header files 576
 heap 187
 Hello World 20
 Hertz (Hz) 150
 Hexadecimal 56
 hexadecimal 53
 Hexadecimal Calculators 62
 Hexadecimal calculators 62
 Hexadecimal input (extended precision) 887
 Hexadecimal numbering system 60
 Hexadecimal output (extended precision) 879
 High order bit 55, 57
 High order byte 58
 High order nibble 57
 High order word 60
 High-speed devices 333
 History of the 80x86 CPU 234
 HLA 4
 Identifiers 19
 HLA pointers 410
 HLA Standard Library 12, 15, 38
 HLA stdlib
 stdin.get 22
 stdout.put 20
 HLA strings 421
 Hybrid control structures 802
 Hybrid parameter passing facilities 838

I

I/O 24, 331
 I/O address bus 140
 I/O and the cache 352
 I/O mapped input/output 331
 I/O port 327
 I/O Speed Hierarchy 333
 I/O subsystem 146
 I/O-mapped input/output 332
 iAPX432 microprocessor 235
 Icon programming language 428
 Identifiers 19
 Identity element for boolean operations 204
 Identity elements 204
 IEEE floating point standard (754 & 854) 90
 IF 30
 IF statement 32
 IF..THEN..ELSE 760, 761
 Implementation section of an abstract data type 1060
 IN instruction 332
 IN operator 31
 INC instruction 190
 INCLUDE directive 570
 Include files 20
 Indexed addressing mode 160

- Indexed addressing mode (x86) 279
- Indirect addressing mode 279
- Indirect calls 839
- Indirect jump 787
- Indirect jump instructions 753
- Indirect Jumps 784
- Indirect jumps 761
- Induction variables 801
- Industry Standard Architecture (ISA) 334
- Infinite loops 787
- Infinite precision arithmetic 87
- Infix notation 634
- Information hiding 1060
- Inheritance 1064, 1075
- INHERITS keyword (classes) 1065
- Inhibition function 1134
- Inhibition operation 207
- Initializing a string 935
- Initializing strings and arrays 946
- Input conditioning 651
- Input/output 24
- Inserting a bit field into some other value 911
- Instance 1376
- Instances (of a class) 1063
- Instruction composition 558
- Instruction pointer register 247
- Instruction set architecture 270
- int16 21
- int32 21
- int8 21
- Integer input 44
- Integer output 41
- Interface section of an abstract data type 1060
- Interrupt enable mask (FPU) 615
- Interrupt service routine 342
- Interrupt service routine (x86) 282
- Interrupt vector 343
- Interrupts 342
- INTMUL instruction 393
- INTO instruction 393
- Invalid operation exception (FPU) 614
- Invariant computations 799
- Inverse element 204
- Inverse element for boolean operations 204
- Inverting bits 68
- Inverting selected bits 913
- IRET instruction 343
- IS operator (object IS someType) 1094
- ISA bus 334
- ISR 342
- Iterators 843

J

- JA instruction 757
- JAE instruction 757
- JB instruction 757

- JBE instruction 757
- JC instruction 756
- JE instruction 757, 758
- JF Instruction 759
- JG instruction 758
- JGE instruction 758
- JL instruction 758
- JLE instruction 758
- JMP instruction 753
- JNA instruction 757
- JNAE instruction 757
- JNB instruction 757
- JNBE instruction 757
- JNC instruction 756
- JNE instruction 757, 758
- JNG instruction 758
- JNGE instruction 758
- JNL instruction 758
- JNLE instruction 758
- JNO instruction 756
- JNP instruction 756
- JNS instruction 756
- JNZ instruction 251, 756
- JO instruction 756
- JP instruction 756
- JPE instruction 756
- JPO instruction 756
- JS instruction 756
- JT instruction 759
- Julian day numbers 512
- JZ instruction 756

K

- Karnaugh Maps 203
- Kost significant bit 57

L

- L.O. 55
- Labels 751
- LAHF instruction 85
- lahf instruction 84
- Large parameters 832
- Large programs 569
- Last-in, first-out data structures 180
- Latency (of a cache access) 307
- Lazy evaluation 1354
- LEA instruction 191
- Leap years 507
- Least recently used (LRU) cache replacement 311
- Least significant bit 57
- Left associative 204
- Left associative operators 601
- Left shift operation 76
- Length (field of an HLA string) 422
- Length-prefixed strings 420

- Level One Cache 304
 - Level Two Cache 304
 - Lexical Nesting 1375
 - Lexical scope 547
 - Lexicographical ordering 437, 944
 - lf 40
 - LIB (library) files 581
 - LIB.EXE program 582
 - Libraries 581
 - Lifetime 170
 - Lifetime (of a variable) 547, 551
 - Lifetime of a variable 1376
 - LIFO 180
 - Linefeed 40
 - LINK.EXE program 582
 - Linker 569
 - Literal record constants 485
 - Literals (boolean) 209
 - Little endian data format 928
 - ln function 640
 - Local symbols in a macro 981
 - Local variables 547, 815
 - Locality of reference 153, 306
 - Locating the Cursor (standard library) 194
 - LOCK prefix 1459
 - LODS 935, 947
 - log function 640
 - Logic Instructions (MMX) 1133
 - Logical AND 204
 - Logical AND operation 65
 - Logical complement 204
 - Logical exclusive-OR 207
 - Logical exclusive-or operation 65, 67
 - Logical inhibition 207
 - Logical NAND 207
 - Logical NOR 207
 - Logical NOT 207
 - Logical NOT operation 65, 67
 - Logical Operations on Binary Numbers 68
 - Logical Operations on Bits 65
 - Logical operators within a constant expression 404
 - Logical OR 204
 - Logical OR operation 65, 66
 - Logical shift right 78
 - Logical XOR operation 65
 - Loop control variables 788
 - LOOP instruction 251
 - Loop invariant computations 799
 - Loop register usage 795
 - Loop termination 796
 - Loop termination test 787
 - Loop unraveling 800
 - Loops 787
 - LOOPZ and LOOPNZ instructions 341
 - Low Level Control Structures 751
 - Low order bit 55, 57
 - Low order byte 58
 - Low order nibble 57
 - Low order word 60
 - Low-speed devices 333
- ## M
- Machine idioms 606
 - Machine state, saving the 543
 - Macro parameter brackets 976
 - Macro parameter expansion 971
 - Macro parameters 971
 - Macros 969
 - Make files 578
 - malloc 187
 - Malloc function 412
 - Managing large programs 569
 - Managing libraries 581
 - Manifest constants 398
 - Mantissa 88
 - Map method for boolean function simplification 214
 - mask 910
 - Masking 68
 - Masking in bits 68
 - Masking out 57
 - Masking out bits 68
 - Masking out bits (setting them to zero) 910
 - MASM 8
 - MASM32 12
 - Maximum addressable memory 139
 - Maximum string length 421
 - MaxStrLen 422
 - Medium-level control structures 759
 - Medium-speed devices 333
 - Megahertz (Mhz) 150
 - Memory 24
 - Memory access 150
 - Memory access time 150
 - Memory access violation exception 414
 - Memory banks 143
 - Memory cells 229
 - Memory Hierarchy 303, 305
 - Memory mapped files 314
 - Memory protection 312
 - Memory subsystem 140
 - MemoryAllocationFailure exception 188
 - Memory-mapped I/O 331
 - Merging bit strings 929
 - Merging source files during assembly 570
 - Metaware Professional Pascal 1307
 - Methods 1061
 - Microprocessor clock 149
 - MIDI 114
 - MIMD (Multiple Instruction, Multiple Data) 268
 - Minimum field width 41
 - Mixed Integer and Floating Point Arithmetic 638
 - MM0, MM1, MM2, MM3, MM4, MM5, MM6, and MM7 (MMX Registers) 1114
 - MMU (memory management unit) 314

- MMX (multimedia extensions) 238
- MMX arithmetic instructions 1131
- MMX Comparison Instructions 1134
- MMX Data Types 1116
- MMX Instruction Operands 1118
- MMX Logic Instructions 1133
- MMX Programming Paradigm 1140
- MMX Registers 1114
- MMX Shift Instructions 1138
- mod (within a constant expression) 404
- MOD calculation using AND 345
- Modulo (floating point remainder) 628
- Monochrome displays 110
- MOV instruction 157
- Mov instruction 27
- MOVD instruction 1123
- Move strings 935
- MOVQ instruction 1123
- MOVS 935, 938
- Movsx instruction 74
- movzx instruction 74
- MP3 files 111
- Multi-precision Division 864
- MUL simulation 606
- Multidimensional arrays 468
- Multi-level page tables 313
- Multi-part macros 985
- Multiplication table 848
- Multiprecision addition 853
- Multi-precision comparisons 857
- Multiprecision operations 853
- Multiprecision subtraction 856
- Multiprocessing 268

N

- Name space pollution 496, 583
- Names of boolean functions 207
- NAMESPACE declarations 584
- Namespaces 496
- NAND gates 221
- NAND operation 207
- Near-Line Storage subsystems 305
- NEG instruction 71
- Negation (floating point) 629
- Negative numbers 70
- Nesting record definitions 488
- Nesting TRY..ENDTRY statements 730
- Network Storage (in the memory hierarchy) 304
- New line 41
- NEW memory allocation operator (C++ or Pascal) 187
- newln 41
- Newsgroups 8
- Nibble 56
- Nibbles 56
- nl 20, 40
- nl (newline) constant 403

- NOALIGNSTK option 813
- Nonuniform Memory Access (NUMA) 304
- NOR operation 207
- Normalized floating point numbers 620
- Normalized values 92
- NOT 605
- NOT IN operator 31
- NOT instruction 68
- NOT operation 65, 67, 204, 207
- NuBus bus 334
- NULL pointer references 165
- NUMA 304, 315
- Number of boolean functions 207
- Numeric Input 44
- Numeric output 41
- Numeric representation 63
- N-way set associative caches 309

O

- Object Initialization 1079
- Objects 1063
- Off-Line storage subsystems 305
- One-way set associative cache 308
- On-line and memory subsystems 304
- Opcodes 247
- Operation codes 247
- Operator precedence 600
- Operator Precedence and Associativity (compile-time operators) 955
- Opposite condition jump conditions 758
- Opposite jumps 758
- Optional macro parameters 975
- OR 65, 605
- OR instruction 68, 911
- OR Operation 66
- OR operation 204
- OUT instruction 332
- Out of Order Execution 266
- Outer product 848
- Outputting register values 176
- Overflow exception (FPU) 614
- Overflow flag 26
- overflow flag 592
- Overlapping blocks (string operations) 940
- Overloading 990
- Overriding a method 1065

P

- Packed arithmetic instructions 1131
- Packed arrays of bit strings 922
- Packed data 81
- Packed decimal arithmetic 901
- Packing and unpacking bit strings 917
- PACKSSDW instruction 1123
- PACKSSWB instruction 1123

- PACKUSDW instruction 1123
- PACKUSWB instruction 1124
- PADDB, PADDW, and PADD instructions 1131
- Padding a record to some number of bytes 491
- Padding parameter data 827
- PADDSB and PADDSW instructions 1131
- PADDUSB and PADDUSW instructions 1132
- Paging 312
- Palette (video card) 109
- PAND instruction 1133
- PANDN instruction 1133
- Parallel computation with MMX instructions 1117
- Parallel execution of instructions 253
- Parallel printer port 337
- Parameter expansion in macros 971
- Parameters 552, 816, 1341
- Parameters (macros) 971
- Parameters, variable length 821
- Parity flag 914
- Parse 286
- Pass by lazy evaluation 1354, 1395
- Pass by name 1395
- Pass by name parameters 1354
- Pass by reference 1395
- Pass by reference parameters 555, 817, 1354
- Pass by result 1395
- Pass by value 1394
- Pass by value parameters 552, 817, 1354
- Pass by value/returned 1354
- Pass by value/returned parameters 1354
- Pass by value-result 1395
- Passing large objects as parameters 832
- Passing parameters as parameters 836
- Passing parameters by name 1360
- Passing parameters by result 1359
- Passing parameters by value 1394
- Passing parameters from one procedure as parameters to another 1363
- Passing parameters in a parameter block 1341, 1353
- Passing parameters in global memory locations 1341
- Passing parameters in global variables 1346
- Passing parameters in registers 818, 1341, 1342
- Passing parameters in the code stream 820, 1341, 1351
- Passing parameters on the stack 822, 1341, 1347
- Passing reference parameters 834
- Passing value parameters 825
- Passing variables from different lex levels as parameters 1394
- Patch panel programming 246
- Pattern matching functions (compile-time) 958
- PCI bus 334
- PCMPEQB, PCMPEQW, and PCMPEQD instructions 1134
- PCMPGTB, PCMPGTW, and PCMPGTD instructions 1134
- PCMPLTx instructions 1136
- Pentium™ Processor 237
- Performance improvements for loops 796
- Performance of Memory Subsystems 306
- Peripheral Connection Interface (PCI) 334
- Pipeline flush 261
- Pipeline stalls 261
- Pipelined instruction execution 237
- Pipelining 259
- PMADDWD instruction 1132
- PMULHUW instruction 1132
- PMULHW instruction 1132
- PMULLW instruction 1132
- Pointer constants and pointer constant expressions 411
- Pointer errors 189
- Pointer problems 413
- POINTER TO type declaration 411
- Pointers 409
- Polled I/O 342
- polymorphism 1066
- POP instruction 177
- POPA and POPAD instructions 183
- POPF and POPFD instructions 184
- POR instruction 1133
- Port 327
- Positioning the Cursor 193
- Postfix notation 635
- Pound sign operator ("#") 100
- Precedence 204, 600
- Precision exception (FPU) 614
- Prefetch Queue 255
- Prefetch queue 256
- Preserving registers 179, 544
- Priming the pump (for output devices) 350
- Principle of duality 205
- Private fields in a class 1062
- Procedural parameters (passing procedures as parameters) 842
- Procedure call syntax 542
- Procedure instance 1376
- Procedure invocation 541, 805
- Procedure Overloading in classes 1085
- Procedure pointers 839
- Procedures and the Stack 807
- Processor size 139
- Product of maxterms representation 209
- Professional Pascal 1307
- Program unit 1380
- Programming in the large 569
- PSARW and PSARD instructions 1138
- Pseudo-opcode 166
- PSLLW, PSLLD, and PSLLQ instructions 1138
- PSLRW, PSLRD, and PSLRQ instructions 1138
- PSUBB, PSUBW, and PSUBD instructions 1132
- PSUBSB and PSUBSW instructions 1132
- PSUBUSB and PSUBUSW instructions 1132
- PUNPCKHBW instruction 1124
- PUNPCKHDQ instruction 1124
- PUNPCKHWD instruction 1124
- PUNPCKLBW instruction 1124
- PUNPCKLDQ instruction 1124

PUNPCKLWD instruction 1124
PUSH instruction 176
PUSHA instruction 183
PUSHAD instruction 183
PUSHD instruction 176
PUSHF and PUSHFD instructions 184
PUSHW instruction 176
Put routine 42
putiXsize 41
PXOR instruction 1133

Q

Quicksort 564
Quicktime 115
QWORD data type 397
qwords 56

R

radix 61
RAISE statement 427, 735
Range of a function 649
RCL instruction 80
RCR instruction 80
Read control line 140
Read/write input/output ports 327
Read/write ports 329
Reading from memory 141
Reading integer values 44
Read-only (input) ports 327
READONLY declaration section 167
Read-only ports 329
READONLY variables as constants 398
Realloc function 413
Rearranging expressions to make them more efficient 773
Record constants 485
Record field alignment 490
Record offsets 489
Records 483
Records as record fields 487
Recursion 563
Reference parameters 831, 834
Register addressing modes 157
Register indirect addressing mode 159
Register indirect jump instruction 753
Register preservation 544
Register preservation in a TRY..ENDTRY statement 739
Register Renaming 266
Register type coercion 175
Register usage in loops 795
Registers 24
Registers (electronic implementation) 230
Registers (in the memory hierarchy) 303
Registers as procedure parameters 818, 1341, 1342
Registers as signed integer values 175
Relational operators 31

Remainder (floating point) 628
Removing unwanted data from the stack 184
REPEAT 30
Repeat Until loop 788
REPEAT..UNTIL loops 787
REPEAT..UNTIL statement 35
Replacement policy (for caches) 310
Representing audio information 111
Required macro parameters 975
Resume frame (for iterators) 1308
RET instruction 541, 805
RETURNS Option 560
Reverse polish notation 634
Reversing a bit string 927
RGB color space 109
Right associative operators 204, 601
Right shift operation 77
Rising edge of a clock 150
ROL instruction 79
ROR instruction 79
Rotate left 79
Rotate right 79
Rounding a floating point value to an integer 628
Rounding control 613
Rounding control (FPU) 613
Row major ordering 469
RPN 634
Run of ones 909
Run of zeros 909
Run-time language 949
Run-time Type Information 1094

S

SAHF instruction 85, 629
sahf instruction 84
SAR instruction 79
Saturation 73
Saturation arithmetic 1118
Saving the machine state 543
SBB instruction 856
Scanning for bits 923
SCAS 935, 946
Schematic Symbols 221
Scope 1375
Scope (of a name) 547
Searching for a bit 923
Searching for a bit pattern 931
Searching for data within a string 935
secant 639
Self-modifying code 386
Separate compilation 569
Sequential logic 228
Set/reset flip-flop (SR flip-flop) 229
SETcc Instructions 593
setOutputAttr routine (standard library) 196
Setting selected bits 911

- Seven segment decoder 223
- Shift arithmetic right operation 79
- Shift Instructions (MMX) 1138
- Shift registers 230
- SHL instruction 76
- SHLD and SHRD instructions 876
- Short circuit boolean evaluation 769
- SHR instruction 77
- SI 24
- Side effects 562
- Sign bit 70
- Sign extension 73, 590
- Sign flag 26
- sign flag 592
- Signed 69
- Signed and unsigned numbers 69
- Signed comparisons 594
- Signed decimal input (extended precision) 895
- Signed decimal output (extended precision) 882
- Signed division 590
- Signed integer output 41
- Significant digits 88
- SIMD 1117
- SIMD (Single Instruction, Multiple Data) 268
- Simplification of boolean functions 214
- Simulating DIV 607
- Simulating MUL 606
- Sine 631
- Single Instruction Multiple Data model 1117
- Single Instruction, Single Data execution model 268
- Single precision floating point format 90
- SISD (single instruction, single data) 268
- Sixteen-bit bus data access 144
- Size of a processor 139
- SNOBOL4 programming language 428
- Source index 936
- SP 24
- Spaghetti code 786
- Spatial locality of reference 153
- Square root 627
- SR (set/reset) flip flop 229
- ST0..ST7 (FPU registers) aliasing with MMX registers 1114
- Stack fault flag (FPU) 616
- Stack frame 810, 1308
- Stack manipulation by procedure calls 807
- Stack Segment 176
- Stack-based parameters for procedures 1341
- Stalls 261
- Standard entry sequence (to a procedure) 813
- Standard exit sequence (from a procedure) 814
- Standard input 40
- Standard Library 38
- Standard Macros 969
- Standard output 40
- State machine 784
- State machines 232
- State variable 784
- Statement Labels 751
- Static data objects in a class 1063
- STATIC declaration section 167
- Static declaration section 21
- Static link 1378
- Static Procedures (in a class) 1066
- std instruction 84
- Stdin.a_gets function 425
- stdin.eoln 103
- Stdin.FlushInput 46
- stdin.FlushInput 103
- stdin.get 22, 65, 102
- Stdin.Get routine 46
- Stdin.getc 43
- stdin.getdw 65
- stdin.getf function 638
- stdin.geth 65
- Stdin.gets function 425
- stdin.getu16 72
- stdin.getu32 72
- stdin.getu8 72
- stdin.getw 65
- Stdin.ReadLn 46
- stdio.bell 40
- stdio.bs 40
- stdio.cr 40
- stdio.lf 40
- stdio.tab 40
- stdlib.hhf 20
- stdout.newln 41, 541
- stdout.newln function 805
- stdout.put 20, 42, 65, 101
- stdout.putc 101
- stdout.putcsize 101
- stdout.putdw 65
- stdout.puth 65
- stdout.puti16 41
- stdout.puti32 41
- stdout.puti8 41
- stdout.putiXsize 41
- stdout.putr32 94
- stdout.putr64 94
- stdout.putr80 94
- stdout.putu16 72
- stdout.putu16size 72
- stdout.putu32 72
- stdout.putu32size 72
- stdout.putu8 72
- stdout.putu8size 72
- stdout.putw 65
- STI instruction 348
- sti instruction 84
- STORAGE declaration section 168
- Stored program computer systems 246
- Storing double words in byte addressable memory 141
- Storing words in byte addressable memory 141
- STOS 935, 946, 947
- str.a_cat function 433

- Str.a_cpy function 432
- str.a_delete function 435
- str.a_insert function 435
- str.a_substr function 435
- str.cat function 433
- str.cpy function 430
- str.delete function 435
- str.eq function 436
- str.ge function 436
- str.gt function 436
- str.ieq function 436
- str.ige function 437
- str.igt function 437
- str.ile function 437
- str.ilt function 437
- str.index function 437
- str.in_ function 437
- str.insert function 435
- str.le function 436
- str.length function 433
- str.lt function 436
- str.ne function 436
- str.strRec data type 422
- str.strRec definition 489
- str.substr function 435
- str.uppercase function 1142
- Stralloc function 423
- Strfree function 424
- String assignment by reference 428
- String comparisons 436
- String concatenation 401, 433
- String constant initializers in the CONST section 402
- String constants 401
- String constants containing control characters 402
- String Functions (compile-time functions) 958
- String instructions 935
- String Operators within a constant expression 404
- String pointers 421
- String primitives 935
- String representation 489
- STRUCT assembler directive 483
- Structure, accessing fields of... 484
- Structured gotos 740
- Structures 483
- Structures as structure fields 487
- SUB 28
- Subroutine instance 1376
- Substring operation 435
- Subtraction table 848
- Sum of minterms representation 209
- Superscalar CPUs 237, 265
- SWITCH Statement 776
- SWITCH statement 747
- Symbol tables 287
- Symbols reserved by HLA 982
- Symbols that begin and end with a single underscore 982
- Synthesizing a While loop 787
- System bus 24, 138

- System Buses 334
- System clock 149
- System clock frequency 150
- System clock period 150
- System date function 509
- System time 514
- System timing 149

T

- tab 40
- Tables 647
- Tag field 495
- Taking the address of a statement label 751
- Tangent 632
- TBYTE data type 397
- Tbyte values (BCD) 902
- Temporal locality of reference 153
- Temporary values in an expression 603
- TenToX function 640
- Term (boolean) 209
- Termination test (for loops) 787
- Termination test for loops 796
- Test for zero (floating point) 630
- TEST Instruction 596
- TEST instruction 338, 914
- Text Attributes (on the display) 195
- Text constants 402, 492
- THEN 30
- Theorems of boolean algebra 204
- THIS 1069
- Thrashing 314
- Thunk 1361
- Time 514
- Time Input/Output 515
- time.curTime function 514
- time.hmsToSecs function 515
- time.secstoHMS function 515
- time.timerec definition 514
- Time-outs on peripheral devices 340
- Translation Lookaside Buffer (TLB) 313
- Treating registers as signed integer values 175
- True (representation) 604
- Truth maps 214, 215
- truth table 66
- Truth tables 205
- TRY..ENDTRY statement 37, 729
- TTL logic levels 138
- Two level caching system 155
- Two's complement 59
- Two's complement representation 70
- TwoToX function 640
- Two-way set associative caches 309
- Type coercion 173, 491
- Type conversion 957
- TYPE declaration section 407
- Type operator 174

U

UCR Standard Library for 80x86 Assembly Language
 Programmers 3
 Underflow exception (FPU) 614
 UNICODE 59, 108, 1124
 Uninitialized pointers 413
 Unions 492
 Unique boolean functions 207
 Unit activation 1376
 UNITS 572
 Universal boolean function (NAND) 221
 Universal boolean functions (NOR) 223
 Unpacking bit strings 917
 Unprotected (try..endtry) 732
 Unraveling loops 800
 Unravelling loops 999
 Unrolling loops 999
 Uns16 72
 Uns32 72
 Uns8 72
 Unsigned comparisons 594
 Unsigned decimal input (extended precision) 891
 Unsigned Decimal Output (extended precision) 879
 Unsigned division 590
 unsigned multiplication 588
 Unsigned numbers 69
 Unsigned variable declarations 72
 UNTIL 30, 35
 Untyped Reference Parameters 843
 Upper case conversion (MMX) 1144
 User-defined exceptions 735

V

VAL (value parameter specification) 554
 VAL declaration section 406
 VAL declarations 397
 Value parameters 824
 VAR (pass by reference parameters) 555
 VAR declarations 169
 Variable length parameters 821
 Variable lifetime 170, 1375, 1376
 Variable number of macro parameters 974
 Variable-length instructions 274
 Variant types 495
 Vars (_vars_) constant in a procedure 816
 Veitch Diagrams 203
 Very Long Instruction Word 267
 Video and audio data 1117
 Video display 109
 Virtual Memory 304, 312
 Virtual method calls 1066
 Virtual method table 1072
 Virtual Method Tables 1073
 Virtual Methods 1066
 VMT 1072, 1075

Von Neuman Architecture 24
 Von Neumann, John 137

W

Wait states 151
 WAV files 111
 WHILE 30
 While loop 787
 WHILE loops 787
 WHILE statement 33
 Word access in byte addressable memory 141
 Word strings 935
 Words 56, 58
 Words stored at odd addresses 145
 Working sets 314
 Wraparound arithmetic 1118
 Write control line 140
 Write-back cache write policy 311
 Write-only ports 329
 Write-through cache write policy 311
 Writing to memory 140

X

x86 conditional jumps 282
 XLAT instruction 648
 XOR 605
 XOR instruction 68, 913
 XOR operation 65, 67

Y

Y2K 83
 Y86 Addressing modes 278
 Y86 Hypothetical Processor 276
 Y86 opcodes 279
 Yield 844
 YtoX function 640

Z

Z80 microprocessor 234
 Z8000 microprocessor 235
 Zero divide exception (FPU) 614
 Zero extension 590
 Zero flag 26
 zero flag 592
 Zero terminating byte (in HLA strings) 421
 Zeroing selected bits 910
 Zero-terminated strings 419

Volume One: Data Representation

Chapter One: Foreword

An introduction to this text and the purpose behind this text.

Chapter Two: Hello, World of Assembly Language

A brief introduction to assembly language programming using the HLA language.

Chapter Three: Data Representation

A discussion of numeric representation on the computer.

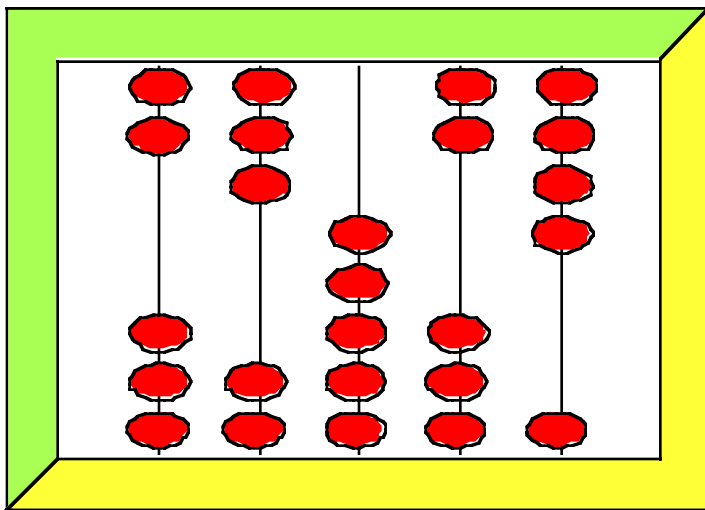
Chapter Four: More Data Representation

Advanced numeric and non-numeric computer data representation.

Chapter Five: Questions, Projects, and Laboratory Exercises

Test what you've learned in the previous chapters!

These five chapters are appropriate for all courses teaching machine organization and assembly language programming.



Nearly every text has a throw-away chapter as Chapter One. Here's my version. Seriously, though, some important copyright, instructional, and support information appears in this chapter. So you'll probably want to read this stuff. Instructors will definitely want to review this material.

1.1 Foreword to the HLA Version of "The Art of Assembly..."

In 1987 I began work on a text I entitled "How to Program the IBM PC, Using 8088 Assembly Language." First, the 8088 faded into history, shortly thereafter the phrase "IBM PC" and even "IBM PC Compatible" became far less dominant in the industry, so I retitled the text "The Art of Assembly Language Programming." I used this text in my courses at Cal Poly Pomona and UC Riverside for many years, getting good reviews on the text (not to mention lots of suggestions and corrections). Sometime around 1994-1995, I converted the text to HTML and posted an electronic version on the Internet. The rest, as they say is history. A week doesn't go by that I don't get several emails praising me for releasing such a fine text on the Internet. Indeed, I only hear three really big complaints about the text: (1) It's a University textbook and some people don't like to read textbooks, (2) It's 16-bit DOS-based, and (3) there isn't a print version of the text. Well, I make no apologies for complaint #1. The whole reason I wrote the text was to support my courses at Cal Poly and UC Riverside. Complaint #2 is quite valid, that's why I wrote this version of the text. As for complaint #3, it was really never cost effective to create a print version; publishers simply cannot justify printing a text 1,500 pages long with a limited market. Furthermore, having a print version would prevent me from updating the text at will for my courses.

The astute reader will note that I haven't updated the electronic version of "The Art of Assembly Language Programming" (or "AoA") since about 1996. If the whole reason for keeping the book in electronic form has been to make updating the text easy, why haven't there been any updates? Well, the story is very similar to Knuth's "The Art of Computer Programming" series: I was sidetracked by other projects¹.

The static nature of AoA over the past several years was never really intended. During the 1995-1996 time frame, I decided it was time to make a major revision to AoA. The first version of AoA was MS-DOS based and by 1995 it was clear that MS-DOS was finally becoming obsolete; almost everyone except a few die-hards had switched over to Windows. So I knew that AoA needed an update for Windows, if nothing else.

I also took some time to evaluate my curriculum to see if I couldn't improve the pedagogical (teaching) material to make it possible for my students to learn even more about 80x86 assembly language in a relatively short 10-week quarter.

One thing I've learned after teaching an assembly language course for over a decade is that support software makes all the difference in the world to students writing their first assembly language programs. When I first began teaching assembly language, my students had to write all their own I/O routines (including numeric to string conversions for numeric I/O). While one could argue that there is some value to having students write this code for themselves, I quickly discovered that they spent a large percentage of their project time over the quarter writing I/O routines. Each moment they spent writing these relatively low-level routines was one less moment available to them for learning more advanced assembly language programming techniques. While, I repeat, there is some value to learning how to write this type of code, it's not all that related to assembly language programming (after all, the same type of problem has to be solved for *any* language that allows numeric I/O). I wanted to free the students from this drudgery so they could learn more about assembly language programming. The result of this observation was "The UCR Standard Library for 80x86 Assembly Language Programmers." This is a library containing several hundred I/O and utility functions that students could use in their assembly language programs. More than nearly anything else, the UCR Standard Library improved the progress students made in my courses.

1. Actually, another problem is the effort needed to maintain the HTML version since it was a manual conversion from Adobe Framemaker. But that's another story...

It should come as no surprise, then, that one of my first projects when rewriting AoA was to create a new, more powerful, version of the UCR Standard Library. This effort (the UCR Stdlib v2.0) ultimately failed (although you can still download the code written for v2.0 from <http://webster.cs.ucr.edu>). The problem was that I was trying to get MASM to do a little bit more than it was capable of and so the project was ultimately doomed.

To condense a really long story, I decided that I needed a new assembler. One that was powerful enough to let me write the new Standard Library the way I felt it should be written. However, this new assembler should also make it much easier to learn assembly language; that is, it should relieve the students of some of the drudgery of assembly language programming just as the UCR Standard Library had. After three years of part-time effort, the end result was the “High Level Assembler,” or HLA.

HLA is a radical step forward in teaching assembly language. It combines the syntax of a high level language with the low-level programming capabilities of assembly language. Together with the HLA Standard Library, it makes learning and programming assembly language almost as easy as learning and programming a High Level Language like Pascal or C++. Although HLA isn't the first attempt to create a hybrid high level/low level language, nor is it even the first attempt to create an assembly language with high level language syntax, it's certainly the first complete system (with library and operating system support) that is suitable for teaching assembly language programming. Recent experiences in my own assembly language courses show that HLA is a major improvement over MASM and other traditional assemblers when teaching machine organization and assembly language programming.

The introduction of HLA is bound to raise lots of questions about its suitability to the task of teaching assembly language programming (as well it should). Today, the primary purpose of teaching assembly language programming at the University level isn't to produce a legion of assembly language programmers; it's to teach machine organization and introduce students to machine architecture. Few instructors realistically expect more than about 5% of their students to wind up working in assembly language as their primary programming language². Doesn't turning assembly language into a high level language defeat the whole purpose of the course? Well, if HLA lets you write C/C++ or Pascal programs and attempted to call these programs “assembly language” then the answer would be “Yes, this defeats the purpose of the course.” However, despite the name and the high level (and very high level) features present in HLA, HLA is still assembly language. An HLA programmer still uses 80x86 machine instructions to accomplish most of the work. And those high level language statements that HLA provides are purely optional; the “purist” can use nothing but 80x86 assembly language, ignoring the high level statements that HLA provides. Those who argue that HLA is not *true* assembly language should note that Microsoft's MASM and Borland's TASM both provide many of the high level control structures found in HLA³.

Perhaps the largest deviation from traditional assemblers that HLA makes is in the declaration of variables and data in a program. HLA uses a very Pascal-like syntax for variable, constant, type, and procedure declarations. However, this does not diminish the fact that HLA is an assembly language. After all, at the machine language (vs. assembly language) level, there is no such thing as a data declaration. Therefore, any syntax for data declaration is an abstraction of data representation in memory. I personally chose to use a syntax that would prove more familiar to my students than the traditional data declarations used by assemblers.

Indeed, perhaps the principle driving force in HLA's design has been to leverage the student's existing knowledge when teaching them assembly language. Keep in mind, when a student first learns assembly language programming, there is so much more for them to learn than a handful of 80x86 machine instructions and the machine language programming paradigm. They've got to learn assembler directives, how to declare variables, how to write and call procedures, how to comment their code, what constitutes good programming style in an assembly language program, etc. Unfortunately, with most assemblers, these concepts are completely different in assembly language than they are in a language like Pascal or C/C++. For example, the indentation techniques students master in order to write readable code in Pascal just don't apply to (traditional) assembly language programs. That's where HLA deviates from traditional assemblers. By using a

2. My experience suggests that only about 10-20% of my students will ever write *any* assembly language again once they graduate; less than 5% ever become regular assembly language users.

3. Indeed, in some respects the MASM and TASM HLL control structures are actually *higher* level than HLA's. I *specifically* restricted the statements in HLA because I did not want students writing “C/C++ programs with MOV instructions.”

high level syntax, HLA lets students leverage their high level language knowledge to write good readable programs. HLA will not let them avoid learning machine instructions, but it doesn't force them to learn a whole new set of programming style guidelines, new ways to comment your code, new ways to create identifiers, etc. HLA lets them use the knowledge they already possess in those areas that really have little to do with assembly language programming so they can concentrate on learning the important issues in assembly language.

So let there be no question about it: HLA is an assembly language. It is not a high level language masquerading as an assembler⁴. However, it is a system that makes learning and using assembly language easier than ever before possible.

Some long-time assembly language programmers, and even many instructors, would argue that making a subject easier to learn diminishes the educational content. Students don't get as much out of a course if they don't have to work very hard at it. Certainly, students who don't apply themselves as well aren't going to learn as much from a course. I would certainly agree that if HLA's only purpose was to make it easier to learn a fixed amount of material in a course, then HLA would have the negative side-effect of reducing what the students learn in their course. However, the real purpose of HLA is to make the educational process more efficient; not so the students spend less time learning a fixed amount of material (although HLA could certainly achieve this), but to allow the students to learn the same amount of material in less time *so they can use the additional time available to them to advance their study of assembly language*. Remember what I said earlier about the UCR Standard Library- its introduction into my course allowed me to teach even more advanced topics in my course. The same is true, even more so, for HLA. Keep in mind, I've got ten weeks in a quarter. If using HLA lets me teach the same material in seven weeks that took ten weeks with MASM, I'm not going to dismiss the course after seven weeks. Instead, I'll use this additional time to cover more advanced topics in assembly language programming. That's the real benefit to using pedagogical tools like HLA.

Of course, once I've addressed the concerns of assembly language instructors and long-time assembly language programmers, the need arises to address questions a student might have about HLA. Without question, the number one concern my students have had is "If I spend all this time learning HLA, will I be able to use this knowledge once I get out of school?" A more blunt way of putting this is "Am I wasting my time learning HLA?" Let me address these questions using three points.

First, as pointed out above, most people (instructors and experienced programmers) view learning assembly language as an educational process. Most students will probably never program full-time in assembly language, indeed, few programmers write more than a tiny fraction (less than 1%) of their code in assembly language. One of the main reasons most Universities require their students to take an assembly language course is so they will be familiar with the low-level operation of their machine and so they can appreciate what the compiler is doing for them (and help them to write better HLL code once they realize how the compiler processes HLL statements). HLA is an assembly language and learning HLA will certainly teach you the concepts of machine organization, the real purpose behind most assembly language courses.

The second point to ponder is that learning assembly language consists of two main activities; learning the assembler's syntax and learning the assembly language programming paradigm (that is, learning to *think* in assembly language). Of these two, the second activity is, by far, the more difficult. HLA, since it uses a high level language-like syntax, simplifies learning the assembly language syntax. HLA also simplifies the initial process of learning to program in assembly language by providing a crutch, the HLA high level statements, that allows students to use high level language semantics when writing their first programs. However, HLA does allow students to write "pure" assembly language programs, so a good instructor will ensure that they master the full assembly language programming paradigm before they complete the course. Once a student masters the semantics (i.e., the programming paradigm) of assembly language, learning a new syntax is relatively easy. Therefore, a typical student should be able to pick up MASM in about a week after mastering HLA⁵.

As for the third and final point: to those that would argue that this is still extra effort that isn't worthwhile, I would simply point out that none of the existing assemblers have more than a cursory level of com-

4. The C-- language is a good example of a low-level non-assembly language, if you need a comparison.

5. This is very similar to mastering C after learning C++.

patibility. Yes, TASM can assemble most MASM programs, but the reverse is not true. And it's certainly not the case that NASM, A86, GAS, MASM, and TASM let you write interchangeable code. If you master the syntax of one of these assemblers and someone expects you to write code in a different assembler, you're still faced with the prospect of having to learn the syntax of the new assembler. And that's going to take you about a week (assuming the presence of well-written documentation). In this respect, HLA is no different than any of the other assemblers.

Having addressed these concerns you might have, it's now time to move on and start teaching assembly language programming using HLA.

1.2 Intended Audience

No single textbook can be all things to all people. This text is no exception. I've geared this text and the accompanying software to University level students who've never previously learned assembly language programming. This is not to say that others cannot benefit from this work; it simply means that as I've had to make choices about the presentation, I've made choices that should prove most comfortable for this audience I've chosen.

A secondary audience who could benefit from this presentation is any motivated person that really wants to learn assembly language. Although I assume a certain level of mathematical maturity from the reader (i.e., high school algebra), most of the "tough math" in this textbook is incidental to learning assembly language programming and you can easily skip over it without fear that you'll miss too much. High school students and those who haven't seen a school in 40 years have effectively used this text (and its DOS counterpart) to learn assembly language programming.

The organization of this text reflects the diverse audience for which it is intended. For example, in a standard textbook each chapter typically has its own set of questions, programming exercises, and laboratory exercises. Since the primary audience for this text is University students, such pedagogical material does appear within this text. However, recognizing that not everyone who reads this text wants to bother with this material (e.g., downloading it), this text moves such pedagogical material to the end of each volume in the text and places this material in a separate chapter. This is somewhat of an unusual organization, but I feel that University instructors can easily adapt to this organization and it saves burdening those who aren't interested in this material.

One audience to whom this book is specifically not directed are those persons who are already comfortable programming in 80x86 assembly language. Undoubtedly, there is a lot of material such programmers will find of use in this textbook. However, my experience suggests that those who've already learned x86 assembly language with an assembler like MASM, TASM, or NASM rebel at the thought of having to relearn basic assembly language syntax (as they would have to learn HLA). If you fall into this category, I humbly apologize for not writing a text more to your liking. However, my goal has always been to teach those who don't already know assembly language, not extend the education of those who do. If you happen to fall into this category and you don't particularly like this text's presentation, there is some good news: there are dozens of texts on assembly language programming that use MASM and TASM out there. So you don't really need this one.

1.3 Teaching From This Text

The first thing any instructor will notice when reviewing this text is that it's far too large for any reasonable course. That's because assembly language courses generally come in two flavors: a machine organization course (more hardware oriented) and an assembly language programming course (more software oriented). No text that is "just the right size" is suitable for both types of classes. Combining the information for both courses, plus advanced information students may need after they finish the course, produces a large text, like this one.

If you're an instructor with a limited schedule for teaching this subject, you'll have to carefully select the material you choose to present over the time span of your course. To help, I've included some brief notes

at the beginning of each Volume in this text that suggests whether a chapter in that Volume is appropriate for a machine organization course, an assembly language programming course, or an advanced assembly programming course. These brief course notes can help you choose which chapters you want to cover in your course.

If you would like to offer hard copies of this text in the bookstore for your students, I will attempt to arrange with some “Custom Textbook Publishing” houses to make this material available on an “as-requested” basis. As I work out arrangements with such outfits, I’ll post ordering information on Webster (<http://webster.cs.ucr.edu>). If your school has a printing and reprographics department, or you have a local business that handles custom publishing, you can certainly request copyright clearance to print the text locally.

If you’re not taking a formal course, just keep in mind that you don’t have to read this text straight through, chapter by chapter. If you want to learn assembly language programming and some of the machine organization chapters seem a little too hardware oriented for your tastes, feel free to skip those chapters and come back to them later on, when you understand the need to learn this information.

1.4 Copyright Notice

The full contents of this text is copyrighted material. Here are the rights I hereby grant concerning this material. You have the right to

- Read this text on-line from the <http://webster.cs.ucr.edu> web site or any other approved web site.
- Download an electronic version of this text for your own personal use and view this text on your own personal computer.
- Make a single printed copy for your own personal use.

I usually grant instructors permission to use this text in conjunction with their courses at recognized academic institutions. There are two types of reproduction I allow in this instance: electronic and printed. I grant electronic reproduction rights for one school term; after which the institution must remove the electronic copy of the text and obtain new permission to repost the electronic form (I require a new copy for each term so that corrections, changes, and additions propagate across the net). If your institution has reproduction facilities, I will grant hard copy reproduction rights for one academic year (for the same reasons as above). You may obtain copyright clearance by emailing me at

rhyde@cs.ucr.edu

I will respond with clearance via email. My returned email plus this page should provide sufficient acknowledgement of copyright clearance. If, for some reason, your reproduction department needs to have me physically sign a copyright clearance, I will have to charge \$75.00 U.S. to cover my time and effort needed to deal with this. To obtain such clearance, please email me at the address above. Presumably, your printing and reproduction department can handle producing a master copy from PDF files. If not, I can print a master copy on a laser printer (800x400dpi), please email me for the current cost of this service.

All other rights to this text are expressly reserved by the author. In particular, it is a copyright violation to

- Post this text (or some portion thereof) on some web site without prior approval.
- Reproduce this text in printed or electronic form for non-personal (e.g., commercial) use.

The software accompanying this text is all public domain material unless an explicit copyright notice appears in the software. Feel free to use the accompanying software in any way you feel fit.

1.5 How to Get a Hard Copy of This Text

This text is distributed in electronic form only. It is not available in hard copy form nor do I personally intend to have it published. If you want a hard copy of this text, the copyright allows you to print one for yourself. The PDF distribution format makes this possible (though the length of the text will make it somewhat expensive).

If you're wondering why I don't get this text published, there's a very simple reason: it's too long. Publishing houses generally don't want to get involved with texts for specialized subjects as it is; the cost of producing this text is prohibitive given its limited market. Rather than cut it down to the 500 or so 6" x 9" pages that most publishers would accept, my decision was to stick with the full text and release the text in electronic form on the Internet. The upside is that you can get a free copy of this text; the downside is that you can't readily get a hard copy.

Note that the copyright notice forbids you from copying this text for anything other than personal use (without permission, of course). If you run a "Print to Order/Custom Textbook" publishing house and would like to make copies for people, feel free to contact me and maybe we can work out a deal for those who just have to have a hard copy of this text.

1.6 Obtaining Program Source Listings and Other Materials in This Text

All of the software appearing in this text is available from the Webster web site. The URL is

<http://webster.cs.ucr.edu>

The exact filename(s) of this material may change with time, and different services use different names for these files. Check on Webster for any important changes in addresses. If for some reason, Webster disappears in the future, you should use a web-based search engine like "AltaVista" and search for "Art of Assembly" to locate the current home site of this material.

1.7 Where to Get Help

If you're reading this text and you've got questions about how to do something, please post a message to one of the following Internet newsgroups:

`comp.lang.asm.x86`
`alt.lang.asm`

Hundreds of knowledgeable individuals frequent these newsgroups and as long as you're not simply asking them to do your homework assignment for you, they'll probably be more than happy to help you with any problems that you have with assembly language programming.

I certainly welcome corrections and bug reports concerning this text at my email address. However, I regret that I do not have the time to answer general assembly language programming questions via email. I do provide support in public forums (e.g., the newsgroups above and on Webster at <http://webster.cs.ucr.edu>) so please use those avenues rather than emailing questions directly to me. Due to the volume of email I receive daily, I regret that I cannot reply to all emails that I receive; so if you're looking for a response to a question, the newsgroup is your best bet (not to mention, others might benefit from the answer as well).

1.8 Other Materials You Will Need (Windows Version)

In addition to this text and the software I provide, you will need a machine running a 32-bit version of Windows (Windows 9x, NT, 2000, ME, etc.), a copy of Microsoft's MASM and a 32-bit linker, some sort of

text editor, and other rudimentary general-purpose software tools you normally use. MASM and MS-Link are freely available on the internet. Alas, the procedure you must follow to download these files from Microsoft seems to change on a monthly basis. However, a quick post to comp.lang.asm.x86 should turn up the current site from which you may obtain this software. Almost all the software you need to use this text is part of Windows (e.g., a simple text editor like Notepad.exe) or is freely available on the net (MASM, LINK, and HLA). You shouldn't have to purchase anything.

1.9 Other Materials You Will Need (Linux Version)

In addition to this text and the software I provide, you will need a machine running Linux (preferably Linux 2.4 or later), “as” and “ld” (if you can compile GCC programs, you've got these, they come standard with most distributions), some sort of text editor, and other rudimentary general-purpose software tools you normally use. Although not necessary, it helps if you've got superuser privileges during installation so you can put the software in a reasonable spot.

Hello, World of Assembly Language

Chapter Two

2.1 Chapter Overview

This chapter is a “quick-start” chapter that lets you start writing basic assembly language programs right away. This chapter presents the basic syntax of an HLA (High Level Assembly) program, introduces you to the Intel CPU architecture, provides a handful of data declarations and machine instructions, describes some utility routines you can call in the HLA Standard Library, and then shows you how to write some simple assembly language programs. By the conclusion of this chapter, you should understand the basic syntax of an HLA program and be prepared to start learning new language features in subsequent chapters.

2.2 Installing the HLA Distribution Package

Before you can learn assembly language programming using HLA, you must first successfully install HLA on your system. Currently, HLA is available for the Linux and Windows operating systems. This section explains how to install HLA on these two systems. If HLA is already running on your system, you may skip to the next major section in this chapter.

The latest version of HLA is available from the Webster web server at

<http://webster.cs.ucr.edu>

Go to this web site and following the HLA links to the “HLA Download” page. From here you should select the latest version of HLA for download to your computer. The HLA distribution is provided in a “Zip File” compressed format. Under Windows, you will need a decompressor program like PKUNZIP or WinZip in order to extract the HLA files from this zipped archive file; under Linux, you will use the GZIP and TAR programs to decompress and extract HLA. A detailed description of the use of these decompression products is beyond the scope of this manual, please consult the software vendor’s documentation or their web page for information concerning the use of these products; this discussion will only briefly describe how to use them to extract important HLA files.

This text assumes that you will unzip the HLA distribution into the root directory of your C: drive under Windows, or to the “/usr/hla” directory under Linux. You can certainly install HLA anywhere you want, but you will have to adjust the following descriptions if you install HLA somewhere else. If possible, you should install HLA using root/administrator privileges; regardless, you should make sure the permissions are set properly on the files so everyone has read and execute access to the HLA files; if you are unsure how to do this, please consult your operating system’s documentation or consult a system administrator.

HLA is a console application. In order to run the HLA compiler you must run the command window program (this is “command.com” on Windows 95 and 98, or “cmd.exe” on Windows NT and Windows 2000; Linux users typically run “bash” or some other shell). This also means that you should be familiar with some simple “command line interface” (CLI) or “shell” commands.

Most Windows distributions let you run the command prompt windows from the Start menu or from a submenu hanging off the start menu (you may also select “RUN” from the Start menu and type “cmd” as the program name). This text assumes that you are familiar with the Windows command window and you know how to use some basic command window commands (e.g., dir, del, rename, etc.). If you have never before used the Windows command line interpreter, you should consult an appropriate text to learn a few basic commands.

Most Linux distributions run “bash” or some other shell program whenever you open up a terminal window (e.g., a GNOME or KDE terminal window or an X-TERM window). There are some minor differences between the shells running under Linux, this document assumes that you are using GNU’s “bash” shell. Again, this text assumes that you are comfortable with a few commands like ls, rm, and mv. If you have never used a Unix shell program before, you should consult an appropriate text or the on-line documentation to learn a few basic commands.

Before you can actually run the HLA compiler, you must set the system execution path and set up various environment variables. The following subsections explain how to do this under Windows and then Linux.

2.2.1 Installation Under Windows

HLA is not a stand alone program. It is a compiler that translates HLA source code into a lower-level assembly language. A separate assembler, such as MASM, then completes the processing of this low-level intermediate code to produce an object code file. Finally, you must link the object code output from the assembler using a linker program. Typically you will link the object code produced by one or more HLA source files with the HLA Standard Library (hlalib.lib) and, possibly, several operating system specific library files (e.g., kernel32.lib under Windows). Most of this activity takes place transparently whenever you ask HLA to compile your HLA source file(s). However, for the whole process to run smoothly, you must have installed HLA and all the support files correctly. This section will discuss how to set up HLA on your Windows system.

First, you will need an HLA distribution for Windows. The latest version of HLA is always available on Webster at <http://webster.cs.ucr.edu>. You should go there and download the latest version if you do not already possess it.

As noted earlier, HLA is not a stand alone assembler. The HLA package contains the HLA compiler, the HLA Standard Library, and a set of include files for the HLA Standard Library. If you write an HLA program with just this code, HLA will produce an "ASM" file and then stop. To produce an executable file you will need Microsoft's MASM and LINK programs, along with some Windows library files, to complete the process. The easiest way to get all the files you need is to download the "MASM32" package from <http://www.pdq.com.au/home/hutch/masm.htm> or any of the other places on the net where you can find the MASM32 package (Webster maintains a current link if this link is dead). Once you unzip this file, it's easy to install the MASM32 package using the install program it supplies. ***You must install MASM32 (or MASM/LINK/Win32 library files) before HLA will function properly.***

Here are the steps I went through to install MASM32 on my system:

- I downloaded `masm32v6.zip` from the URL above (later versions are probably okay too, although there is a slight chance that the installation will be different).
- I double-clicked on the `masm32v6.zip` file (which runs WinZip on my system).
- I choose to extract "install.exe". I told WinZip to extract this file to C:\.
- I double-clicked on the "install.exe" icon and selected the "C:" drive in the window that popped up. Then I hit the install button and waited while MASM32 extracted all the pertinent files. This produced a directory called "MASM32". MASM32 is a powerful assembly language development subsystem in its own right; but it uses the traditional MASM syntax rather than the HLA syntax. So we'll use MASM32 mainly for the assembler, linker, and library files. MASM32 also includes a simple editor/IDE and several other tools that may be useful to an HLA programmer. Feel free to check this software out and see if it is useful to you. For now, note that the executable files you will ultimately need are `ML.EXE`, `ML.ERR`, `LINK.EXE`, and a couple of DLLs. You can find them in the `MASM32\BIN` subdirectory. Leave them there for the time being. The `MASM32\LIB` directory also contains many Win32 library files you will need. Again, leave them alone for the time being.
- Next, if you haven't already done so, download the HLA executables file from Webster at <http://webster.cs.ucr.edu>. On Webster you can download several different ZIP files associated with HLA from the HLA download page. The "Executables" is the only one you'll absolutely need; however, you'll probably want to grab the documentation and examples files as well. If you're curious, or you want some more example code, you can download the source listings to the HLA Standard Library. If you're *really* curious (or masochistic), you can download the HLA compiler source listings to (this is *not* for casual browsing!).
- I downloaded the `HLA1_32.zip` file while writing this. Most likely, there is a much later version available as you're reading this. Be sure to get the latest version. I chose to download this file to my "C:\" root directory.

- After downloading HLA1_32.zip to my C: drive, I double-clicked on the icon to run WinZip. I selected "Extract" and told WinZip to extract all the files to my C:\ directory. This created an "HLA" subdirectory in my root on C: with two subdirectories (include and lib) and two EXE files (HLA.EXE and HLAPARSE.EXE. The HLA program is a "shell" program that runs the HLA compiler (HLAPARSE.EXE), MASM (ML.EXE), the linker (LINK.EXE), and other programs. You can think of HLA.EXE as the "HLA Compiler".
- Next, I created the following text file and named it "IHLA.BAT" (note that you may need to change the default drive letters if you want to install HLA on a drive other than "C:");

```
path=c:\hla;c:\masm32\bin;%path%
set lib=c:\masm32\lib;c:\hla\hlalib;%lib%
set include=c:\hla\include;c:\masm32\include;%include%
set hlainc=c:\hla\include
set hlalib=c:\hla\hlalib\hlalib.lib
```

- Be sure you've typed all the lines exactly as written or HLA will fail to run properly. You may use any reasonable TEXT editor (e.g., NOTEPAD.EXE) to create this file. Do not use a word processing program (since they generally don't save their data as a TEXT file). Be sure the file is named "IHLA.BAT" and not "IHLA.BAT.TXT" or some other variation.
- This batch file tells the system where to find all the files you will need when running HLA. Advanced Win32 users should note that you can set all these environment variables up inside the Windows system control panel in the "Advanced->Environment Variables" area. This is far more convenient (ultimately) than using this batch file (for reasons you'll soon see). However, you can mess up your system if you don't know what you're doing when playing with the system control panel, so only advanced users who've done this stuff before should attempt this.
- HLA is a Win32 Console Window program. To run HLA you must open up a console Window. Under Windows 2000, Microsoft has hidden this away in Start->Programs->Accessories->Command Prompt. You might find it in another location. You can also start the command prompt processor by selecting Start->Run and entering "cmd".
- Once you've got the command prompt, ("C:>" or something similar), execute the IHLA.BAT file you've created by typing "IHLA" at the command line prompt. Hit the ENTER key to execute the command.
- At this point, HLA should be properly installed and ready to run. Try typing "hla -?" at the command line prompt and verify that you get the HLA help message. If not, go back and figure out what you've done wrong up to this point (it doesn't hurt to start over from the beginning if you're lost).
- Thus far, you've verified that HLA.EXE is operational. Now try the following command: "ML /?" This should run the Microsoft Macro Assembler (MASM) and display the help screen. You can ignore the information that appears; you will probably never need to know this stuff.
- Next, let's verify the correct operation of the linker. Type "link /?" and verify that the linker program runs. Again, you can ignore the help screen that appears. You don't need to know about this stuff.
- Now it's time to try your hand at writing an honest to goodness HLA program and verify that the whole system is working. Here's the canonical "Hello World" program written in HLA (we will revisit this program a little later in this chapter, don't worry about what it means just yet). Enter it into a text editor and save it using the filename "HW.HLA":

```
program HelloWorld;
#include( "stdlib.hhf" )
begin HelloWorld;

    stdout.put( "Hello, World of Assembly Language", nl );

end HelloWorld;
```

- Make sure you're in the same directory containing the HW.HLA file and type the following command at the "C:>" prompt: "HLA -v HW". The "-v" option tells HLA to produce VERBOSE output during compilation. This is helpful for determining what went wrong if the system fails somewhere along the line. This command should produce the following output:

```
HLA (High Level Assembler)
Written by Randall Hyde and released to the public domain.
Version Version 1.32 build 4904 (prototype)

Files:
1: hw.hla

Compiling "hw.hla" to "hw.asm"

Assembling hw.asm via "ml /c /coff /Cp hw.asm"

Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997. All rights reserved.

Assembling: hw.asm
Linking via "link -subsystem:console /heap:0x1000000,0x1000000
/stack:0x1000000,0x1000000 /BASE:0x3000000 /machine:IX86 -entry:?HLAMain @hw.link
-out:hw.exe kernel32.lib user32.lib c:\hla\hlalib\hlalib.lib hw.obj"
Microsoft (R) Incremental Linker Version 5.12.8078
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

/section:.text,ER
/section:readonly,R
/section:.edata,R
/section:.data,RW
/section:.bss,RW
```

- If you get all of this output, you're in business. You can run the "HW" program using the following CLI (command line interpreter) command:

```
HW
```

- One thing to remember is that unless you set the environment variables permanently in the System control panel, you will have to run the IHLA.BAT file every time you open up a new command prompt window. Since this is a pain, here are some instructions I've taken from the Internet that describe how to set up the environment variables (*DO THIS AT YOUR OWN RISK!*)
- 1) Open System Properties (Winkey-Break is a convenient shortcut) and go to Advanced tab, then Environment Variables. Add "c:\hla" to the Path in SYSTEM VARIABLES, not in "User variables for <your win2k login name>". Click OK, but keep the Environment Variables window open, we're not done.
 - 2) Look at the contents of ihla.bat (ABOVE):
 - 3) In "User Variables for <your login name>", you must end up with each of these settings. For example, to create hlainc, you click the "New..." button, type "hlainc" as the name of the variable, and type "c:\hla\include" as the Variable value (all without quotes of course). If there is already a path set, and it already has some value, add this immediately to the end: ";c:\hla;%path%" and that will preserve your existing User and System paths as well as adding c:\hla.

For example, suppose you opened up your User Variables for <login name> and it already said "C:\Private

Files\PantiePix;c:\winnt\system32;c:\winnt;c:\winnt\System32\Wbem;d:\lcc\bin;D:\PROGRA~1\ULTRAE~1;D:\4NT300;C:\msoffice\Office;c:/hla",

you would click on Edit and type "C:\Private Files\PantiePix;c:\hla;%path%"

(Same advice for preserving existing lib and include settings)

- 4) Once you reboot the computer, you should be all set for "Hello world of assembly language"! (without having to run the IHLA.BAT file.)

Installing HLA is a complex and slightly involved process. Unfortunately, this is necessary because I don't have the rights to distribute MASM, LINK, and other Microsoft files. Fortunately, HUTCH has collected all of these files together so they are easy to download. If you are concerned about possible legal issues with the download, you may legally download MASM and LINK from Microsoft's site. A link on Webster (at the URL above) describes how to do this. At the time this was being written, work was progressing on HLA to produce TASM compatible output and plans were in the works to produce NASM and Gas versions as well. However, you will still have to obtain the Microsoft library files from some source if you intend to produce a Win32 application. Versions of HLA may appear for other Operating Systems as well. Check out Webster to see if any progress has been made in this direction.

The most common two problems people have running HLA involve the location of the Win32 library files and the choice of linker. During the linking phase, HLA (well, link.exe actually) requires the kernel32.lib, user32.lib, and gdi32.lib library files. These must be present in the pathname(s) specified by the LIB environment variable. If, during the linker phase, HLA complains about missing object modules, make sure that the LIB path specifies the directory containing these files. If you're a MS VC++ user, installation of VC++ should have set up the LIB path for you. If not, then locate these files (they are part of the MASM32 distribution) and copy them to the HLA\HLALIB directory (note that the ihla.bat file includes c:\hla\hlalib as part of the LIB path).

Another common problem with running HLA is the use of the wrong link.exe program. Microsoft has distributed several different versions of link.exe; in particular, there are 16-bit linkers and 32-bit linkers. You must use a 32-bit segmented linker with HLA. If you get complaints about "stack size exceeded" or other errors during the linker phase, this is a good indication that you're using a 16-bit version of the linker. Obtain and use a 32-bit version and things will work. Don't forget that the 32-bit linker must appear in the execution path (specified by the PATH environment variable) before the 16-bit linker.

2.2.2 Installation Under Linux

HLA is not a stand alone program. It is a compiler that translates HLA source code into a lower-level assembly language. A separate assembler, such as Gas (as), then completes the processing of this low-level intermediate code to produce an object code file. Finally, you must link the object code output from the assembler using a linker program. Typically you will link the object code produced by one or more HLA source files with the HLA Standard Library (hlalib.a). Most of this activity takes place transparently whenever you ask HLA to compile your HLA source file(s). However, for the whole process to run smoothly, you must have installed HLA and all the support files correctly. This section will discuss how to set up HLA on your system.

First, you will need an HLA distribution for Linux. The latest version of HLA is always available on Webster at <http://webster.cs.ucr.edu>. You should go there and download the latest version if you do not already possess it.

As noted earlier, HLA is not a stand alone assembler. The HLA package contains the HLA compiler, the HLA Standard Library, and a set of include files for the HLA Standard Library. If you write an HLA program with just this code, HLA will produce an "ASM" file and then stop. To produce an executable file

you will need GNU's `as` and `ld` programs (these come with any Linux distribution that supports compiling C/C++ programs). Note that HLA only works with `Gas` v2.10 or later. The `Gas` assembler is part of the `Binutils` package. If you don't have version 2.10 or later, download an appropriate `binutils` package from the internet. HLA will generate errors when it attempts to assemble its output via an invocation of the `as` (`Gas`) executable if you don't have `Gas` v2.10 or later installed in your system.

Here are the steps I went through to install HLA on my Linux system:

- First, if you haven't already done so, download the HLA executables file from Webster at <http://webster.cs.ucr.edu>. On Webster you can download several different ZIP files associated with HLA from the HLA download page. The "Linux Executables" is the only one you'll absolutely need; however, you'll probably want to grab the documentation and examples files as well. If you're curious, or you want some more example code, you can download the source listings to the HLA Standard Library. If you're *really* curious (or masochistic), you can download the HLA compiler source listings to (this is *not* for casual browsing!).
- I downloaded the `HLA1_39.tar.gz` file while writing this. Most likely, there is a much later version available as you're reading this. Be sure to get the latest version. I chose to download this file to my root directory; you can put the file wherever you like, though this documentation assumes that all HLA files wind up in the `/usr/hla/...` directory tree. If you do not already have a `/usr/hla` subdirectory, you can create one with the `"mkdir"` command (it's best to do this using the `"root"` or `"superuser"` account; if you do not have superuser privileges, you should have your system administrator do this for you).
- After downloading `HLA1_39.tar.gz` to my root directory, I executed the following shell command: `"gzip -d HLA1_39.tar.gz"`. Once decompression was complete, I extracted the individual files using the command `"tar xvf HLA1_39.tar"`. This extracted a couple of executable files (`"hla"` and `"hlaparse"`) along with two subdirectories (`include` and `hlalib`). The HLA program is a "shell" program that runs the HLA compiler (`hlaparse`), `Gas` (`as`), the linker (`ld`), and other programs. You can think of `"hla"` as the "HLA Compiler". It would be a real good idea, at this point, to set the permissions on `"hla"` and `"hlaparse"` so that everyone can read and execute them. You should also set read and execute permissions on the two subdirectories and read permissions on all the files within the directories (if this isn't the default state). Do a `"man chmod"` from the Linux command-line if you don't know how to change permissions.
- Next, (logged in as a plain user rather than root or the super-user), I edited the `".bashrc"` file in my home directory (`/home/rhyde` in my particular case, this will probably be different for you). I found the line that defined the `"path"` variable, it originally looked like this on my system

```
"PATH=$DBROOT/bin:$DBROOT/pgm:$PATH"
```

I edited this line to add the path to the HLA directory, producing the following:

```
"PATH=$DBROOT/bin:$DBROOT/pgm:/usr/hla:$PATH"
```

Without this modification, Linux will probably not find HLA when you attempt to execute it unless you type a full path (e.g., `/usr/hla/hla`) when running the program. Since this is a pain, you'll definitely want to add `/usr/hla` to your path.

- Next, I added the following four lines to `".bashrc"` (note that Linux filenames beginning with a period don't normally show up in directory listings unless you supply the `"-a"` option to `ls`):


```
hlalib=/usr/hla/hlalib/hlalib.a
export hlalib
hlainc=/usr/hla/include
export hlainc
```

These four lines define (and export) environment variables that HLA needs during compilation. Without these environment variables, HLA will probably complain about not being able to find include files, or the linker (`ld`) will complain about strange undefined symbols when you attempt to compile your programs.

After saving the `".bashrc"` shell, you can tell Linux to make the changes to the system by using the command:

```
source .bashrc
```

Note: this discussion only applies to users who run the BASH shell. If you are using a different shell (like the C-Shell or the Korn Shell), then the directions for setting the path and environment variables differs slightly. Please see the documentation for your particular shell if you don't know how to do this. Also note that Linux does not normally display files whose name begins with a period when you use the "ls" command; to see such files, use the "ls -a" shell command.

- At this point, HLA should be properly installed and ready to run. Try typing "hla -?" at the command line prompt and verify that you get the HLA help message. If not, go back and figure out what you've done wrong up to this point (it doesn't hurt to start over from the beginning if you're lost).
- Now it's time to try your hand at writing an honest to goodness HLA program and verify that the whole system is working. Here's the canonical "Hello World" program written in HLA (we'll discuss this program in detail a little later in this chapter). Enter it into a text editor and save it using the filename "hw.hla":

```
program HelloWorld;
#include( "stdlib.hhf" )
begin HelloWorld;

    stdout.put( "Hello, World of Assembly Language", nl );

end HelloWorld;
```

- Make sure you're in the same directory containing the "hw.hla" file and type the following command at the prompt: "hla -v hw". The "-v" option tells HLA to produce VERBOSE output during compilation. This is helpful for determining what went wrong if the system fails somewhere along the line. This command should produce the following output:

```
HLA (High Level Assembler) Parser
Written by Randall Hyde and released to the public domain.
Version Version 1.39 build 6845 (prototype)
-t active
File: t.hla

Compiling "t.hla" to "t.asm"
HLA (High Level Assembler)
Copyright 1999, by Randall Hyde, all rights reserved.
Version Version 1.39 build 6845 (prototype)
ELF output
Using GAS assembler
GAS output
-test active

Files:
1: t.hla

Compiling 't.hla' to 't.asm'
using command line [hlaparse -v -sg -test "t.hla"]

Assembling "t.asm" via [as -o t.o "t.asm"]
Linking via [ld -o "t" "t.o" "/usr/hla/hlalib/hlalib.a"]
```

Installing HLA is a complex and slightly involved process; though take heart, it's a lot simpler to install HLA under Linux than Windows! (See the previous section if you need proof.) Versions of HLA may appear for other operating systems (beyond Windows and Linux) as well. Check out Webster to see if any progress has been made in this direction. Note a very unique thing about HLA: Carefully written (console) applications will compile and run on all supported operating systems without change. This is unheard of for

assembly language! So if you are using multiple operating systems supported by HLA, you'll probably want to download files for all supported OSes.

Note: to run the HelloWorld program, a Linux user would type "hw" (or possibly "./hw") at the command line prompt.

2.2.3 Installing "Art of Assembly" Related Files

Although HLA is relatively flexible about where you put it on your system, this text assumes you've installed HLA in the "hla" directory on your C: drive under a Win32 operating system or in "/usr/hla" under Linux. This text also assumes the standard directory placement for the HLA files, which has the following layout

- HLA directory
- AoA directory
- Doc directory
- Examples directory
- hllib directory
- hllibsrc directory
- include directory
- Tests directory

The "Art of Assembly" (AoA) software distribution has the following directory tree structure:

- AoA directory
- volume1
- ch01 directory
- ch02 directory
- etc.
- volume2
- ch01 directory
- ch02 directory
- etc.
- etc.

The main *HLA* directory contains the executable code for the compiler. This consists of two files, HLA.EXE/hla and HLAPARSE.EXE/hlaparse (Windows/Linux). These two programs must be in the current execution path in order to run the compiler. Under Windows, it wouldn't hurt to put the ml.exe, ml.err, link.exe, mspdbX0.dll (x=5, 6, or greater), and msvcrt.dll files in this directory as well. Under Linux, the "as" and "ld" programs are already in the execution path, assuming your Linux system supports C/C++ development.

The *Doc* directory contains reference material for HLA in PDF and HTML formats. If you have a copy of Adobe Acrobat Reader, you will probably want to read the PDF versions since they are much nicer than the HTML versions. These documents contain the most up-to-date information about the HLA language; you should consult them if you have a question about the HLA language or the HLA Standard Library. Generally, material in this documentation supersedes information appearing in this text since the HLA document is electronic and is probably more up to date.

The *Examples* directory contains a large set of HLA programs that demonstrate various features in the HLA language. If you have a question about an HLA feature, you can probably find an example program that demonstrates that feature in the *Examples* directory. Such examples provide invaluable insight that is often superior to a written description of the feature. Note that some of these programs may be specific to Windows or Linux, not all will compile and run under either operating system.

The *hlalib* directory contains the object code for the HLA Standard Library. As you become more competent with HLA, you may want to take a look at how HLA implements various library functions by checking out the library source code in the *hlalibsrc* subdirectory.

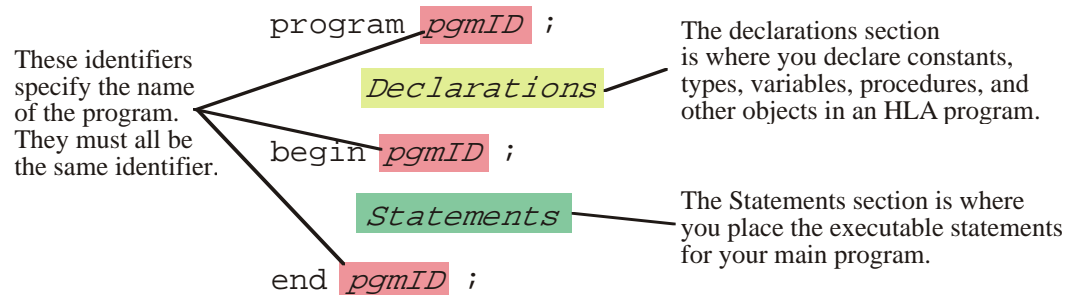
The *include* directory contains the HLA Standard Library include files. These special files (that end with a “.hhf” suffix, for “HLA Header File”) are needed during assembly to provide prototype and other information to your program. The example programs in this chapter all include the HLA header file “stdlib.hhf” that, in turn, includes all the other HLA header files in the standard library.

The *Tests* directory contains various test files that test the correct operation of the HLA system. HLA includes these files as part of the distribution package because they provide additional examples of HLA coding.

The *AoA* directory contains the code specific to this textbook. This directory contains all the source code to the (complete) programs appearing in this text. It also contains the programs appearing in the Laboratory Exercises section of each chapter. Therefore, this directory is very important to you. Within this subdirectory, the information is further divided up by volume and chapter. The material for Chapter One appears in the “ch01” subdirectory of the “volume1” directory in the AoA directory tree, the material for Chapter Two appears in the “ch02” subdirectory of the “volume1” directory, etc..

2.3 The Anatomy of an HLA Program

An HLA program typically takes the following form:



PROGRAM, BEGIN, and END are HLA reserved words that delineate the program. Note the placement of the semicolons in this program.

Figure 2.1 Basic HLA Program Layout

The *pgmID* in the template above is a user-defined program identifier. You must pick an appropriate, descriptive, name for your program. In particular, *pgmID* would be a horrible choice for any real program. If you are writing programs as part of a course assignment, your instructor will probably give you the name to use for your main program. If you are writing your own HLA program, you will have to choose this name.

Identifiers in HLA are very similar to identifiers in most high level languages. HLA identifiers may begin with an underscore or an alphabetic character, and may be followed by zero or more alphanumeric or underscore characters. HLA’s identifiers are *case neutral*. This means that the identifiers are case sensitive insofar as you must always spell an identifier exactly the same way in your program (even with respect to upper and lower case). However, unlike other case sensitive languages, like C/C++, you may not declare two identifiers in the program whose name differs only by the case of alphabetic characters appearing in an identifier. Case neutrality enforces the good programming style of always spelling your names exactly the same

way (with respect to case) and never declaring two identifiers whose only difference is the case of certain alphabetic characters.

A traditional first program people write, popularized by K&R's "The C Programming Language" is the "Hello World" program. This program makes an excellent concrete example for someone who is learning a new language. Here's what the "Hello World" program looks like in HLA:

```

program helloWorld;
#include( "stdlib.hhf" );

begin helloWorld;

    stdout.put( "Hello, World of Assembly Language", nl );

end helloWorld;

```

Program 2.1 The Hello World Program

The `#include` statement in this program tells the HLA compiler to include a set of declarations from the `stdlib.hhf` (standard library, HLA Header File). Among other things, this file contains the declaration of the `stdout.put` code that this program uses.

The `stdout.put` statement is the "print" statement for the HLA language. You use it to write data to the standard output device (generally the console). To anyone familiar with I/O statements in a high level language, it should be obvious that this statement prints the phrase "Hello, World of Assembly Language". The `nl` appearing at the end of this statement is a constant, also defined in "stdlib.hhf", that corresponds to the newline sequence.

Note that semicolons follow the program, `BEGIN`, `stdout.put`, and `END` statements¹. Technically speaking, a semicolon is not necessary after the `#INCLUDE` statement. It is possible to create include files that generate an error if a semicolon follows the `#INCLUDE` statement, so you may want to get in the habit of not putting a semicolon here (note, however, that the HLA standard library include files always allow a semicolon after the corresponding `#INCLUDE` statement).

The `#INCLUDE` is your first introduction to HLA declarations. The `#INCLUDE` itself isn't actually a declaration, but it does tell the HLA compiler to substitute the file "stdlib.hhf" in place of the `#INCLUDE` directive, thus inserting several declarations at this point in your program. Most HLA programs you will write will need to include at least some of the HLA Standard Library header files ("stdlib.hhf" actually includes all the standard library definitions into your program; for more efficient compiles, you might want to be more selective about which files you include. You will see how to do this in a later chapter).

Compiling this program produces a *console* application. Running this program in a command window prints the specified string and then control returns back to the command line interpreter (or *shell* in Unix terminology).

Note that HLA is a free-format language. Therefore, you may split statement across multiple lines (just like high level languages) if this helps to make your programs more readable. For example, the `stdout.put` statement in the HelloWorld program could also be written as follows:

```

stdout.put
(
    "Hello, World of Assembly Language",
    nl
);

```

1. Technically, from a language design point of view, these are not all statements. However, this chapter will not make that distinction.

Another item worth noting, since you'll see it cropping up in example code throughout this text, is that HLA automatically concatenates any adjacent string constants it finds in your source file. Therefore, the statement above is also equivalent to:

```
stdout.put
(
    "Hello, "
    "World of Assembly Language",
    nl
);
```

Indeed, "nl" (the newline) is really nothing more than a string constant, so (technically) the comma between the *nl* and the preceding string isn't necessary. You'll often see the above written as:

```
stdout.put( "Hello, World of Assembly Language" nl );
```

Notice the lack of a comma between the string constant and *nl*; this turns out to be perfectly legal in HLA, though it only applies to certain symbol string constants; you may not, in general, drop the comma. The chapter on Strings, later in this text, will explain in detail how this works. This discussion appears here because you'll probably see this "trick" employed by sample code prior to the formal discussion in the chapter on Strings.

2.4 Some Basic HLA Data Declarations

HLA provides a wide variety of constant, type, and data declaration statements. Later chapters will cover the declaration section in more detail but it's important to know how to declare a few simple variables in an HLA program.

HLA predefines three different signed integer types: *int8*, *int16*, and *int32*, corresponding to eight-bit (one byte) signed integers, 16-bit (two byte) signed integers, and 32-bit (four byte) signed integers respectively². Typical variable declarations occur in the HLA *static variable section*. A typical set of variable declarations takes the following form

```
static
i8, i16, and i32
are the names of
the variables to
declare here.
i8: int8;
i16: int16;
i32: int32;
```

"static" is the keyword that begins the variable declaration section.

int8, int16, and int32 are the names of the data types for each declaration

Figure 2.2 Static Variable Declarations

Those who are familiar with the Pascal language should be comfortable with this declaration syntax. This example demonstrates how to declare three separate integers, *i8*, *i16*, and *i32*. Of course, in a real program you should use variable names that are a little more descriptive. While names like "i8" and "i32" describe the type of the object, they do not describe its purpose. Variable names should describe the purpose of the object.

In the *STATIC* declaration section, you can also give a variable an initial value that the operating system will assign to the variable when it loads the program into memory. The following figure demonstrates the syntax for this:

2. A discussion of bits and bytes will appear in the next chapter if you are unfamiliar with these terms.

```

static
    i8:  int8  := 8;
    i16: int16 := 1600;
    i32: int32 := -320000;

```

The constant assignment operator, ":= " tells HLA that you wish to initialize the specified variable with an initial value.

The operand after the constant assignment operator must be a constant whose type is compatible with the variable you are initializing

Figure 2.3 Static Variable Initialization

It is important to realize that the expression following the assignment operator (":=") must be a constant expression. You cannot assign the values of other variables within a STATIC variable declaration.

Those familiar with other high level languages (especially Pascal) should note that you may only declare one variable per statement. That is, HLA does not allow a comma delimited list of variable names followed by a colon and a type identifier. Each variable declaration consists of a single identifier, a colon, a type ID, and a semicolon.

Here is a simple HLA program that demonstrates the use of variables within an HLA program:

```

Program DemoVars;
#include( "stdlib.hhf" );

static
    InitDemo:      int32 := 5;
    NotInitialized: int32;

begin DemoVars;

    // Display the value of the pre-initialized variable:

    stdout.put( "InitDemo's value is ", InitDemo, nl );

    // Input an integer value from the user and display that value:

    stdout.put( "Enter an integer value: " );
    stdin.get( NotInitialized );
    stdout.put( "You entered: ", NotInitialized, nl );

end DemoVars;

```

Program 2.2 Variable Declaration and Use

In addition to STATIC variable declarations, this example introduces three new concepts. First, the *stdout.put* statement allows multiple parameters. If you specify an integer value, *stdout.put* will convert that value to the string representation of that integer's value on output. The second new feature this sample program introduces is the *stdin.get* statement. This statement reads a value from the standard input device (usually the keyboard), converts the value to an integer, and stores the integer value into the *NotInitialized* variable. Finally, this program also introduces the syntax for (one form of) HLA comments. The HLA compiler ignores all text from the "/" sequence to the end of the current line. Those familiar with C++ and Delphi should recognize these comments.

2.5 Boolean Values

HLA and the HLA Standard Library provides limited support for boolean objects. You can declare boolean variables, use boolean literal constants, use boolean variables in boolean expressions (e.g., in an IF statement), and you can print the values of boolean variables.

Boolean literal constants consist of the two predefined identifiers *true* and *false*. Internally, HLA represents the value true using the numeric value one; HLA represents false using the value zero. Most programs treat zero as false and anything else as true, so HLA's representations for *true* and *false* should prove sufficient.

To declare a boolean variable, you use the *boolean* data type. HLA uses a single byte (the least amount of memory it can allocate) to represent boolean values. The following example demonstrates some typical declarations:

```
static
  BoolVar:  boolean;
  HasClass: boolean := false;
  IsClear:  boolean := true;
```

As you can see in this example, you may declare initialized as well as uninitialized variables.

Since boolean variables are byte objects, you can manipulate them using eight-bit registers and any instructions that operate directly on eight-bit values. Furthermore, as long as you ensure that your boolean variables only contain zero and one (for false and true, respectively), you can use the 80x86 AND, OR, XOR, and NOT instructions to manipulate these boolean values (we'll describe these instructions a little later).

You can print boolean values by making a call to the *stdout.put* routine, e.g.,

```
stdout.put( BoolVar )
```

This routine prints the text "true" or "false" depending upon the value of the boolean parameter (zero is false, anything else is true). Note that the HLA Standard Library does not allow you to read boolean values via *stdin.get*.

2.6 Character Values

HLA lets you declare one-byte ASCII character objects using the *char* data type. You may initialize character variables with a literal character value by surrounding the character with a pair of apostrophes. The following example demonstrates how to declare and initialize character variables in HLA:

```
static
  c: char;
  LetterA: char := 'A';
```

You can print character variables using the *stdout.put* routine. We'll return to the subject of character constants a little later.

2.7 An Introduction to the Intel 80x86 CPU Family

Thus far, you've seen a couple of HLA programs that will actually compile and run. However, all the statements utilized to this point have been either data declarations or calls to HLA Standard Library routines. There hasn't been any *real* assembly language up to this point. Before we can progress any farther and learn

some real assembly language, a detour is necessary. For unless you understand the basic structure of the Intel 80x86 CPU family, the machine instructions will seem mysterious indeed.

The Intel CPU family is generally classified as a *Von Neumann Architecture Machine*. Von Neumann computer systems contain three main building blocks: the *central processing unit* (CPU), *memory*, and *input/output devices* (I/O). These three components are connected together using the *system bus*. The following block diagram shows this relationship:

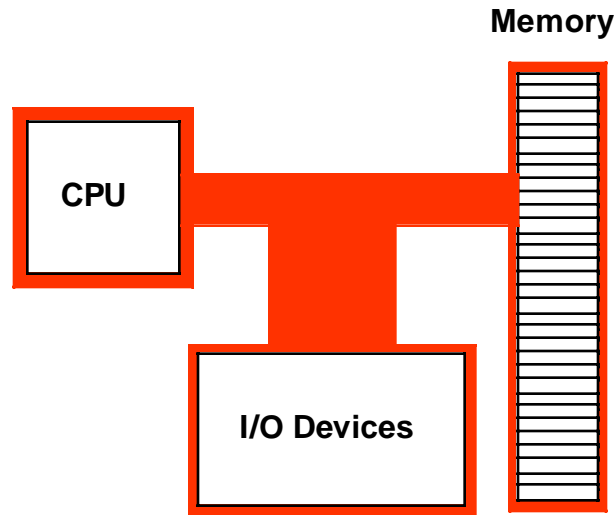


Figure 2.4 Von Neumann Computer System Block Diagram

Memory and I/O devices will be the subjects of later chapters; for now, let's take a look inside the CPU portion of the computer system, at least at the components that are visible to the assembly language programmer.

The most prominent items within the CPU are the registers. The Intel CPU registers can be broken down into four categories: general purpose registers, special purpose application accessible registers, segment registers, and special purpose kernel mode registers. This text will not consider the last two sets of registers. The segment registers are not used much in modern 32-bit operating systems (e.g., Windows, BeOS, and Linux); since this text is geared around programs written for 32-bit operating systems, there is little need to discuss the segment registers. The special purpose kernel mode registers are intended for use by people who write operating systems, debuggers, and other system level tools. Such software construction is well beyond the scope of this text, so once again there is little need to discuss the special purpose kernel mode registers.

The 80x86 (Intel family) CPUs provide several general purpose registers for application use. These include eight 32-bit registers that have the following names:

EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP

The "E" prefix on each name stands for *extended*. This prefix differentiates the 32-bit registers from the eight 16-bit registers that have the following names:

AX, BX, CX, DX, SI, DI, BP, and SP

Finally, the 80x86 CPUs provide eight 8-bit registers that have the following names:

AL, AH, BL, BH, CL, CH, DL, and DH

Unfortunately, these are not all separate registers. That is, the 80x86 does not provide 24 independent registers. Instead, the 80x86 overlays the 32-bit registers with the 16-bit registers and it overlays the 16-bit registers with the 8-bit registers. The following diagram shows this relationship:

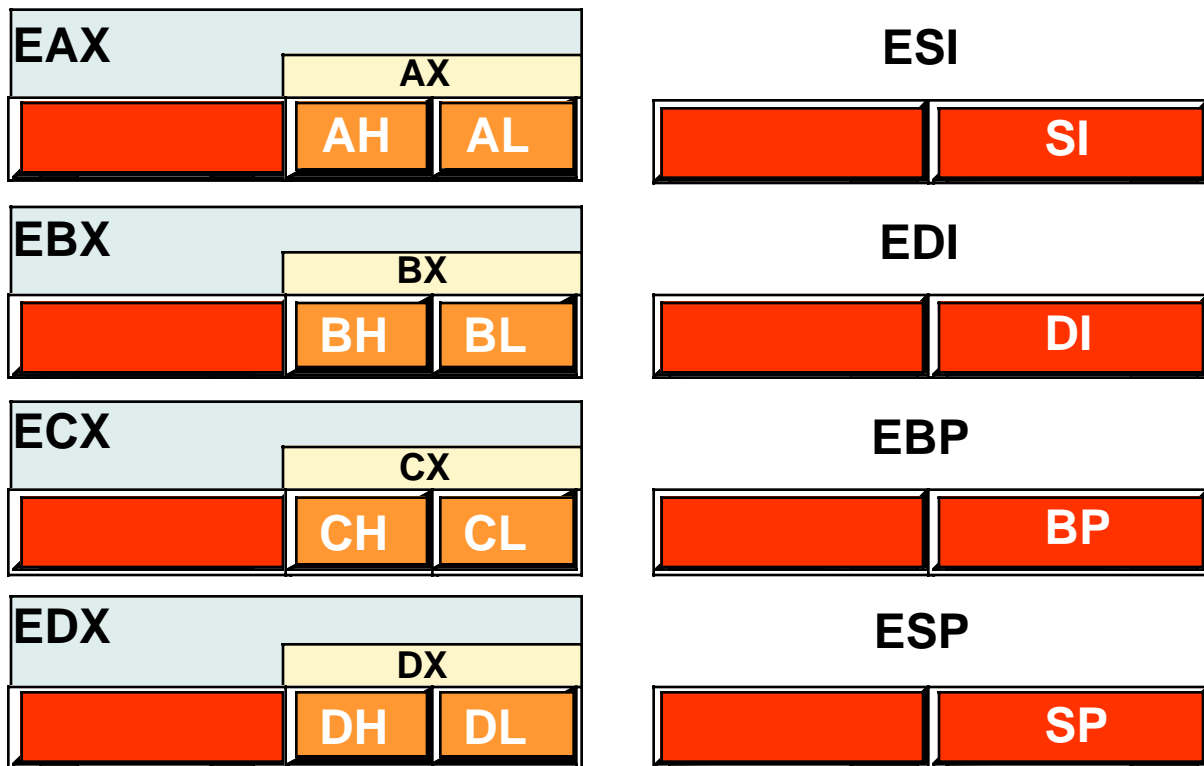


Figure 2.5 80x86 (Intel CPU) General Purpose Registers

The most important thing to note about the general purpose registers is that they are not independent. Modifying one register will modify at least one other register and may modify as many as three other registers. For example, modification of the EAX register may very well modify the AL, AH, and AX registers as well. This fact cannot be overemphasized here. A very common mistake in programs written by beginning assembly language programmers is register value corruption because the programmer did not fully understand the ramifications of the above diagram.

The EFLAGS register is a 32-bit register that encapsulates several single-bit boolean (true/false) values. Most of the bits in the EFLAGS register are either reserved for kernel mode (operating system) functions, or are of little interest to the application programmer. Eight of these bits (or *flags*) are of interest to application programmers writing assembly language programs. These are the overflow, direction, interrupt disable³, sign, zero, auxiliary carry, parity, and carry flags. The following diagram shows their layout within the lower 16-bits of the EFLAGS register.

3. Application programs cannot modify the interrupt flag, but we'll look at this flag later in this text, hence the discussion of this flag here.

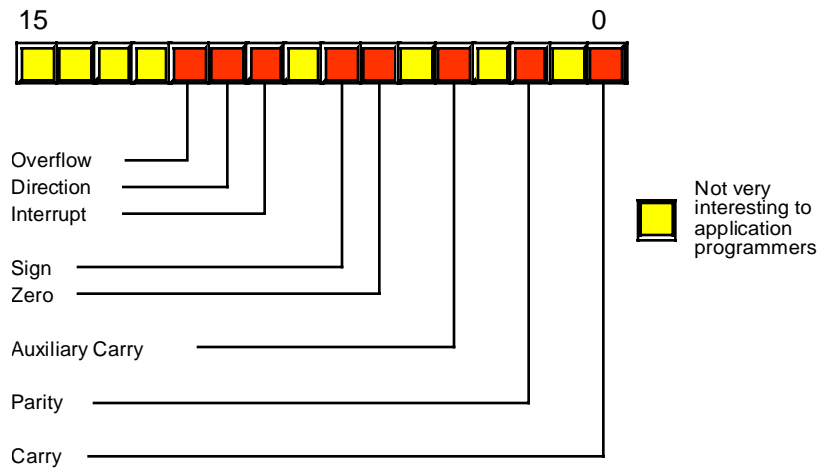


Figure 2.6 Layout of the FLAGS Register (Lower 16 bits of EFLAGS)

Of the eight flags that are usable by application programmers, four flags in particular are extremely valuable: the overflow, carry, sign, and zero flags. Collectively, we will call these four flags the *condition codes*⁴. The state of these flags (boolean variables) will let you test the results of previous computations and allow you to make decisions in your programs. For example, after comparing two values, the state of the condition code flags will tell you if one value is less than, equal to, or greater than a second value. The 80x86 CPUs provide special machine instructions that let you test the flags, alone or in various combinations.

The last register of interest is the EIP (instruction pointer) register. This 32-bit register contains the *memory address* of the next machine instruction to execute. Although you will manipulate this register directly in your programs, the instructions that modify its value treat this register as an implicit operand. Therefore, you will not need to remember much about this register since the 80x86 instruction set effectively hides it from you.

One important fact that comes as a surprise to those just learning assembly language is that almost all calculations on the 80x86 CPU must involve a register. For example, to add two (memory) variables together, storing the sum into a third location, you must load one of the memory operands into a register, add the second operand to the value in the register, and then store the register away in the destination memory location. Registers are a *middleman* in nearly every calculation. Therefore, registers are very important in 80x86 assembly language programs.

Another thing you should be aware of is that although the general purpose registers have the name “general purpose” you should not infer that you can use any register for any purpose. The SP/ESP register for example, has a very special purpose (it’s the *stack pointer*) that effectively prevents you from using it for any other purpose. Likewise, the BP/EBP register has a special purpose that limits its usefulness as a general purpose register. All the 80x86 registers have their own special purposes that limit their use in certain contexts. For the time being, you should simply avoid the use of the ESP and EBP registers for generic calculations and keep in mind that the remaining registers are not completely interchangeable in your programs.

2.8 Some Basic Machine Instructions

The 80x86 CPUs provide just over a hundred to many thousands of different machine instructions, depending on how you define a machine instruction. Even at the low end of the count (greater than 100), it appears as though there are far too many machine instructions to learn in a short period of time. Fortunately,

4. Technically the parity flag is also a condition code, but we will not use that flag in this text.

you don't need to know all the machine instructions. In fact, most assembly language programs probably use around 30 different machine instructions⁵. Indeed, you can certainly write several meaningful programs with only a small handful of machine instructions. The purpose of this section is to provide a small handful of machine instructions so you can start writing simple HLA assembly language programs right away.

Without question, the MOV instruction is the most often-used assembly language statement. In a typical program, anywhere from 25-40% of the instructions are typically MOV instructions. As its name suggests, this instruction moves data from one location to another⁶. The HLA syntax for this instruction is

```
mov( source_operand, destination_operand );
```

The *source_operand* can be a register, a memory variable, or a constant. The *destination_operand* may be a register or a memory variable. Technically the 80x86 instruction set does not allow both operands to be memory variables; HLA, however, will automatically translate a MOV instruction with two 16- or 32-bit memory operands into a pair of instructions that will copy the data from one location to another. In a high level language like Pascal or C/C++, the MOV instruction is roughly equivalent to the following assignment statement:

```
destination_operand = source_operand ;
```

Perhaps the major restriction on the MOV instruction's operands is that they must both be the same size. That is, you can move data between two eight-bit objects, between two 16-bit objects, or between two 32-bit objects; you may not, however, mix the sizes of the operands. The following table lists all the legal combinations:

Table 1: Legal 80x86 MOV Instruction Operands

Source	Destination
Reg ₈ ^a	Reg ₈
Reg ₈	Mem ₈
Mem ₈	Reg ₈
constant ^b	Reg ₈
constant	Mem ₈
Reg ₁₆	Reg ₁₆
Reg ₁₆	Mem ₁₆
Mem ₁₆	Reg ₁₆
constant	Reg ₁₆
constant	Mem ₁₆
Reg ₃₂	Reg ₃₂

5. Different programs may use a different set of 30 instructions, but few programs use more than 30 distinct instructions.

6. Technically, MOV actually copies data from one location to another. It does not destroy the original data in the source operand. Perhaps a better name for this instruction should have been COPY. Alas, it's too late to change it now.

Table 1: Legal 80x86 MOV Instruction Operands

Reg ₃₂	Mem ₃₂
Mem ₃₂	Reg ₃₂
constant	Reg ₃₂
constant	Mem ₃₂

- a. The suffix denotes the size of the register or memory location.
- b. The constant must be small enough to fit in the specified destination operand

You should study this table carefully. Most of the general purpose 80x86 instructions use this same syntax. Note that in addition to the forms above, the HLA MOV instruction lets you specify two memory operands as the source and destination. However, this special translation that HLA provides only applies to the MOV instruction; it does not generalize to the other instructions.

The 80x86 ADD and SUB instructions let you add and subtract two operands. Their syntax is nearly identical to the MOV instruction:

```
add( source_operand, destination_operand );
sub( source_operand, destination_operand );
```

The ADD and SUB operands must take the same form as the MOV instruction, listed in the table above⁷. The ADD instruction does the following:

```
destination_operand = destination_operand + source_operand ;
destination_operand += source_operand; // For those who prefer C syntax
```

Similarly, the SUB instruction does the calculation:

```
destination_operand = destination_operand - source_operand ;
destination_operand -= source_operand ; // For C fans.
```

With nothing more than these three instructions, plus the HLA control structures that the next section discusses, you can actually write some sophisticated programs. Here's a sample HLA program that demonstrates these three instructions:

```
program DemoMOVaddSUB;

#include( "stdlib.hhf" );

static
    i8:    int8    := -8;
    i16:   int16   := -16;
    i32:   int32   := -32;

begin DemoMOVaddSUB;

    // First, print the initial values
    // of our variables.

    stdout.put
    (
        nl,
```

7. Remember, though, that ADD and SUB do not support memory-to-memory operations.

```

        "Initialized values: i8=", i8,
        ", i16=", i16,
        ", i32=", i32,
        nl
    );

    // Compute the absolute value of the
    // three different variables and
    // print the result.
    // Note, since all the numbers are
    // negative, we have to negate them.
    // Using only the MOV, ADD, and SUB
    // instruction, we can negate a value
    // by subtracting it from zero.

    mov( 0, al ); // Compute i8 := -i8;
    sub( i8, al );
    mov( al, i8 );

    mov( 0, ax ); // Compute i16 := -i16;
    sub( i16, ax );
    mov( ax, i16 );

    mov( 0, eax ); // Compute i32 := -i32;
    sub( i32, eax );
    mov( eax, i32 );

    // Display the absolute values:

    stdout.put
    (
        nl,
        "After negation: i8=", i8,
        ", i16=", i16,
        ", i32=", i32,
        nl
    );

    // Demonstrate ADD and constant-to-memory
    // operations:

    add( 32323200, i32 );
    stdout.put( nl, "After ADD: i32=", i32, nl );

end DemoMOVaddSUB;

```

Program 2.3 Demonstration of MOV, ADD, and SUB Instructions

2.9 Some Basic HLA Control Structures

The MOV, ADD, and SUB instructions, while valuable, aren't sufficient to let you write meaningful programs. You will need to complement these instructions with the ability to make decisions and create loops in your HLA programs before you can write anything other than a trivial program. HLA provides several high level control structures that are very similar to control structures found in high level languages. These

include IF..THEN..ELSEIF..ELSE..ENDIF, WHILE..ENDWHILE, REPEAT..UNTIL, and so on. By learning these statements you will be armed and ready to write some real programs.

Before discussing these high level control structures, it's important to point out that these are not real 80x86 assembly language statements. HLA compiles these statements into a sequence of one or more real assembly language statements for you. Later in this text, you'll learn how HLA compiles the statements and you'll learn how to write pure assembly language code that doesn't use them. However, you'll need to learn many new concepts before you get to that point, so we'll stick with these high level language statements for now since you're probably already familiar with statements like these from your exposure to high level languages.

Another important fact to mention is that HLA's high level control structures are *not* as high level as they first appear. The purpose behind HLA's high level control structures is to let you start writing assembly language programs as quickly as possible, not to let you avoid the use of real assembly language altogether. You will soon discover that these statements have some severe restrictions associated with them and you will quickly outgrow their capabilities (at least the restricted forms appearing in this section). This is intentional. Once you reach a certain level of comfort with HLA's high level control structures and decide you need more power than they have to offer, it's time to move on and learn the real 80x86 instructions behind these statements.

2.9.1 Boolean Expressions in HLA Statements

Several HLA statements require a boolean (true or false) expression to control their execution. Examples include the IF, WHILE, and REPEAT..UNTIL statements. The syntax for these boolean expressions represents the greatest limitation of the HLA high level control structures. This is one area where your familiarity with a high level language will work against you – you'll want to use the same boolean expressions you use in a high level language and HLA only supports some basic forms.

HLA boolean expressions always take the following forms⁸:

```

flag_specification
!flag_specification
register
!register
Boolean_variable
!Boolean_variable
mem_reg relop mem_reg_const
register in LowConst..HiConst
register not in LowConst..HiConst

```

A *flag_specification* may be one of the following symbols:

- @c carry: True if the carry is set (1), false if the carry is clear (0).
- @nc no carry: True if the carry is clear (0), false if the carry is set (1).
- @z zero: True if the zero flag is set, false if it is clear.
- @nz not zero: True if the zero flag is clear, false if it is set.
- @o overflow: True if the overflow flag is set, false if it is clear.
- @no no overflow: True if the overflow flag is clear, false if it is set.
- @s sign: True if the sign flag is set, false if it is clear.
- @ns no sign: True if the sign flag is clear, false if it is set.

8. Technically, there are a few more, advanced, forms, but you'll have to wait a few chapters before seeing these additional formats.

The use of the flag values in a boolean expression is somewhat advanced. You will begin to see how to use these boolean expression operands in the next chapter.

A register operand can be any of the 8-bit, 16-bit, or 32-bit general purpose registers. The expression evaluates false if the register contains a zero; it evaluates true if the register contains a non-zero value.

If you specify a boolean variable as the expression, the program tests it for zero (false) or non-zero (true). Since HLA uses the values zero and one to represent false and true, respectively, the test works in an intuitive fashion. Note that HLA requires that stand-alone variables be of type *boolean*. HLA rejects other data types. If you want to test some other type against zero/not zero, then use the general boolean expression discussed next.

The most general form of an HLA boolean expression has two operands and a relational operator. The following table lists the legal combinations:

Table 2: Legal Boolean Expressions

Left Operand	Relational Operator	Right Operand
Memory Variable or Register	= or ==	Memory Variable, Register, or Constant
	<> or !=	
	<	
	<=	
	>	
	>=	

Note that both operands cannot be memory operands. In fact, if you think of the Right Operand as the source operand and the Left Operand as the destination operand, then the two operands must be the same as those allowed for the ADD and SUB instructions.

Also like the ADD and SUB instructions, the two operands must be the same size. That is, they must both be eight-bit operands, they must both be 16-bit operands, or they must both be 32-bit operands. If the Right Operand is a constant, it's value must be in the range that is compatible with the Left Operand.

There is one other issue of which you need to be aware. If the Left Operand is a register and the Right Operand is a positive constant or another register, HLA uses an *unsigned* comparison. The next chapter will discuss the ramifications of this; for the time being, do not compare negative values in a register against a constant or another register. You may not get an intuitive result.

The IN and NOT IN operators let you test a register to see if it is within a specified range. For example, the expression "EAX in 2000..2099" evaluates true if the value in the EAX register is between 2000 and 2099 (inclusive). The NOT IN (two words) operator lets you check to see if the value in a register is outside the specified range. For example, "AL not in 'a'..'z'" evaluates true if the character in the AL register is not a lower case alphabetic character.

Here are some examples of legal boolean expressions in HLA:

```
@c
Bool_var
al
ESI
EAX < EBX
```

```

EBX > 5
i32 < -2
i8 > 128
al < i8
eax in 1..100
ch not in 'a'..'z'

```

2.9.2 The HLA IF..THEN..ELSEIF..ELSE..ENDIF Statement

The HLA IF statement uses the following syntax:

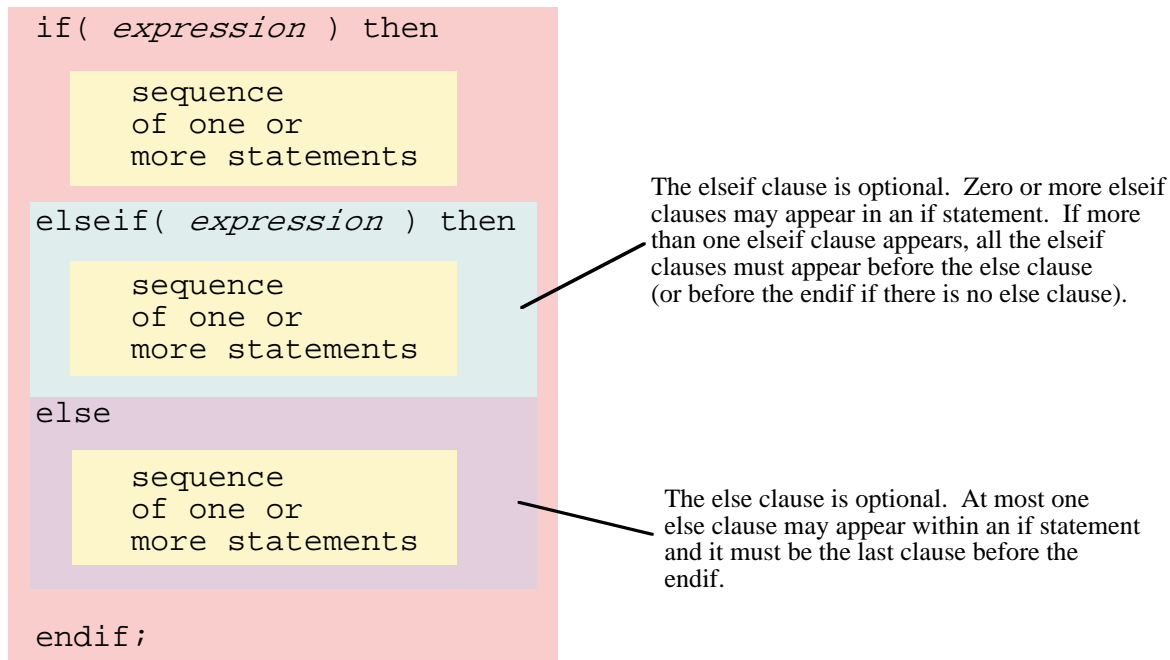


Figure 2.7 HLA IF Statement Syntax

The expressions appearing in this statement must take one of the forms from the previous section. If the associated expression is true, the code after the THEN executes, otherwise control transfers to the next ELSEIF or ELSE clause in the statement.

Since the ELSEIF and ELSE clauses are optional, an IF statement could take the form of a single IF..THEN clause, followed by a sequence of statements, and a closing ENDIF clause. The following is an example of just such a statement:

```

if( eax = 0 ) then

    stdout.put( "error: NULL value", nl );

endif;

```

If, during program execution, the expression evaluates true, then the code between the THEN and the ENDIF executes. If the expression evaluates false, then the program skips over the code between the THEN and the ENDIF.

Another common form of the IF statement has a single ELSE clause. The following is an example of an IF statement with an optional ELSE:

```
if( eax = 0 ) then
    stdout.put( "error: NULL pointer encountered", nl );
else
    stdout.put( "Pointer is valid", nl );
endif;
```

If the expression evaluates true, the code between the THEN and the ELSE executes; otherwise the code between the ELSE and the ENDIF clauses executes.

You can create sophisticated decision-making logic by incorporating the ELSEIF clause into an IF statement. For example, if the CH register contains a character value, you can select from a menu of items using code like the following:

```
if( ch = 'a' ) then
    stdout.put( "You selected the 'a' menu item", nl );
elseif( ch = 'b' ) then
    stdout.put( "You selected the 'b' menu item", nl );
elseif( ch = 'c' ) then
    stdout.put( "You selected the 'c' menu item", nl );
else
    stdout.put( "Error: illegal menu item selection", nl );
endif;
```

Although this simple example doesn't demonstrate it, HLA does not require an ELSE clause at the end of a sequence of ELSEIF clauses. However, when making multi-way decisions, it's always a good idea to provide an ELSE clause just in case an error arises. Even if you think it's impossible for the ELSE clause to execute, just keep in mind that future modifications to the code could possibly void this assertion, so it's a good idea to have error reporting statements built into your code.

2.9.3 The WHILE..ENDWHILE Statement

The WHILE statement uses the following basic syntax:

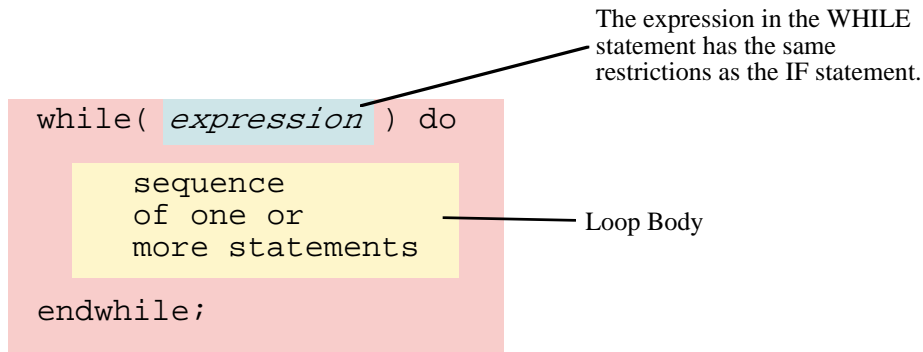


Figure 2.8 HLA While Statement Syntax

This statement evaluates the boolean expression. If it is false, control immediately transfers to the first statement following the ENDWHILE clause. If the value of the expression is true, then control falls through to the body of the loop. After the loop body executes, control transfers back to the top of the loop where the WHILE statement retests the loop control expression. This process repeats until the expression evaluates false.

Note that the WHILE loop, like its high level language siblings, tests for loop termination at the top of the loop. Therefore, it is quite possible that the statements in the body of the loop will not execute (if the expression is false when the code first executes the WHILE statement). Also note that the body of the WHILE loop must, at some point, modify the value of the boolean expression or an infinite loop will result.

```

mov( 0, i );
while( i < 10 ) do

    stdout.put( "i=", i, nl );
    add( 1, i );

endwhile;

```

2.9.4 The FOR..ENDFOR Statement

The HLA FOR loop takes the following general form:

```

for( Initial Stmt; Termination Expression; Post Body Statement ) do

    << Loop Body >>

endfor;

```

This is equivalent to the following WHILE statement:

```

Initial Stmt;
while( Termination expression ) do

    << loop_body >>

    Post Body Statement;

endwhile;

```

Initial Stmt can be any single HLA/80x86 instruction. Generally this statement initializes a register or memory location (the loop counter) with zero or some other initial value. *Termination_expression* is an

HLA boolean expression (same format that WHILE allows). This expression determines whether the loop body will execute. The *Post_Body_Statement* executes at the bottom of the loop (as shown in the WHILE example above). This is a single HLA statement. Usually it is an instruction like ADD that modifies the value of the loop control variable.

The following gives a complete example:

```
for( mov( 0, i ); i < 10; add(1, i ) ) do

    stdout.put( "i=", i, nl );

endfor;

// The above, rewritten as a while loop, becomes:

mov( 0, i );
while( i < 10 ) do

    stdout.put( "i=", i, nl );

    add( 1, i );

endwhile;
```

2.9.5 The REPEAT..UNTIL Statement

The HLA repeat..until statement uses the following syntax:

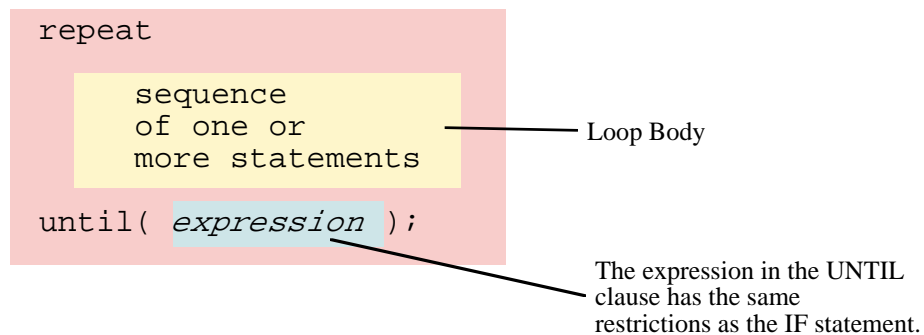


Figure 2.9 HLA Repeat..Until Statement Syntax

The HLA REPEAT..UNTIL statement tests for loop termination at the bottom of the loop. Therefore, the statements in the loop body always execute at least once. Upon encountering the UNTIL clause, the program will evaluate the expression and repeat the loop if the expression is false (that is, it repeats while false). If the expression evaluates true, the control transfers to the first statement following the UNTIL clause.

The following simple example demonstrates one use for the REPEAT..UNTIL statement:

```
mov( 10, ecx );
repeat

    stdout.put( "ecx = ", ecx, nl );
    sub( 1, ecx );

until( ecx = 0 );
```

If the loop body will always execute at least once, then it is more efficient to use a REPEAT..UNTIL loop rather than a WHILE loop.

2.9.6 The BREAK and BREAKIF Statements

The BREAK and BREAKIF statements provide the ability to prematurely exit from a loop. They use the following syntax:

```
break;
```

```
breakif( expression );
```

The expression in the BREAKIF statement has the same restrictions as the IF statement.

Figure 2.10 HLA Break and Breakif Syntax

The BREAK statement exits the loop that immediately contains the break; The BREAKIF statement evaluates the boolean expression and terminates the containing loop if the expression evaluates true.

2.9.7 The FOREVER..ENDFOR Statement

The FOREVER statement uses the following syntax:

```
forever
```

```
sequence
```

```
of one or
```

```
more statements
```

```
endfor;
```

Loop Body

Figure 2.11 HLA Forever Loop Syntax

This statement creates an infinite loop. You may also use the BREAK and BREAKIF statements along with FOREVER..ENDFOR to create a loop that tests for loop termination in the middle of the loop. Indeed, this is probably the most common use of this loop as the following example demonstrates:

```
forever
```

```
    stdout.put( "Enter an integer less than 10: ");
```

```
    stdin.get( i );
```

```
    breakif( i < 10 );
```

```
    stdout.put( "The value needs to be less than 10!", nl );
```

```
endfor;
```

2.9.8 The TRY..EXCEPTION..ENDTRY Statement

The HLA TRY..EXCEPTION..ENDTRY statement provides very powerful *exception handling* capabilities. The syntax for this statement is the following:

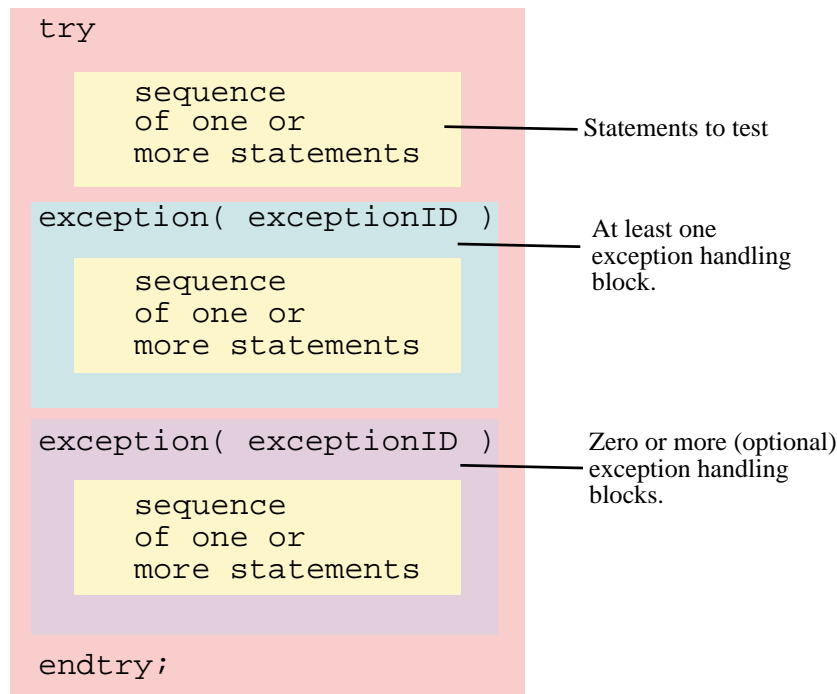


Figure 2.12 HLA Try..Except..Endtry Statement Syntax

The TRY..ENDTRY statement protects a block of statements during execution. If these statements, between the TRY clause and the first EXCEPTION clause, execute without incident, control transfers to the first statement after the ENDTRY immediately after executing the last statement in the protected block. If an error (exception) occurs, then the program interrupts control at the point of the exception (that is, the program *raises* an exception). Each exception has an unsigned integer constant associated with it, known as the exception ID. The “excepts.hhf” header file in the HLA Standard Library predefines several exception IDs, although you may create new ones for your own purposes. When an exception occurs, the system compares the exception ID against the values appearing in each of the one or more EXCEPTION clauses following the protected code. If the current exception ID matches one of the EXCEPTION values, control continues with the block of statements immediately following that EXCEPTION. After the exception handling code completes execution, control transfers to the first statement following the ENDTRY.

If an exception occurs and there is no active TRY..ENDTRY statement, or the active TRY..ENDTRY statements do not handle the specific exception, the program will abort with an error message.

The following sample program demonstrates how to use the TRY..ENDTRY statement to protect the program from bad user input:

```

repeat

    mov( false, GoodInteger );    // Note: GoodInteger must be a boolean var.
    try

        stdout.put( "Enter an integer: " );
        stdin.get( i );
        mov( true, GoodInteger );

    exception( ex.ConversionError );

        stdout.put( "Illegal numeric value, please re-enter", nl );

    exception( ex.ValueOutOfRange );

        stdout.put( "Value is out of range, please re-enter", nl );

    endtry;

until( GoodInteger );

```

The REPEAT..UNTIL loop repeats this code as long as there is an error during input. Should an exception occur, control transfers to the EXCEPTION clauses to see if a conversion error (e.g., illegal characters in the number) or a numeric overflow occurs. If either of these exceptions occur, then they print the appropriate message and control falls out of the TRY..ENDTRY statement and the REPEAT..UNTIL loop repeats since *GoodInteger* was never set to true. If a different exception occurs (one that is not handled in this code), then the program aborts with the specified error message⁹.

Please see the “excepts.hhf” header file that accompanies the HLA release for a complete list of all the exception ID codes. The HLA documentation will describe the purpose of each of these exception codes.

2.10 Introduction to the HLA Standard Library

There are two reasons HLA is much easier to learn and use than standard assembly language. The first reason is HLA’s high level syntax for declarations and control structures. This HLA feature leverages your high level language knowledge, reducing the need to learn arcane syntax, allowing you to learn assembly language more efficiently. The other half of the equation is the HLA Standard Library. The HLA Standard Library provides lot of commonly needed, easy to use, assembly language routines that you can call without having to write this code yourself (or even learn how to write yourself). This eliminates one of the larger stumbling blocks many people have when learning assembly language: the need for sophisticated I/O and support code in order to write basic statements. Prior to the advent of a standardized assembly language library, it often took weeks of study before a new assembly language programmer could do as much as print a string to the display. With the HLA Standard Library, this roadblock is removed and you can concentrate on learning assembly language concepts rather than learning low-level I/O details that are specific to a given operating system.

A wide variety of library routines is only part of HLA’s support. After all, assembly language libraries have been around for quite some time¹⁰. HLA’s Standard Library continues the HLA tradition by providing a high level language interface to these routines. Indeed, the HLA language itself was originally designed specifically to allow the creation of a high-level accessible set of library routines¹¹. This high level interface,

9. An experienced programmer may wonder why this code uses a boolean variable rather than a BREAKIF statement to exit the REPEAT..UNTIL loop. There are some technical reasons for this that you will learn about later in this text.

10. E.g., the UCR Standard Library for 80x86 Assembly Language Programmers.

11. HLA was created because MASM was insufficient to support the creation of the UCR StdLib v2.0.

combined with the high level nature of many of the routines in the library, packs a surprising amount of power in an easy to use package.

The HLA Standard Library consists of several modules organized by category. The following table lists many of the modules that are available¹²:

Table 3: HLA Standard Library Modules

Name	Description
args	Command line parameter parsing support routines.
conv	Various conversions between strings and other values.
cset	Character set functions.
DateTime	Calendar, date, and time functions.
excepts	Exception handling routines.
fileio	File input and output routines
hla	Special HLA constants and other values.
Linux	Linux system calls (HLA Linux version only).
math	Transcendental and other mathematical functions.
memory	Memory allocation, deallocation, and support code.
misctypes	Miscellaneous data types.
patterns	The HLA pattern matching library.
rand	Pseudo-random number generators and support code.
stdin	User input routines
stdout	Provides user output and several other support routines.
stdlib	A special include file that links in all HLA standard library modules.
strings	HLA's powerful string library.
tables	Table (associative array) support routines.
win32	Constants used in Windows calls (HLA Win32 version, only)
x86	Constants and other items specific to the 80x86 CPU.

Later sections of this text will explain many of these modules in greater detail. This section will concentrate on the most important routines (at least to beginning HLA programmers), the *stdio* library.

12. Since the HLA Standard Library is expanding, this list is probably out of date. Please see the HLA documentation for a current list of Standard Library modules.

2.10.1 Predefined Constants in the STDIO Module

Perhaps the first place to start is with a description of some common constants that the `STDIO` module defines for you. One constant you've seen already in code appearing in this chapter. Consider the following (typical) example:

```
stdout.put( "Hello World", nl );
```

The `nl` appearing at the end of this statement stands for *newline*. The `nl` identifier is not a special HLA reserved word, nor is it specific to the `stdout.put` statement. Instead, it's simply a predefined constant that corresponds to the string containing a single linefeed character (the standard Windows end of line sequence).

In addition to the `nl` constant, the HLA standard I/O library module defines several other useful character constants. They are

- `stdio.bell` The ASCII bell character. Beeps the speaker when printed.
- `stdio.bs` The ASCII backspace character.
- `stdio.tab` The ASCII tab character.
- `stdio.eoln` A linefeed character (even under Windows).
- `stdio.lf` The ASCII linefeed character.
- `stdio.cr` The ASCII carriage return character.

Except for `nl`, these characters appear in the `stdio` namespace (and, therefore, require the “`stdio.`” prefix). The placement of these ASCII constants within the `stdio` namespace is to help avoid naming conflicts with your own variables. The `nl` name does not appear within a namespace because you will use it very often and typing `stdio.nl` would get tiresome very quickly.

2.10.2 Standard In and Standard Out

Many of the HLA I/O routines have a `stdin` or `stdout` prefix. Technically, this means that the standard library defines these names in a *namespace*¹³. In practice, this prefix suggests where the input is coming from (the *standard input* device) or going to (the *standard output* device). By default, the standard input device is the system keyboard. Likewise, the default standard output device is the console display. So, in general, statements that have `stdin` or `stdout` prefixes will read and write data on the console device.

When you run a program from the command line window (or shell), you have the option of *redirecting* the standard input and/or standard output devices. A command line parameter of the form “`>outfile`” redirects the standard output device to the specified file (outfile). A command line parameter of the form “`<infile`” redirects the standard input so that its data comes from the specified input file (infile). The following examples demonstrate how to use these parameters when running a program named “`testpgm`” in the command window¹⁴:

```
testpgm <input.data
testpgm >output.txt
testpgm <in.txt >output.txt
```

2.10.3 The `stdout.newln` Routine

The `stdout.newln` procedure prints a newline sequence to the standard output device. This is functionally equivalent to saying “`stdout.put(nl);`” Of course, the call to `stdout.newln` is sometimes a little more convenient. Example of call:

13. Namespaces will be the subject of a later chapter.

14. Note for Linux users: depending on how your system is set up, you may need to type “`./`” in front of the program's name to actually execute the program, e.g., “`./testpgm <input.data`”.

```
stdout.newln();
```

2.10.4 The `stdout.putiX` Routines

The `stdout.puti8`, `stdout.puti16`, and `stdout.puti32` library routines print a single parameter (one byte, two bytes, or four bytes, respectively) as a signed integer value. The parameter may be a constant, a register, or a memory variable, as long as the size of the actual parameter is the same as the size of the formal parameter.

These routines print the value of their specified parameter to the standard output device. These routines will print the value using the minimum number of print positions possible. If the number is negative, these routines will print a leading minus sign. Here are some examples of calls to these routines:

```
stdout.puti8( 123 );
stdout.puti16( DX );
stdout.puti32( i32Var );
```

2.10.5 The `stdout.putiXSize` Routines

The `stdout.puti8Size`, `stdout.puti16Size`, and `stdout.puti32Size` routines output signed integer values to the standard output, just like the `stdout.putiX` routines. These routines, however, provide more control over the output; they let you specify the (minimum) number of print positions the value will require on output. These routines also let you specify a padding character should the print field be larger than the minimum needed to display the value. These routines require the following parameters:

```
stdout.puti8Size( Value8, width, padchar );
stdout.puti16Size( Value16,width, padchar );
stdout.puti32Size( Value32, width, padchar );
```

The *ValueX* parameter can be a constant, a register, or a memory location of the specified size. The *width* parameter can be any signed integer constant that is between -256 and +256; this parameter may be a constant, register (32-bit), or memory location (32-bit). The *padchar* parameter should be a single character value.

Like the `stdout.putiX` routines, these routines print the specified value as a signed integer constant to the standard output device. These routines, however, let you specify the *field width* for the value. The field width is the minimum number of print positions these routines will use when printing the value. The *width* parameter specifies the minimum field width. If the number would require more print positions (e.g., if you attempt to print “1234” with a field width of two), then these routines will print however many characters are necessary to properly display the value. On the other hand, if the *width* parameter is greater than the number of character positions required to display the value, then these routines will print some extra padding characters to ensure that the output has at least *width* character positions. If the *width* value is negative, the number is left justified in the print field; if the *width* value is positive, the number is right justified in the print field.

If the absolute value of the *width* parameter is greater than the minimum number of print positions, then these `stdout.putiXSize` routines will print a padding character before or after the number. The *padchar* parameter specifies which character these routines will print. Most of the time you would specify a space as the pad character; for special cases, you might specify some other character. Remember, the *padchar* parameter is a character value; in HLA character constants are surrounded by apostrophes, not quotation marks. You may also specify an eight-bit register as this parameter.

Here is a short HLA program that demonstrates the use of the `puti32Size` routine to display a list of values in tabular form:

```
program NumsInColumns;
```

```

#include( "stdlib.hhf" );

var
  i32:    int32;
  ColCnt: int8;

begin NumsInColumns;

  mov( 96, i32 );
  mov( 0, ColCnt );
  while( i32 > 0 ) do

    if( ColCnt = 8 ) then

      stdout.newln();
      mov( 0, ColCnt );

    endif;
    stdout.puti32Size( i32, 5, ' ' );
    sub( 1, i32 );
    add( 1, ColCnt );

  endwhile;
  stdout.newln();

end NumsInColumns;

```

Program 2.4 Columnar Output Demonstration Using `stdio.Puti32Size`

2.10.6 The `stdout.put` Routine

The `stdout.put` routine¹⁵ is one of the most flexible output routines in the standard output library module. It combines most of the other output routines into a single, easy to use, procedure.

The generic form for the `stdout.put` routine is the following:

```
stdout.put( list_of_values_to_output );
```

The `stdout.put` parameter list consists of one or more constants, registers, or memory variables, each separated by a comma. This routine displays the value associated with each parameter appearing in the list. Since we've already been using this routine throughout this chapter, you've already seen lots of examples of this routine's basic form. It is worth pointing out that this routine has several additional features not apparent in the examples appearing in this chapter. In particular, each parameter can take one of the following two forms:

```

value
value:width

```

The *value* may be any legal constant, register, or memory variable object. In this chapter, you've seen string constants and memory variables appearing in the `stdout.put` parameter list. These parameters correspond to the first form above. The second parameter form above lets you specify a minimum field width, similar to the `stdout.putiXSize` routines¹⁶. The following sample program produces the same output as the previous program; however, it uses `stdout.put` rather than `stdout.puti32Size`:

15. `Stdout.put` is actually a macro, not a procedure. The distinction between the two is beyond the scope of this chapter. However, this text will describe their differences a little later.

```

program NumsInColumns2;

#include( "stdlib.hhf" );

var
    i32:    int32;
    ColCnt: int8;

begin NumsInColumns2;

    mov( 96, i32 );
    mov( 0, ColCnt );
    while( i32 > 0 ) do

        if( ColCnt = 8 ) then

            stdout.newln();
            mov( 0, ColCnt );

        endif;
        stdout.put( i32:5 );
        sub( 1, i32 );
        add( 1, ColCnt );

    endwhile;
    stdout.put( nl );

end NumsInColumns2;

```

Program 2.5 Demonstration of stdout.put Field Width Specification

The *stdout.put* routine is capable of much more than the few attributes this section describes. This text will introduce those additional capabilities as appropriate.

2.10.7 The stdin.getc Routine.

The *stdin.getc* routine reads the next available character from the standard input device's input buffer¹⁷. It returns this character in the CPU's AL register. The following example program demonstrates a simple use of this routine:

```

program charInput;

#include( "stdlib.hhf" );

var
    counter: int32;

```

16. Note that you cannot specify a padding character when using the *stdout.put* routine; the padding character defaults to the space character. If you need to use a different padding character, call the *stdout.putiXSize* routines.

17. "Buffer" is just a fancy term for an array.

```

begin charInput;

    // The following repeats as long as the user
    // confirms the repetition.

    repeat

        // Print out 14 values.

        mov( 14, counter );
        while( counter > 0 ) do

            stdout.put( counter:3 );
            sub( 1, counter );

        endwhile;

        // Wait until the user enters 'y' or 'n'.

        stdout.put( nl, nl, "Do you wish to see it again? (y/n):" );
        forever

            stdin.readLine();
            stdin.getc();
            breakif( al = 'n' );
            breakif( al = 'y' );
            stdout.put( "Error, please enter only 'y' or 'n': " );

        endfor;
        stdout.newln();

    until( al = 'n' );

end charInput;

```

Program 2.6 Demonstration of the `stdin.getc()` Routine

This program uses the `stdin.ReadLine` routine to force a new line of input from the user. A description of `stdin.ReadLine` appears just a little later in this chapter.

2.10.8 The `stdin.getiX` Routines

The `stdin.geti8`, `stdin.geti16`, and `stdin.geti32` routines read eight, 16, and 32-bit signed integer values from the standard input device. These routines return their values in the AL, AX, or EAX register, respectively. They provide the standard mechanism for reading signed integer values from the user in HLA.

Like the `stdin.getc` routine, these routines read a sequence of characters from the standard input buffer. They begin by skipping over any white space characters (spaces, tabs, etc.) and then convert the following stream of decimal digits (with an optional, leading, minus sign) into the corresponding integer. These routines raise an exception (that you can trap with the TRY..ENDTRY statement) if the input sequence is not a valid integer string or if the user input is too large to fit in the specified integer size. Note that values read by `stdin.geti8` must be in the range -128..+127; values read by `stdin.geti16` must be in the range -32,768..+32,767; and values read by `stdin.geti32` must be in the range -2,147,483,648..+2,147,483,647.

The following sample program demonstrates the use of these routines:

```

program intInput;

#include( "stdlib.hhf" );

var
    i8:      int8;
    i16:     int16;
    i32:     int32;

begin intInput;

    // Read integers of varying sizes from the user:

    stdout.put( "Enter a small integer between -128 and +127: " );
    stdin.geti8();
    mov( al, i8 );

    stdout.put( "Enter a small integer between -32768 and +32767: " );
    stdin.geti16();
    mov( ax, i16 );

    stdout.put( "Enter an integer between +/- 2 billion: " );
    stdin.geti32();
    mov( eax, i32 );

    // Display the input values.

    stdout.put
    (
        nl,
        "Here are the numbers you entered:", nl, nl,
        "Eight-bit integer: ", i8:12, nl,
        "16-bit integer:    ", i16:12, nl,
        "32-bit integer:     ", i32:12, nl
    );

end intInput;

```

Program 2.7 stdin.getiX Example Code

You should compile and run this program and test what happens when you enter a value that is out of range or enter an illegal string of characters.

2.10.9 The `stdin.readLine` and `stdin.flushInput` Routines

Whenever you call an input routine like `stdin.getc` or `stdin.geti32`, the program does not necessarily read the value from the user at that moment. Instead, the HLA Standard Library buffers the input by reading a whole line of text from the user. Calls to input routines will fetch data from this input buffer until the buffer is empty. While this buffering scheme is efficient and convenient, sometimes it can be confusing. Consider the following code sequence:

```

stdout.put( "Enter a small integer between -128 and +127: " );

```

```

stdin.geti8();
mov( al, i8 );

stdout.put( "Enter a small integer between -32768 and +32767: " );
stdin.geti16();
mov( ax, i16 );

```

Intuitively, you would expect the program to print the first prompt message, wait for user input, print the second prompt message, and wait for the second user input. However, this isn't exactly what happens. For example if you run this code (from the sample program in the previous section) and enter the text "123 456" in response to the first prompt, the program will not stop for additional user input at the second prompt. Instead, it will read the second integer (456) from the input buffer read during the execution of the *stdin.geti8* call.

In general, the *stdin* routines only read text from the user when the input buffer is empty. As long as the input buffer contains additional characters, the input routines will attempt to read their data from the buffer. You may take advantage of this behavior by writing code sequences such as the following:

```

stdout.put( "Enter two integer values: " );
stdin.geti32();
mov( eax, intval );
stdin.geti32();
mov( eax, AnotherIntVal );

```

This sequence allows the user to enter both values on the same line (separated by one or more white space characters) thus preserving space on the screen. So the input buffer behavior is desirable every now and then.

Unfortunately, the buffered behavior of the input routines is definitely counter-intuitive at other times. Fortunately, the HLA Standard Library provides two routines, *stdin.readLine* and *stdin.flushInput*, that let you control the standard input buffer. The *stdin.readLine* routine discards everything that is in the input buffer and immediately requires the user to enter a new line of text. The *stdin.flushInput* routine simply discards everything that is in the buffer. The next time an input routine executes, the system will require a new line of input from the user. You would typically call *stdin.readLine* immediately before some standard input routine; you would normally call *stdin.flushInput* immediately after a call to a standard input routine.

Note: If you are calling *stdin.readLine* and you find that you are having to input your data twice, this is a good indication that you should be calling *stdin.flushInput* rather than *stdin.readLine*. In general, you should always be able to call *stdin.flushInput* to flush the input buffer and read a new line of data on the next input call. The *stdin.readLine* routine is rarely necessary, so you should use *stdin.flushInput* unless you really need to immediately force the input of a new line of text.

2.10.10 The *stdin.get* Macro

The *stdin.get* macro combines many of the standard input routines into a single call, in much the same way that *stdout.put* combines all of the output routines into a single call. Actually, *stdin.get* is much easier to use than *stdout.put* since the only parameters to this routine are a list of variable names.

Let's rewrite the example given in the previous section:

```

stdout.put( "Enter two integer values: " );
stdin.geti32();
mov( eax, intval );
stdin.geti32();
mov( eax, AnotherIntVal );

```

Using the *stdin.get* macro, we could rewrite this code as:

```

stdout.put( "Enter two integer values: " );
stdin.get( intval, AnotherIntVal );

```

As you can see, the *stdin.get* routine is a little more convenient to use.

Note that *stdin.get* stores the input values directly into the memory variables you specify in the parameter list; it does not return the values in a register unless you actually specify a register as a parameter. The *stdin.get* parameters must all be variables or registers¹⁸.

2.11 Putting It All Together

This chapter has covered a lot of ground! While you've still got a lot to learn about assembly language programming, this chapter, combined with your knowledge of high level languages, provides just enough information to let you start writing real assembly language programs.

In this chapter, you've seen the basic format for an HLA program. You've seen how to declare integer, character, and boolean variables. You have taken a look at the internal organization of the Intel 80x86 CPU family and learned about the MOV, ADD, and SUB instructions. You've looked at the basic HLA high level language control structures (IF, WHILE, REPEAT, FOR, BREAK, BREAKIF, FOREVER, and TRY) as well as what constitutes a legal boolean expression in these statements. Finally, this chapter has introduced several commonly-used routines in the HLA Standard Library.

You might think that knowing only three machine instructions is hardly sufficient to write meaningful programs. However, those three instructions (*mov*, *add*, and *sub*), combined with the HLA high level control structures and the HLA Standard Library routines are actually equivalent to knowing several dozen machine instructions. Certainly enough to write simple programs. Indeed, with only a few more arithmetic instructions plus the ability to write your own procedures, you'll be able to write almost any program. Of course, your journey into the world of assembly language has only just begun; you'll learn some more instructions, and how to use them, starting in the next chapter.

2.12 Sample Programs

This section contains several little HLA programs that demonstrate some of HLA's features appearing in this chapter. These short examples also demonstrate that it is possible to write meaningful (if simple) programs in HLA using nothing more than the information appearing in this chapter. You may find all of the sample programs appearing in this section in the subdirectory of the "volume1" directory in the software that accompanies this text.

2.12.1 Powers of Two Table Generation

The following sample program generates a table listing all the powers of two between 2**0 and 2**30.

```
// PowersOfTwo-
//
// This program generates a nicely-formatted
// "Powers of Two" table. It computes the
// various powers of two by successively
// doubling the value in the pwrOf2 variable.

program PowersOfTwo;
#include( "stdlib.hhf" );

static
```

18. Note that register input is always in hexadecimal or base 16. The next chapter will discuss hexadecimal numbers.

```

    pwrOf2:    int32;
    LoopCntr:  int32;

begin PowersOfTwo;

    // Print a start up banner.

    stdout.put( "Powers of two: ", nl, nl );

    // Initialize "pwrOf2" with 2**0 (two raised to the zero power).

    mov( 1, pwrOf2 );

    // Because of the limitations of 32-bit signed integers,
    // we can only display 2**0..2**30.

    mov( 0, LoopCntr );
    while( LoopCntr < 31 ) do

        stdout.put( "2**(", LoopCntr:2, ") = ", pwrOf2:10, nl );

        // Double the value in pwrOf2 to compute the
        // next power of two.

        mov( pwrOf2, eax );
        add( eax, eax );
        mov( eax, pwrOf2 );

        // Move on to the next loop iteration.

        inc( LoopCntr );

    endwhile;
    stdout.newln();

end PowersOfTwo;

```

Program 2.8 Powers of Two Table Generator Program

2.12.2 Checkerboard Program

This short little program demonstrates how to generate a checkerboard pattern with HLA.

```

// CheckerBoard-
//
// This program demonstrates how to draw a
// checkerboard using a set of nested while
// loops.

program CheckerBoard;
#include( "stdlib.hhf" );

static

```

```

xCoord:    int8;    // Counts off eight squares in each row.
yCoord:    int8;    // Counts off four pairs of squares in each column.
ColCntr:   int8;    // Counts off four rows in each square.

begin CheckerBoard;

    mov( 0, yCoord );
    while( yCoord < 4 ) do

        // Display a row that begins with black.

        mov( 4, ColCntr );
        repeat

            // Each square is a 4x4 group of
            // spaces (white) or asterisks (black).
            // Print out one row of asterisks/spaces
            // for the current row of squares:

            mov( 0, xCoord );
            while( xCoord < 4 ) do

                stdout.put( "****  " );
                add( 1, xCoord );

            endwhile;
            stdout.newln();
            sub( 1, ColCntr );

        until( ColCntr = 0 );

        // Display a row that begins with white.

        mov( 4, ColCntr );
        repeat

            // Print out a single row of
            // spaces/asterisks for this
            // row of squares:

            mov( 0, xCoord );
            while( xCoord < 4 ) do

                stdout.put( "  ****" );
                add( 1, xCoord );

            endwhile;
            stdout.newln();
            sub( 1, ColCntr );

        until( ColCntr = 0 );

        add( 1, yCoord );

    endwhile;

end CheckerBoard;

```

 Program 2.9 Checkerboard Generation Program

2.12.3 Fibonacci Number Generation

The Fibonacci sequence is very important to certain algorithms in Computer Science and other fields. The following sample program generates a sequence of Fibonacci numbers for $n=1..40$.

```

// This program generates the fibonacci
// sequence for n=1..40.
//
// The fibonacci sequence is defined recursively
// for positive integers as follows:
//
// fib(1) = 1;
// fib(2) = 1;
// fib( n ) = fib( n-1 ) + fib( n-2 ).
//
// This program provides an iterative solution.

program fib;
#include( "stdlib.hhf" );

static

    FibCntr:    int32;
    CurFib:     int32;
    LastFib:    int32;
    TwoFibsAgo: int32;

begin fib;

    // Some simple initialization:

    mov( 1, LastFib );
    mov( 1, TwoFibsAgo );

    // Print fib(1) and fib(2) as a special case:

    stdout.put
    (
        "fib( 1) =          1", nl
        "fib( 2) =          1", nl
    );

    // Use a loop to compute the remaining fib values:

    mov( 3, FibCntr );
    while( FibCntr <= 40 ) do

        // Get the last two computed fibonacci values
        // and add them together:

        mov( LastFib, ebx );

```

```
mov( TwoFibsAgo, eax );
add( ebx, eax );

// Save the result and print it:

mov( eax, CurFib );
stdout.put( "fib(", FibCntr:2, ") =", CurFib:10, nl );

// Recycle current LastFib (in ebx) as TwoFibsAgo,
// and recycle CurFib as LastFib.

mov( eax, LastFib );
mov( ebx, TwoFibsAgo );

// Bump up our loop counter:

add( 1, FibCntr );

endwhile;

end fib;
```

Program 2.10 Fibonacci Sequence Generator

Data Representation

Chapter Three

A major stumbling block many beginners encounter when attempting to learn assembly language is the common use of the binary and hexadecimal numbering systems. Many programmers think that hexadecimal (or hex¹) numbers represent absolute proof that God never intended anyone to work in assembly language. While it is true that hexadecimal numbers are a little different from what you may be used to, their advantages outweigh their disadvantages by a large margin. Nevertheless, understanding these numbering systems is important because their use simplifies other complex topics including boolean algebra and logic design, signed numeric representation, character codes, and packed data.

3.1 Chapter Overview

This chapter discusses several important concepts including the binary and hexadecimal numbering systems, binary data organization (bits, nibbles, bytes, words, and double words), signed and unsigned numbering systems, arithmetic, logical, shift, and rotate operations on binary values, bit fields and packed data. This is basic material and the remainder of this text depends upon your understanding of these concepts. If you are already familiar with these terms from other courses or study, you should at least skim this material before proceeding to the next chapter. If you are unfamiliar with this material, or only vaguely familiar with it, you should study it carefully before proceeding. *All of the material in this chapter is important!* Do not skip over any material. In addition to the basic material, this chapter also introduces some new HLA statements and HLA Standard Library routines.

3.2 Numbering Systems

Most modern computer systems do not represent numeric values using the decimal system. Instead, they typically use a binary or two's complement numbering system. To understand the limitations of computer arithmetic, you must understand how computers represent numbers.

3.2.1 A Review of the Decimal System

You've been using the decimal (base 10) numbering system for so long that you probably take it for granted. When you see a number like "123", you don't think about the value 123; rather, you generate a mental image of how many items this value represents. In reality, however, the number 123 represents:

$$1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$

or

$$100 + 20 + 3$$

In the positional numbering system, each digit appearing to the left of the decimal point represents a value between zero and nine times an increasing power of ten. Digits appearing to the right of the decimal point represent a value between zero and nine times an increasing negative power of ten. For example, the value 123.456 means:

$$1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2} + 6 \cdot 10^{-3}$$

or

$$100 + 20 + 3 + 0.4 + 0.05 + 0.006$$

1. Hexadecimal is often abbreviated as *hex* even though, technically speaking, hex means base six, not base sixteen.

3.2.2 The Binary Numbering System

Most modern computer systems (including PCs) operate using binary logic. The computer represents values using two voltage levels (usually 0v and +2.4..5v). With two such levels we can represent exactly two different values. These could be any two different values, but they typically represent the values zero and one. These two values, coincidentally, correspond to the two digits used by the binary numbering system. Since there is a correspondence between the logic levels used by the 80x86 and the two digits used in the binary numbering system, it should come as no surprise that the PC employs the binary numbering system.

The binary numbering system works just like the decimal numbering system, with two exceptions: binary only allows the digits 0 and 1 (rather than 0-9), and binary uses powers of two rather than powers of ten. Therefore, it is very easy to convert a binary number to decimal. For each “1” in the binary string, add in 2^n where “n” is the zero-based position of the binary digit. For example, the binary value 11001010_2 represents:

$$\begin{aligned} 1*2^7 + 1*2^6 + 0*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 \\ = \\ 128 + 64 + 8 + 2 \\ = \\ 202_{10} \end{aligned}$$

To convert decimal to binary is slightly more difficult. You must find those powers of two which, when added together, produce the decimal result. One method is to work from a large power of two down to 2^0 . Consider the decimal value 1359:

- $2^{10}=1024$, $2^{11}=2048$. So 1024 is the largest power of two less than 1359. Subtract 1024 from 1359 and begin the binary value on the left with a “1” digit. Binary = ”1”, Decimal result is $1359 - 1024 = 335$.
- The next lower power of two ($2^9 = 512$) is greater than the result from above, so add a “0” to the end of the binary string. Binary = “10”, Decimal result is still 335.
- The next lower power of two is 256 (2^8). Subtract this from 335 and add a “1” digit to the end of the binary number. Binary = “101”, Decimal result is 79.
- $128 (2^7)$ is greater than 79, so tack a “0” to the end of the binary string. Binary = “1010”, Decimal result remains 79.
- The next lower power of two ($2^6 = 64$) is less than 79, so subtract 64 and append a “1” to the end of the binary string. Binary = “10101”, Decimal result is 15.
- 15 is less than the next power of two ($2^5 = 32$) so simply add a “0” to the end of the binary string. Binary = “101010”, Decimal result is still 15.
- $16 (2^4)$ is greater than the remainder so far, so append a “0” to the end of the binary string. Binary = “1010100”, Decimal result is 15.
- 2^3 (eight) is less than 15, so stick another “1” digit on the end of the binary string. Binary = “10101001”, Decimal result is 7.
- 2^2 is less than seven, so subtract four from seven and append another one to the binary string. Binary = “101010011”, decimal result is 3.
- 2^1 is less than three, so append a one to the end of the binary string and subtract two from the decimal value. Binary = “1010100111”, Decimal result is now 1.
- Finally, the decimal result is one, which is 2^0 , so add a final “1” to the end of the binary string. The final binary result is “10101001111”

If you actually have to convert a decimal number to binary by hand, the algorithm above probably isn’t the easiest to master. A simpler solution is the “even/odd – divide by two” algorithm. This algorithm uses the following steps:

- If the number is even, emit a zero. If the number is odd, emit a one.
- Divide the number by two and throw away any fractional component or remainder.

- If the quotient is zero, the algorithm is complete.
- If the quotient is not zero and is odd, insert a one before the current string; if the number is even, prefix your binary string with zero.
- Go back to step two above and repeat.

Fortunately, you'll rarely need to convert decimal numbers directly to binary strings, so neither of these algorithms is particularly important in real life.

Binary numbers, although they have little importance in high level languages, appear everywhere in assembly language programs (even if you don't convert between decimal and binary). So you should be somewhat comfortable with them.

3.2.3 Binary Formats

In the purest sense, every binary number contains an infinite number of digits (or *bits* which is short for binary digits). For example, we can represent the number five by:

101 00000101 0000000000101 ... 000000000000101

Any number of leading zero bits may precede the binary number without changing its value.

We will adopt the convention of ignoring any leading zeros if present in a value. For example, 101_2 represents the number five but since the 80x86 works with groups of eight bits, we'll find it much easier to zero extend all binary numbers to some multiple of four or eight bits. Therefore, following this convention, we'd represent the number five as 0101_2 or 00000101_2 .

In the United States, most people separate every three digits with a comma to make larger numbers easier to read. For example, 1,023,435,208 is much easier to read and comprehend than 1023435208. We'll adopt a similar convention in this text for binary numbers. We will separate each group of four binary bits with an underscore. For example, we will write the binary value 1010111110110010 as 1010_1111_1011_0010.

We often pack several values together into the same binary number. One form of the 80x86 MOV instruction uses the binary encoding 1011 0rrr dddd dddd to pack three items into 16 bits: a five-bit operation code (1_0110), a three-bit register field (rrr), and an eight-bit immediate value (dddd_dddd). For convenience, we'll assign a numeric value to each bit position. We'll number each bit as follows:

- 1) The rightmost bit in a binary number is bit position zero.
- 2) Each bit to the left is given the next successive bit number.

An eight-bit binary value uses bits zero through seven:

$X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0$

A 16-bit binary value uses bit positions zero through fifteen:

$X_{15} X_{14} X_{13} X_{12} X_{11} X_{10} X_9 X_8 X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0$

A 32-bit binary value uses bit positions zero through 31, etc.

Bit zero is usually referred to as the *low order* (L.O.) bit (some refer to this as the *least significant bit*). The left-most bit is typically called the *high order* (H.O.) bit (or the *most significant bit*). We'll refer to the intermediate bits by their respective bit numbers.

3.3 Data Organization

In pure mathematics a value may take an arbitrary number of bits. Computers, on the other hand, generally work with some specific number of bits. Common collections are single bits, groups of four bits (called *nibbles*), groups of eight bits (*bytes*), groups of 16 bits (*words*), groups of 32 bits (double words or *dwords*), groups of 64-bits (quad words or *qwords*), and more. The sizes are not arbitrary. There is a good reason for these particular values. This section will describe the bit groups commonly used on the Intel 80x86 chips.

3.3.1 Bits

The smallest “unit” of data on a binary computer is a single *bit*. Since a single bit is capable of representing only two different values (typically zero or one) you may get the impression that there are a very small number of items you can represent with a single bit. Not true! There are an infinite number of items you can represent with a single bit.

With a single bit, you can represent any two distinct items. Examples include zero or one, true or false, on or off, male or female, and right or wrong. However, you are *not* limited to representing binary data types (that is, those objects which have only two distinct values). You could use a single bit to represent the numbers 723 and 1,245. Or perhaps 6,254 and 5. You could also use a single bit to represent the colors red and blue. You could even represent two unrelated objects with a single bit. For example, you could represent the color red and the number 3,256 with a single bit. You can represent *any* two different values with a single bit. However, you can represent *only two* different values with a single bit.

To confuse things even more, different bits can represent different things. For example, one bit might be used to represent the values zero and one, while an adjacent bit might be used to represent the values true and false. How can you tell by looking at the bits? The answer, of course, is that you can't. But this illustrates the whole idea behind computer data structures: *data is what you define it to be*. If you use a bit to represent a boolean (true/false) value then that bit (by your definition) represents true or false. For the bit to have any real meaning, you must be consistent. That is, if you're using a bit to represent true or false at one point in your program, you shouldn't use the true/false value stored in that bit to represent red or blue later.

Since most items you'll be trying to model require more than two different values, single bit values aren't the most popular data type you'll use. However, since everything else consists of groups of bits, bits will play an important role in your programs. Of course, there are several data types that require two distinct values, so it would seem that bits are important by themselves. However, you will soon see that individual bits are difficult to manipulate, so we'll often use other data types to represent boolean values.

3.3.2 Nibbles

A *nibble* is a collection of four bits. It wouldn't be a particularly interesting data structure except for two items: BCD (*binary coded decimal*) numbers² and hexadecimal numbers. It takes four bits to represent a single BCD or hexadecimal digit. With a nibble, we can represent up to 16 distinct values since there are 16 unique combinations of a string of four bits:

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
```

2. Binary coded decimal is a numeric scheme used to represent decimal numbers using four bits for each decimal digit.

```

1001
1010
1011
1100
1101
1110
1111

```

In the case of hexadecimal numbers, the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are represented with four bits (see “The Hexadecimal Numbering System” on page 60). BCD uses ten different digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) and requires four bits (since you can only represent eight different values with three bits). In fact, any sixteen distinct values can be represented with a nibble, but hexadecimal and BCD digits are the primary items we can represent with a single nibble.

3.3.3 Bytes

Without question, the most important data structure used by the 80x86 microprocessor is the byte. A byte consists of eight bits and is the smallest addressable datum (data item) on the 80x86 microprocessor. Main memory and I/O addresses on the 80x86 are all byte addresses. This means that the smallest item that can be individually accessed by an 80x86 program is an eight-bit value. To access anything smaller requires that you read the byte containing the data and mask out the unwanted bits. The bits in a byte are normally numbered from zero to seven as shown in Figure 3.1.

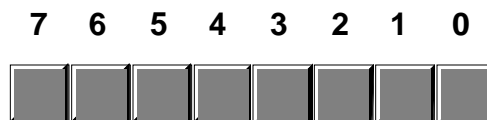


Figure 3.1 Bit Numbering

Bit 0 is the *low order bit* or *least significant bit*, bit 7 is the *high order bit* or *most significant bit* of the byte. We'll refer to all other bits by their number.

Note that a byte also contains exactly two nibbles (see Figure 3.2).

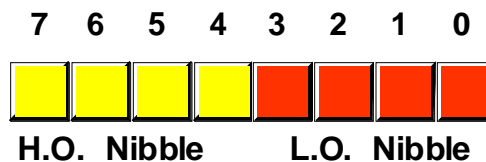


Figure 3.2 The Two Nibbles in a Byte

Bits 0..3 comprise the *low order nibble*, bits 4..7 form the *high order nibble*. Since a byte contains exactly two nibbles, byte values require two hexadecimal digits.

Since a byte contains eight bits, it can represent 2^8 , or 256, different values. Generally, we'll use a byte to represent numeric values in the range 0..255, signed numbers in the range -128..+127 (see “Signed and Unsigned Numbers” on page 69), ASCII/IBM character codes, and other special data types requiring no more than 256 different values. Many data types have fewer than 256 items so eight bits is usually sufficient.

Since the 80x86 is a byte addressable machine (see the next volume), it turns out to be more efficient to manipulate a whole byte than an individual bit or nibble. For this reason, most programmers use a whole byte to represent data types that require no more than 256 items, even if fewer than eight bits would suffice. For example, we'll often represent the boolean values true and false by 00000001_2 and 00000000_2 (respectively).

Probably the most important use for a byte is holding a character code. Characters typed at the keyboard, displayed on the screen, and printed on the printer all have numeric values. To allow it to communicate with the rest of the world, PCs use a variant of the ASCII character set (see “The ASCII Character Encoding” on page 97). There are 128 defined codes in the ASCII character set. PCs typically use the remaining 128 possible values for extended character codes including European characters, graphic symbols, Greek letters, and math symbols.

Because bytes are the smallest unit of storage in the 80x86 memory space, bytes also happen to be the smallest variable you can create in an HLA program. As you saw in the last chapter, you can declare an eight-bit signed integer variable using the *int8* data type. Since *int8* objects are signed, you can represent values in the range -128..+127 using an *int8* variable (see “Signed and Unsigned Numbers” on page 69 for a discussion of signed number formats). You should only store signed values into *int8* variables; if you want to create an arbitrary byte variable, you should use the *byte* data type, as follows:

```
static
    byteVar: byte;
```

The *byte* data type is a partially untyped data type. The only type information associated with *byte* objects is their size (one byte). You may store any one-byte object (small signed integers, small unsigned integers, characters, etc.) into a byte variable. It is up to you to keep track of the type of object you've put into a byte variable.

3.3.4 Words

A word is a group of 16 bits. We'll number the bits in a word starting from zero on up to fifteen. The bit numbering appears in Figure 3.3.

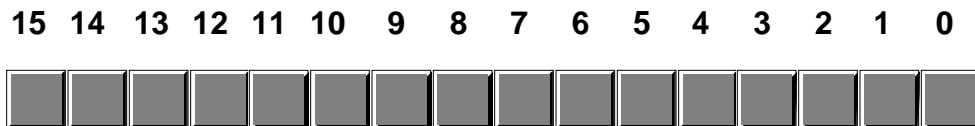


Figure 3.3 Bit Numbers in a Word

Like the byte, bit 0 is the low order bit. For words, bit 15 is the high order bit. When referencing the other bits in a word, use their bit position number.

Notice that a word contains exactly two bytes. Bits 0 through 7 form the low order byte, bits 8 through 15 form the high order byte (see Figure 3.4).

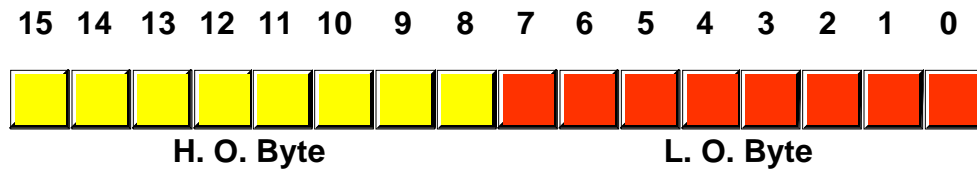


Figure 3.4 The Two Bytes in a Word

Naturally, a word may be further broken down into four nibbles as shown in Figure 3.5.

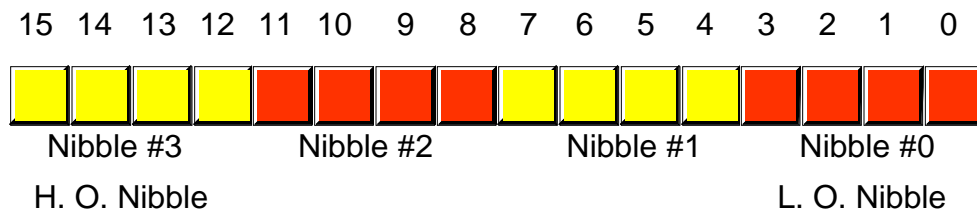


Figure 3.5 Nibbles in a Word

Nibble zero is the low order nibble in the word and nibble three is the high order nibble of the word. We'll simply refer to the other two nibbles as "nibble one" or "nibble two."

With 16 bits, you can represent 2^{16} (65,536) different values. These could be the values in the range 0..65,535 or, as is usually the case, -32,768..+32,767, or any other data type with no more than 65,536 values. The three major uses for words are signed integer values, unsigned integer values, and UNICODE characters.

Words can represent integer values in the range 0..65,535 or -32,768..32,767. Unsigned numeric values are represented by the binary value corresponding to the bits in the word. Signed numeric values use the two's complement form for numeric values (see "Signed and Unsigned Numbers" on page 69). As UNICODE characters, words can represent up to 65,536 different characters, allowing the use of non-Roman character sets in a computer program. UNICODE is an international standard, like ASCII, that allows computers to process non-Roman characters like Asian, Greek, and Russian characters.

Like bytes, you can also create word variables in an HLA program. Of course, in the last chapter you saw how to create sixteen-bit signed integer variables using the *int16* data type. To create an arbitrary word variable, just use the *word* data type, as follows:

```
static
    w: word;
```

3.3.5 Double Words

A double word is exactly what its name implies, a pair of words. Therefore, a double word quantity is 32 bits long as shown in Figure 3.6.



Figure 3.6 Bit Numbers in a Double Word

Naturally, this double word can be divided into a high order word and a low order word, four different bytes, or eight different nibbles (see Figure 3.7).

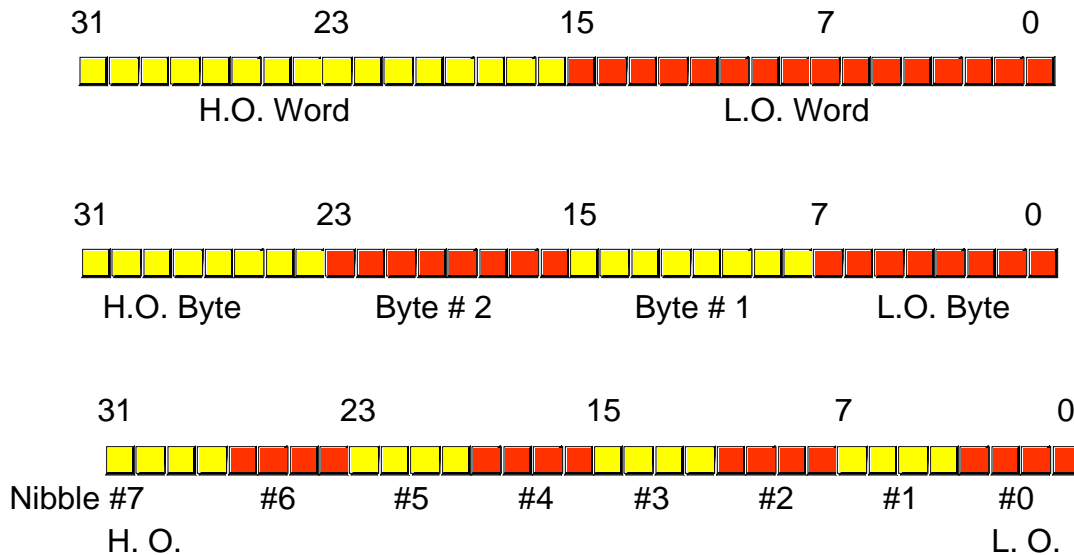


Figure 3.7 Nibbles, Bytes, and Words in a Double Word

Double words can represent all kinds of different things. A common item you will represent with a double word is a 32-bit integer value (which allows unsigned numbers in the range 0..4,294,967,295 or signed numbers in the range -2,147,483,648..2,147,483,647). 32-bit floating point values also fit into a double word. Another common use for dword objects is to store pointer variables.

In the previous chapter, you saw how to create 32-bit (dword) signed integer variables using the *int32* data type. You can also create an arbitrary double word variable using the *dword* data type as the following example demonstrates:

```
static
    d: dword;
```

3.4 The Hexadecimal Numbering System

A big problem with the binary system is verbosity. To represent the value 202₁₀ requires eight binary digits. The decimal version requires only three decimal digits and, thus, represents numbers much more compactly than does the binary numbering system. This fact was not lost on the engineers who designed binary computer systems. When dealing with large values, binary numbers quickly become too unwieldy.

Unfortunately, the computer thinks in binary, so most of the time it is convenient to use the binary numbering system. Although we can convert between decimal and binary, the conversion is not a trivial task. The hexadecimal (base 16) numbering system solves these problems. Hexadecimal numbers offer the two features we're looking for: they're very compact, and it's simple to convert them to binary and vice versa. Because of this, most computer systems engineers use the hexadecimal numbering system. Since the radix (base) of a hexadecimal number is 16, each hexadecimal digit to the left of the hexadecimal point represents some value times a successive power of 16. For example, the number 1234_{16} is equal to:

$$1 * 16^3 + 2 * 16^2 + 3 * 16^1 + 4 * 16^0$$

or

$$4096 + 512 + 48 + 4 = 4660_{10}.$$

Each hexadecimal digit can represent one of sixteen values between 0 and 15_{10} . Since there are only ten decimal digits, we need to invent six additional digits to represent the values in the range 10_{10} through 15_{10} . Rather than create new symbols for these digits, we'll use the letters A through F. The following are all examples of valid hexadecimal numbers:

1234_{16} $DEAD_{16}$ $BEEF_{16}$ $0AFB_{16}$ $FEED_{16}$ $DEAF_{16}$

Since we'll often need to enter hexadecimal numbers into the computer system, we'll need a different mechanism for representing hexadecimal numbers. After all, on most computer systems you cannot enter a subscript to denote the radix of the associated value. We'll adopt the following conventions:

- All hexadecimal values begin with a "\$" character, e.g., \$123A4.
- All binary values begin with a percent sign ("%").
- Decimal numbers do not have a prefix character.
- If the radix is clear from the context, this text may drop the leading "\$" or "%" character.

Examples of valid hexadecimal numbers:

\$1234 \$DEAD \$BEEF \$AFB \$FEED \$DEAF

As you can see, hexadecimal numbers are compact and easy to read. In addition, you can easily convert between hexadecimal and binary. Consider the following table:

Table 4: Binary/Hex Conversion

Binary	Hexadecimal
%0000	\$0
%0001	\$1
%0010	\$2
%0011	\$3
%0100	\$4
%0101	\$5
%0110	\$6
%0111	\$7
%1000	\$8
%1001	\$9

Table 4: Binary/Hex Conversion

Binary	Hexadecimal
% 1010	\$A
% 1011	\$B
% 1100	\$C
% 1101	\$D
% 1110	\$E
% 1111	\$F

This table provides all the information you'll ever need to convert any hexadecimal number into a binary number or vice versa.

To convert a hexadecimal number into a binary number, simply substitute the corresponding four bits for each hexadecimal digit in the number. For example, to convert \$ABCD into a binary value, simply convert each hexadecimal digit according to the table above:

0	A	B	C	D	Hexadecimal
0000	1010	1011	1100	1101	Binary

To convert a binary number into hexadecimal format is almost as easy. The first step is to pad the binary number with zeros to make sure that there is a multiple of four bits in the number. For example, given the binary number 1011001010, the first step would be to add two bits to the left of the number so that it contains 12 bits. The converted binary value is 001011001010. The next step is to separate the binary value into groups of four bits, e.g., 0010_1100_1010. Finally, look up these binary values in the table above and substitute the appropriate hexadecimal digits, i.e., \$2CA. Contrast this with the difficulty of conversion between decimal and binary or decimal and hexadecimal!

Since converting between hexadecimal and binary is an operation you will need to perform over and over again, you should take a few minutes and memorize the table above. Even if you have a calculator that will do the conversion for you, you'll find manual conversion to be a lot faster and more convenient when converting between binary and hex.

3.5 Arithmetic Operations on Binary and Hexadecimal Numbers

There are several operations we can perform on binary and hexadecimal numbers. For example, we can add, subtract, multiply, divide, and perform other arithmetic operations. Although you needn't become an expert at it, you should be able to, in a pinch, perform these operations manually using a piece of paper and a pencil. Having just said that you should be able to perform these operations manually, the correct way to perform such arithmetic operations is to have a calculator that does them for you. There are several such calculators on the market; the following table lists some of the manufacturers who produce such devices:

Some manufacturers of Hexadecimal Calculators (circa 2002):

- Casio
- Hewlett-Packard
- Sharp
- Texas Instruments

This list is by no means exhaustive. Other calculator manufacturers probably produce these devices as well. The Hewlett-Packard devices are arguably the best of the bunch. However, they are more expensive

than the others. Sharp and Casio produce units which sell for well under \$50. If you plan on doing any assembly language programming at all, owning one of these calculators is essential.

To understand why you should spend the money on a calculator, consider the following arithmetic problem:

```

  $9
+ $1
----

```

You're probably tempted to write in the answer "\$10" as the solution to this problem. But that is not correct! The correct answer is ten, which is "\$A", not sixteen which is "\$10". A similar problem exists with the arithmetic problem:

```

 $10
- $1
----

```

You're probably tempted to answer "\$9" even though the true answer is "\$F". Remember, this problem is asking "what is the difference between sixteen and one?" The answer, of course, is fifteen which is "\$F".

Even if the two problems above don't bother you, in a stressful situation your brain will switch back into decimal mode while you're thinking about something else and you'll produce the incorrect result. Moral of the story – if you must do an arithmetic computation using hexadecimal numbers by hand, take your time and be careful about it. Either that, or convert the numbers to decimal, perform the operation in decimal, and convert them back to hexadecimal.

3.6 A Note About Numbers vs. Representation

Many people confuse numbers and their representation. A common question beginning assembly language students have is "I've got a binary number in the EAX register, how do I convert that to a hexadecimal number in the EAX register?" The answer is "you don't." Although a strong argument could be made that numbers in memory or in registers are represented in binary, it's best to view values in memory or in a register as *abstract numeric quantities*. Strings of symbols like 128, \$80, or %1000_0000 are not different numbers; they are simply different representations for the same abstract quantity that we often refer to as "one hundred twenty-eight." Inside the computer, a number is a number regardless of representation; the only time representation matters is when you input or output the value in a human readable form.

Human readable forms of numeric quantities are always strings of characters. To print the value 128 in human readable form, you must convert the numeric value 128 to the three-character sequence '1' followed by '2' followed by '8'. This would provide the decimal representation of the numeric quantity. If you prefer, you could convert the numeric value 128 to the three character sequence "\$80". It's the same number, but we've converted it to a different sequence of characters because (presumably) we wanted to view the number using hexadecimal representation rather than decimal. Likewise, if we want to see the number in binary, then we must convert this numeric value to a string containing a one followed by seven zeros.

By default, HLA displays all byte, word, and dword variables using the hexadecimal numbering system when you use the *stdout.put* routine. Likewise, HLA's *stdout.put* routine will display all register values in hex. Consider the following program that converts values input as decimal numbers to their hexadecimal equivalents:

```

program ConvertToHex;
#include( "stdlib.hhf" );
static
    value: int32;

begin ConvertToHex;

```

```

stdout.put( "Input a decimal value:" );
stdin.get( value );
mov( value, eax );
stdout.put( "The value ", value, " converted to hex is $", eax, nl );

end ConvertToHex;

```

Program 3.11 Decimal to Hexadecimal Conversion Program

In a similar fashion, the default input base is also hexadecimal for registers and byte, word, or dword variables. The following program is the converse of the one above- it inputs a hexadecimal value and outputs it as decimal:

```

program ConvertToDecimal;
#include( "stdlib.hhf" );
static
    value: int32;

begin ConvertToDecimal;

    stdout.put( "Input a hexadecimal value: " );
    stdin.get( ebx ); mov( ebx, value );
    stdout.put( "The value $", ebx, " converted to decimal is ", value, nl );

end ConvertToDecimal;

```

Program 3.12 Hexadecimal to Decimal Conversion Program

Just because the HLA *stdout.put* routine chooses decimal as the default output base for *int8*, *int16*, and *int32* variables doesn't mean that these variables hold "decimal" numbers. Remember, memory and registers hold numeric values, not hexadecimal or decimal values. The *stdout.put* routine converts these numeric values to strings and prints the resulting strings. The choice of hexadecimal vs. decimal output was a design choice in the HLA language, nothing more. You could very easily modify HLA so that it outputs registers and *byte*, *word*, or *dword* variables as decimal values rather than as hexadecimal. If you need to print the value of a register or *byte*, *word*, or *dword* variable as a decimal value, simply call one of the *putiX* routines to do this. The *stdout.puti8* routine will output its parameter as an eight-bit signed integer. Any eight-bit parameter will work. So you could pass an eight-bit register, an *int8* variable, or a *byte* variable as the parameter to *stdout.puti8* and the result will always be decimal. The *stdout.puti16* and *stdout.puti32* provide the same capabilities for 16-bit and 32-bit objects. The following program demonstrates the decimal conversion program (Program 3.12 above) using only the EAX register (i.e., it does not use the variable *iValue*):

```

program ConvertToDecimal2;
#include( "stdlib.hhf" );
begin ConvertToDecimal2;

    stdout.put( "Input a hexadecimal value: " );
    stdin.get( ebx );
    stdout.put( "The value $", ebx, " converted to decimal is " );
    stdout.puti32( ebx );
    stdout.newln();

```

```
end ConvertToDecimal2;
```

Program 3.13 Variable-less Hexadecimal to Decimal Converter

Note that HLA's *stdin.get* routine uses the same default base for input as *stdout.put* uses for output. That is, if you attempt to read an *int8*, *int16*, or *int32* variable, the default input base is decimal. If you attempt to read a register or *byte*, *word*, or *dword* variable, the default input base is hexadecimal. If you want to change the default input base to decimal when reading a register or a *byte*, *word*, or *dword* variable, then you can use *stdin.geti8*, *stdin.geti16*, or *stdin.geti32*.

If you want to go in the opposite direction, that is you want to input or output an *int8*, *int16*, or *int32* variable as a hexadecimal value, you can call the *stdout.putb*, *stdout.putw*, *stdout.putd*, *stdin.getb*, *stdin.getw*, or *stdin.getd* routines. The *stdout.putb*, *stdout.putw*, and *stdout.putd* routines write eight-bit, 16-bit, or 32-bit objects as hexadecimal values. The *stdin.getb*, *stdin.getw*, and *stdin.getd* routines read eight-bit, 16-bit, and 32-bit values respectively; they return their results in the AL, AX, or EAX registers. The following program demonstrates the use of a few of these routines:

```
program HexIO;

#include( "stdlib.hhf" );

static
    i32: int32;

begin HexIO;

    stdout.put( "Enter a hexadecimal value: " );
    stdin.getd();
    mov( eax, i32 );
    stdout.put( "The value you entered was $" );
    stdout.putd( i32 );
    stdout.newln();

end HexIO;
```

Program 3.14 Demonstration of *stdin.getd* and *stdout.putd*

3.7 Logical Operations on Bits

There are four main logical operations we'll need to perform on hexadecimal and binary numbers: AND, OR, XOR (exclusive-or), and NOT. Unlike the arithmetic operations, a hexadecimal calculator isn't necessary to perform these operations. It is often easier to do them by hand than to use an electronic device

to compute them. The logical AND operation is a dyadic³ operation (meaning it accepts exactly two operands). These operands are single binary (base 2) bits. The AND operation is:

$$0 \text{ and } 0 = 0$$

$$0 \text{ and } 1 = 0$$

$$1 \text{ and } 0 = 0$$

$$1 \text{ and } 1 = 1$$

A compact way to represent the logical AND operation is with a truth table. A truth table takes the following form:

Table 5: AND Truth Table

AND	0	1
0	0	0
1	0	1

This is just like the multiplication tables you encountered in elementary school. The values in the left column correspond to the leftmost operand of the AND operation. The values in the top row correspond to the rightmost operand of the AND operation. The value located at the intersection of the row and column (for a particular pair of input values) is the result of logically ANDing those two values together.

In English, the logical AND operation is, “If the first operand is one and the second operand is one, the result is one; otherwise the result is zero.” We could also state this as “If either or both operands are zero, the result is zero.”

One important fact to note about the logical AND operation is that you can use it to force a zero result. If one of the operands is zero, the result is always zero regardless of the other operand. In the truth table above, for example, the row labelled with a zero input contains only zeros and the column labelled with a zero only contains zero results. Conversely, if one operand contains a one, the result is exactly the value of the second operand. These features of the AND operation are very important, particularly when we want to force individual bits in a bit string to zero. We will investigate these uses of the logical AND operation in the next section.

The logical OR operation is also a dyadic operation. Its definition is:

$$0 \text{ or } 0 = 0$$

$$0 \text{ or } 1 = 1$$

$$1 \text{ or } 0 = 1$$

$$1 \text{ or } 1 = 1$$

3. Many texts call this a binary operation. The term dyadic means the same thing and avoids the confusion with the binary numbering system.

The truth table for the OR operation takes the following form:

Table 6: OR Truth Table

OR	0	1
0	0	1
1	1	1

Colloquially, the logical OR operation is, “If the first operand or the second operand (or both) is one, the result is one; otherwise the result is zero.” This is also known as the *inclusive-OR* operation.

If one of the operands to the logical-OR operation is a one, the result is always one regardless of the second operand’s value. If one operand is zero, the result is always the value of the second operand. Like the logical AND operation, this is an important side-effect of the logical-OR operation that will prove quite useful when working with bit strings since it lets you force individual bits to one.

Note that there is a difference between this form of the inclusive logical OR operation and the standard English meaning. Consider the phrase “I am going to the store *or* I am going to the park.” Such a statement implies that the speaker is going to the store or to the park but not to both places. Therefore, the English version of logical OR is slightly different than the inclusive-OR operation; indeed, it is closer to the *exclusive-OR* operation.

The logical XOR (exclusive-or) operation is also a dyadic operation. It is defined as follows:

$$0 \text{ xor } 0 = 0$$

$$0 \text{ xor } 1 = 1$$

$$1 \text{ xor } 0 = 1$$

$$1 \text{ xor } 1 = 0$$

The truth table for the XOR operation takes the following form:

Table 7: XOR Truth Table

XOR	0	1
0	0	1
1	1	0

In English, the logical XOR operation is, “If the first operand or the second operand, but not both, is one, the result is one; otherwise the result is zero.” Note that the exclusive-or operation is closer to the English meaning of the word “or” than is the logical OR operation.

If one of the operands to the logical exclusive-OR operation is a one, the result is always the *inverse* of the other operand; that is, if one operand is one, the result is zero if the other operand is one and the result is one if the other operand is zero. If the first operand contains a zero, then the result is exactly the value of the second operand. This feature lets you selectively invert bits in a bit string.

The logical NOT operation is a monadic operation (meaning it accepts only one operand). It is:

$$\text{NOT } 0 = 1$$

$$\text{NOT } 1 = 0$$

The truth table for the NOT operation takes the following form:

Table 8: NOT Truth Table

NOT	0	1
	1	0

3.8 Logical Operations on Binary Numbers and Bit Strings

As described in the previous section, the logical functions work only with single bit operands. Since the 80x86 uses groups of eight, sixteen, or thirty-two bits, we need to extend the definition of these functions to deal with more than two bits. Logical functions on the 80x86 operate on a *bit-by-bit* (or *bitwise*) basis. Given two values, these functions operate on bit zero producing bit zero of the result. They operate on bit one of the input values producing bit one of the result, etc. For example, if you want to compute the logical AND of the following two eight-bit numbers, you would perform the logical AND operation on each column independently of the others:

```

%1011_0101
%1110_1110
-----
%1010_0100

```

This bit-by-bit form of execution can be easily applied to the other logical operations as well.

Since we've defined logical operations in terms of binary values, you'll find it much easier to perform logical operations on binary values than on values in other bases. Therefore, if you want to perform a logical operation on two hexadecimal numbers, you should convert them to binary first. This applies to most of the basic logical operations on binary numbers (e.g., AND, OR, XOR, etc.).

The ability to force bits to zero or one using the logical AND/OR operations and the ability to invert bits using the logical XOR operation is very important when working with strings of bits (e.g., binary numbers). These operations let you selectively manipulate certain bits within some value while leaving other bits unaffected. For example, if you have an eight-bit binary value *X* and you want to guarantee that bits four through seven contain zeros, you could logically AND the value *X* with the binary value %0000_1111. This bitwise logical AND operation would force the H.O. four bits to zero and pass the L.O. four bits of *X* through unchanged. Likewise, you could force the L.O. bit of *X* to one and invert bit number two of *X* by logically ORing *X* with %0000_0001 and logically exclusive-ORing *X* with %0000_0100, respectively. Using the logical AND, OR, and XOR operations to manipulate bit strings in this fashion is known as *masking* bit strings. We use the term *masking* because we can use certain values (one for AND, zero for OR/XOR) to 'mask out' or 'mask in' certain bits from the operation when forcing bits to zero, one, or their inverse.

The 80x86 CPUs support four instructions that apply these bitwise logical operations to their operands. The instructions are AND, OR, XOR, and NOT. The AND, OR, and XOR instructions use the same syntax as the ADD and SUB instructions, that is,

```

and( source, dest );
or( source, dest );
xor( source, dest );

```

These operands have the same limitations as the ADD operands. Specifically, the *source* operand has to be a constant, memory, or register operand and the *dest* operand must be a memory or register operand. Also, the operands must be the same size and they cannot both be memory operands. These instructions compute the obvious bitwise logical operation via the equation:

$$dest = dest \text{ operator } source$$

The 80x86 logical NOT instruction, since it has only a single operand, uses a slightly different syntax. This instruction takes the following form:

```
not( dest );
```

Note that this instruction has a single operand. It computes the following result:

$$dest = \text{NOT}(dest)$$

The *dest* operand (for *not*) must be a register or memory operand. This instruction inverts all the bits in the specified destination operand.

The following program inputs two hexadecimal values from the user and calculates their logical AND, OR, XOR, and NOT:

```

program LogicalOp;
#include( "stdlib.hhf" );
begin LogicalOp;

    stdout.put( "Input left operand: " );
    stdin.get( eax );
    stdout.put( "Input right operand: " );
    stdin.get( ebx );

    mov( eax, ecx );
    and( ebx, ecx );
    stdout.put( "$", eax, " AND $", ebx, " = $", ecx, nl );

    mov( eax, ecx );
    or( ebx, ecx );
    stdout.put( "$", eax, " OR $", ebx, " = $", ecx, nl );

    mov( eax, ecx );
    xor( ebx, ecx );
    stdout.put( "$", eax, " XOR $", ebx, " = $", ecx, nl );

    mov( eax, ecx );
    not( ecx );
    stdout.put( "NOT $", eax, " = $", ecx, nl );

    mov( ebx, ecx );
    not( ecx );
    stdout.put( "NOT $", ebx, " = $", ecx, nl );

end LogicalOp;

```

Program 3.15 AND, OR, XOR, and NOT Example

3.9 Signed and Unsigned Numbers

So far, we've treated binary numbers as unsigned values. The binary number ...00000 represents zero, ...00001 represents one, ...00010 represents two, and so on toward infinity. What about negative numbers? Signed values have been tossed around in previous sections and we've mentioned the two's complement

numbering system, but we haven't discussed how to represent negative numbers using the binary numbering system. That is what this section is all about!

To represent signed numbers using the binary numbering system we have to place a restriction on our numbers: they must have a finite and fixed number of bits. For our purposes, we're going to severely limit the number of bits to eight, 16, 32, or some other small number of bits.

With a fixed number of bits we can only represent a certain number of objects. For example, with eight bits we can only represent 256 different values. Negative values are objects in their own right, just like positive numbers; therefore, we'll have to use some of the 256 different eight-bit values to represent negative numbers. In other words, we've got to use up some of the (unsigned) positive numbers to represent negative numbers. To make things fair, we'll assign half of the possible combinations to the negative values and half to the positive values and zero. So we can represent the negative values $-128..-1$ and the non-negative values $0..127$ with a single eight bit byte. With a 16-bit word we can represent values in the range $-32,768..+32,767$. With a 32-bit double word we can represent values in the range $-2,147,483,648..+2,147,483,647$. In general, with n bits we can represent the signed values in the range -2^{n-1} to $+2^{n-1}-1$.

Okay, so we can represent negative values. Exactly how do we do it? Well, there are many ways, but the 80x86 microprocessor uses the two's complement notation. In the two's complement system, the H.O. bit of a number is a *sign bit*. If the H.O. bit is zero, the number is positive; if the H.O. bit is one, the number is negative. Examples:

For 16-bit numbers:

\$8000 is negative because the H.O. bit is one.

\$100 is positive because the H.O. bit is zero.

\$7FFF is positive.

\$FFFF is negative.

\$FFF is positive.

If the H.O. bit is zero, then the number is positive and is stored as a standard binary value. If the H.O. bit is one, then the number is negative and is stored in the two's complement form. To convert a positive number to its negative, two's complement form, you use the following algorithm:

- 1) Invert all the bits in the number, i.e., apply the logical NOT function.
- 2) Add one to the inverted result.

For example, to compute the eight-bit equivalent of -5:

```
%0000_0101    Five (in binary).
%1111_1010    Invert all the bits.
%1111_1011    Add one to obtain result.
```

If we take minus five and perform the two's complement operation on it, we get our original value, %0000_0101, back again, just as we expect:

```
%1111_1011    Two's complement for -5.
%0000_0100    Invert all the bits.
%0000_0101    Add one to obtain result (+5).
```

The following examples provide some positive and negative 16-bit signed values:

\$7FFF: +32767, the largest 16-bit positive number.

\$8000: -32768, the smallest 16-bit negative number.

\$4000: +16,384.

To convert the numbers above to their negative counterpart (i.e., to negate them), do the following:

```

$7FFF:    %0111_1111_1111_1111    +32,767
          %1000_0000_0000_0000    Invert all the bits (8000h)
          %1000_0000_0000_0001    Add one (8001h or -32,767)

4000h:    %0100_0000_0000_0000    16,384
          %1011_1111_1111_1111    Invert all the bits ($BFFF)
          %1100_0000_0000_0000    Add one ($C000 or -16,384)

$8000:    %1000_0000_0000_0000    -32,768
          %0111_1111_1111_1111    Invert all the bits ($7FFF)
          %1000_0000_0000_0000    Add one (8000h or -32768)

```

\$8000 inverted becomes \$7FFF. After adding one we obtain \$8000! Wait, what's going on here? $-(-32,768)$ is $-32,768$? Of course not. But the value $+32,768$ cannot be represented with a 16-bit signed number, so we cannot negate the smallest negative value.

Why bother with such a miserable numbering system? Why not use the H.O. bit as a sign flag, storing the positive equivalent of the number in the remaining bits? The answer lies in the hardware. As it turns out, negating values is the only tedious job. With the two's complement system, most other operations are as easy as the binary system. For example, suppose you were to perform the addition $5+(-5)$. The result is zero. Consider what happens when we add these two values in the two's complement system:

```

% 0000_0101
% 1111_1011
-----
%1_0000_0000

```

We end up with a carry into the ninth bit and all other bits are zero. As it turns out, if we ignore the carry out of the H.O. bit, adding two signed values always produces the correct result when using the two's complement numbering system. This means we can use the same hardware for signed and unsigned addition and subtraction. This wouldn't be the case with some other numbering systems.

Except for the questions associated with this chapter, you will not need to perform the two's complement operation by hand. The 80x86 microprocessor provides an instruction, NEG (negate), that performs this operation for you. Furthermore, all the hexadecimal calculators will perform this operation by pressing the change sign key (+/- or CHS). Nevertheless, performing a two's complement by hand is easy, and you should know how to do it.

Once again, you should note that the data represented by a set of binary bits depends entirely on the context. The eight bit binary value `%1100_0000` could represent an IBM/ASCII character, it could represent the unsigned decimal value 192, or it could represent the signed decimal value -64. As the programmer, it is your responsibility to use this data consistently.

The 80x86 negate instruction, NEG, uses the same syntax as the NOT instruction; that is, it takes a single destination operand:

```
neg( dest );
```

This instruction computes "dest = -dest;" and the operand has the same limitations as for NOT (it must be a memory location or a register). NEG operates on byte, word, and dword-sized objects. Of course, since this is a signed integer operation, it only makes sense to operate on signed integer values. The following program demonstrates the two's complement operation by using the NEG instruction:

```

program twosComplement;
#include( "stdlib.hhf" );

static

```

```

PosValue:  int8;
NegValue:  int8;

begin twosComplement;

  stdout.put( "Enter an integer between 0 and 127: " );
  stdin.get( PosValue );

  stdout.put( nl, "Value in hexadecimal: $" );
  stdout.putb( PosValue );

  mov( PosValue, al );
  not( al );
  stdout.put( nl, "Invert all the bits: $", al, nl );
  add( 1, al );
  stdout.put( "Add one: $", al, nl );
  mov( al, NegValue );
  stdout.put( "Result in decimal: ", NegValue, nl );

  stdout.put
  (
    nl,
    "Now do the same thing with the NEG instruction: ",
    nl
  );
  mov( PosValue, al );
  neg( al );
  mov( al, NegValue );
  stdout.put( "Hex result = $", al, nl );
  stdout.put( "Decimal result = ", NegValue, nl );

end twosComplement;

```

Program 3.16 The Two's Complement Operation

As you saw in the previous chapters, you use the *int8*, *int16*, and *int32* data types to reserve storage for signed integer variables. Those chapters also introduced routines like *stdout.puti8* and *stdin.geti32* that read and write signed integer values. Since this section has made it abundantly clear that you must differentiate signed and unsigned calculations in your programs, you should probably be asking yourself about now “how do I declare and use unsigned integer variables?”

The first part of the question, “how do you declare unsigned integer variables,” is the easiest to answer. You simply use the *uns8*, *uns16*, and *uns32* data types when declaring the variables, for example:

```

static
  u8:    uns8;
  u16:   uns16;
  u32:   uns32;

```

As for using these unsigned variables, the HLA Standard Library provides a complementary set of input/output routines for reading and displaying unsigned variables. As you can probably guess, these routines include *stdout.putu8*, *stdout.putu16*, *stdout.putu32*, *stdout.putu8Size*, *stdout.putu16Size*, *stdout.putu32Size*, *stdin.getu8*, *stdin.getu16*, and *stdin.getu32*. You use these routines just as you would use their signed integer counterparts except, of course, you get to use the full range of the unsigned values with these routines. The following source code demonstrates unsigned I/O as well as demonstrating what can happen if you mix signed and unsigned operations in the same calculation:

```

program UnsExample;
#include( "stdlib.hhf" );

static
    UnsValue:    uns16;

begin UnsExample;

    stdout.put( "Enter an integer between 32,768 and 65,535: " );
    stdin.getu16();
    mov( ax, UnsValue );

    stdout.put
    (
        "You entered ",
        UnsValue,
        ". If you treat this as a signed integer, it is "
    );
    stdout.puti16( UnsValue );
    stdout.newln();

end UnsExample;

```

Program 3.17 Unsigned I/O

3.10 Sign Extension, Zero Extension, Contraction, and Saturation

Since two's complement format integers have a fixed length, a small problem develops. What happens if you need to convert an eight bit two's complement value to 16 bits? This problem, and its converse (converting a 16 bit value to eight bits) can be accomplished via *sign extension* and *contraction* operations. Likewise, the 80x86 works with fixed length values, even when processing unsigned binary numbers. *Zero extension* lets you convert small unsigned values to larger unsigned values.

Consider the value "-64". The eight bit two's complement value for this number is \$C0. The 16-bit equivalent of this number is \$FFC0. Now consider the value "+64". The eight and 16 bit versions of this value are \$40 and \$0040, respectively. The difference between the eight and 16 bit numbers can be described by the rule: "If the number is negative, the H.O. byte of the 16 bit number contains \$FF; if the number is positive, the H.O. byte of the 16 bit quantity is zero."

To sign extend a value from some number of bits to a greater number of bits is easy, just copy the sign bit into all the additional bits in the new format. For example, to sign extend an eight bit number to a 16 bit number, simply copy bit seven of the eight bit number into bits 8..15 of the 16 bit number. To sign extend a 16 bit number to a double word, simply copy bit 15 into bits 16..31 of the double word.

You must use sign extension when manipulating signed values of varying lengths. Often you'll need to add a byte quantity to a word quantity. You must sign extend the byte quantity to a word before the operation takes place. Other operations (multiplication and division, in particular) may require a sign extension to 32-bits. You must not sign extend unsigned values.

```

Sign Extension:
Eight Bits  Sixteen Bits  Thirty-two Bits

$80         $FF80         $FFFF_FF80
$28         $0028         $0000_0028
$9A         $FF9A         $FFFF_FF9A

```

\$7F	\$007F	\$0000_007F
---	\$1020	\$0000_1020
---	\$8086	\$FFFF_8086

To extend an unsigned byte you must zero extend the value. Zero extension is very easy – just store a zero into the H.O. byte(s) of the larger operand. For example, to zero extend the value \$82 to 16-bits you simply add a zero to the H.O. byte yielding \$0082.

Zero Extension:

Eight Bits	Sixteen Bits	Thirty-two Bits
\$80	\$0080	\$0000_0080
\$28	\$0028	\$0000_0028
\$9A	\$009A	\$0000_009A
\$7F	\$007F	\$0000_007F
---	\$1020	\$0000_1020
---	\$8086	\$0000_8086

The 80x86 provides several instructions that will let you sign or zero extend a smaller number to a larger number. The first group of instructions we will look at will sign extend the AL, AX, or EAX register. These instructions are

- `cbw();` // Converts the byte in AL to a word in AX via sign extension.
- `cwd();` // Converts the word in AX to a double word in DX:AX
- `cdq();` // Converts the double word in EAX to the quad word in EDX:EAX
- `cwde();` // Converts the word in AX to a doubleword in EAX.

Note that the CWD (convert word to doubleword) instruction does not sign extend the word in AX to the doubleword in EAX. Instead, it stores the H.O. doubleword of the sign extension into the DX register (the notation “DX:AX” tells you that you have a double word value with DX containing the upper 16 bits and AX containing the lower 16 bits of the value). If you want the sign extension of AX to go into EAX, you should use the CWDE (convert word to doubleword, extended) instruction.

The four instructions above are unusual in the sense that these are the first instructions you’ve seen that do not have any operands. These instructions’ operands are *implied* by the instructions themselves.

Within a few chapters you will discover just how important these instructions are, and why the CWD and CDQ instructions involve the DX and EDX registers. However, for simple sign extension operations, these instructions have a few major drawbacks - you do not get to specify the source and destination operands and the operands must be registers.

For general sign extension operations, the 80x86 provides an extension of the MOV instruction, MOVZX (move with sign extension), that copies data and sign extends the data while copying it. The MOVZX instruction’s syntax is very similar to the MOV instruction:

```
movsx( source, dest );
```

The big difference in syntax between this instruction and the MOV instruction is the fact that the destination operand must be larger than the source operand. That is, if the source operand is a byte, the destination operand must be a word or a double word. Likewise, if the source operand is a word, the destination operand must be a double word. Another difference is that the destination operand has to be a register; the source operand, however, can be a memory location⁴.

To zero extend a value, you can use the MOVZX instruction. It has the same syntax and restrictions as the MOVZX instruction. Zero extending certain eight-bit registers (AL, BL, CL, and DL) into their corresponding 16-bit registers is easily accomplished without using MOVZX by loading the complementary H.O.

4. This doesn’t turn out to be much of a limitation because sign extension almost always precedes an arithmetic operation which must take place in a register.

register (AH, BH, CH, or DH) with zero. Obviously, to zero extend AX into DX:AX or EAX into EDX:EAX, all you need to do is load DX or EDX with zero⁵.

The following sample program demonstrates the use of the sign extension instructions:

```

program signExtension;
#include( "stdlib.hhf" );

static
    i8:    int8;
    i16:   int16;
    i32:   int32;

begin signExtension;

    stdout.put( "Enter a small negative number: " );
    stdin.get( i8 );

    stdout.put( nl, "Sign extension using CBW and CWDE:", nl, nl );

    mov( i8, al );
    stdout.put( "You entered ", i8, " ($", al, ")", nl );

    cbw();
    mov( ax, i16 );
    stdout.put( "16-bit sign extension: ", i16, " ($", ax, ")", nl );

    cwde();
    mov( eax, i32 );
    stdout.put( "32-bit sign extension: ", i32, " ($", eax, ")", nl );

    stdout.put( nl, "Sign extension using MOVSX:", nl, nl );

    movsx( i8, ax );
    mov( ax, i16 );
    stdout.put( "16-bit sign extension: ", i16, " ($", ax, ")", nl );

    movsx( i8, eax );
    mov( eax, i32 );
    stdout.put( "32-bit sign extension: ", i32, " ($", eax, ")", nl );

end signExtension;

```

Program 3.18 Sign Extension Instructions

Sign contraction, converting a value with some number of bits to the identical value with a fewer number of bits, is a little more troublesome. Sign extension never fails. Given an m -bit signed value you can always convert it to an n -bit number (where $n > m$) using sign extension. Unfortunately, given an n -bit number, you cannot always convert it to an m -bit number if $m < n$. For example, consider the value -448. As a 16-bit hexadecimal number, its representation is \$FE40. Unfortunately, the magnitude of this number is too large to fit into an eight bit value, so you cannot sign contract it to eight bits. This is an example of an overflow condition that occurs upon conversion.

5. Zero extending into DX:AX or EDX:EAX is just as necessary as the CWD and CDQ instructions, as you will eventually see.

To properly sign contract one value to another, you must look at the H.O. byte(s) that you want to discard. The H.O. bytes you wish to remove must all contain either zero or \$FF. If you encounter any other values, you cannot contract it without overflow. Finally, the H.O. bit of your resulting value must match *every* bit you've removed from the number. Examples (16 bits to eight bits):

```
$FF80 can be sign contracted to $80.
$0040 can be sign contracted to $40.
$FE40 cannot be sign contracted to 8 bits.
$0100 cannot be sign contracted to 8 bits.
```

Another way to reduce the size of an integer is through saturation. Saturation is useful in situations where you must convert a larger object to a smaller object and you're willing to live with possible loss of precision. To convert a value via saturation you simply copy the larger value to the smaller value if it is not outside the range of the smaller object. If the larger value is outside the range of the smaller value, then you *clip* the value by setting it to the largest (or smallest) value within the range of the smaller object.

For example, when converting a 16-bit signed integer to an eight-bit signed integer, if the 16-bit value is in the range -128..+127 you simply copy the L.O. byte of the 16-bit object to the eight-bit object. If the 16-bit signed value is greater than +127, then you clip the value to +127 and store +127 into the eight-bit object. Likewise, if the value is less than -128, you clip the final eight bit object to -128. Saturation works the same way when clipping 32-bit values to smaller values. If the larger value is outside the range of the smaller value, then you simply set the smaller value to the value closest to the out of range value that you can represent with the smaller value.

Obviously, if the larger value is outside the range of the smaller value, then there will be a loss of precision during the conversion. While clipping the value to the limits the smaller object imposes is never desirable, sometimes this is acceptable as the alternative is to raise an exception or otherwise reject the calculation. For many applications, such as audio or video processing, the clipped result is still recognizable, so this is a reasonable conversion to use.

3.11 Shifts and Rotates

Another set of logical operations which apply to bit strings are the *shift* and *rotate* operations. These two categories can be further broken down into *left shifts*, *left rotates*, *right shifts*, and *right rotates*. These operations turn out to be extremely useful to assembly language programmers.

The left shift operation moves each bit in a bit string one position to the left (see Figure 3.8).

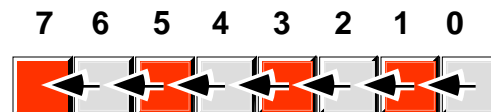


Figure 3.8 Shift Left Operation

Bit zero moves into bit position one, the previous value in bit position one moves into bit position two, etc. There are, of course, two questions that naturally arise: “What goes into bit zero?” and “Where does bit seven wind up?” We’ll shift a zero into bit zero and the previous value of bit seven will be the *carry* out of this operation.

The 80x86 provides a shift left instruction, SHL, that performs this useful operation. The syntax for the SHL instruction is the following:

```
shl( count, dest );
```

The count operand is either “CL” or a constant in the range 0..n, where n is one less than the number of bits in the destination operand (i.e., n=7 for eight-bit operands, n=15 for 16-bit operands, and n=31 for 32-bit operands). The dest operand is a typical dest operand, it can be either a memory location or a register.

When the count operand is the constant one, the SHL instruction does the following:

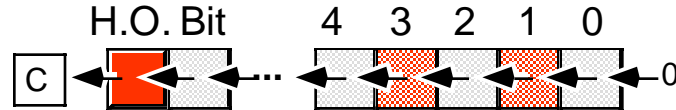


Figure 3.9 Operation of the SHL(1, Dest) Instruction

In Figure 3.9, the “C” represents the carry flag. That is, the bit shifted out of the H.O. bit of the operand is moved into the carry flag. Therefore, you can test for overflow after a SHL(1, dest) instruction by testing the carry flag immediately after executing the instruction (e.g., by using “if(@c) then...” or “if(@nc) then...”).

Intel’s literature suggests that the state of the carry flag is undefined if the shift count is a value other than one. Usually, the carry flag contains the last bit shifted out of the destination operand, but Intel doesn’t seem to guarantee this. If you need to shift more than one bit out of an operand and you need to capture all the bits you shift out, you should take a look at the SHLD and SHRD instructions in the appendices.

Note that shifting a value to the left is the same thing as multiplying it by its radix. For example, shifting a decimal number one position to the left (adding a zero to the right of the number) effectively multiplies it by ten (the radix):

```
1234 shl 1 = 12340 (shl 1 means shift one digit position to the left)
```

Since the radix of a binary number is two, shifting it left multiplies it by two. If you shift a binary value to the left twice, you multiply it by two twice (i.e., you multiply it by four). If you shift a binary value to the left three times, you multiply it by eight ($2*2*2$). In general, if you shift a value to the left n times, you multiply that value by 2^n .

A right shift operation works the same way, except we’re moving the data in the opposite direction. Bit seven moves into bit six, bit six moves into bit five, bit five moves into bit four, etc. During a right shift, we’ll move a zero into bit seven, and bit zero will be the carry out of the operation (see Figure 3.10).

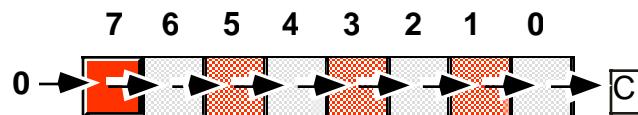


Figure 3.10 Shift Right Operation

As you would probably expect by now, the 80x86 provides a SHR instruction that will shift the bits to the right in a destination operand. The syntax is the same as the SHL instruction except, of course, you specify SHR rather than SHL:

```
SHR( count, dest );
```

This instruction shifts a zero into the H.O. bit of the destination operand, it shifts all the other bits one place to the right (that is, from a higher bit number to a lower bit number). Finally, bit zero is shifted into the carry flag. If you specify a count of one, the SHR instruction does the following:

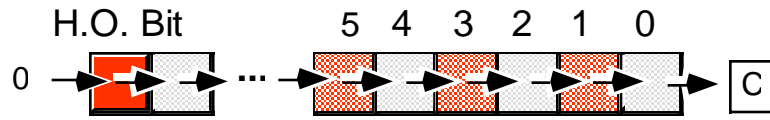


Figure 3.11 SHR(1, Dest) Operation

Once again, Intel's documents suggest that shifts of more than one bit leave the carry in an undefined state.

Since a left shift is equivalent to a multiplication by two, it should come as no surprise that a right shift is roughly comparable to a division by two (or, in general, a division by the radix of the number). If you perform n right shifts, you will divide that number by 2^n .

There is one problem with shift rights with respect to division: as described above a shift right is only equivalent to an *unsigned* division by two. For example, if you shift the unsigned representation of 254 (0FEh) one place to the right, you get 127 (07Fh), exactly what you would expect. However, if you shift the binary representation of -2 (0FEh) to the right one position, you get 127 (07Fh), which is *not* correct. This problem occurs because we're shifting a zero into bit seven. If bit seven previously contained a one, we're changing it from a negative to a positive number. Not a good thing when dividing by two.

To use the shift right as a division operator, we must define a third shift operation: *arithmetic shift right*⁶. An arithmetic shift right works just like the normal shift right operation (a *logical shift right*) with one exception: instead of shifting a zero into bit seven, an arithmetic shift right operation leaves bit seven alone, that is, during the shift operation it does not modify the value of bit seven as Figure 3.12 shows.

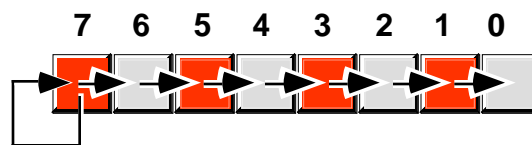


Figure 3.12 Arithmetic Shift Right Operation

This generally produces the result you expect. For example, if you perform the arithmetic shift right operation on -2 (0FEh) you get -1 (0FFh). Keep one thing in mind about arithmetic shift right, however. This operation always rounds the numbers to the closest integer *which is less than or equal to the actual result*. Based on experiences with high level programming languages and the standard rules of integer truncation, most people assume this means that a division always truncates towards zero. But this simply isn't the case. For example, if you apply the arithmetic shift right operation on -1 (0FFh), the result is -1, not zero. -1 is less than zero so the arithmetic shift right operation rounds towards minus one. This is not a "bug" in the arithmetic shift right operation, it's just uses a different (though valid) definition of integer division.

6. There is no need for an arithmetic shift left. The standard shift left operation works for both signed and unsigned numbers, assuming no overflow occurs.

The 80x86 provides an arithmetic shift right instruction, SAR (shift arithmetic right). This instruction's syntax is nearly identical to SHL and SHR. The syntax is

```
SAR( count, dest );
```

The usual limitations on the count and destination operands apply. This instruction does the following if the count is one:

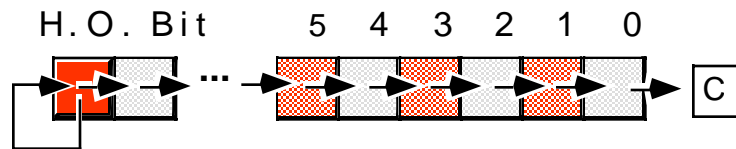


Figure 3.13 SAR(1, dest) Operation

Once again, Intel's documents suggest that shifts of more than one bit leave the carry in an undefined state.

Another pair of useful operations are *rotate left* and *rotate right*. These operations behave like the shift left and shift right operations with one major difference: the bit shifted out from one end is shifted back in at the other end.

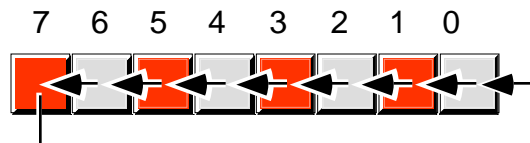


Figure 3.14 Rotate Left Operation

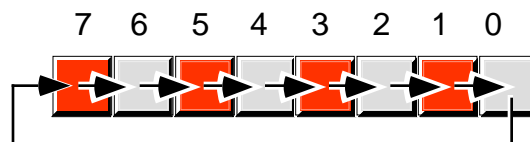


Figure 3.15 Rotate Right Operation

The 80x86 provides ROL (rotate left) and ROR (rotate right) instructions that do these basic operations on their operands. The syntax for these two instructions is similar to the shift instructions:

```
rol( count, dest );
ror( count, dest );
```

Once again, these instructions provide a special behavior if the shift count is one. Under this condition these two instructions also copy the bit shifted out of the destination operand into the carry flag as the following two figures show:

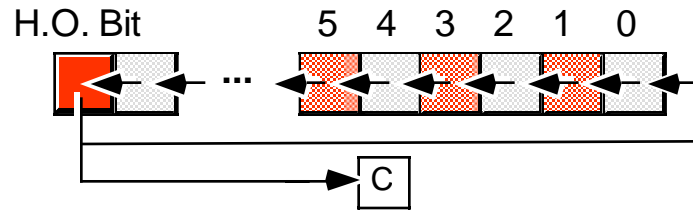


Figure 3.16 ROL(1, Dest) Operation

Note that, Intel's documents suggest that rotates of more than one bit leave the carry in an undefined state.

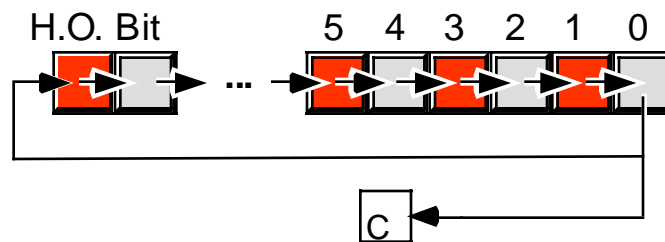


Figure 3.17 ROR(1, Dest) Operation

It will turn out that it is often more convenient for the rotate operation to shift the output bit through the carry and shift the previous carry value back into the input bit of the shift operation. The 80x86 RCL (rotate through carry left) and RCR (rotate through carry right) instructions achieve this for you. These instructions use the following syntax:

```
RCL( count, dest );
RCR( count, dest );
```

As is true for the other shift and rotate instructions, the count operand is either a constant or the CL register and the destination operand is a memory location or register. The count operand must be a value that is less than the number of bits in the destination operand. For a count value of one, these two instructions do the following:

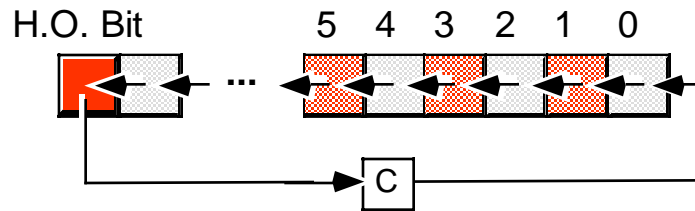


Figure 3.18 RCL(1, Dest) Operation

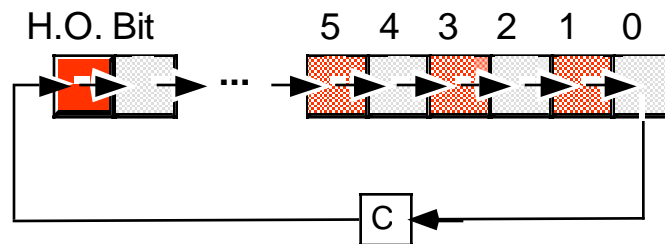


Figure 3.19 RCR(1, Dest) Operation

Again, Intel's documents suggest that rotates of more than one bit leave the carry in an undefined state.

3.12 Bit Fields and Packed Data

Although the 80x86 operates most efficiently on byte, word, and double word data types, occasionally you'll need to work with a data type that uses some number of bits other than eight, 16, or 32. For example, consider a date of the form "04/02/01". It takes three numeric values to represent this date: a month, day, and year value. Months, of course, take on the values 1..12. It will require at least four bits (maximum of sixteen different values) to represent the month. Days range between 1..31. So it will take five bits (maximum of 32 different values) to represent the day entry. The year value, assuming that we're working with values in the range 0..99, requires seven bits (which can be used to represent up to 128 different values). Four plus five plus seven is 16 bits, or two bytes. In other words, we can pack our date data into two bytes rather than the three that would be required if we used a separate byte for each of the month, day, and year values. This saves one byte of memory for each date stored, which could be a substantial saving if you need to store a lot of dates. The bits could be arranged as shown in the following figure:



Figure 3.20 Short Packed Date Format (Two Bytes)

MMMM represents the four bits making up the month value, DDDDD represents the five bits making up the day, and YYYYYYY is the seven bits comprising the year. Each collection of bits representing a data item is a *bit field*. April 2nd, 2001 would be represented as \$4101:

```
0100  00010 0000001 = %0100_0001_0000_0001 or $4101
 4      2      01
```

Although packed values are *space efficient* (that is, very efficient in terms of memory usage), they are computationally *inefficient* (slow!). The reason? It takes extra instructions to unpack the data packed into the various bit fields. These extra instructions take additional time to execute (and additional bytes to hold the instructions); hence, you must carefully consider whether packed data fields will save you anything. The following sample program demonstrates the effort that must go into packing and unpacking this 16-bit date format:

```
program dateDemo;

#include( "stdlib.hhf" );

static
    day:      uns8;
    month:    uns8;
    year:     uns8;

    packedDate: word;

begin dateDemo;

    stdout.put( "Enter the current month, day, and year: " );
    stdin.get( month, day, year );

    // Pack the data into the following bits:
    //
    // 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    //  m m m m d d d d d y y y y y y y
    //

    mov( 0, ax );
    mov( ax, packedDate ); //Just in case there is an error.
    if( month > 12 ) then

        stdout.put( "Month value is too large", nl );

    elseif( month = 0 ) then

        stdout.put( "Month value must be in the range 1..12", nl );

    elseif( day > 31 ) then

        stdout.put( "Day value is too large", nl );
```

```

elseif( day = 0 ) then

    stdout.put( "Day value must be in the range 1..31", nl );

elseif( year > 99 ) then

    stdout.put( "Year value must be in the range 0..99", nl );

else

    mov( month, al );
    shl( 5, ax );
    or( day, al );
    shl( 7, ax );
    or( year, al );
    mov( ax, packedDate );

endif;

// Okay, display the packed value:

stdout.put( "Packed data = $", packedDate, nl );

// Unpack the date:

mov( packedDate, ax );
and( $7f, al );          // Retrieve the year value.
mov( al, year );

mov( packedDate, ax ); // Retrieve the day value.
shr( 7, ax );
and( %1_1111, al );
mov( al, day );

mov( packedDate, ax ); // Retrieve the month value.
rol( 4, ax );
and( %1111, al );
mov( al, month );

stdout.put( "The date is ", month, "/", day, "/", year, nl );

end dateDemo;

```

Program 3.19 Packing and Unpacking Date Data

Of course, having gone through the problems with Y2K, using a date format that limits you to 100 years (or even 127 years) would be quite foolish at this time. If you're concerned about your software running 100 years from now, perhaps it would be wise to use a three-byte date format rather than a two-byte format. As you will see in the chapter on arrays, however, you should always try to create data objects whose length is an even power of two (one byte, two bytes, four bytes, eight bytes, etc.) or you will pay a performance penalty. Hence, it is probably wise to go ahead and use four bytes and pack this data into a dword variable. Figure 3.21 shows a possible data organization for a four-byte date.

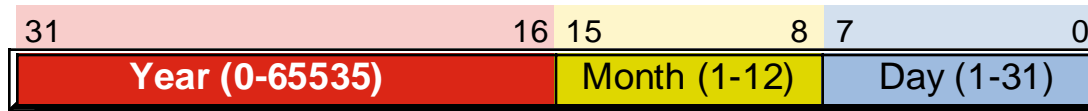


Figure 3.21 Long Packed Date Format (Four Bytes)

In this long packed data format several changes were made beyond simply extending the number of bits associated with the year. First, since there are lots of extra bits in a 32-bit dword variable, this format allots extra bits to the month and day fields. Since these two fields consist of eight bits each, they can be easily extracted as a byte object from the dword. This leaves fewer bits for the year, but 65,536 years is probably sufficient; you can probably assume without too much concern that your software will not still be in use 63 thousand years from now when this date format will wrap around.

Of course, you could argue that this is no longer a packed date format. After all, we needed three numeric values, two of which fit just nicely into one byte each and one that should probably have at least two bytes. Since this “packed” date format consumes the same four bytes as the unpacked version, what is so special about this format? Well, another difference you will note between this long packed date format and the short date format appearing in Figure 3.20 is the fact that this long date format rearranges the bits so the *Year* is in the H.O. bit positions, the *Month* field is in the middle bit positions, and the *Day* field is in the L.O. bit positions. This is important because it allows you to very easily compare two dates to see if one date is less than, equal to, or greater than another date. Consider the following code:

```
mov( Date1, eax );      // Assume Date1 and Date2 are dword variables
if( eax > Date2 ) then // using the Long Packed Date format.

    << do something if Date1 > Date2 >>

endif;
```

Had you kept the different date fields in separate variables, or organized the fields differently, you would not have been able to compare *Date1* and *Date2* in such a straight-forward fashion. Therefore, this example demonstrates another reason for packing data even if you don't realize any space savings- it can make certain computations more convenient or even more efficient (contrary to what normally happens when you pack data).

Examples of practical packed data types abound. You could pack eight boolean values into a single byte, you could pack two BCD digits into a byte, etc. Of course, a classic example of packed data is the FLAGS register (see Figure 3.22). This register packs nine important boolean objects (along with seven important system flags) into a single 16-bit register. You will commonly need to access many of these flags. For this reason, the 80x86 instruction set provides many ways to manipulate the individual bits in the FLAGS register. Of course, you can test many of the condition code flags using the HLA @c, @nc, @z, @nz, etc., pseudo-boolean variables in an IF statement or other statement using a boolean expression.

In addition to the condition codes, the 80x86 provides instructions that directly affect certain flags. These instructions include the following:

- `cld()`; Clears (sets to zero) the direction flag.
- `std()`; Sets (to one) the direction flag.
- `cli()`; Clears the interrupt disable flag.
- `sti()`; Sets the interrupt disable flag.
- `clc()`; Clears the carry flag.
- `stc()`; Sets the carry flag.
- `cmc()`; Complements (inverts) the carry flag.
- `sahf()`; Stores the AH register into the L.O. eight bits of the FLAGS register.
- `lahf()`; Loads AH from the L.O. eight bits of the FLAGS register.

There are other instructions that affect the FLAGS register as well; these, however, demonstrate how to access several of the packed boolean values in the FLAGS register. The LAHF and SAHF instructions, in particular, provide a convenient way to access the L.O. eight bits of the FLAGS register as an eight-bit byte (rather than as eight separate one-bit values).

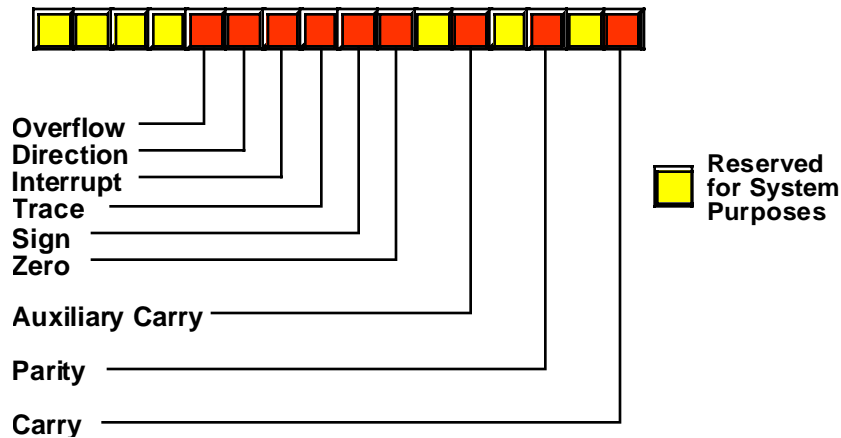


Figure 3.22 The FLAGS Register as a Packed Data Type

The LAHF (load AH with the L.O. eight bits of the FLAGS register) and the SAHF (store AH into the L.O. byte of the FLAGS register) use the following syntax:

```
lahf();
sahf();
```

3.13 Putting It All Together

In this chapter you've seen how we represent numeric values inside the computer. You've seen how to represent values using the decimal, binary, and hexadecimal numbering systems as well as the difference between signed and unsigned numeric representation. Since we represent nearly everything else inside a computer using numeric values, the material in this chapter is very important. Along with the base representation of numeric values, this chapter discusses the finite bit-string organization of data on typical computer systems, specifically bytes, words, and doublewords. Next, this chapter discusses arithmetic and logical operations on the numbers and presents some new 80x86 instructions to apply these operations to values inside the CPU. Finally, this chapter concludes by showing how you can pack several different numeric values into a fixed-length object (like a byte, word, or doubleword).

Absent from this chapter is any discussion of non-integer data. For example, how do we represent real numbers as well as integers? How do we represent characters, strings, and other non-numeric data? Well, that's the subject of the next chapter, so keep on reading...

More Data Representation

Chapter Four

4.1 Chapter Overview

Although the basic machine data objects (bytes, words, and double words) appear to represent nothing more than signed or unsigned numeric values, we can employ these data types to represent many other types of objects. This chapter discusses some of the other objects and their internal computer representation.

This chapter begins by discussing the floating point (real) numeric format. After integer representation, floating point representation is the second most popular numeric format in use on modern computer systems¹. Although the floating point format is somewhat complex, the necessity to handle non-integer calculations in modern programs requires that you understand this numeric format and its limitations.

Binary Coded Decimal (BCD) is another numeric data representation that is useful in certain contexts. Although BCD is not suitable for general purpose arithmetic, it is useful in some embedded applications. The principle benefit of the BCD format is the ease with which you can convert between string and BCD format. When we look at the BCD format a little later in this chapter, you'll see why this is the case.

Computers can represent all kinds of different objects, not just numeric values. Characters are, unquestionably, one of the more popular data types a computer manipulates. In this chapter you will take a look at a couple of different ways we can represent individual characters on a computer system. This chapter discusses two of the more common character sets in use today: the ASCII character set and the Unicode character set.

This chapter concludes by discussing some common non-numeric data types like pixel colors on a video display, audio data, video data, and so on. Of course, there are lots of different representations for any kind of standard data you could envision; there is no way two chapters in a textbook can cover them all. (And that's not even considering specialized data types you could create). Nevertheless, this chapter (and the last) should give you the basic idea behind representing data on a computer system.

4.2 An Introduction to Floating Point Arithmetic

Integer arithmetic does not let you represent fractional numeric values. Therefore, modern CPUs support an approximation of *real* arithmetic: floating point arithmetic. A big problem with floating point arithmetic is that it does not follow the standard rules of algebra. Nevertheless, many programmers apply normal algebraic rules when using floating point arithmetic. This is a source of defects in many programs. One of the primary goals of this section is to describe the limitations of floating point arithmetic so you will understand how to use it properly.

Normal algebraic rules apply only to *infinite precision* arithmetic. Consider the simple statement “ $x:=x+1$,” x is an integer. On any modern computer this statement follows the normal rules of algebra *as long as overflow does not occur*. That is, this statement is valid only for certain values of x ($minint \leq x < maxint$). Most programmers do not have a problem with this because they are well aware of the fact that integers in a program do not follow the standard algebraic rules (e.g., $5/2 \neq 2.5$).

Integers do not follow the standard rules of algebra because the computer represents them with a finite number of bits. You cannot represent any of the (integer) values above the maximum integer or below the minimum integer. Floating point values suffer from this same problem, only worse. After all, the integers are a subset of the real numbers. Therefore, the floating point values must represent the same infinite set of integers. However, there are an infinite number of values between any two real values, so this problem is infinitely worse. Therefore, as well as having to limit your values between a maximum and minimum range, you cannot represent all the values between those two ranges, either.

1. There are other numeric formats, such as fixed point formats and binary coded decimal format.

To represent real numbers, most floating point formats employ scientific notation and use some number of bits to represent a *mantissa* and a smaller number of bits to represent an *exponent*. The end result is that floating point numbers can only represent numbers with a specific number of *significant* digits. This has a big impact on how floating point arithmetic operates. To easily see the impact of limited precision arithmetic, we will adopt a simplified decimal floating point format for our examples. Our floating point format will provide a mantissa with three significant digits and a decimal exponent with two digits. The mantissa and exponents are both signed values as shown in Figure 4.1

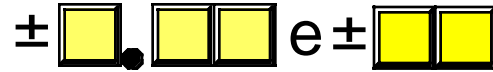


Figure 4.1 Simple Floating Point Format

When adding and subtracting two numbers in scientific notation, you must adjust the two values so that their exponents are the same. For example, when adding $1.23e1$ and $4.56e0$, you must adjust the values so they have the same exponent. One way to do this is to convert $4.56e0$ to $0.456e1$ and then add. This produces $1.686e1$. Unfortunately, the result does not fit into three significant digits, so we must either *round* or *truncate* the result to three significant digits. Rounding generally produces the most accurate result, so let's round the result to obtain $1.69e1$. As you can see, the lack of *precision* (the number of digits or bits we maintain in a computation) affects the accuracy (the correctness of the computation).

In the previous example, we were able to round the result because we maintained *four* significant digits *during* the calculation. If our floating point calculation is limited to three significant digits *during* computation, we would have had to truncate the last digit of the smaller number, obtaining $1.68e1$ which is even less correct. To improve the accuracy of floating point calculations, it is necessary to add extra digits for use during the calculation. Extra digits available during a computation are known as *guard digits* (or *guard bits* in the case of a binary format). They greatly enhance accuracy during a long chain of computations.

The accuracy loss during a single computation usually isn't enough to worry about unless you are greatly concerned about the accuracy of your computations. However, if you compute a value which is the result of a sequence of floating point operations, the error can *accumulate* and greatly affect the computation itself. For example, suppose we were to add $1.23e3$ with $1.00e0$. Adjusting the numbers so their exponents are the same before the addition produces $1.23e3 + 0.001e3$. The sum of these two values, even after rounding, is $1.23e3$. This might seem perfectly reasonable to you; after all, we can only maintain three significant digits, adding in a small value shouldn't affect the result at all. However, suppose we were to add $1.00e0$ to $1.23e3$ *ten times*. The first time we add $1.00e0$ to $1.23e3$ we get $1.23e3$. Likewise, we get this same result the second, third, fourth, ..., and tenth time we add $1.00e0$ to $1.23e3$. On the other hand, had we added $1.00e0$ to itself ten times, then added the result ($1.00e1$) to $1.23e3$, we would have gotten a different result, $1.24e3$. This is an important thing to know about limited precision arithmetic:

- The order of evaluation can effect the accuracy of the result.

You will get more accurate results if the relative magnitudes (that is, the exponents) are close to one another. If you are performing a chain calculation involving addition and subtraction, you should attempt to group the values appropriately.

Another problem with addition and subtraction is that you can wind up with *false precision*. Consider the computation $1.23e0 - 1.22e0$. This produces $0.01e0$. Although this is mathematically equivalent to $1.00e-2$, this latter form suggests that the last two digits are exactly zero. Unfortunately, we've only got a single significant digit at this time. Indeed, some FPUs or floating point software packages might actually insert random digits (or bits) into the L.O. positions. This brings up a second important rule concerning limited precision arithmetic:

- Whenever subtracting two numbers with the same signs or adding two numbers with different signs, the accuracy of the result may be less than the precision available in the floating point format.

Multiplication and division do not suffer from the same problems as addition and subtraction since you do not have to adjust the exponents before the operation; all you need to do is add the exponents and multiply the mantissas (or subtract the exponents and divide the mantissas). By themselves, multiplication and division do not produce particularly poor results. However, they tend to multiply any error that already exists in a value. For example, if you multiply 1.23e0 by two, when you should be multiplying 1.24e0 by two, the result is even less accurate. This brings up a third important rule when working with limited precision arithmetic:

- When performing a chain of calculations involving addition, subtraction, multiplication, and division, try to perform the multiplication and division operations first.

Often, by applying normal algebraic transformations, you can arrange a calculation so the multiply and divide operations occur first. For example, suppose you want to compute $x*(y+z)$. Normally you would add y and z together and multiply their sum by x . However, you will get a little more accuracy if you transform $x*(y+z)$ to get $x*y+x*z$ and compute the result by performing the multiplications first.

Multiplication and division are not without their own problems. When multiplying two very large or very small numbers, it is quite possible for *overflow* or *underflow* to occur. The same situation occurs when dividing a small number by a large number or dividing a large number by a small number. This brings up a fourth rule you should attempt to follow when multiplying or dividing values:

- When multiplying and dividing sets of numbers, try to arrange the multiplications so that they multiply large and small numbers together; likewise, try to divide numbers that have the same relative magnitudes.

Comparing floating point numbers is very dangerous. Given the inaccuracies present in any computation (including converting an input string to a floating point value), you should *never* compare two floating point values to see if they are equal. In a binary floating point format, different computations which produce the same (mathematical) result may differ in their least significant bits. For example, adding 1.31e0+1.69e0 should produce 3.00e0. Likewise, adding 1.50e0+1.50e0 should produce 3.00e0. However, were you to compare (1.31e0+1.69e0) against (1.50e0+1.50e0) you might find out that these sums are *not* equal to one another. The test for equality succeeds if and only if all bits (or digits) in the two operands are exactly the same. Since this is not necessarily true after two different floating point computations which should produce the same result, a straight test for equality may not work.

The standard way to test for equality between floating point numbers is to determine how much error (or tolerance) you will allow in a comparison and check to see if one value is within this error range of the other. The straight-forward way to do this is to use a test like the following:

```
if Value1 >= (Value2-error) and Value1 <= (Value2+error) then ...
```

Another common way to handle this same comparison is to use a statement of the form:

```
if abs(Value1-Value2) <= error then ...
```

Most texts, when discussing floating point comparisons, stop immediately after discussing the problem with floating point equality, assuming that other forms of comparison are perfectly okay with floating point numbers. This isn't true! If we are assuming that $x=y$ if x is within $y\pm error$, then a simple bitwise comparison of x and y will claim that $x<y$ if y is greater than x but less than $y+error$. However, in such a case x should really be treated as equal to y , not less than y . Therefore, we must always compare two floating point numbers using ranges, regardless of the actual comparison we want to perform. Trying to compare two floating point numbers directly can lead to an error. To compare two floating point numbers, x and y , against one another, you should use one of the following forms:

```
= if abs(x-y) <= error then ...
≠ if abs(x-y) > error then ...
< if (x-y) < -error then ...
≤ if (x-y) <= error then ...
> if (x-y) > error then ...
≥ if (x-y) >= -error then ...
```

You must exercise care when choosing the value for *error*. This should be a value slightly greater than the largest amount of error which will creep into your computations. The exact value will depend upon the

particular floating point format you use, but more on that a little later. The final rule we will state in this section is

- When comparing two floating point numbers, always compare one value to see if it is in the range given by the second value plus or minus some small error value.

There are many other little problems that can occur when using floating point values. This text can only point out some of the major problems and make you aware of the fact that you cannot treat floating point arithmetic like real arithmetic – the inaccuracies present in limited precision arithmetic can get you into trouble if you are not careful. A good text on numerical analysis or even scientific computing can help fill in the details that are beyond the scope of this text. If you are going to be working with floating point arithmetic, *in any language*, you should take the time to study the effects of limited precision arithmetic on your computations.

HLA's IF statement does not support boolean expressions involving floating point operands. Therefore, you cannot use statements like "IF($x < 3.141$) THEN..." in your programs. In a later chapter that discusses floating point operations on the 80x86 you'll learn how to do floating point comparisons.

4.2.1 IEEE Floating Point Formats

When Intel planned to introduce a floating point coprocessor for their new 8086 microprocessor, they were smart enough to realize that the electrical engineers and solid-state physicists who design chips were, perhaps, not the best people to do the necessary numerical analysis to pick the best possible binary representation for a floating point format. So Intel went out and hired the best numerical analyst they could find to design a floating point format for their 8087 FPU. That person then hired two other experts in the field and the three of them (Kahn, Coonan, and Stone) designed Intel's floating point format. They did such a good job designing the KCS Floating Point Standard that the IEEE organization adopted this format for the IEEE floating point format².

To handle a wide range of performance and accuracy requirements, Intel actually introduced *three* floating point formats: single precision, double precision, and extended precision. The single and double precision formats corresponded to C's float and double types or FORTRAN's real and double precision types. Intel intended to use extended precision for long chains of computations. Extended precision contains 16 extra bits that the calculations could use as guard bits before rounding down to a double precision value when storing the result.

The single precision format uses a one's complement 24 bit mantissa and an eight bit excess-127 exponent. The mantissa usually represents a value between 1.0 to just under 2.0. The H.O. bit of the mantissa is always assumed to be one and represents a value just to the left of the *binary point*³. The remaining 23 mantissa bits appear to the right of the binary point. Therefore, the mantissa represents the value:

$$1.\text{mmmmmmmm mmmmmmmmm mmmmmmmmm}$$

The "mmmm..." characters represent the 23 bits of the mantissa. Keep in mind that we are working with binary numbers here. Therefore, each position to the right of the binary point represents a value (zero or one) times a successive negative power of two. The implied one bit is always multiplied by 2^0 , which is one. This is why the mantissa is always greater than or equal to one. Even if the other mantissa bits are all zero, the implied one bit always gives us the value one⁴. Of course, even if we had an almost infinite number of one bits after the binary point, they still would not add up to two. This is why the mantissa can represent values in the range one to just under two.

Although there are an infinite number of values between one and two, we can only represent eight million of them because we use a 23 bit mantissa (the 24th bit is always one). This is the reason for inaccuracy

2. There were some minor changes to the way certain degenerate operations were handled, but the bit representation remained essentially unchanged.

3. The binary point is the same thing as the decimal point except it appears in binary numbers rather than decimal numbers.

4. Actually, this isn't necessarily true. The IEEE floating point format supports *denormalized* values where the H.O. bit is not zero. However, we will ignore denormalized values in our discussion.

in floating point arithmetic – we are limited to 23 bits of precision in computations involving single precision floating point values.

The mantissa uses a *one's complement* format rather than two's complement. This means that the 24 bit value of the mantissa is simply an unsigned binary number and the sign bit determines whether that value is positive or negative. One's complement numbers have the unusual property that there are two representations for zero (with the sign bit set or clear). Generally, this is important only to the person designing the floating point software or hardware system. We will assume that the value zero always has the sign bit clear.

To represent values outside the range 1.0 to just under 2.0, the exponent portion of the floating point format comes into play. The floating point format raises two to the power specified by the exponent and then multiplies the mantissa by this value. The exponent is eight bits and is stored in an *excess-127* format. In excess-127 format, the exponent 2^0 is represented by the value 127 (7f). Therefore, to convert an exponent to excess-127 format simply add 127 to the exponent value. The use of excess-127 format makes it easier to compare floating point values. The single precision floating point format takes the form shown in Figure 4.2.

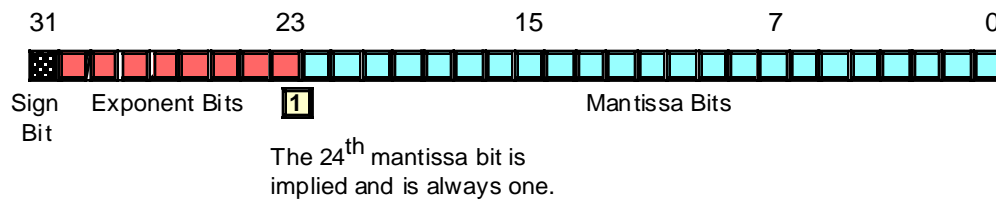


Figure 4.2 Single Precision (32-bit) Floating Point Format

With a 24 bit mantissa, you will get approximately $6\frac{1}{2}$ digits of precision (one half digit of precision means that the first six digits can all be in the range 0..9 but the seventh digit can only be in the range 0..x where $x < 9$ and is generally close to five). With an eight bit excess-127 exponent, the dynamic range of single precision floating point numbers is approximately $2^{\pm 128}$ or about $10^{\pm 38}$.

Although single precision floating point numbers are perfectly suitable for many applications, the dynamic range is somewhat limited for many scientific applications and the very limited precision is unsuitable for many financial, scientific, and other applications. Furthermore, in long chains of computations, the limited precision of the single precision format may introduce serious error.

The double precision format helps overcome the problems of single precision floating point. Using twice the space, the double precision format has an 11-bit excess-1023 exponent and a 53 bit mantissa (with an implied H.O. bit of one) plus a sign bit. This provides a dynamic range of about $10^{\pm 308}$ and $14\frac{1}{2}$ digits of precision, sufficient for most applications. Double precision floating point values take the form shown in Figure 4.3.

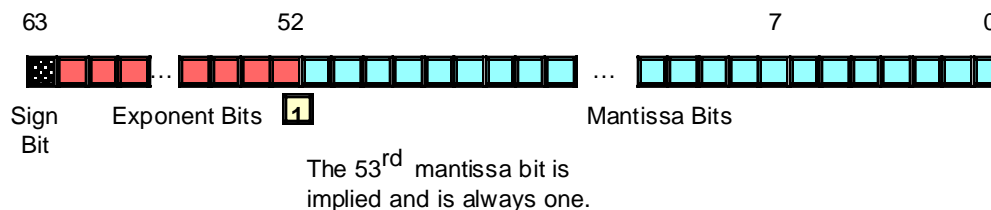


Figure 4.3 64-Bit Double Precision Floating Point Format

In order to help ensure accuracy during long chains of computations involving double precision floating point numbers, Intel designed the extended precision format. The extended precision format uses 80 bits. Twelve of the additional 16 bits are appended to the mantissa, four of the additional bits are appended to the

end of the exponent. Unlike the single and double precision values, the extended precision format's mantissa does not have an implied H.O. bit which is always one. Therefore, the extended precision format provides a 64 bit mantissa, a 15 bit excess-16383 exponent, and a one bit sign. The format for the extended precision floating point value is shown in Figure 4.4:

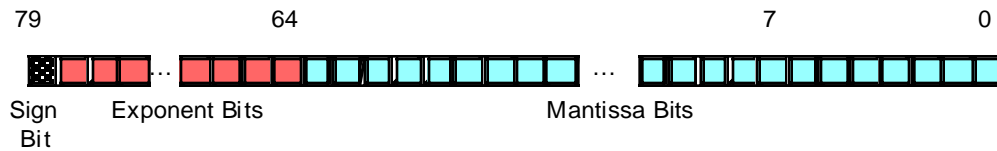


Figure 4.4 80-bit Extended Precision Floating Point Format

On the FPUs all computations are done using the extended precision form. Whenever you load a single or double precision value, the FPU automatically converts it to an extended precision value. Likewise, when you store a single or double precision value to memory, the FPU automatically rounds the value down to the appropriate size before storing it. By always working with the extended precision format, Intel guarantees a large number of guard bits are present to ensure the accuracy of your computations. Some texts erroneously claim that you should never use the extended precision format in your own programs, because Intel only guarantees accurate computations when using the single or double precision formats. This is foolish. By performing all computations using 80 bits, Intel helps ensure (but not guarantee) that you will get full 32 or 64 bit accuracy in your computations. Since the FPUs do not provide a large number of guard bits in 80 bit computations, some error will inevitably creep into the L.O. bits of an extended precision computation. However, if your computation is correct to 64 bits, the 80 bit computation will always provide *at least* 64 accurate bits. Most of the time you will get even more. While you cannot assume that you get an accurate 80 bit computation, you can usually do better than 64 when using the extended precision format.

To maintain maximum precision during computation, most computations use *normalized* values. A normalized floating point value is one whose H.O. mantissa bit contains one. Almost any non-normalized value can be normalized by shifting the mantissa bits to the left and decrementing the exponent until a one appears in the H.O. bit of the mantissa. Remember, the exponent is a binary exponent. Each time you increment the exponent, you multiply the floating point value by two. Likewise, whenever you decrement the exponent, you divide the floating point value by two. By the same token, shifting the mantissa to the left one bit position multiplies the floating point value by two; likewise, shifting the mantissa to the right divides the floating point value by two. Therefore, shifting the mantissa to the left one position *and* decrementing the exponent does not change the value of the floating point number at all.

Keeping floating point numbers normalized is beneficial because it maintains the maximum number of bits of precision for a computation. If the H.O. bits of the mantissa are all zero, the mantissa has that many fewer bits of precision available for computation. Therefore, a floating point computation will be more accurate if it involves only normalized values.

There are two important cases where a floating point number cannot be normalized. The value 0.0 is a special case. Obviously it cannot be normalized because the floating point representation for zero has no one bits in the mantissa. This, however, is not a problem since we can exactly represent the value zero with only a single bit.

The second case is when we have some H.O. bits in the mantissa which are zero but the biased exponent is also zero (and we cannot decrement it to normalize the mantissa). Rather than disallow certain small values, whose H.O. mantissa bits and biased exponent are zero (the most negative exponent possible), the IEEE standard allows special *denormalized* values to represent these smaller values⁵. Although the use of denormalized values allows IEEE floating point computations to produce better results than if underflow occurred, keep in mind that denormalized values offer less bits of precision.

5. The alternative would be to underflow the values to zero.

Since the FPU always converts single and double precision values to extended precision, extended precision arithmetic is actually *faster* than single or double precision. Therefore, the expected performance benefit of using the smaller formats is not present on these chips. However, when designing the Pentium/586 CPU, Intel redesigned the built-in floating point unit to better compete with RISC chips. Most RISC chips support a native 64 bit double precision format which is faster than Intel's extended precision format. Therefore, Intel provided native 64 bit operations on the Pentium to better compete against the RISC chips. Therefore, the double precision format is the fastest on the Pentium and later chips.

4.2.2 HLA Support for Floating Point Values

HLA provides several data types and library routines to support the use of floating point data in your assembly language programs. These include built-in types to declare floating point variables as well as routines that provide floating point input, output, and conversion.

Perhaps the best place to start when discussing HLA's floating point facilities is with a description of floating point literal constants. HLA floating point constants allow the following syntax:

- An optional “+” or “-” symbol, denoting the sign of the mantissa (if this is not present, HLA assumes that the mantissa is positive),
- Followed by one or more decimal digits,
- Optionally followed by a decimal point and one or more decimal digits,
- Optionally followed by an “e” or “E”, optionally followed by a sign (“+” or “-”) and one or more decimal digits.

Note: the decimal point or the “e”/“E” must be present in order to differentiate this value from an integer or unsigned literal constant. Here are some examples of legal literal floating point constants:

```
1.234      3.75e2      -1.0      1.1e-1      1e+4      0.1      -123.456e+789      +25e0
```

Notice that a floating point literal constant cannot begin with a decimal point; it must begin with a decimal digit so you must use “0.1” to represent “.1” in your programs.

HLA also allows you to place an underscore character (“_”) between any two consecutive decimal digits in a floating point literal constant. You may use the underscore character in place of a comma (or other language-specific separator character) to help make your large floating point numbers easier to read. Here are some examples:

```
1_234_837.25      1_000.00      789_934.99      9_999.99
```

To declare a floating point variable you use the *real32*, *real64*, or *real80* data types. Like their integer and unsigned brethren, the number at the end of these data type declarations specifies the number of bits used for each type's binary representation. Therefore, you use *real32* to declare single precision real values, *real64* to declare double precision floating point values, and *real80* to declare extended precision floating point values. Other than the fact that you use these types to declare floating point variables rather than integers, their use is nearly identical to that for *int8*, *int16*, *int32*, etc. The following examples demonstrate these declarations and their syntax:

```
static

    fltVar1:  real32;
    fltVar1a: real32 := 2.7;
    pi:      real32 := 3.14159;
    DblVar:  real64;
    DblVar2: real64 := 1.23456789e+10;
    XPVar:   real80;
    XPVar2:  real80 := -1.0e-104;
```

To output a floating point variable in ASCII form, you would use one of the *stdout.putr32*, *stdout.putr64*, or *stdout.putr80* routines. These procedures display a number in decimal notation, that is, a string of digits, an optional decimal point and a closing string of digits. Other than their names, these three routines use exactly the same calling sequence. Here are the calls and parameters for each of these routines:

```
stdout.putr80( r:real80; width:uns32; decpts:uns32 );
stdout.putr64( r:real64; width:uns32; decpts:uns32 );
stdout.putr32( r:real32; width:uns32; decpts:uns32 );
```

The first parameter to these procedures is the floating point value you wish to print. The size of this parameter must match the procedure's name (e.g., the *r* parameter must be an 80-bit extended precision floating point variable when calling the *stdout.putr80* routine). The second parameter specifies the field width for the output text; this is the number of print positions the number will require when the procedure displays it. Note that this width must include print positions for the sign of the number and the decimal point. The third parameter specifies the number of print positions after the decimal point. For example,

```
stdout.putr32( pi, 10, 4 );
```

displays the value

```
_ _ _ _ 3.1416
```

(the underscores represent leading spaces in this example).

Of course, if the number is very large or very small, you will want to use scientific notation rather than decimal notation for your floating point numeric output. The HLA Standard Library *stdout.pute32*, *stdout.pute64*, and *stdout.pute80* routines provide this facility. These routines use the following procedure prototypes:

```
stdout.pute80( r:real80; width:uns32 );
stdout.pute64( r:real64; width:uns32 );
stdout.pute32( r:real32; width:uns32 );
```

Unlike the decimal output routines, these scientific notation output routines do not require a third parameter specifying the number of digits after the decimal point to display. The width parameter, indirectly, specifies this value since all but one of the mantissa digits always appears to the right of the decimal point. These routines output their values in decimal notation, similar to the following:

```
1.23456789e+10   -1.0e-104   1e+2
```

You can also output floating point values using the HLA Standard Library *stdout.put* routine. If you specify the name of a floating point variable in the *stdout.put* parameter list, the *stdout.put* code will output the value using scientific notation. The actual field width varies depending on the size of the floating point variable (the *stdout.put* routine attempts to output as many significant digits as possible, in this case). Example:

```
stdout.put( "XPVar2 = ", XPVar2 );
```

If you specify a field width specification, by using a colon followed by a signed integer value, then the *stdout.put* routine will use the appropriate *stdout.puteXX* routine to display the value. That is, the number will still appear in scientific notation, but you get to control the field width of the output value. Like the field width for integer and unsigned values, a positive field width right justifies the number in the specified field, a negative number left justifies the value. Here is an example that prints the *XPVar2* variable using ten print positions:

```
stdout.put( "XPVar2 = ", XPVar2:10 );
```

If you wish to use *stdout.put* to print a floating point value in decimal notation, you need to use the following syntax:

```
Variable_Name : Width : DecPts
```


Note that the *DecPts* field must be a non-negative integer value.

When *stdout.put* contains a parameter of this form, it calls the corresponding *stdout.putrXX* routine to display the specified floating point value. As an example, consider the following call:

```
stdout.put( "Pi = ", pi:5:3 );
```

The corresponding output is

```
3.142
```

The HLA Standard Library provides several other useful routines you can use when outputting floating point values. Consult the HLA Standard Library reference manual for more information on these routines.

The HLA Standard Library provides several routines to let you display floating point values in a wide variety of formats. In contrast, the HLA Standard Library only provides two routines to support floating point input: *stdin.getf()* and *stdin.get()*. The *stdin.getf()* routine requires the use of the 80x86 FPU stack, a hardware component that this chapter is not going to cover. Therefore, this chapter will defer the discussion of the *stdin.getf()* routine until the chapter on arithmetic, later in this text. Since the *stdin.get()* routine provides all the capabilities of the *stdin.getf()* routine, this deference will not prove to be a problem.

You've already seen the syntax for the *stdin.get()* routine; its parameter list simply contains a list of variable names. *Stdin.get()* reads appropriate values for the user for each of the variables appearing in the parameter list. If you specify the name of a floating point variable, the *stdin.get()* routine automatically reads a floating point value from the user and stores the result into the specified variable. The following example demonstrates the use of this routine:

```
stdout.put( "Input a double precision floating point value: " );
stdin.get( DblVar );
```

Warning: This section has discussed how you would declare floating point variables and how you would input and output them. It did not discuss arithmetic. Floating point arithmetic is different than integer arithmetic; you cannot use the 80x86 ADD and SUB instructions to operate on floating point values. Floating point arithmetic will be the subject of a later chapter in this text.

4.3 Binary Coded Decimal (BCD) Representation

Although the integer and floating point formats cover most of the numeric needs of an average program, there are some special cases where other numeric representations are convenient. In this section we'll discuss the Binary Coded Decimal (BCD) format since the 80x86 CPU provides a small amount of hardware support for this data representation.

BCD values are a sequence of nibbles with each nibble representing a value in the range zero through nine. Of course you can represent values in the range 0..15 using a nibble; the BCD format, however, uses only 10 of the possible 16 different values for each nibble.

Each nibble in a BCD value represents a single decimal digit. Therefore, with a single byte (i.e., two digits) we can represent values containing two decimal digits, or values in the range 0..99. With a word, we can represent values having four decimal digits, or values in the range 0..9999. Likewise, with a double word we can represent values with up to eight decimal digits (since there are eight nibbles in a double word value).

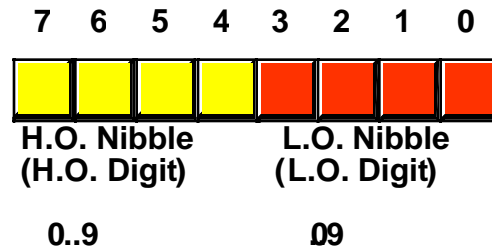


Figure 4.5 BCD Data Representation in Memory

As you can see, BCD storage isn't particularly memory efficient. For example, an eight-bit BCD variable can represent values in the range 0..99 while that same eight bits, when holding a binary value, can represent values in the range 0..255. Likewise, a 16-bit binary value can represent values in the range 0..65535 while a 16-bit BCD value can only represent about $\frac{1}{6}$ of those values (0..9999). Inefficient storage isn't the only problem. BCD calculations tend to be slower than binary calculations.

At this point, you're probably wondering why anyone would ever use the BCD format. The BCD format does have two saving graces: it's very easy to convert BCD values between the internal numeric representation and their string representation; also, it's very easy to encode multi-digit decimal values in hardware (e.g., using a "thumb wheel" or dial) using BCD than it is using binary. For these two reasons, you're likely to see people using BCD in embedded systems (e.g., toaster ovens and alarm clocks) but rarely in general purpose computer software.

A few decades ago people mistakenly thought that calculations involving BCD (or just 'decimal') arithmetic was more accurate than binary calculations. Therefore, they would often perform 'important' calculations, like those involving dollars and cents (or other monetary units) using decimal-based arithmetic. While it is true that certain calculations can produce more accurate results in BCD, this statement is not true in general. Indeed, for most calculations (even those involving fixed point decimal arithmetic), the binary representation is more accurate. For this reason, most modern computer programs represent all values in a binary form. For example, the Intel x86 floating point unit (FPU) supports a pair of instructions for loading and storing BCD values. Internally, however, the FPU converts these BCD values to binary and performs all calculations in binary. It only uses BCD as an external data format (external to the FPU, that is). This generally produces more accurate results and requires far less silicon than having a separate coprocessor that supports decimal arithmetic.

This text will take up the subject of BCD arithmetic in a later chapter. Until then, you can safely ignore BCD unless you find yourself converting a COBOL program to assembly language (which is quite unlikely).

4.4 Characters

Perhaps the most important data type on a personal computer is the character data type. The term "character" refers to a human or machine readable symbol that is typically a non-numeric entity. In general, the term "character" refers to any symbol that you can normally type on a keyboard (including some symbols that may require multiple key presses to produce) or display on a video display. Many beginners often confuse the terms "character" and "alphabetic character." These terms are not the same. Punctuation symbols, numeric digits, spaces, tabs, carriage returns (enter), other control characters, and other special symbols are also characters. When this text uses the term "character" it refers to any of these characters, not just the alphabetic characters. When this text refers to alphabetic characters, it will use phrases like "alphabetic characters," "upper case characters," or "lower case characters."⁶

Another common problem beginners have when they first encounter the character data type is differentiating between numeric characters and numbers. The character '1' is distinct and different from the value one. The computer (generally) uses two different internal, binary, representations for numeric characters ('0', '1', ..., '9') versus the numeric values zero through nine. You must take care not to confuse the two.

Most computer systems use a one or two byte sequence to encode the various characters in binary form. Windows and Linux certainly fall into this category, using either the ASCII or Unicode encodings for characters. This section will discuss the ASCII character set and the character declaration facilities that HLA provides.

4.4.1 The ASCII Character Encoding

The ASCII (American Standard Code for Information Interchange) Character set maps 128 textual characters to the unsigned integer values 0..127 (\$0..\$7F). Internally, of course, the computer represents everything using binary numbers; so it should come as no surprise that the computer also uses binary values to represent non-numeric entities such as characters. Although the exact mapping of characters to numeric values is arbitrary and unimportant, it is important to use a standardized code for this mapping since you will need to communicate with other programs and peripheral devices and you need to talk the same "language" as these other programs and devices. This is where the ASCII code comes into play; it is a standardized code that nearly everyone has agreed upon. Therefore, if you use the ASCII code 65 to represent the character "A" then you know that some peripheral device (such as a printer) will correctly interpret this value as the character "A" whenever you transmit data to that device.

You should not get the impression that ASCII is the only character set in use on computer systems. IBM uses the EBCDIC character set family on many of its mainframe computer systems. Another common character set in use is the Unicode character set. Unicode is an extension to the ASCII character set that uses 16 bits rather than seven to represent characters. This allows the use of 65,536 different characters in the character set, allowing the inclusion of most symbols in the world's different languages into a single unified character set.

Since the ASCII character set provides only 128 different characters and a byte can represent 256 different values, an interesting question arises: "what do we do with the values 128..255 that one could store into a byte value when working with character data?" One answer is to ignore those extra values. That will be the primary approach of this text. Another possibility is to extend the ASCII character set and add an additional 128 characters to the character set. Of course, this would tend to defeat the whole purpose of having a standardized character set unless you could get everyone to agree upon the extensions. That is a difficult task.

When IBM first created their IBM-PC, they defined these extra 128 character codes to contain various non-English alphabetic characters, some line drawing graphics characters, some mathematical symbols, and several other special characters. Since IBM's PC was the foundation for what we typically call a PC today, that character set has become a pseudo-standard on all IBM-PC compatible machines. Even on modern machines, which are not IBM-PC compatible and cannot run early PC software, the IBM extended character set still survives. Note, however, that this PC character set (an extension of the ASCII character set) is not universal. Most printers will not print the extended characters when using native fonts and many programs (particularly in non-English countries) do not use those characters for the upper 128 codes in an eight-bit value. For these reasons, this text will generally stick to the standard 128 character ASCII character set. However, a few examples and programs in this text will use the IBM PC extended character set, particularly the line drawing graphic characters (see Appendix B).

Should you need to exchange data with other machines which are not PC-compatible, you have only two alternatives: stick to standard ASCII or ensure that the target machine supports the extended IBM-PC character set. Some machines, like the Apple Macintosh, do not provide native support for the extended IBM-PC character set; however you may obtain a PC font which lets you display the extended character set.

6. Upper and lower case characters are always alphabetic characters within this text.

Other machines have similar capabilities. However, the 128 characters in the standard ASCII character set are the only ones you should count on transferring from system to system.

Despite the fact that it is a “standard”, simply encoding your data using standard ASCII characters does not guarantee compatibility across systems. While it’s true that an “A” on one machine is most likely an “A” on another machine, there is very little standardization across machines with respect to the use of the control characters. Indeed, of the 32 control codes plus delete, there are only four control codes commonly supported – backspace (BS), tab, carriage return (CR), and line feed (LF). Worse still, different machines often use these control codes in different ways. End of line is a particularly troublesome example. Windows, MS-DOS, CP/M, and other systems mark end of line by the two-character sequence CR/LF. Apple Macintosh, and many other systems mark the end of line by a single CR character. Linux, BeOS, and other UNIX systems mark the end of a line with a single LF character. Needless to say, attempting to exchange simple text files between such systems can be an experience in frustration. Even if you use standard ASCII characters in all your files on these systems, you will still need to convert the data when exchanging files between them. Fortunately, such conversions are rather simple.

Despite some major shortcomings, ASCII data is *the* standard for data interchange across computer systems and programs. Most programs can accept ASCII data; likewise most programs can produce ASCII data. Since you will be dealing with ASCII characters in assembly language, it would be wise to study the layout of the character set and memorize a few key ASCII codes (e.g., “0”, “A”, “a”, etc.).

The ASCII character set (excluding the extended characters defined by IBM) is divided into four groups of 32 characters. The first 32 characters, ASCII codes 0 through 1F (31), form a special set of non-printing characters called the control characters. We call them control characters because they perform various printer/display control operations rather than displaying symbols. Examples include *carriage return*, which positions the cursor to the left side of the current line of characters⁷, line feed (which moves the cursor down one line on the output device), and back space (which moves the cursor back one position to the left). Unfortunately, different control characters perform different operations on different output devices. There is very little standardization among output devices. To find out exactly how a control character affects a particular device, you will need to consult its manual.

The second group of 32 ASCII character codes comprise various punctuation symbols, special characters, and the numeric digits. The most notable characters in this group include the space character (ASCII code 20) and the numeric digits (ASCII codes 30..39). Note that the numeric digits differ from their numeric values only in the H.O. nibble. By subtracting 30 from the ASCII code for any particular digit you can obtain the numeric equivalent of that digit.

The third group of 32 ASCII characters contains the upper case alphabetic characters. The ASCII codes for the characters “A”..”Z” lie in the range 41..5A (65..90). Since there are only 26 different alphabetic characters, the remaining six codes hold various special symbols.

The fourth, and final, group of 32 ASCII character codes represent the lower case alphabetic symbols, five additional special symbols, and another control character (delete). Note that the lower case character symbols use the ASCII codes 61..7A. If you convert the codes for the upper and lower case characters to binary, you will notice that the upper case symbols differ from their lower case equivalents in exactly one bit position. For example, consider the character code for “E” and “e” in the following figure:

7. Historically, carriage return refers to the *paper carriage* used on typewriters. A carriage return consisted of physically moving the carriage all the way to the right so that the next character typed would appear at the left hand side of the paper.

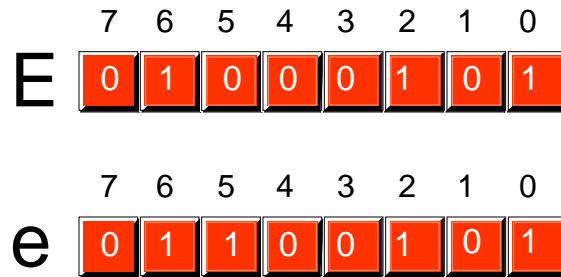


Figure 4.6 ASCII Codes for “E” and “e”

The only place these two codes differ is in bit five. Upper case characters always contain a zero in bit five; lower case alphabetic characters always contain a one in bit five. You can use this fact to quickly convert between upper and lower case. If you have an upper case character you can force it to lower case by setting bit five to one. If you have a lower case character and you wish to force it to upper case, you can do so by setting bit five to zero. You can toggle an alphabetic character between upper and lower case by simply inverting bit five.

Indeed, bits five and six determine which of the four groups in the ASCII character set you’re in:

Table 9: ASCII Groups

Bit 6	Bit 5	Group
0	0	Control Characters
0	1	Digits & Punctuation
1	0	Upper Case & Special
1	1	Lower Case & Special

So you could, for instance, convert any upper or lower case (or corresponding special) character to its equivalent control character by setting bits five and six to zero.

Consider, for a moment, the ASCII codes of the numeric digit characters:

Table 10: ASCII Codes for Numeric Digits

Character	Decimal	Hexadecimal
“0”	48	\$30
“1”	49	\$31
“2”	50	\$32
“3”	51	\$33

Table 10: ASCII Codes for Numeric Digits

Character	Decimal	Hexadecimal
"4"	52	\$34
"5"	53	\$35
"6"	54	\$36
"7"	55	\$37
"8"	56	\$38
"9"	57	\$39

The decimal representations of these ASCII codes are not very enlightening. However, the hexadecimal representation of these ASCII codes reveals something very important – the L.O. nibble of the ASCII code is the binary equivalent of the represented number. By stripping away (i.e., setting to zero) the H.O. nibble of a numeric character, you can convert that character code to the corresponding binary representation. Conversely, you can convert a binary value in the range 0..9 to its ASCII character representation by simply setting the H.O. nibble to three. Note that you can use the logical-AND operation to force the H.O. bits to zero; likewise, you can use the logical-OR operation to force the H.O. bits to %0011 (three).

Note that you *cannot* convert a string of numeric characters to their equivalent binary representation by simply stripping the H.O. nibble from each digit in the string. Converting 123 (\$31 \$32 \$33) in this fashion yields three bytes: \$010203, not the correct value which is \$7B. Converting a string of digits to an integer requires more sophistication than this; the conversion above works only for single digits.

4.4.2 HLA Support for ASCII Characters

Although you could easily store character values in *byte* variables and use the corresponding numeric equivalent ASCII code when using a character literal in your program, such agony is unnecessary - HLA provides good support for character variables and literals in your assembly language programs.

Character literal constants in HLA take one of two forms: a single character surrounded by apostrophes or a pound symbol (“#”) followed by a numeric constant in the range 0..127 specifying the ASCII code of the character. Here are some examples:

```
'A'      #65      #$41      %#0100_0001
```

Note that these examples all represent the same character ('A') since the ASCII code of 'A' is 65.

With a single exception, only a single character may appear between the apostrophes in a literal character constant. That single exception is the apostrophe character itself. If you wish to create an apostrophe literal constant, place four apostrophes in a row (i.e., double up the apostrophe inside the surrounding apostrophes), i.e.,

```
''''
```

The pound sign operator (“#”) must precede a legal HLA numeric constant (either decimal, hexadecimal or binary as the examples above indicate). In particular, the pound sign is not a generic character conversion function; it cannot precede registers or variable names, only constants. As a general rule, you should always use the apostrophe form of the character literal constant for graphic characters (that is, those that are printable or displayable). Use the pound sign form for control characters (that are invisible, or do funny things when you print them) or for extended ASCII characters that may not display or print properly within your source code.

Notice the difference between a character literal constant and a string literal constant in your programs. Strings are sequences of zero or more characters surrounded by quotation marks, characters are surrounded by apostrophes. It is especially important to realize that

```
'A' ≠ "A"
```

The character constant 'A' and the string containing the single character "A" have two completely different internal representations. If you attempt to use a string containing a single character where HLA expects a character constant, HLA will report an error. Strings and string constants will be the subject of a later chapter.

To declare a character variable in an HLA program, you use the *char* data type. The following declaration, for example, demonstrates how to declare a variable named *UserInput*:

```
static
    UserInput:    char;
```

This declaration reserves one byte of storage that you could use to store any character value (including eight-bit extended ASCII characters). You can also initialize character variables as the following example demonstrates:

```
static

    TheCharA:    char := 'A' ;
    ExtendedChar char := #128;
```

Since character variables are eight-bit objects, you can manipulate them using eight-bit registers. You can move character variables into eight-bit registers and you can store the value of an eight-bit register into a character variable.

The HLA Standard Library provides a handful of routines that you can use for character I/O and manipulation; these include *stdout.putc*, *stdout.putcSize*, *stdout.put*, *stdin.getc*, and *stdin.get*.

The *stdout.putc* routine uses the following calling sequence:

```
stdout.putc( chvar );
```

This procedure outputs the single character parameter passed to it as a character to the standard output device. The parameter may be any *char* constant or variable, or a *byte* variable or register⁸.

The *stdout.putcSize* routine provides output width control when displaying character variables. The calling sequence for this procedure is

```
stdout.putcSize( charvar, widthInt32, fillchar );
```

This routine prints the specified character (parameter *c*) using at least *width* print positions⁹. If the absolute value of *width* is greater than one, then *stdout.putcSize* prints the *fill* character as padding. If the value of *width* is positive, then *stdout.putcSize* prints the character right justified in the print field; if *width* is negative, then *stdout.putcSize* prints the character left justified in the print field. Since character output is usually left justified in a field, the *width* value will normally be negative for this call. The space character is the most common *fill* value.

You can also print character values using the generic *stdout.put* routine. If a character variable appears in the *stdout.put* parameter list, then *stdout.put* will automatically print it as a character value, e.g.,

```
stdout.put( "Character c = '", c, "'", nl );
```

You can read characters from the standard input using the *stdin.getc* and *stdin.get* routines. The *stdin.getc* routine does not have any parameters. It reads a single character from the standard input buffer and returns this character in the AL register. You may then store the character value away or otherwise

8. If you specify a byte variable or a byte-sized register as the parameter, the *stdout.putc* routine will output the character whose ASCII code appears in the variable or register.

9. The only time *stdout.putcSize* uses more print positions than you specify is when you specify zero as the width; then this routine uses exactly one print position.

manipulate the character in the AL register. The following program reads a single character from the user, converts it to upper case if it is a lower case character, and then displays the character:

```

program charInputDemo;
#include( "stdlib.hhf" );
static
    c:char;

begin charInputDemo;

    stdout.put( "Enter a character: " );
    stdin.getc();
    if( al >= 'a' ) then

        if( al <= 'z' ) then

            and( $5f, al );

        endif;

    endif;
    stdout.put
    (
        "The character you entered, possibly ", nl,
        "converted to upper case, was '"
    );
    stdout.putc( al );
    stdout.put( "'", nl );

end charInputDemo;

```

Program 4.1 Character Input Sample

You can also use the generic *stdin.get* routine to read character variables from the user. If a *stdin.get* parameter is a character variable, then the *stdin.get* routine will read a character from the user and store the character value into the specified variable. Here is the program above rewritten to use the *stdin.get* routine:

```

program charInputDemo2;
#include( "stdlib.hhf" );
static
    c:char;

begin charInputDemo2;

    stdout.put( "Enter a character: " );
    stdin.get(c);
    if( c >= 'a' ) then

        if( c <= 'z' ) then

            and( $5f, c );

        endif;

    endif;

```



```

endif;
stdout.put
(
    "The character you entered, possibly ", nl,
    "converted to upper case, was '",
    c,
    "'", nl
);

end charInputDemo2;

```

Program 4.2 Stdin.get Character Input Sample

As you may recall from the last chapter, the HLA Standard Library buffers its input. Whenever you read a character from the standard input using *stdin.getc* or *stdin.get*, the library routines read the next available character from the buffer; if the buffer is empty, then the program reads a new line of text from the user and returns the first character from that line. If you want to guarantee that the program reads a new line of text from the user when you read a character variable, you should call the *stdin.flushInput* routine before attempting to read the character. This will flush the current input buffer and force the input of a new line of text on the next input (which should be your *stdin.getc* or *stdin.get* call).

The end of line is problematic. Different operating systems handle the end of line differently on output versus input. From the console device, pressing the ENTER key signals the end of a line; however, when reading data from a file you get an end of line sequence which is typically a line feed or a carriage return/line feed pair. To help solve this problem, HLA's Standard Library provides an "end of line" function. This procedure returns true (one) in the AL register if all the current input characters have been exhausted, it returns false (zero) otherwise. The following sample program demonstrates the use of the *stdin.eoln* function.

```

program eolnDemo2;
#include( "stdlib.hhf" );
begin eolnDemo2;

    stdout.put( "Enter a short line of text: " );
    stdin.flushInput();
    repeat

        stdin.getc();
        stdout.putc( al );
        stdout.put( "=$", al, nl );

    until( stdin.eoln() );

end eolnDemo2;

```

Program 4.3 Testing for End of Line Using Stdin.eoln

The HLA language and the HLA Standard Library provide many other procedures and additional support for character objects. Later chapters in this textbook, as well as the HLA reference documentation, describe how to use these features.

4.4.3 The ASCII Character Set

The following table lists the binary, hexadecimal, and decimal representations for each of the 128 ASCII character codes.

Table 11: ASCII Character Set

Binary	Hex	Decimal	Character
0000_0000	00	0	NULL
0000_0001	01	1	ctrl A
0000_0010	02	2	ctrl B
0000_0011	03	3	ctrl C
0000_0100	04	4	ctrl D
0000_0101	05	5	ctrl E
0000_0110	06	6	ctrl F
0000_0111	07	7	bell
0000_1000	08	8	backspace
0000_1001	09	9	tab
0000_1010	0A	10	line feed
0000_1011	0B	11	ctrl K
0000_1100	0C	12	form feed
0000_1101	0D	13	return
0000_1110	0E	14	ctrl N
0000_1111	0F	15	ctrl O
0001_0000	10	16	ctrl P
0001_0001	11	17	ctrl Q
0001_0010	12	18	ctrl R
0001_0011	13	19	ctrl S
0001_0100	14	20	ctrl T
0001_0101	15	21	ctrl U
0001_0110	16	22	ctrl V
0001_0111	17	23	ctrl W

Table 11: ASCII Character Set

Binary	Hex	Decimal	Character
0001_1000	18	24	ctrl X
0001_1001	19	25	ctrl Y
0001_1010	1A	26	ctrl Z
0001_1011	1B	27	ctrl [
0001_1100	1C	28	ctrl \
0001_1101	1D	29	Esc
0001_1110	1E	30	ctrl ^
0001_1111	1F	31	ctrl _
0010_0000	20	32	space
0010_0001	21	33	!
0010_0010	22	34	"
0010_0011	23	35	#
0010_0100	24	36	\$
0010_0101	25	37	%
0010_0110	26	38	&
0010_0111	27	39	'
0010_1000	28	40	(
0010_1001	29	41)
0010_1010	2A	42	*
0010_1011	2B	43	+
0010_1100	2C	44	,
0010_1101	2D	45	-
0010_1110	2E	46	.
0010_1111	2F	47	/
0011_0000	30	48	0
0011_0001	31	49	1
0011_0010	32	50	2
0011_0011	33	51	3

Table 11: ASCII Character Set

Binary	Hex	Decimal	Character
0011_0100	34	52	4
0011_0101	35	53	5
0011_0110	36	54	6
0011_0111	37	55	7
0011_1000	38	56	8
0011_1001	39	57	9
0011_1010	3A	58	:
0011_1011	3B	59	;
0011_1100	3C	60	<
0011_1101	3D	61	=
0011_1110	3E	62	>
0011_1111	3F	63	?
0100_0000	40	64	@
0100_0001	41	65	A
0100_0010	42	66	B
0100_0011	43	67	C
0100_0100	44	68	D
0100_0101	45	69	E
0100_0110	46	70	F
0100_0111	47	71	G
0100_1000	48	72	H
0100_1001	49	73	I
0100_1010	4A	74	J
0100_1011	4B	75	K
0100_1100	4C	76	L
0100_1101	4D	77	M
0100_1110	4E	78	N
0100_1111	4F	79	O

Table 11: ASCII Character Set

Binary	Hex	Decimal	Character
0101_0000	50	80	P
0101_0001	51	81	Q
0101_0010	52	82	R
0101_0011	53	83	S
0101_0100	54	84	T
0101_0101	55	85	U
0101_0110	56	86	V
0101_0111	57	87	W
0101_1000	58	88	X
0101_1001	59	89	Y
0101_1010	5A	90	Z
0101_1011	5B	91	[
0101_1100	5C	92	\
0101_1101	5D	93]
0101_1110	5E	94	^
0101_1111	5F	95	_
0110_0000	60	96	`
0110_0001	61	97	a
0110_0010	62	98	b
0110_0011	63	99	c
0110_0100	64	100	d
0110_0101	65	101	e
0110_0110	66	102	f
0110_0111	67	103	g
0110_1000	68	104	h
0110_1001	69	105	i
0110_1010	6A	106	j
0110_1011	6B	107	k

Table 11: ASCII Character Set

Binary	Hex	Decimal	Character
0110_1100	6C	108	l
0110_1101	6D	109	m
0110_1110	6E	110	n
0110_1111	6F	111	o
0111_0000	70	112	p
0111_0001	71	113	q
0111_0010	72	114	r
0111_0011	73	115	s
0111_0100	74	116	t
0111_0101	75	117	u
0111_0110	76	118	v
0111_0111	77	119	w
0111_1000	78	120	x
0111_1001	79	121	y
0111_1010	7A	122	z
0111_1011	7B	123	{
0111_1100	7C	124	
0111_1101	7D	125	}
0111_1110	7E	126	~
0111_1111	7F	127	

4.5 The UNICODE Character Set

Although the ASCII character set is, unquestionably, the most popular character representation on computers, it is certainly not the only format around. For example, IBM uses the EBCDIC code on many of its mainframe and minicomputer lines. Since EBCDIC appears mainly on IBM's big iron and you'll rarely encounter it on personal computer systems, we will not consider that character set in this text. Another character representation that is becoming popular on small computer systems (and large ones, for that matter) is the Unicode character set. Unicode overcomes two of ASCII's greatest limitations: the limited character space (i.e., a maximum of 128/256 characters in an eight-bit byte) and the lack of international (beyond the USA) characters.

Unicode uses a 16-bit word to represent a single character. Therefore, Unicode supports up to 65,536 different character codes. This is obviously a huge advance over the 256 possible codes we can represent with an eight-bit byte. Unicode is upwards compatible from ASCII. Specifically, if the H.O. 17 bits of a

Unicode character contain zero, then the L.O. seven bits represent the same character as the ASCII character with the same character code. If the H.O. 17 bits contain some non-zero value, then the character represents some other value. If you're wondering why so many different character codes are necessary, simply note that certain Asian character sets contain 4096 characters (at least, their Unicode subset).

This text will stick to the ASCII character set except for a few brief mentions of Unicode here and there. Unfortunately, many string algorithms are not as conveniently written for Unicode as for ASCII (especially character set functions) so we'll stick with ASCII in this text as long as possible.

4.6 Other Data Representations

Of course, we can represent many different objects other than numbers and characters in a computer system. The following subsections provide a brief description of the different real-world data types you might encounter.

4.6.1 Representing Colors on a Video Display

As you're probably aware, color images on a computer display are made up of a series of dots known as *pixels* (which is short for "picture elements."). Different display modes (depending on the capability of the display adapter) use different data representations for each of these pixels. The one thing in common between these data types is that they control the mixture of the three additive primary colors (red, green, and blue) to form a specific color on the display. The question, of course, is how much of each of these colors do they mix together?

Color depth is the term video card manufacturers use to describe how much red, green, and blue they mix together for each pixel. Modern video cards generally provide three color depths of eight, sixteen, or twenty-four bits, allowing 256, 65536, or over 16 million colors per pixel on the display. This produces images that are somewhat coarse and grainy (eight-bit images) to "Polaroid quality" (16-bit images), on up to "photographic quality" (24-bit images)¹⁰.

One problem with these color depths is that two of the three formats do not contain a number of bits that is evenly divisible by three. Therefore, in each of these formats at least one of the three primary colors will have fewer bits than the others. For example, with an eight-bit color depth, two of the colors can have three bits (or eight different shades) associated with them while one of the colors must have only two bits (or four shades). Therefore, when distributing the bits there are three formats possible: 2-3-3 (two bits red, three bits green, and three bits blue), 3-2-3, or 3-3-2. Likewise, with a 16 bit color depth, two of the three colors can have five bits while the third color can have six bits. This lets us generate three different palettes using the bit values 5-5-6, 5-6-5, or 6-5-5. For 24-bit displays, each primary color can have eight bits, so there is an even distribution of the colors for each pixel.

A 24-bit display produces amazingly good results. A 16-bit display produces okay images. Eight-bit displays, to put it bluntly, produce horrible photographic images (they do produce good synthetic images like those you would manipulate with a draw program). To produce better images when using an eight-bit display, most cards provide a hardware *palette*. A palette is nothing more than an array of 256 values containing 256 elements¹¹. The system uses the eight-bit pixel value as an index into this array of 256 values and displays the color associated with the 24-bit entry in the palette table. Although the display can still display only 256 different colors at one time, the palette mechanism lets users select exactly which colors they

10. Some graphic artists would argue that 24 bit images are not of a sufficient quality. There are some display/printer/scanner devices capable of working with 32-bit, 36-bit, and even 48-bit images; if, of course, you're willing to pay for them.

11. Actually, the color depth of each palette entry is not necessarily fixed at 24 bits. Some display devices, for example, use 18-bit entries in their palette.

want to display. For example, they could display 250 shades of blue and six shades of purple if such a mixture produces a better image for them.

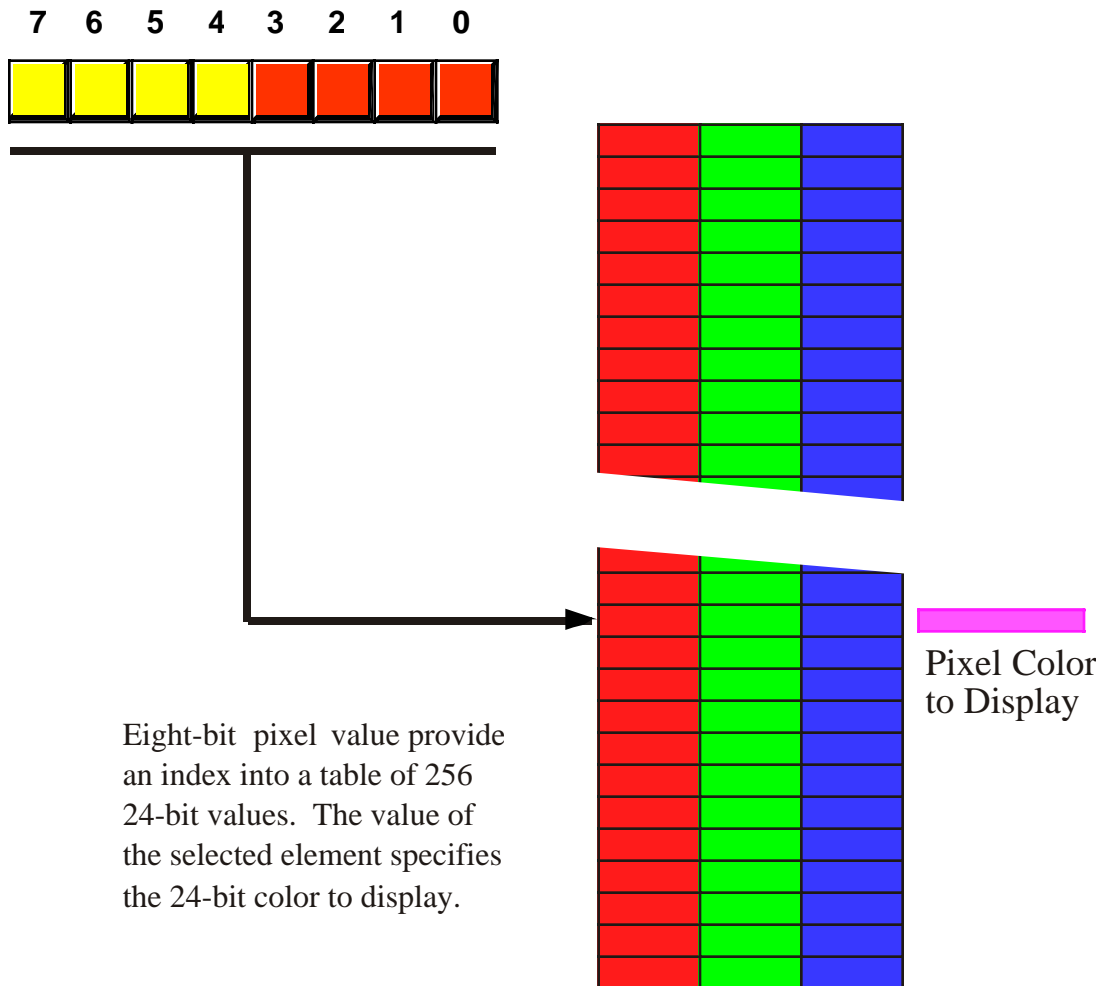


Figure 4.7 Extending the Number of Colors Using a Palette

Unfortunately, the palette scheme only works for displays with minimal color depths. For example, attempting to use a palette with 16-bit images would require a lookup table with 65,536 different three-byte entries – a bit much for today’s operating systems (since they may have to reload the palette every time you select a window on the display). Fortunately, the higher bit depths don’t require the palette concept as much as the eight-bit color depth.

Obviously, we could dream up other schemes for representing pixel color on the display. Some display systems, for example, use the subtractive primary colors (Cyan, Yellow, and Magenta, plus Black, the so-called CMYK color space). Other display system use fewer or more bits to represent the values. Some distribute the bits between various shades. Monochrome displays typically use one, four, or eight bit pixels to display various gray scales (e.g., two, sixteen, or 256 shades of gray). However, the bit organizations of this section are among the more popular in use by display adapters.

4.6.2 Representing Audio Information

Another real-world quantity you'll often find in digital form on a computer is audio information. WAV files, MP3 files, and other audio formats are quite popular on personal computers. An interesting question is "how do we represent audio information inside the computer?" While many sound formats are far too complex to discuss here (e.g., the MP3 format), it is relatively easy to represent sound using a simple sound data format (something similar to the WAV file format). In this section we'll explore a couple of possible ways to represent audio information; but before we take a look at the digital format, perhaps it's a wise idea to study the analog format first.

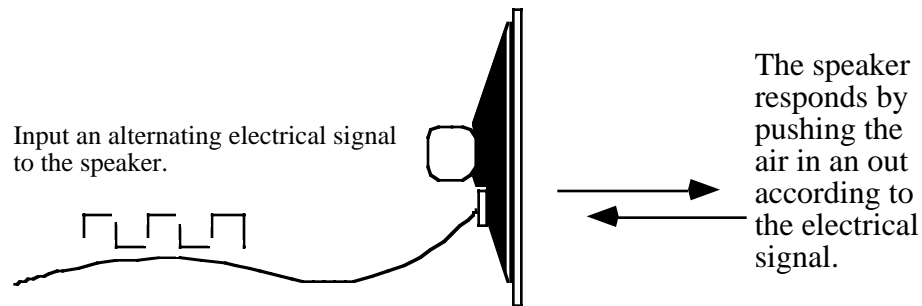
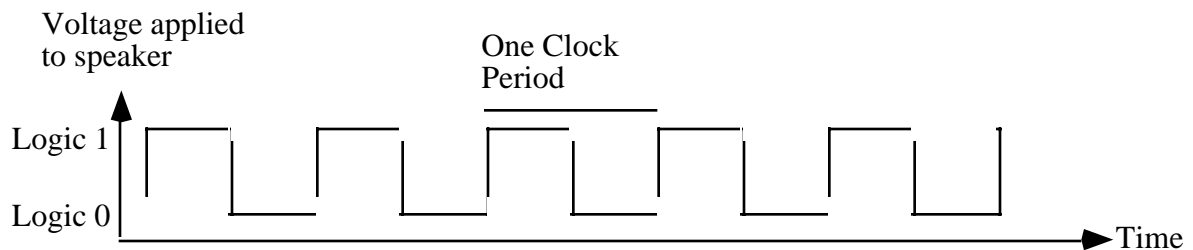


Figure 4.8 Operation of a Speaker

Sounds you hear are the result of vibrating air molecules. When air molecules quickly vibrate back and forth between 20 and 20,000 times per second, we interpret this as some sort of sound. A speaker (see Figure 4.8) is a device which vibrates air in response to an electrical signal. That is, it converts an electric signal which alternates between 20 and 20,000 times per second (Hz) to an audible tone. Alternating a signal is very easy on a computer, all you have to do is apply a logic one to an output port for some period of time and then write a logic zero to the output port for a short period. Then repeat this over and over again. A plot of this activity over time appears in Figure 4.9.



Note: Frequency is equal to the reciprocal of the clock period. Audible sounds are between 20 and 20,000 Hz.

Figure 4.9 An Audible Sound Wave

Although many humans are capable of hearing tones in the range 20-20Khz, the PC's speaker is not capable of faithfully reproducing the tones in this range. It works pretty good for sounds in the range 100-10Khz, but the volume drops off dramatically outside this range. Fortunately, most modern PCs contain a sound card that is quite capable (with appropriate external speakers) of faithfully representing "CD-Quality" sound. Of course, a good question might be "what is CD-Quality sound, anyway?" Well, to answer

that question, we've got to decide how we're going to represent sound information in a binary format (see "What is "Digital Audio" Anyway?" on page 112).

Take another look at Figure 4.9. This is a graph of amplitude (volume level) over time. If logic one corresponds to a fully extended speaker cone and logic zero corresponds to a fully retracted speaker cone, then the graph in Figure 4.9 suggests that we are constantly pushing the speaker cone in and out as time progresses. This analog data, by the way, produces what is known as a "square wave" which tends to be a very bright sound at high frequencies and a very buzzy sound at low frequencies. One advantage of a square wave tone is that we only need to alternate a single bit of data over time in order to produce a tone. This is very easy to do and very inexpensive. These two reasons are why the PC's built-in speaker (not the sound card) uses exactly this technique for producing beeps and squawks.

To produce different tones with a square wave sound system is very easy. All you've got to do is write a one and a zero to some bit connected to the speaker somewhere between 20 and 20,000 times per second. You can even produce "warbling" sounds by varying the frequency at which you write those zeros and ones to the speaker.

One easy data format we can develop to represent digitized (or, should we say, "binarized") audio data is to create a stream of bits that we feed to the speaker every $\frac{1}{40,000}$ seconds. By alternating ones and zeros in this bit stream, we get a 20 KHz tone (remember, it takes a high and a low section to give us one clock period, hence it will take two bits to produce a single cycle on the output). To get a 20 Hz tone, you would create a bit stream that alternates between 1,000 zeros and 1,000 ones. With 1,000 zeros, the speaker will remain in the retracted position for $\frac{1}{40}$ seconds, following that with 1,000 ones leaves the speaker in the fully extended position for $\frac{1}{40}$ seconds. The end result is that the speaker moves in and out 20 times a second (giving us our 20 Hz frequency). Of course, you don't have to emit a regular pattern of zeros and ones. By varying the positions of the ones and zeros in your data stream you can dramatically affect the type of sound the system will produce.

The length of your data stream will determine how long the sound plays. With 40,000 bits, the sound will play for one second (assuming each bit's duration is $\frac{1}{40,000}$ seconds). As you can see, this sound format will consume 5,000 bytes per second. This may seem like a lot, but it's relatively modest by digital audio standards.

Unfortunately, square waves are very limited with respect to the sounds they can produce and are not very high fidelity (certainly not "CD-Quality"). Real analog audio signals are much more complex and you cannot represent them with two different voltage levels on a speaker. Figure 4.10 provides a typical example

What is "Digital Audio" Anyway?

"Digital Audio" or "digitized audio" is the conventional term the consumer electronics industry uses to describe audio information encoded for use on a computer. What exactly does the term "digital" mean in this case. Historically, the term "digit" refers to a finger. A digital numbering system is one based on counting one's fingers. Traditionally, then, a "digital number" was a base ten number (since the numbering system we most commonly use is based on the ten digits with which God endowed us). In the early days of computer systems the terms "digital computer" and "binary computer" were quite prevalent, with digital computers describing decimal computer systems (i.e., BCD-based systems). Binary computers, of course, were those based on the binary numbering system. Although BCD computers are mainly an artifact in the historical dust bin, the name "digital computer" lives on and is the common term to describe all computer systems, binary or otherwise. Therefore, when people talk about the logic gates computer designers use to create computer systems, they call them "digital logic." Likewise, when they refer to computerized data (like audio data), they refer to it as "digital." Technically, the term "digital" should mean base ten, not base two. Therefore, we should really refer to "digital audio" as "binary audio" to be technically correct. However, it's a little late in the game to change this term, so "digital XXXXX" lives on. Just keep in mind that the two terms "digital audio" and "binary audio" really do mean the same thing, even though they shouldn't.

of an audio waveform. Notice that the frequency and the amplitude (the height of the signal) varies considerably over time. To capture the height of the waveform at any given point in time we will need more than two values; hence, we'll need more than a single bit.

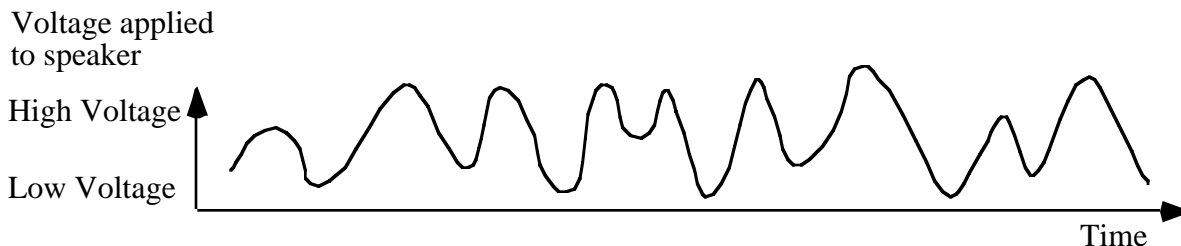


Figure 4.10 A Typical Audio Waveform

An obvious first approximation is to use a byte, rather than a single bit, to represent each point in time on our waveform. We can convert this byte data to an analog signal using a “digital to analog converter” (how obvious) or DAC. This accepts some binary number as input and produces an analog voltage on its output. This allows us to represent an impressive 256 different voltage levels in the waveform. By using eight bits, we can produce a far wider range of sounds than are possible with a single bit. Of course, our data stream now consumes 40,000 bytes per second; quite a big step up from the 5,000 bytes/second in the previous example, but still relatively modest in terms of digital audio data rates.

You might think that 256 levels would be sufficient to produce some impressive audio. Unfortunately, our hearing is logarithmic in nature and it takes an order of magnitude difference in signal for a sound to appear just a little bit louder. Therefore, our 256 different analog levels aren't as impressive to our ears. Although you can produce some decent sounds with an eight-bit data stream, it's still not high fidelity and certainly not “CD-Quality” audio.

The next obvious step up the ladder is a 16-bit value for each point of our digital audio stream. With 65,536 different analog levels we finally reach the realm of “CD-Quality” audio. Of course, we're now consuming 80,000 bytes per second to achieve this! For technical reasons, the Compact Disc format actually requires 44,100 16-bit samples per second. For a stereo (rather than monaural) data stream, you need two 16-bit values each $\frac{1}{44,100}$ seconds. This produces a whopping data rate of over 160,000 bytes per second. Now you understand the claim a littler earlier that 5,000 bytes per second is a relatively modest data rate.

Some very high quality digital audio systems use 20 or 24 bits of information and record the data at a higher frequency than 44.1 KHz (48 KHz is popular, for example). Such data formats record a better signal at the expense of a higher data rate. Some sound systems don't require anywhere near the fidelity levels of even a CD-Quality recording. Telephone conversations, for example, require only about 5,000 eight-bit samples per second (this, by the way, is why phone modems are limited to approximately 56,000 bits per second, which is about 5,000 bytes per second plus some overhead). Some common “digitizing” rates for audio include the following:

- Eight-bit samples at 11 KHz
- Eight-bit samples at 22 KHz
- Eight-bit samples at 44.1 KHz
- 16-bit samples at 32 KHz
- 16-bit samples at 44.1 KHz
- 16-bit samples at 48 KHz
- 24-bit samples at 44.1 KHz (generally in professional recording systems)
- 24-bit samples at 48 KHz (generally in professional recording systems)

The fidelity increases as you move down this list.

The exact format for various audio file formats is way beyond the scope of this text since many of the formats incorporate data compression. Some simple audio file formats like WAV and AIFF consist of little more than the digitized byte stream, but other formats are nearly indecipherable in their complexity. The exact nature of a sound data type is highly dependent upon the sound hardware in your system, so we won't delve any farther into this subject. There are several books available on computer audio and sound file formats if you're interested in pursuing this subject farther.

4.6.3 Representing Musical Information

Although it is possible to compress an audio data stream somewhat, high-quality audio will consume a large amount of data. CD-Quality audio consumes just over 160 Kilobytes per second, so a CD at 650 Megabytes holds enough data for just over an hour of audio (in stereo). Earlier, you saw that we could use a palette to allow higher quality color images on an eight-bit display. An interesting question is "can we create a sound palette to let us encode higher quality audio?" Unfortunately, the general answer is no because audio information is much less redundant than video information and you cannot produce good results with rough approximation (which using a sound palette would require). However, if you're trying to produce a specific sound, rather than trying to faithfully reproduce some recording, there are some possibilities open to you.

The advantage to the digitized audio format is that it records *everything*. In a music track, for example, the digital information records all the instruments, the vocalists, the background noise, and, well, *everything*. Sometimes you might not need to retain all this information. For example, if all you want to record is a keyboard player's synthesizer, the ability to record all the other audio information simultaneously is not necessary. In fact, with an appropriate interface to the computer, recording the audio signal from the keyboard is completely unnecessary. A far more cost-effective approach (from a memory usage point of view) is to simply record the notes the keyboardist plays (along with the duration of each note and the velocity at which the keyboardist plays the note) and then simply feed this keyboard information back to the synthesizer to play the music at a later time. Since it only takes a few bytes to record each note the keyboardist plays, and the keyboardist generally plays fewer than 100 notes per second, the amount of data needed to record a complex piece of music is tiny compared to a digitized audio recording of the same performance.

One very popular format for recording musical information in this fashion is the MIDI format (MIDI stands for Musical Instrument Digital Interface and it specifies how to connect musical instruments, computers, and other equipment together). The MIDI protocol uses multi-byte values to record information about a series of instruments (a simple MIDI file can actually control up to 16 or more instruments simultaneously).

Although the internal data format of the MIDI protocol is beyond the scope of this chapter, it is interesting to note that a MIDI command is effectively equivalent to a "palette look-up" for an audio signal. When a musical instrument receives a MIDI command telling it to play back some note, that instrument generally plays back some waveform stored in the synthesizer.

Note that you don't actually need an external keyboard/synthesizer to play back MIDI files. Most sound cards contain software that will interpret MIDI commands and play the accompanying notes. These cards definitely use the MIDI command as an index into a "wave table" (short for waveform lookup table) to play the accompanying sound. Although the quality of the sound these cards reproduce is often inferior to that a professional synthesizer produces, they do let you play MIDI files without purchasing an expensive synthesizer module¹².

If you're interested in the actual data format that MIDI uses, there are dozens of texts available on the MIDI format. Any local music store should carry several of these. You should also be able to find lots of information on MIDI on the Internet (try Roland's web site as a good starting point).

12. For those who would like a better MIDI experience using a sound card, some synthesizer manufacturers produce sound cards with an integrated synthesizer on-board.

4.6.4 Representing Video Information

Recent increases in disk space, computer speed, and network access have allowed an explosion in the popularity of multimedia on personal computers. Although the term “multimedia” suggests that the data format deals with many different types of media, most people use this term to describe digital video recording and playback on a computer system. In fact, most multimedia formats support at least two mediums: video and audio. The more popular formats like Apple’s Quicktime support other concurrent media streams as well (e.g., a separate subtitle track, time codes, and device control). To simplify matters, we limit the discussion in this section to digital video streams.

Fundamentally, a video image is nothing more than a succession of still pictures that the system displays at some rate like 30 images per second. Therefore, if we want to create a digitized video image format, all we really need to do is store 30 or so pictures for each second of video we wish to view. This may not seem like a big deal, but consider that a typical “full screen” video display has 640x480 pixels or a total of 307,200 pixels. If we use a 24-bit RGB color space, then each pixel will require three bytes, raising the total to 921,600 bytes per image. Displaying 30 of these images per second means our video format will consume 27,648,000 bytes per second. Digital audio, at 160 Kilobytes per second is virtually nothing compared to the data requirements for digital video.

Although computer systems and hard disk systems have advanced tremendously over the past decade, maintaining a 30 MByte/second data rate from disk to display is a little too much to expect from all but the most expensive workstations currently available (at least, in the year 2000 as this was written). Therefore, most multimedia systems use various techniques (or combinations of these techniques) to get the data rate down to something more reasonable. In stock computer systems, a common technique is to display a 320x240 quarter screen image rather than a full-screen 640x480 image. This reduces the data rate to about seven megabytes per second.

Another technique digital video formats use is to *compress* the video data. Video data tends to contain lots of redundant information that the system can eliminate through the use of compression. The popular DV format for digital video camcorders, for example, compresses the data stream by almost 90%, requiring only a 3.3 MByte/second data rate for full-screen video. This type of compression is not without cost. There is a detectable, though slight, loss in image quality when employing DV compression on a video image. Nevertheless, this compression makes it possible to deal with digital video data streams on a contemporary computer system. Compressed data formats are a little beyond the scope of this chapter; however, by the time you finish this text you should be well-prepared to deal with compressed data formats. Programmers writing video data compression algorithms often use assembly language because compression and decompression algorithms need to be very fast to process a video stream in real time. Therefore, keep reading this text if you’re interested in working on these types of algorithms.

4.6.5 Where to Get More Information About Data Types

Since there are many ways to represent a particular real-world object inside the computer, and nearly an infinite variety of real-world objects, this text cannot even begin to cover all the possibilities. In fact, one of the most important steps in writing a piece of computer software is to carefully consider what objects the software needs to represent and then choose an appropriate internal representation for that object. For some objects or processes, an internal representation is fairly obvious; for other objects or processes, developing an appropriate data type representation is a difficult task. Although we will continue to look at different data representations throughout this text, if you’re really interested in learning more about data representation of real world objects, activities, and processes, you should consult a good “Data Structures and Algorithms” textbook. This text does not have the space to treat these subjects properly (since it still has to teach assembly language). Most texts on data structures present their material in a high level language. Adopting this material to assembly language is not difficult, especially once you’ve digested a large percentage of this text. For something a little closer to home, you might consider reading Knuth’s “The Art of Computer Programming” that describes data structures and algorithms using a synthetic assembly language called *MIX*. Although *MIX* isn’t the same as HLA or even x86 assembly language, you will probably find it easier to

convert algorithms in this text to x86 than it would be to convert algorithms written in Pascal, Java, or C++ to assembly language.

4.7 Putting It All Together

Perhaps the most important fact this chapter and the last chapter present is that computer programs all use strings of binary bits to represent data internally. It is up to an application program to distinguish between the possible representations. For example, the bit string %0100_0001 could represent the numeric value 65, an ASCII character ('A'), or the mantissa portion of a floating point value (\$41). The CPU cannot and does not distinguish between these different representations, it simply processes this eight-bit value as a bit string and leaves the interpretation of the data to the application.

Beginning assembly language programmers often have trouble comprehending that they are responsible for interpreting the type of data found in memory; after all, one of the most important abstractions that high level languages provide is to associate a data type with a bit string in memory. This allows the compiler to do the interpretation of data representation rather than the programmer. Therefore, an important point this chapter makes is that assembly language programmers must handle this interpretation themselves. The HLA language provides built-in data types that seem to provide these abstractions, but keep in mind that once you've loaded a value into a register, HLA can no longer interpret that data for you, it is your responsibility to use the appropriate machine instructions that operate on the specified data.

One small amount of checking that HLA and the CPU does enforce is size checking - HLA will not allow you to mix sizes of operands within most instructions¹³. That is, you cannot specify a byte operand and a word operand in the same instruction that expects its two operands to be the same size. However, as the following program indicates, you can easily write a program that treats the same value as completely different types.

```

program dataInterpretation;
#include( "stdlib.hhf" );
static
    r: real32 := -1.0;

begin dataInterpretation;

    stdout.put( "`r' interpreted as a real32 value: ", r:5:2, nl );

    stdout.put( "`r' interpreted as an uns32 value: " );
    mov( r, eax );
    stdout.putu32( eax );
    stdout.newln();

    stdout.put( "`r' interpreted as an int32 value: " );
    mov( r, eax );
    stdout.puti32( eax );
    stdout.newln();

    stdout.put( "`r' interpreted as a dword value: $" );
    mov( r, eax );
    stdout.putd( eax );
    stdout.newln();

end dataInterpretation;

```

13. The sign and zero extension instructions are an obvious exception, though HLA still checks the operand sizes to ensure they are appropriate.

Program 4.4 Interpreting a Single Value as Several Different Data Types

As this sample program demonstrates, you can get completely different results by interpreting your data differently during your program's execution. So always remember, it is your responsibility to interpret the data in your program. HLA helps a little by allowing you to declare data types that are slightly more abstract than bytes, words, or double words; HLA also provides certain support routines, like `stdout.put`, that will automatically interpret these abstract data types for you; however, it is generally your responsibility to use the appropriate machine instructions to consistently manipulate memory objects according to their data type.

Volume Two: An Introduction to Machine Architecture

- Chapter One: System Organization
A gentle introduction to the components that make up a typical PC.
- Chapter Two: Memory Access and Organization
A discussion of the 80x86 memory addressing modes and how HLA organizes your data in memory.
- Chapter Three: Introduction to Digital Design
A low-level description of how computer designers build CPUs and other system components.
- Chapter Four: CPU Architecture
A look at the internal operation of the CPU.
- Chapter Five: Instruction Set Architecture
This chapter describes how Intel's engineers designed the 80x86 instruction set. It also explains many of their design decisions, good and bad.
- Chapter Six: Memory Architecture
How memory is organized for high performance computing systems.
- Chapter Seven: The I/O Subsystem
Input and output are two of the most important functions on a PC. This chapter describes how input and output occurs on a typical 80x86 system.
- Chapter Eight: Questions, Projects, and Laboratory Exercises
See what you've learned in this topic!

This topic, as its title suggests, is primarily targeted towards a machine organization course. Those who wish to study assembly language programming should at least read Chapter Two and possibly Chapter One. Chapter Three is a low-level discussion of digital logic. This information is important to those who are interested in design-

Volume Two:

Machine Architecture

ing CPUs and other system components. Those individuals who are main interested in programming can safely skip this chapter. Chapters Four, Five, and Six provide a more in-depth look at computer systems' architecture. Those wanting to know how things work "under the hood" will want to read these chapters. However, programmers who just want to learn assembly language programming can safely skip these chapters. Chapter Seven discusses I/O on the 80x86. Under modern 32-bit operating systems you will not be able to utilize much of this information unless you are writing device drivers. However, those interested in learning how low-level I/O takes place in assembly language will want to read this chapter.

System Organization

Chapter One

To write even a modest 80x86 assembly language program requires considerable familiarity with the 80x86 family. To write *good* assembly language programs requires a strong knowledge of the underlying hardware. Unfortunately, the underlying hardware is not consistent. Techniques that are crucial for 8088 programs may not be useful on Pentium systems. Likewise, programming techniques that provide big performance boosts on the Pentium chip may not help at all on an 80486. Fortunately, some programming techniques work well no matter which microprocessor you're using. This chapter discusses the effect hardware has on the performance of computer software.

1.1 Chapter Overview

This chapter describes the basic components that make up a computer system: the CPU, memory, I/O, and the bus that connects them. Although you can write software that is ignorant of these concepts, high performance software requires a complete understanding of this material. This chapter also discusses the 80x86 memory addressing modes and how you access memory data from your programs.

This chapter begins by discussing bus organization and memory organization. These two hardware components will probably have a bigger performance impact on your software than the CPU's speed. Understanding the organization of the system bus will allow you to design data structures and algorithms that operate at maximum speed. Similarly, knowing about memory performance characteristics, data locality, and cache operation can help you design software that runs as fast as possible. Of course, if you're not interested in writing code that runs as fast as possible, you can skip this discussion; however, most people do care about speed at one point or another, so learning this information is useful.

With the generic hardware issues out of the way, this chapter then discusses the program-visible components of the memory architecture - specifically the 80x86 addressing modes and how a program can access memory. In addition to the addressing modes, this chapter introduces several new 80x86 instructions that are quite useful for manipulating memory. This chapter also presents several new HLA Standard Library calls you can use to allocate and deallocate memory.

Some might argue that this chapter gets too involved with computer architecture. They feel such material should appear in an architectural book, not an assembly language programming book. This couldn't be farther from the truth! Writing *good* assembly language programs requires a strong knowledge of the architecture. Hence the emphasis on computer architecture in this chapter.

1.2 The Basic System Components

The basic operational design of a computer system is called its *architecture*. John Von Neumann, a pioneer in computer design, is given credit for the architecture of most computers in use today. For example, the 80x86 family uses the *Von Neumann architecture* (VNA). A typical Von Neumann system has three major components: the *central processing unit* (or *CPU*), *memory*, and *input/output* (or *I/O*). The way a system designer combines these components impacts system performance (See Figure 1.1).

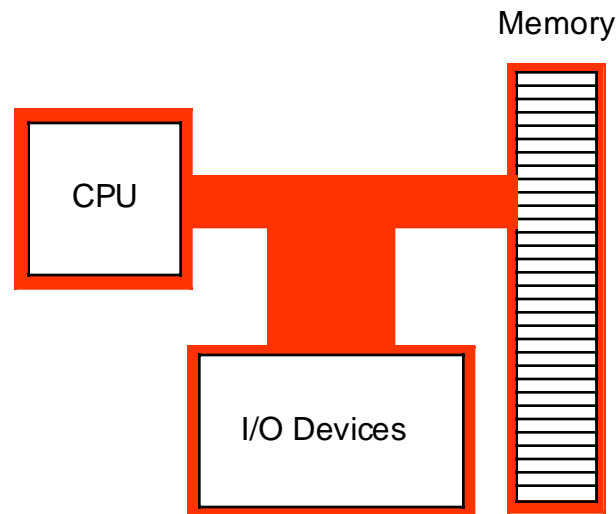


Figure 1.1 Typical Von Neumann Machine

In VNA machines, like the 80x86 family, the CPU is where all the action takes place. All computations occur inside the CPU. Data and machine instructions reside in memory until required by the CPU. To the CPU, most I/O devices look like memory because the CPU can store data to an output device and read data from an input device. The major difference between memory and I/O locations is the fact that I/O locations are generally associated with external devices in the outside world.

1.2.1 The System Bus

The *system bus* connects the various components of a VNA machine. The 80x86 family has three major busses: the *address bus*, the *data bus*, and the *control bus*. A bus is a collection of wires on which electrical signals pass between components in the system. These busses vary from processor to processor. However, each bus carries comparable information on all processors; e.g., the data bus may have a different implementation on the 80386 than on the 8088, but both carry data between the processor, I/O, and memory.

A typical 80x86 system component uses *standard TTL logic levels*¹. This means each wire on a bus uses a standard voltage level to represent zero and one². We will always specify zero and one rather than the electrical levels because these levels vary on different processors (especially laptops).

1.2.1.1 The Data Bus

The 80x86 processors use the *data bus* to shuffle data between the various components in a computer system. The size of this bus varies widely in the 80x86 family. Indeed, this bus defines the “size” of the processor.

Every modern x86 CPU from the Pentium on up employs a 64-bit wide data bus. Some of the earlier processors used 8-bit, 16-bit, or 32-bit data busses, but such machines are sufficiently obsolete that we do not need to consider them here..

1. Actually, newer members of the family tend to use lower voltage signals, but these remain compatible with TTL signals.

2. TTL logic represents the value zero with a voltage in the range 0.0-0.8v. It represents a one with a voltage in the range 2.4-5v. If the signal on a bus line is between 0.8v and 2.4v, it's value is indeterminate. Such a condition should only exist when a bus line is changing from one state to the other.

You'll often hear a processor called an *eight, 16, 32, or 64 bit processor*. While there is a mild controversy concerning the size of a processor, most people now agree that the minimum of either the number of data lines on the processor or the size of the largest general purpose integer register determines the processor size. The modern x86 CPUs all have 64-bit busses, but only provide 32-bit general purpose integer registers, so most people classify these devices as 32-bit processors.

Although the 80x86 family members with eight, 16, 32, and 64 bit data busses *can* process data up to the width of the bus, they can also access smaller memory units of eight, 16, or 32 bits. Therefore, anything you can do with a small data bus can be done with a larger data bus as well; the larger data bus, however, may access memory faster and can access larger chunks of data in one memory operation. You'll read about the exact nature of these memory accesses a little later (see "The Memory Subsystem" on page 140).

1.2.1.2 The Address Bus

The data bus on an 80x86 family processor transfers information between a particular memory location or I/O device and the CPU. The only question is, "Which memory location or I/O device?" The address bus answers that question. To differentiate memory locations and I/O devices, the system designer assigns a unique memory address to each memory element and I/O device. When the software wants to access some particular memory location or I/O device, it places the corresponding address on the address bus. Circuitry associated with the memory or I/O device recognizes this address and instructs the memory or I/O device to read the data from or place data on to the data bus. In either case, all other memory locations ignore the request. Only the device whose address matches the value on the address bus responds.

With a single address line, a processor could create exactly two unique addresses: zero and one. With n address lines, the processor can provide 2^n unique addresses (since there are 2^n unique values in an n -bit binary number). Therefore, the number of bits on the address bus will determine the *maximum* number of addressable memory and I/O locations. Early x86 processors, for example, provided only 20 bit address busses. Therefore, they could only access up to 1,048,576 (or 2^{20}) memory locations. Larger address busses can access more memory.

Table 12: 80x86 Family Address Bus Sizes

Processor	Address Bus Size	Max Addressable Memory	In English!
8088, 8086, 80186, 80188	20	1,048,576	One Megabyte
80286, 80386sx	24	16,777,216	Sixteen Megabytes
80386dx	32	4,294,976,296	Four Gigabytes
80486, Pentium	32	4,294,976,296	Four Gigabytes
Pentium Pro, II, III, IV	36	68,719,476,736	64 Gigabytes

Future 80x86 processors (e.g., the AMD "Hammer") will probably support 40, 48, and 64-bit address busses. The time is coming when most programmers will consider four gigabytes of storage to be too small, much like they consider one megabyte insufficient today. (There was a time when one megabyte was considered far more than anyone would ever need!).

1.2.1.3 The Control Bus

The control bus is an eclectic collection of signals that control how the processor communicates with the rest of the system. Consider for a moment the data bus. The CPU sends data to memory and receives data from memory on the data bus. This prompts the question, "Is it sending or receiving?" There are two lines on

the control bus, *read* and *write*, which specify the direction of data flow. Other signals include system clocks, interrupt lines, status lines, and so on. The exact make up of the control bus varies among processors in the 80x86 family. However, some control lines are common to all processors and are worth a brief mention.

The *read* and *write* control lines control the direction of data on the data bus. When both contain a logic one, the CPU and memory-I/O are not communicating with one another. If the read line is low (logic zero), the CPU is reading data from memory (that is, the system is transferring data from memory to the CPU). If the write line is low, the system transfers data from the CPU to memory.

The *byte enable lines* are another set of important control lines. These control lines allow 16, 32, and 64 bit processors to deal with smaller chunks of data. Additional details appear in the next section.

The 80x86 family, unlike many other processors, provides two distinct address spaces: one for memory and one for I/O. While the memory address busses on various 80x86 processors vary in size, the I/O address bus on all 80x86 CPUs is 16 bits wide. This allows the processor to address up to 65,536 different I/O *locations*. As it turns out, most devices (like the keyboard, printer, disk drives, etc.) require more than one I/O location. Nonetheless, 65,536 I/O locations are more than sufficient for most applications. The original IBM PC design only allowed the use of 1,024 of these.

Although the 80x86 family supports two address spaces, it does not have two address busses (for I/O and memory). Instead, the system shares the address bus for both I/O and memory addresses. Additional control lines decide whether the address is intended for memory or I/O. When such signals are active, the I/O devices use the address on the L.O. 16 bits of the address bus. When inactive, the I/O devices ignore the signals on the address bus (the memory subsystem takes over at that point).

1.2.2 The Memory Subsystem

A typical 80x86 processor addresses a maximum of 2^n different memory locations, where n is the number of bits on the address bus³. As you've seen already, 80x86 processors have 20, 24, 32, and 36 bit address busses (with 64 bits on the way).

Of course, the first question you should ask is, "What exactly is a memory location?" The 80x86 supports *byte addressable memory*. Therefore, the basic memory unit is a byte. So with 20, 24, 32, and 36 address lines, the 80x86 processors can address one megabyte, 16 megabytes, four gigabytes, and 64 gigabytes of memory, respectively.

Think of memory as a linear array of bytes. The address of the first byte is zero and the address of the last byte is $2^n - 1$. For an 8088 with a 20 bit address bus, the following pseudo-Pascal array declaration is a good approximation of memory:

Memory: array [0..1048575] of byte;

To execute the equivalent of the Pascal statement "Memory [125] := 0;" the CPU places the value zero on the data bus, the address 125 on the address bus, and asserts the write line (since the CPU is writing data to memory), see Figure 1.2.

3. This is the *maximum*. Most computer systems built around 80x86 family do not include the maximum addressable amount of memory.

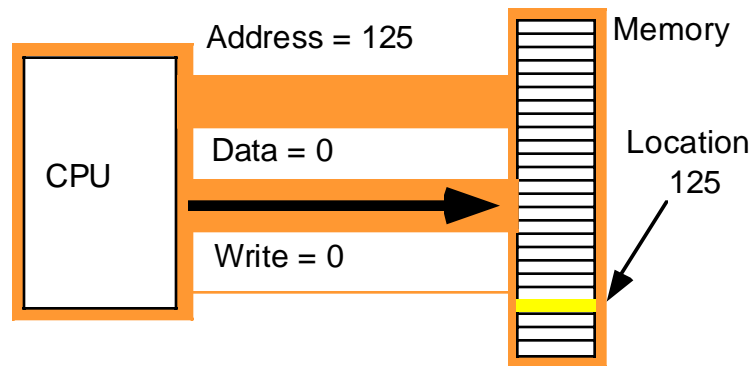


Figure 1.2 Memory Write Operation

To execute the equivalent of “CPU := Memory [125];” the CPU places the address 125 on the address bus, asserts the read line (since the CPU is reading data from memory), and then reads the resulting data from the data bus (see Figure 1.3).

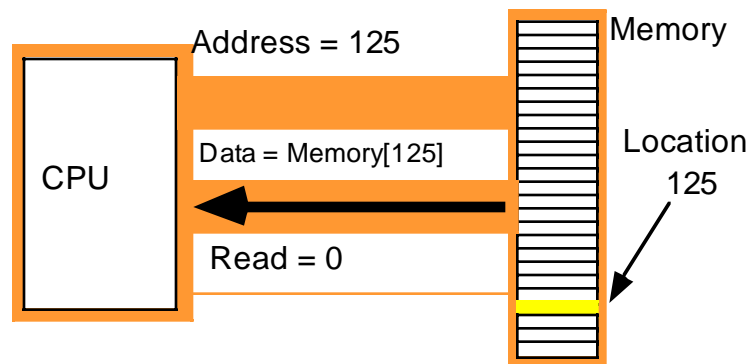


Figure 1.3 Memory Read Operation

The above discussion applies *only* when accessing a single byte in memory. So what happens when the processor accesses a word or a double word? Since memory consists of an array of bytes, how can we possibly deal with values larger than eight bits?

Different computer systems have different solutions to this problem. The 80x86 family deals with this problem by storing the L.O. byte of a word at the address specified and the H.O. byte at the next location. Therefore, a word consumes two consecutive memory addresses (as you would expect, since a word consists of two bytes). Similarly, a double word consumes four consecutive memory locations. The address for the double word is the address of its L.O. byte. The remaining three bytes follow this L.O. byte, with the H.O. byte appearing at the address of the double word *plus three* (see Figure 1.4). Bytes, words, and double words may begin at *any* valid address in memory. We will soon see, however, that starting larger objects at an arbitrary address is not a good idea.

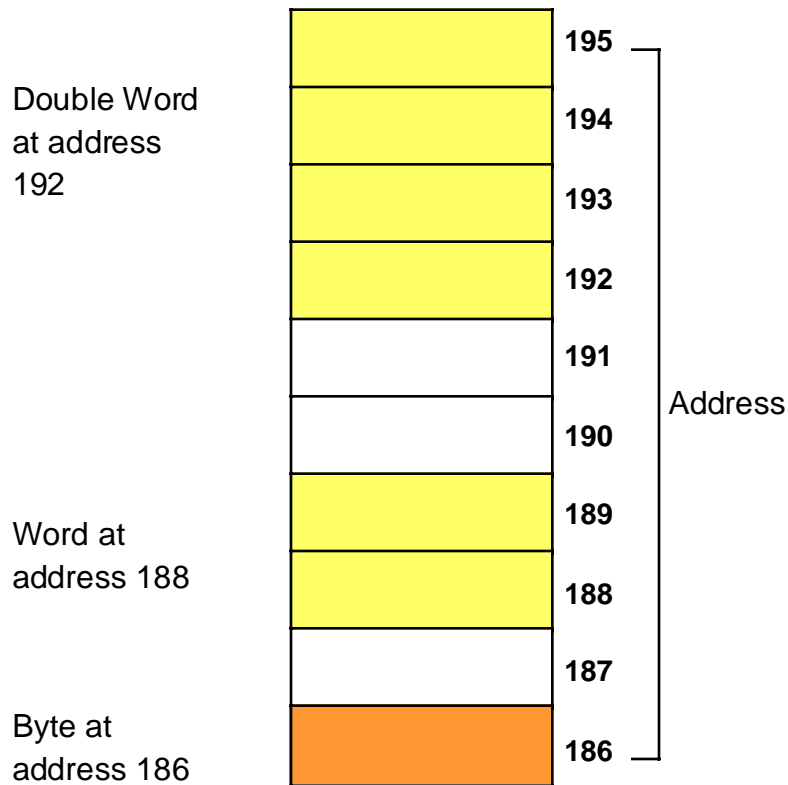


Figure 1.4 Byte, Word, and DWord Storage in Memory

Note that it is quite possible for byte, word, and double word values to overlap in memory. For example, in Figure 1.4 you could have a word variable beginning at address 193, a byte variable at address 194, and a double word value beginning at address 192. These variables would all overlap.

A processor with an eight-bit bus (like the old 8088 CPU) can transfer eight bits of data at a time. Since each memory address corresponds to an eight bit byte, this turns out to be the most convenient arrangement (from the hardware perspective), see Figure 1.5.

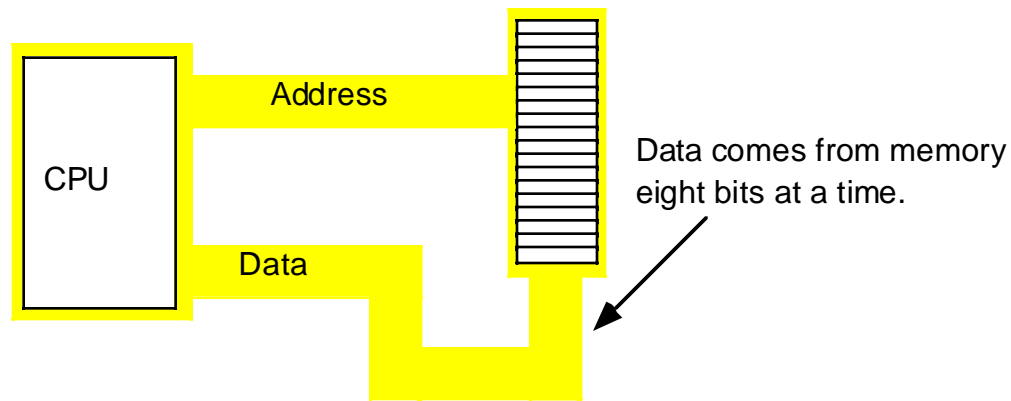


Figure 1.5 Eight-Bit CPU <-> Memory Interface

The term “byte addressable memory array” means that the CPU can address memory in chunks as small as a single byte. It also means that this is the *smallest* unit of memory you can access at once with the processor. That is, if the processor wants to access a four bit value, it must read eight bits and then ignore the extra four bits. Also realize that byte addressability does not imply that the CPU can access eight bits on any arbitrary bit boundary. When you specify address 125 in memory, you get the entire eight bits at that address, nothing less, nothing more. Addresses are integers; you cannot, for example, specify address 125.5 to fetch fewer than eight bits.

CPUs with an eight-bit bus can manipulate word and double word values, even through their data bus is only eight bits wide. However, this requires multiple memory operations because these processors can only move eight bits of data at once. To load a word requires two memory operations; to load a double word requires four memory operations.

Some older x86 CPUs (e.g., the 8086 and 80286) have a 16 bit data bus. This allows these processors to access twice as much memory in the same amount of time as their eight bit brethren. These processors organize memory into two *banks*: an “even” bank and an “odd” bank (see Figure 1.6). Figure 1.7 illustrates the connection to the CPU (D0-D7 denotes the L.O. byte of the data bus, D8-D15 denotes the H.O. byte of the data bus):

	Even	Odd
Word 3	6	7
Word 2	4	5
Word 1	2	3
Word 0	0	1

Numbers in cells represent the byte addresses

Figure 1.6 Byte Addressing in Word Memory

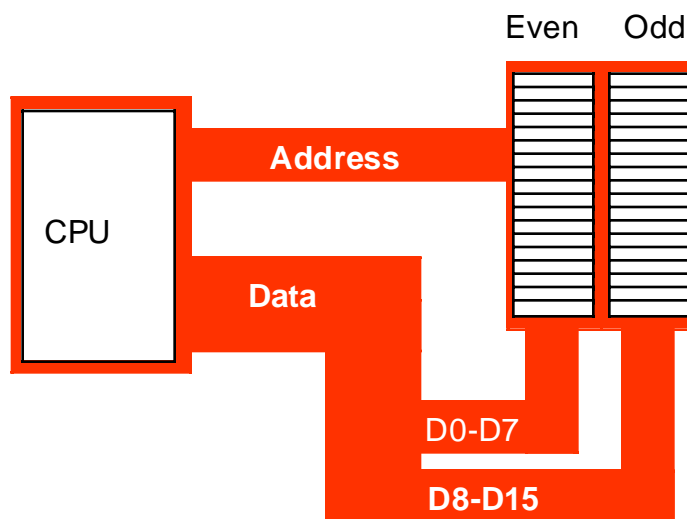


Figure 1.7 Sixteen-Bit Processor (8086, 80186, 80286, 80386sx) Memory Organization

The 16 bit members of the 80x86 family can load a word from any arbitrary address. As mentioned earlier, the processor fetches the L.O. byte of the value from the address specified and the H.O. byte from the next consecutive address. This creates a subtle problem if you look closely at the diagram above. What happens when you access a word on an odd address? Suppose you want to read a word from location 125. Okay, the L.O. byte of the word comes from location 125 and the H.O. word comes from location 126. What's the big deal? It turns out that there are two problems with this approach.

First, look again at Figure 1.7. Data bus lines eight through 15 (the H.O. byte) connect to the odd bank, and data bus lines zero through seven (the L.O. byte) connect to the even bank. Accessing memory location 125 will transfer data to the CPU on the H.O. byte of the data bus; yet we want this data in the L.O. byte! Fortunately, the 80x86 CPUs recognize this situation and automatically transfer the data on D8-D15 to the L.O. byte.

The second problem is even more obscure. When accessing words, we're really accessing two separate bytes, each of which has its own byte address. So the question arises, "What address appears on the address bus?" The 16 bit 80x86 CPUs always place even addresses on the bus. Even bytes always appear on data lines D0-D7 and the odd bytes always appear on data lines D8-D15. If you access a word at an even address, the CPU can bring in the entire 16 bit chunk in one memory operation. Likewise, if you access a single byte,

the CPU activates the appropriate bank (using a “byte enable” control line). If the byte appeared at an odd address, the CPU will automatically move it from the H.O. byte on the bus to the L.O. byte.

So what happens when the CPU accesses a *word* at an odd address, like the example given earlier? Well, the CPU cannot place the address 125 onto the address bus and read the 16 bits from memory. There are no odd addresses coming out of a 16 bit 80x86 CPU. The addresses are always even. So if you try to put 125 on the address bus, this will put 124 on to the address bus. Were you to read the 16 bits at this address, you would get the word at addresses 124 (L.O. byte) and 125 (H.O. byte) – not what you’d expect. Accessing a word at an odd address requires two memory operations. First the CPU must read the byte at address 125, then it needs to read the byte at address 126. Finally, it needs to swap the positions of these bytes internally since both entered the CPU on the wrong half of the data bus.

Fortunately, the 16 bit 80x86 CPUs hide these details from you. Your programs can access words at *any* address and the CPU will properly access and swap (if necessary) the data in memory. However, to access a word at an odd address requires two memory operations (just like the 8088/80188). Therefore, accessing words at odd addresses on a 16 bit processor is slower than accessing words at even addresses. **By carefully arranging how you use memory, you can improve the speed of your program on these CPUs.**

Accessing 32 bit quantities always takes at least two memory operations on the 16 bit processors. If you access a 32 bit quantity at an odd address, a 16-bit processor will require three memory operations to access the data.

The 80x86 processors with a 32-bit data bus (e.g., the 80386 and 80486) use four banks of memory connected to the 32 bit data bus (see Figure 1.8).

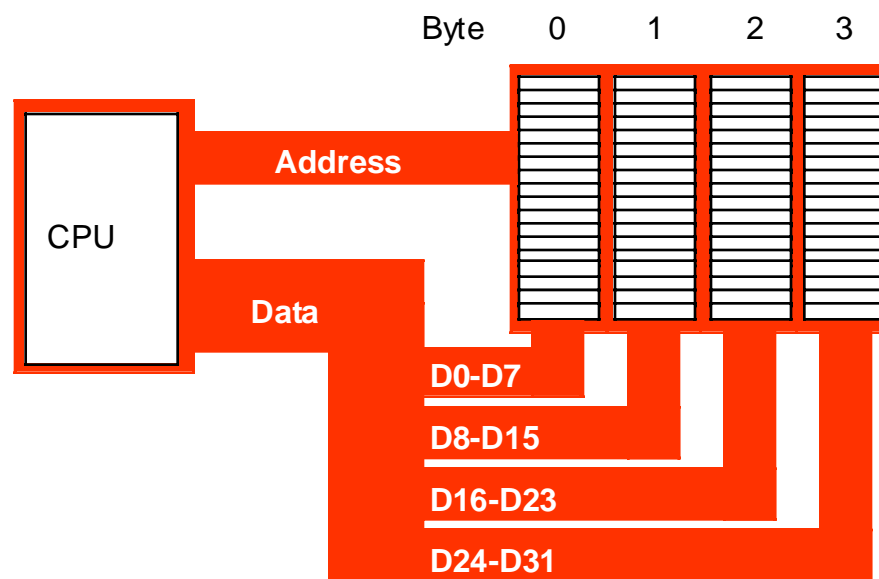


Figure 1.8 32-Bit Processor (80386, 80486, Pentium Overdrive) Memory Organization

The address placed on the address bus is always some multiple of four. Using various “byte enable” lines, the CPU can select which of the four bytes at that address the software wants to access. As with the 16 bit processor, the CPU will automatically rearrange bytes as necessary.

With a 32 bit memory interface, the 80x86 CPU can access any byte with one memory operation. If (address MOD 4) does not equal three, then a 32 bit CPU can access a word at that address using a single memory operation. However, if the remainder is three, then it will take two memory operations to access that word (see Figure 1.9). This is the same problem encountered with the 16 bit processor, except it occurs half as often.

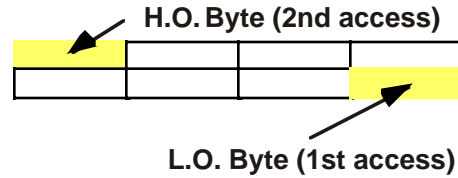


Figure 1.9 Accessing a Word at (Address mod 4) = 3.

A 32 bit CPU can access a double word in a single memory operation *if* the address of that value is evenly divisible by four. If not, the CPU will require two memory operations.

Once again, the CPU handles all of this automatically. In terms of loading correct data the CPU handles everything for you. However, there is a performance benefit to proper data alignment. As a general rule you should always place word values at even addresses and double word values at addresses which are evenly divisible by four. This will speed up your program.

The Pentium and later processors provide a 64-bit data bus and special cache memory that reduces the impact of non-aligned data access. Although there may still be a penalty for accessing data at an inappropriate address, modern x86 CPUs suffer from the problem less frequently than the earlier CPUs. The discussion of cache memory in a later chapter will discuss the details.

1.2.3 The I/O Subsystem

Besides the 20, 24, or 32 address lines which access memory, the 80x86 family provides a 16 bit I/O address bus. This gives the 80x86 CPUs two separate address spaces: one for memory and one for I/O operations. Lines on the control bus differentiate between memory and I/O addresses. Other than separate control lines and a smaller bus, I/O addressing behaves exactly like memory addressing. Memory and I/O devices both share the same data bus and the L.O. 16 lines on the address bus.

There are three limitations to the I/O subsystem on the PC: first, the 80x86 CPUs require special instructions to access I/O devices; second, the designers of the PC used the “best” I/O locations for their own purposes, forcing third party developers to use less accessible locations; third, 80x86 systems can address no more than 65,536 (2^{16}) I/O addresses. When you consider that a typical video display card requires over eight megabytes of addressable locations, you can see a problem with the size of I/O bus.

Fortunately, hardware designers can map their I/O devices into the memory address space as easily as they can the I/O address space. So by using the appropriate circuitry, they can make their I/O devices look just like memory. This is how, for example, display adapters on the PC work.

1.3 HLA Support for Data Alignment

In order to write the fastest running programs, you need to ensure that your data objects are properly aligned in memory. Data becomes misaligned whenever you allocate storage for different sized objects in adjacent memory locations. Since it is nearly impossible to write a (large) program that uses objects that are all the same size, some other facility is necessary in order to realign data that would normally be unaligned in memory.

Consider the following HLA variable declarations:

```

static
    dw:    dword;
    b:     byte;
    w:     word;
    dw2:   dword;
    w2:    word;
    b2:    byte;
    dw3:   dword;

```

The first static declaration in a program (running under Windows, Linux, and most 32-bit operating systems) places its variables at an address that is an even multiple of 4096 bytes. Since 4096 is a power of two, whatever variable first appears in the static declaration is guaranteed to be aligned on a reasonable address. Each successive variable is allocated at an address that is the sum of the sizes of all the preceding variables plus the starting address. Therefore, assuming the above variables are allocated at a starting address of 4096, then each variable will be allocated at the following addresses:

		// Start Adrs	Length
dw:	dword;	// 4096	4
b:	byte;	// 4100	1
w:	word;	// 4101	2
dw2:	dword;	// 4103	4
w2:	word;	// 4107	2
b2:	byte;	// 4109	1
dw3:	dword;	// 4110	4

With the exception of the first variable (which is aligned on a 4K boundary) and the byte variables (whose alignment doesn't matter), all of these variables are misaligned in memory. The *w*, *w2*, and *dw2* variables are aligned on odd addresses and the *dw3* variable is aligned on an even address that is not an even multiple of four.

An easy way to guarantee that your variables are aligned on an appropriate address is to put all the dword variables first, the word variables second, and the byte variables last in the declaration:

```

static
    dw:    dword;
    dw2:   dword;
    dw3:   dword;
    w:     word;
    w2:    word;
    b:     byte;
    b2:    byte;

```

This organization produces the following addresses in memory (again, assuming the first variable is allocated at address 4096):

		// Start Adrs	Length
dw:	dword;	// 4096	4
dw2:	dword;	// 4100	4
dw3:	dword;	// 4104	4
w:	word;	// 4108	2
w2:	word;	// 4110	2
b:	byte;	// 4112	1
b2:	byte;	// 4113	1

As you can see, these variables are all aligned at reasonable addresses.

Unfortunately, it is rarely possible for you to arrange your variables in this manner. While there are lots of technical reasons that make this alignment impossible, a good practical reason for not doing this is because it doesn't let you organize your variable declarations by logical function (that is, you probably want to keep related variables next to one another regardless of their size).

To resolve this problem, HLA provides two solutions. The first is an alignment option whenever you encounter a *static* section. If you follow the *static* keyword by an integer constant inside parentheses, HLA will align the very next variable declaration at an address that is an even multiple of the specified constant, e.g.,

```
static( 4 )
    dw:    dword;
    b:     byte;
    w:     word;
    dw2:   dword;
    w2:    word;
    b2:    byte;
    dw3:   dword;
```

Of course, if you have only a single *static* section in your entire program, this declaration doesn't buy you much because the first declaration in the section is already aligned on a 4096 byte boundary. However, HLA does allow you to put multiple *static* sections into your program, so you can specify an alignment constant for each *static* section:

```
static( 4 )
    dw:    dword;
    b:     byte;

static( 2 )
    w:     word;

static( 4 )
    dw2:   dword;
    w2:    word;
    b2:    byte;

static( 4 )
    dw3:   dword;
```

This particular sequence guarantees that all double word variables are aligned on addresses that are multiples of four and all word variables are aligned on even addresses (note that a special section was not created for *w2* since its address is going to be an even multiple of four).

While the alignment parameter to the *static* directive is useful on occasion, there are two problems with it: The first problem is that inserting so many *static* directives into the middle of your variable declarations tends to disrupt the readability of your variable declarations. Part of this problem can be overcome by simply placing a *static* directive before every variable declaration:

```
static( 4 )    dw:    dword;
static( 1 )    b:     byte;
static( 2 )    w:     word;
static( 4 )    dw2:   dword;
static( 2 )    w2:    word;
static( 1 )    b2:    byte;
static( 4 )    dw3:   dword;
```

While this approach can, arguably, make a program easier to read, it certainly involves more typing and it doesn't address the second problem: variables appearing in separate *static* sections are not guaranteed to be allocated in adjacent memory locations. Once in a while it is very important to ensure that two variables are allocated in adjacent memory cells and most programmers assume that variables declared next to one another in the source code are allocated in adjacent memory cells. The mechanism above does not guarantee this.

The second facility HLA provides to help align adjacent memory locations is the *align* directive. The *align* directive uses the following syntax:

```
align( integer_constant );
```

The integer constant must be one of the following small unsigned integer values: 1, 2, 4, 8, or 16. If HLA encounters the *align* directive in a *static* section, it will align the very next variable on an address that is an even multiple of the specified alignment constant. The previous example could be rewritten, using the *align* directive, as follows:

```
static( 4 )
    dw:    dword;
    b:     byte;
    align( 2 );
    w:     word;
    align( 4 );
    dw2:   dword;
    w2:    word;
    b2:    byte;
    align( 4 );
    dw3:   dword;
```

If you're wondering how the *align* directive works, it's really quite simple. If HLA determines that the current address is not an even multiple of the specified value, HLA will quietly emit extra bytes of padding after the previous variable declaration until the current address in the *static* section is an even multiple of the specified value. This has the effect of making your program slightly larger (by a few bytes) in exchange for faster access to your data; Given that your program will only grow by a small number of bytes when you use this feature, this is a good trade off.

1.4 System Timing

Although modern computers are quite fast and getting faster all the time, they still require a finite amount of time to accomplish even the smallest tasks. On Von Neumann machines like the 80x86, most operations are *serialized*. This means that the computer executes commands in a prescribed order. It wouldn't do, for example, to execute the statement `I:=I*5+2;` before `I:=J;` in the following sequence:

```
I := J;
I := I * 5 + 2;
```

Clearly we need some way to control which statement executes first and which executes second.

Of course, on real computer systems, operations do not occur instantaneously. Moving a copy of `J` into `I` takes a certain amount of time. Likewise, multiplying `I` by five and then adding two and storing the result back into `I` takes time. As you might expect, the second Pascal statement above takes quite a bit longer to execute than the first. For those interested in writing fast software, a natural question to ask is, "How does the processor execute statements, and how do we measure how long they take to execute?"

The CPU is a very complex piece of circuitry. Without going into too many details, let us just say that operations inside the CPU must be very carefully coordinated or the CPU will produce erroneous results. To ensure that all operations occur at just the right moment, the 80x86 CPUs use an alternating signal called the *system clock*.

1.4.1 The System Clock

At the most basic level, the *system clock* handles all synchronization within a computer system. The system clock is an electrical signal on the control bus which alternates between zero and one at a periodic rate (see Figure 1.10). All activity within the CPU is synchronized with the edges (rising or falling) of this clock signal.

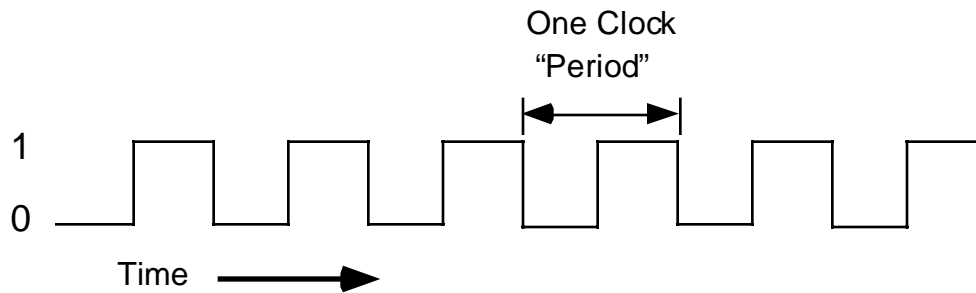


Figure 1.10 The System Clock

The frequency with which the system clock alternates between zero and one is the *system clock frequency*. The time it takes for the system clock to switch from zero to one and back to zero is the *clock period*. One full period is also called a *clock cycle*. On most modern systems, the system clock switches between zero and one at rates exceeding several hundred million times per second to several billion times per second. The clock frequency is simply the number of clock cycles which occur each second. A typical Pentium IV chip, circa 2002, runs at speeds of 2 billion cycles per second or faster. “Hertz” (Hz) is the technical term meaning one cycle per second. Therefore, the aforementioned Pentium chip runs at 2000 million hertz, or 2000 megahertz (MHz), also known as two gigahertz. Typical frequencies for 80x86 parts range from 5 MHz up to several Gigahertz (GHz, or billions of cycles per second) and beyond. Note that one clock period (the amount of time for one complete clock cycle) is the reciprocal of the clock frequency. For example, a 1 MHz clock would have a clock period of one microsecond ($1/1,000,000^{\text{th}}$ of a second). Likewise, a 10 MHz clock would have a clock period of 100 nanoseconds (100 billionths of a second). A CPU running at 1 GHz would have a clock period of one nanosecond. Note that we usually express clock periods in millionths or billionths of a second.

To ensure synchronization, most CPUs start an operation on either the *falling edge* (when the clock goes from one to zero) or the *rising edge* (when the clock goes from zero to one). The system clock spends most of its time at either zero or one and very little time switching between the two. Therefore clock edge is the perfect synchronization point.

Since all CPU operations are synchronized around the clock, the CPU cannot perform tasks any faster than the clock. However, just because a CPU is running at some clock frequency doesn’t mean that it is executing that many operations each second. Many operations take multiple clock cycles to complete so the CPU often performs operations at a significantly lower rate.

1.4.2 Memory Access and the System Clock

Memory access is one of the most common CPU activities. Memory access is definitely an operation synchronized around the system clock or some submultiple of the system clock. That is, reading a value from memory or writing a value to memory occurs no more often than once every clock cycle. Indeed, on many 80x86 processors, it takes several clock cycles to access a memory location. The *memory access time* is the number of clock cycles the system requires to access a memory location; this is an important value since longer memory access times result in lower performance..

Memory access time is the amount of time between a memory operation request (read or write) and the time the memory operation completes. Modern x86 CPUs are so much faster than memory that systems built around these CPUs often use a second clock, the bus clock, that is some sub-multiple of the CPU speed. For example, typical processors in the 100 MHz to 2 GHz range use 400MHz, 133MHz, 100MHz, or 66 MHz bus clocks (often, the bus speed is selectable on the CPU).

When reading from memory, the memory access time is the amount of time from the point that the CPU places an address on the address bus and the CPU takes the data off the data bus. On typical x86 CPU with a

one cycle memory access time, a read looks something like shown in Figure 1.11. Writing data to memory is similar (see Figure 1.12).

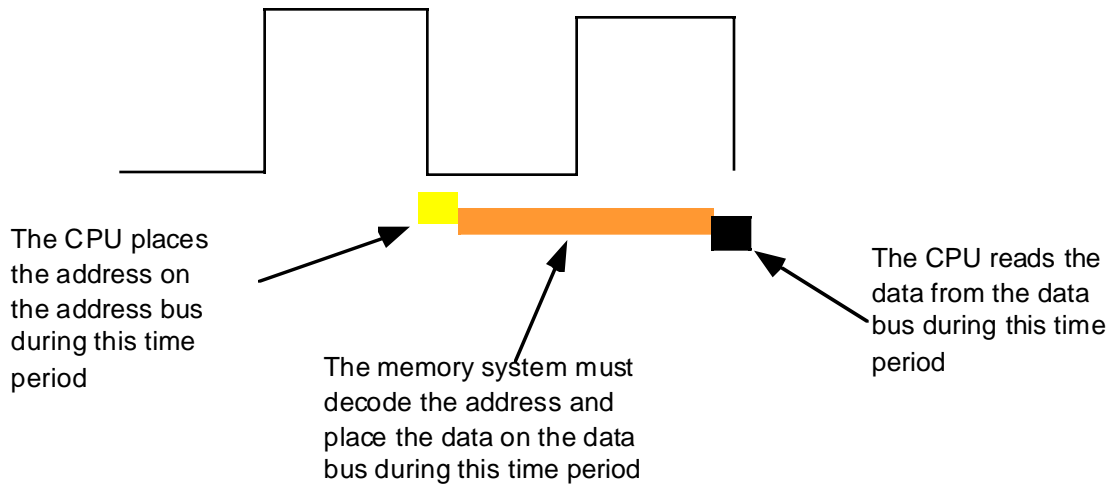


Figure 1.11 The 80x86 Memory Read Cycle

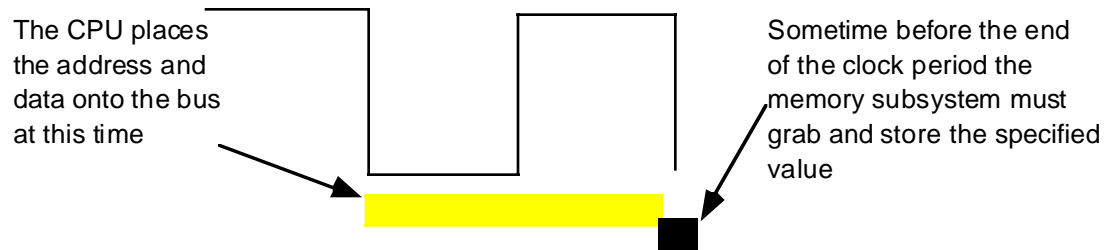


Figure 1.12 The 80x86 Memory Write Cycle

Note that the CPU doesn't wait for memory. The access time is specified by the bus clock frequency. If the memory subsystem doesn't work fast enough, the CPU will read garbage data on a memory read operation and will not properly store the data on a memory write operation. This will surely cause the system to fail.

Memory devices have various ratings, but the two major ones are capacity and speed (access time). Typical dynamic RAM (random access memory) devices have capacities of 512 (or more) megabytes and speeds of 0.25-100 ns. You can buy bigger or faster devices, but they are much more expensive. A typical 2 GHz Pentium system uses 2.5 ns (400 MHz) memory devices.

Wait just a second here! At 2 GHz the clock period is roughly 0.5 ns. How can a system designer get away with using 2.5 ns memory? The answer is *wait states*.

1.4.3 Wait States

A wait state is nothing more than an extra clock cycle to give some device time to complete an operation. For example, a 100 MHz Pentium system has a 10 ns clock period. This implies that you need 10 ns memory. In fact, the situation is worse than this. In most computer systems there is additional circuitry

between the CPU and memory: decoding and buffering logic. This additional circuitry introduces additional delays into the system (see Figure 1.13). In this diagram, the system loses 10ns to buffering and decoding. So if the CPU needs the data back in 10 ns, the memory must respond in less than 0 ns (which is impossible).

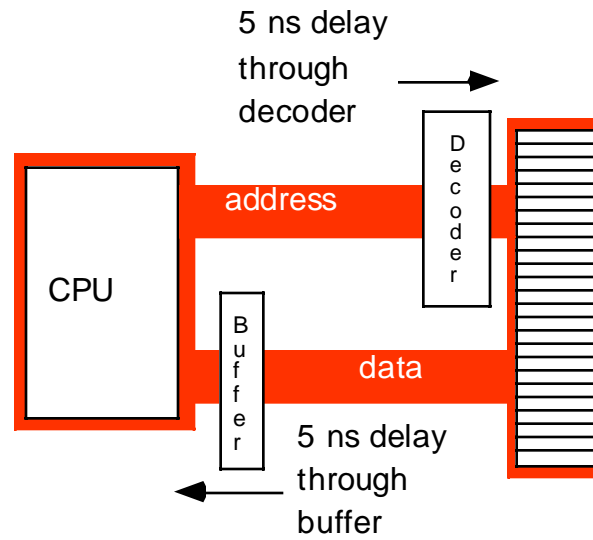


Figure 1.13 Decoding and Buffer Delays

If cost-effective memory won't work with a fast processor, how do companies manage to sell fast PCs? One part of the answer is the wait state. For example, if you have a 2 GHz processor with a memory cycle time of 0.5 ns and you lose 2 ns to buffering and decoding, you'll need 2.5 ns memory. What if your system can only support 10 ns memory (i.e., a 100 MHz system bus)? Adding three wait states to extend the memory cycle to 10 ns (one bus clock cycle) will solve this problem.

Almost every general purpose CPU in existence provides a signal on the control bus to allow the insertion of wait states. Generally, the decoding circuitry asserts this line to delay one additional clock period, if necessary. This gives the memory sufficient access time, and the system works properly (see Figure 1.14).

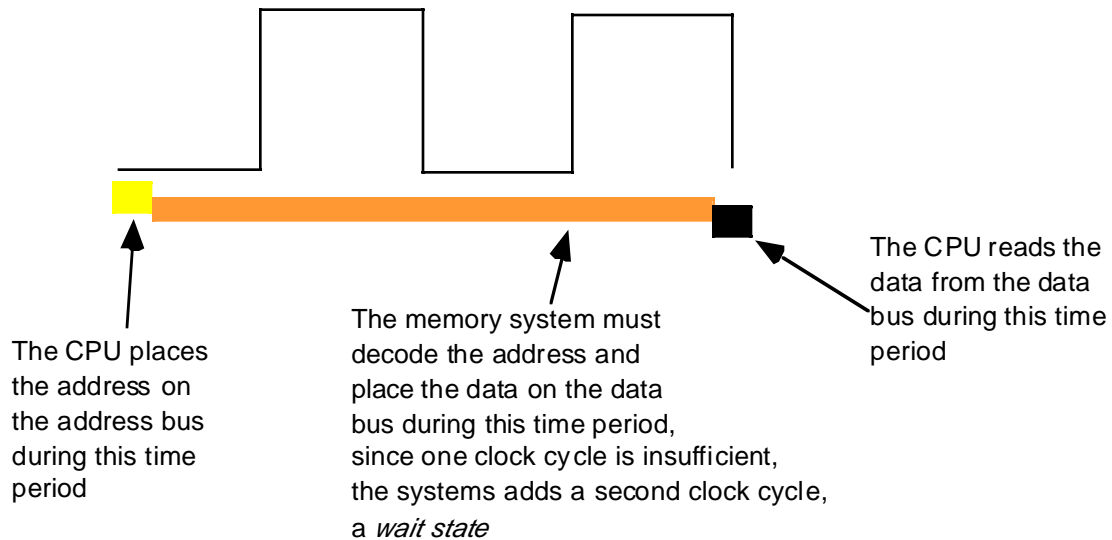


Figure 1.14 Inserting a Wait State into a Memory Read Operation

Needless to say, from the system performance point of view, wait states are *not* a good thing. While the CPU is waiting for data from memory it cannot operate on that data. Adding a single wait state to a memory cycle on a typical CPU *doubles* the amount of time required to access the data. This, in turn, *halves* the speed of the memory access. Running with a wait state on every memory access is almost like cutting the processor clock frequency in half. You're going to get a lot less work done in the same amount of time.

However, we're not doomed to slow execution because of added wait states. There are several tricks hardware designers can play to achieve zero wait states *most* of the time. The most common of these is the use of *cache* (pronounced "cash") memory.

1.4.4 Cache Memory

If you look at a typical program (as many researchers have), you'll discover that it tends to access the same memory locations repeatedly. Furthermore, you also discover that a program often accesses adjacent memory locations. The technical names given to this phenomenon are *temporal locality of reference* and *spatial locality of reference*. When exhibiting spatial locality, a program accesses neighboring memory locations. When displaying temporal locality of reference a program repeatedly accesses the same memory location during a short time period. Both forms of locality occur in the following Pascal code segment:

```
for i := 0 to 10 do
  A [i] := 0;
```

There are two occurrences each of spatial and temporal locality of reference within this loop. Let's consider the obvious ones first.

In the Pascal code above, the program references the variable *i* several times. The for loop compares *i* against 10 to see if the loop is complete. It also increments *i* by one at the bottom of the loop. The assignment statement also uses *i* as an array index. This shows temporal locality of reference in action since the CPU accesses *i* at three points in a short time period.

This program also exhibits spatial locality of reference. The loop itself zeros out the elements of array *A* by writing a zero to the first location in *A*, then to the second location in *A*, and so on. Assuming that Pascal stores the elements of *A* into consecutive memory locations, each loop iteration accesses adjacent memory locations.

There is an additional example of temporal and spatial locality of reference in the Pascal example above, although it is not so obvious. Computer instructions that tell the system to do the specified task also reside in memory. These instructions appear sequentially in memory – the spatial locality part. The computer also executes these instructions repeatedly, once for each loop iteration – the temporal locality part.

If you look at the execution profile of a typical program, you'd discover that the program typically executes less than half the statements. Generally, a typical program might only use 10-20% of the memory allotted to it. At any one given time, a one megabyte program might only access four to eight kilobytes of data and code. So if you paid an outrageous sum of money for expensive zero wait state RAM, you wouldn't be using most of it at any one given time! Wouldn't it be nice if you could buy a small amount of fast RAM and dynamically reassign its address(es) as the program executes?

This is exactly what cache memory does for you. Cache memory sits between the CPU and main memory. It is a small amount of very fast (zero wait state) memory. Unlike normal memory, the bytes appearing within a cache do not have fixed addresses. Instead, cache memory can reassign the address of a data object. This allows the system to keep recently accessed values in the cache. Addresses that the CPU has never accessed or hasn't accessed in some time remain in main (slow) memory. Since most memory accesses are to recently accessed variables (or to locations near a recently accessed location), the data generally appears in cache memory.

Cache memory is not perfect. Although a program may spend considerable time executing code in one place, eventually it will call a procedure or wander off to some section of code outside cache memory. In such an event the CPU has to go to main memory to fetch the data. Since main memory is slow, this will require the insertion of wait states.

A cache *hit* occurs whenever the CPU accesses memory and finds the data in the cache. In such a case the CPU can usually access data with zero wait states. A cache *miss* occurs if the CPU accesses memory and the data is not present in cache. Then the CPU has to read the data from main memory, incurring a performance loss. To take advantage of locality of reference, the CPU copies data into the cache whenever it accesses an address not present in the cache. Since it is likely the system will access that same location shortly, the system will save wait states by having that data in the cache.

As described above, cache memory handles the temporal aspects of memory access, but not the spatial aspects. Caching memory locations *when you access them* won't speed up the program if you constantly access consecutive locations (spatial locality of reference). To solve this problem, most caching systems read several consecutive bytes from memory when a cache miss occurs⁴. 80x86 CPUs, for example, read between 16 and 64 bytes at a shot (depending upon the CPU) upon a cache miss. If you read 16 bytes, why read them in blocks rather than as you need them? As it turns out, most memory chips available today have special modes which let you quickly access several consecutive memory locations on the chip. The cache exploits this capability to reduce the average number of wait states needed to access memory.

If you write a program that randomly accesses memory, using a cache might actually slow you down. Reading 16 bytes on each cache miss is expensive if you only access a few bytes in the corresponding cache line. Nonetheless, cache memory systems work quite well in the average case.

It should come as no surprise that the ratio of cache hits to misses increases with the size (in bytes) of the cache memory subsystem. The 80486 chip, for example, has 8,192 bytes of on-chip cache. Intel claims to get an 80-95% hit rate with this cache (meaning 80-95% of the time the CPU finds the data in the cache). This sounds very impressive. However, if you play around with the numbers a little bit, you'll discover it's not all *that* impressive. Suppose we pick the 80% figure. Then one out of every five memory accesses, on the average, will not be in the cache. If you have a 50 MHz processor and a 90 ns memory access time, four out of five memory accesses require only one clock cycle (since they are in the cache) and the fifth will require about 10 wait states⁵. Altogether, the system will require 15 clock cycles to access five memory locations,

4. Engineers call this block of data a cache *line*.

5. Ten wait states were computed as follows: five clock cycles to read the first four bytes (10+20+20+20+20=90). However, the cache always reads 16 consecutive bytes. Most memory subsystems let you read consecutive addresses in about 40 ns after accessing the first location. Therefore, the 80486 will require an additional six clock cycles to read the remaining three double words. The total is 11 clock cycles or 10 wait states.

or three clock cycles per access, on the average. That's equivalent to two wait states added to every memory access. Doesn't sound as impressive, does it?

There are a couple of ways to improve the situation. First, you can add more cache memory. This improves the cache hit ratio, reducing the number of wait states. For example, increasing the hit ratio from 80% to 90% lets you access 10 memory locations in 20 cycles. This reduces the average number of wait states per memory access to one wait state in our 80486 example – a substantial improvement. Alas, you can't pull an 80486 chip apart and solder more cache onto the chip. However, modern Pentium CPUs have a significantly larger cache than the 80486 and operates with fewer average wait states.

Another way to improve performance is to build a *two-level* caching system. Many 80486 systems work in this fashion. The first level is the on-chip 8,192 byte cache. The next level, between the on-chip cache and main memory, is a secondary cache built on the computer system circuit board (see Figure 1.15). Pentiums and later chips typically move the secondary cache onto the same chip carrier as the CPU (that is, Intel's designers have included the secondary cache as part of the CPU module).

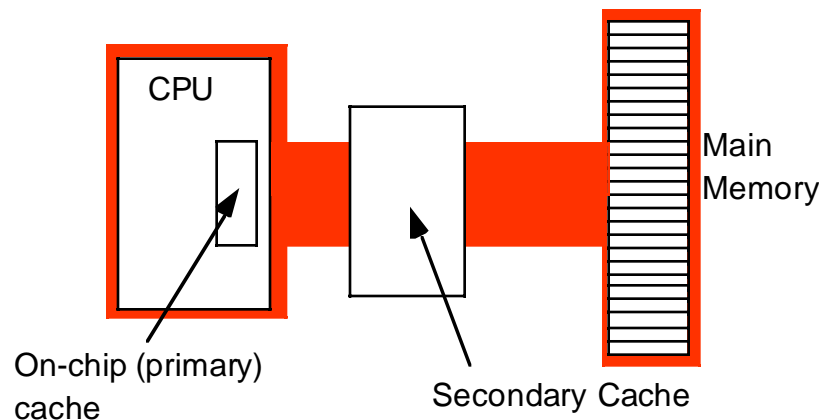


Figure 1.15 A Two Level Caching System

A typical secondary cache contains anywhere from 32,768 bytes to one megabyte of memory. Common sizes on PC subsystems are 256K, 512K, and 1024 Kbytes (1 MB) of cache.

You might ask, “Why bother with a two-level cache? Why not use a 262,144 byte cache to begin with?” Well, the secondary cache generally does not operate at zero wait states. The circuitry to support 262,144 bytes fast memory would be *very* expensive. So most system designers use slower memory which requires one or two wait states. This is still *much* faster than main memory. Combined with the on-chip cache, you can get better performance from the system.

Consider the previous example with an 80% hit ratio. If the secondary cache requires two cycles for each memory access and three cycles for the first access, then a cache miss on the on-chip cache will require a total of six clock cycles. All told, the average system performance will be two clocks per memory access. Quite a bit faster than the three required by the system without the secondary cache. Furthermore, the secondary cache can update its values in parallel with the CPU. So the number of cache misses (which affect CPU performance) goes way down.

You're probably thinking, “So far this all sounds interesting, but what does it have to do with programming?” Quite a bit, actually. By writing your program carefully to take advantage of the way the cache memory system works, you can improve your program's performance. By co-locating variables you commonly use together in the same cache line, you can force the cache system to load these variables as a group, saving extra wait states on each access.

If you organize your program so that it tends to execute the same sequence of instructions repeatedly, it will have a high degree of temporal locality of reference and will, therefore, execute faster.

1.5 Putting It All Together

This chapter has provided a quick overview of the components that make up a typical computer system. The remaining chapters in this volume will expand upon these comments to give you a complete overview of computer system organization.

Memory Access and Organization

Chapter Two

2.1 Chapter Overview

In earlier chapters you saw how to declare and access simple variables in an assembly language program. In this chapter you will learn how the 80x86 CPUs actually access memory (e.g., variables). You will also learn how to efficiently organize your variable declarations so the CPU can access them faster. In this chapter you will also learn about the 80x86 stack and how to manipulate data on the stack with some 80x86 instructions this chapter introduces. Finally, you will learn about dynamic memory allocation.

2.2 The 80x86 Addressing Modes

The 80x86 processors let you access memory in many different ways. Until now, you've only seen a single way to access a variable, the so-called *displacement-only* addressing mode that you can use to access scalar variables. Now it's time to look at the many different ways that you can access memory on the 80x86.

The 80x86 *memory addressing modes* provide flexible access to memory, allowing you to easily access variables, arrays, records, pointers, and other complex data types. Mastery of the 80x86 addressing modes is the first step towards mastering 80x86 assembly language.

When Intel designed the original 8086 processor, they provided it with a flexible, though limited, set of memory addressing modes. Intel added several new addressing modes when it introduced the 80386 microprocessor. Note that the 80386 retained all the modes of the previous processors. However, in 32-bit environments like Win32, BeOS, and Linux, these earlier addressing modes are not very useful; indeed, HLA doesn't even support the use of these older, 16-bit only, addressing modes. Fortunately, anything you can do with the older addressing modes can be done with the new addressing modes as well (even better, as a matter of fact). Therefore, you won't need to bother learning the old 16-bit addressing modes on today's high-performance processors. Do keep in mind, however, that if you intend to work under MS-DOS or some other 16-bit operating system, you will need to study up on those old addressing modes.

2.2.1 80x86 Register Addressing Modes

Most 80x86 instructions can operate on the 80x86's general purpose register set. By specifying the name of the register as an operand to the instruction, you may access the contents of that register. Consider the 80x86 MOV (move) instruction:

```
mov( source, destination );
```

This instruction copies the data from the *source* operand to the *destination* operand. The eight-bit, 16-bit, and 32-bit registers are certainly valid operands for this instruction. The only restriction is that both operands must be the same size. Now let's look at some actual 80x86 MOV instructions:

```
mov( bx, ax ); // Copies the value from BX into AX
mov( al, dl ); // Copies the value from AL into DL
mov( edx, esi ); // Copies the value from EDX into ESI
mov( bp, sp ); // Copies the value from BP into SP
mov( cl, dh ); // Copies the value from CL into DH
mov( ax, ax ); // Yes, this is legal!
```

Remember, the registers are the best place to keep often used variables. As you'll see a little later, instructions using the registers are shorter and faster than those that access memory. Throughout this chapter you'll see the abbreviated operands *reg* and *r/m* (register/memory) used wherever you may use one of the 80x86's general purpose registers.

2.2.2 80x86 32-bit Memory Addressing Modes

The 80x86 provides hundreds of different ways to access memory. This may seem like quite a bit at first, but fortunately most of the addressing modes are simple variants of one another so they're very easy to learn. And learn them you should! The key to good assembly language programming is the proper use of memory addressing modes.

The addressing modes provided by the 80x86 family include displacement-only, base, displacement plus base, base plus indexed, and displacement plus base plus indexed. Variations on these five forms provide the many different addressing modes on the 80x86. See, from 256 down to five. It's not so bad after all!

2.2.2.1 The Displacement Only Addressing Mode

The most common addressing mode, and the one that's easiest to understand, is the *displacement-only* (or *direct*) addressing mode. The displacement-only addressing mode consists of a 32-bit constant that specifies the address of the target location. Assuming that variable *J* is an *int8* variable allocated at address \$8088, the instruction “`mov(J, al);`” loads the AL register with a copy of the byte at memory location \$8088. Likewise, if *int8* variable *K* is at address \$1234 in memory, then the instruction “`mov(dl, K);`” stores the value in the DL register to memory location \$1234 (see Figure 2.1).

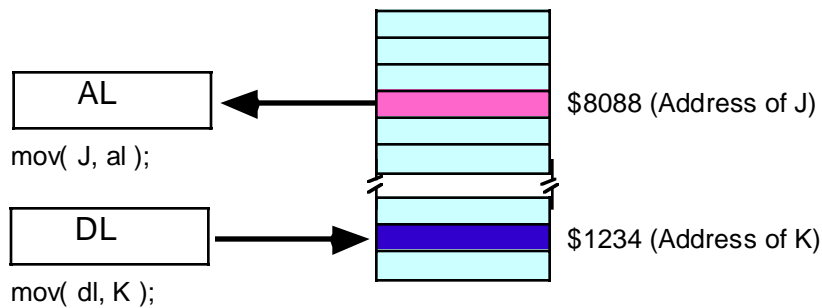


Figure 2.1 Displacement Only (Direct) Addressing Mode

The displacement-only addressing mode is perfect for accessing simple scalar variables.

Intel named this the displacement-only addressing mode because a 32-bit constant (displacement) follows the MOV opcode in memory. On the 80x86 processors, this displacement is an offset from the beginning of memory (that is, address zero). The examples in this chapter will typically access bytes in memory. Don't forget, however, that you can also access words and double words on the 80x86 processors (see Figure 2.2).

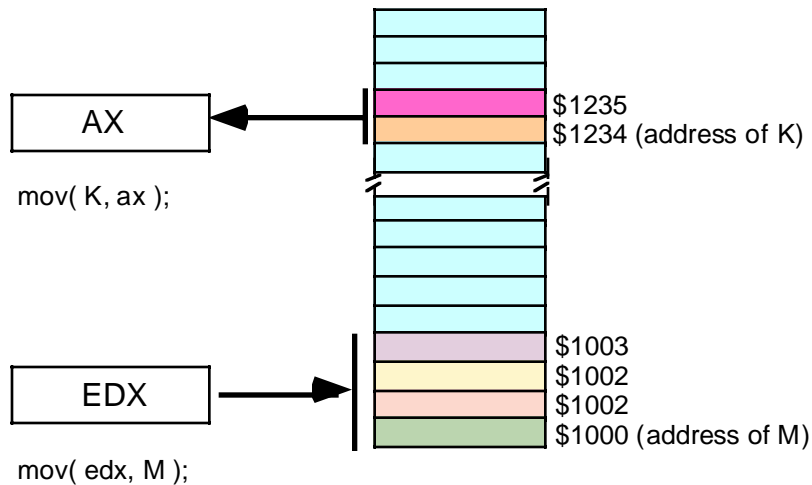


Figure 2.2 Accessing a Word or DWord Using the Displacement Only Addressing Mode

2.2.2.2 The Register Indirect Addressing Modes

The 80x86 CPUs let you access memory indirectly through a register using the register indirect addressing modes. The term indirect means that the operand is not the actual address, but rather, the operand's value specifies the memory address to use. In the case of the register indirect addressing modes, the register's value is the memory location to access. For example, the instruction “mov(eax, [ebx]);” tells the CPU to store EAX's value at the location whose address is in EBX (the square brackets around EBX tell HLA to use the register indirect addressing mode).

There are eight forms of this addressing mode on the 80x86, best demonstrated by the following instructions:

```
mov( [eax], al );
mov( [ebx], al );
mov( [ecx], al );
mov( [edx], al );
mov( [edi], al );
mov( [esi], al );
mov( [ebp], al );
mov( [esp], al );
```

These eight addressing modes reference the memory location at the offset found in the register enclosed by brackets (EAX, EBX, ECX, EDX, EDI, ESI, EBP, or ESP, respectively).

Note that the register indirect addressing modes require a 32-bit register. You cannot specify a 16-bit or eight-bit register when using an indirect addressing mode¹. Technically, you could load a 32-bit register with an arbitrary numeric value and access that location indirectly using the register indirect addressing mode:

```
mov( $1234_5678, ebx );
mov( [ebx], al );           // Attempts to access location $1234_5678.
```

Unfortunately (or fortunately, depending on how you look at it), this will probably cause the operating system to generate a protection fault since it's not always legal to access arbitrary memory locations.

1. Actually, the 80x86 does support addressing modes involving certain 16-bit registers, as mentioned earlier. However, HLA does not support these modes and they are not particularly useful under 32-bit operating systems.

The register indirect addressing mode has lots of uses. You can use it to access data referenced by a pointer, you can use it to step through array data, and, in general, you can use it whenever you need to modify the address of a variable while your program is running.

The register indirect addressing mode provides an example of a *anonymous* variable. When using the register indirect addressing mode you refer to the value of a variable by its numeric memory address (e.g., the value you load into a register) rather than by the name of the variable. Hence the phrase anonymous variable.

HLA provides a simple operator that you can use to take the address of a `STATIC` variable and put this address into a 32-bit register. This is the “&” (address of) operator (note that this is the same symbol that C/C++ uses for the address-of operator). The following example loads the address of variable *J* into `EBX` and then stores the value in `EAX` into *J* using the register indirect addressing mode:

```
mov( &J, ebx );           // Load address of J into EBX.
mov( eax, [ebx] );       // Store EAX into J.
```

Of course, it would have been simpler to store the value in `EAX` directly into *J* rather than using two instructions to do this indirectly. However, you can easily imagine a code sequence where the program loads one of several different addresses into `EBX` prior to the execution of the “`mov(eax, [ebx]);`” statement, thus storing `EAX` into one of several different locations depending on the execution path of the program.

Warning: the “&” (address-of) operator is not a general address-of operator like the “&” operator in C/C++. You may only apply this operator to static variables². It cannot be applied to generic address expressions or other types of variables. For more information on taking the address of such objects, see “Obtaining the Address of a Memory Object” on page 191.

2.2.2.3 Indexed Addressing Modes

The indexed addressing modes use the following syntax:

```
mov( VarName[ eax ], al );
mov( VarName[ ebx ], al );
mov( VarName[ ecx ], al );
mov( VarName[ edx ], al );
mov( VarName[ edi ], al );
mov( VarName[ esi ], al );
mov( VarName[ ebp ], al );
mov( VarName[ esp ], al );
```

VarName is the name of some variable in your program.

The indexed addressing mode computes an *effective address*³ by adding the address of the specified variable to the value of the 32-bit register appearing inside the square brackets. This sum is the actual address in memory that the instruction will access. So if *VarName* is at address \$1100 in memory and `EBX` contains eight, then “`mov(VarName[ebx], al);`” loads the byte at address \$1108 into the `AL` register (see Figure 2.3).

2. Note: the term “static” here indicates a `STATIC`, `READONLY`, or `STORAGE` object.

3. The effective address is the ultimate address in memory that an instruction will access, once all the address calculations are complete.

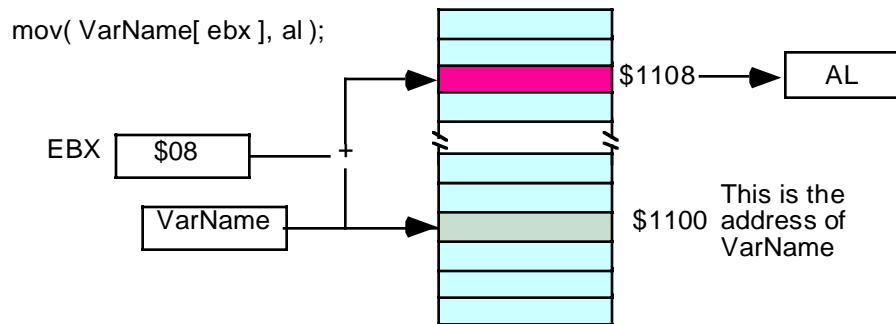


Figure 2.3 Indexed Addressing Mode

The indexed addressing mode is really handy for accessing elements of arrays. You will see how to use this addressing mode for that purpose a little later in this text. A little later in this chapter you will see how to use the indexed addressing mode to step through data values in a table.

2.2.2.4 Variations on the Indexed Addressing Mode

There are two important syntactical variations of the indexed addressing mode. Both forms generate the same basic machine instructions, but their syntax suggests other uses for these variants.

The first variant uses the following syntax:

```
mov( [ ebx + constant ], al );
mov( [ ebx - constant ], al );
```

These examples use only the EBX register. However, you can use any of the other 32-bit general purpose registers in place of EBX. This addressing mode computes its effective address by adding the value in EBX to the specified constant, or subtracting the specified constant from EBX (See Figure 2.4 and Figure 2.5).

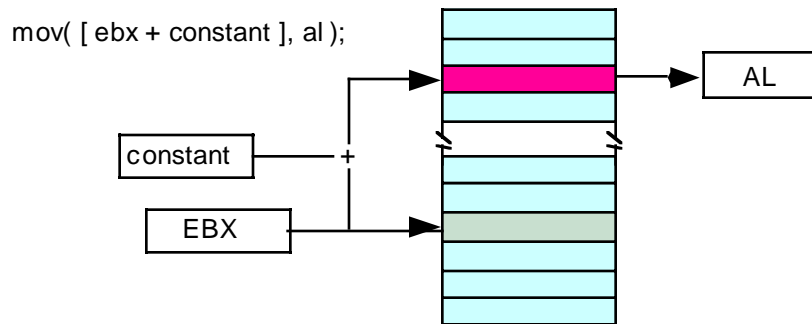


Figure 2.4 Indexed Addressing Mode Using a Register Plus a Constant

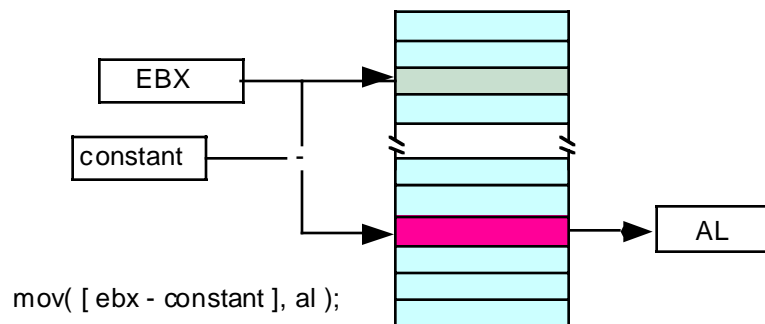


Figure 2.5 Indexed Addressing Mode Using a Register Minus a Constant

This particular variant of the addressing mode is useful if a 32-bit register contains the *base address* of a multi-byte object and you wish to access a memory location some number of bytes before or after that location. One important use of this addressing mode is accessing fields of a record (or structure) when you have a pointer to the record data. You'll see a little later in this text that this addressing mode is also invaluable for accessing automatic (local) variables in procedures.

The second variant of the indexed addressing mode is actually a combination of the previous two forms. The syntax for this version is the following:

```
mov( VarName[ ebx + constant ], al );
mov( VarName[ ebx - constant ], al );
```

Once again, this example uses only the EBX register. You may, however, substitute any of the 32-bit general purpose registers in place of EBX in these two examples. This particular form is quite useful when accessing elements of an array of records (structures) in an assembly language program (more on that in a few chapters).

These instructions compute their effective address by adding or subtracting the *constant* value from *VarName* and then adding the value in EBX to this result. Note that HLA, not the CPU, computes the sum or difference of *VarName* and *constant*. The actual machine instructions above contain a single constant value that the instructions add to the value in EBX at run-time. Since HLA substitutes a constant for *VarName*, it can reduce an instruction of the form

```
mov( VarName[ ebx + constant ], al );
```

to an instruction of the form:

```
mov( constant1[ ebx + constant2], al );
```

Because of the way these addressing modes work, this is semantically equivalent to

```
mov( [ebx + (constant1 + constant2)], al );
```

HLA will add the two constants together at compile time, effectively producing the following instruction:

```
mov( [ebx + constant_sum], al );
```

So, HLA converts the first addressing mode of this sequence to the last in this sequence.

Of course, there is nothing special about subtraction. You can easily convert the addressing mode involving subtraction to addition by simply taking the two's complement of the 32-bit constant and then adding this complemented value (rather than subtracting the uncomplemented value). Other transformations are equally possible and legal. The end result is that these three variations on the indexed addressing mode are indeed equivalent.

2.2.2.5 Scaled Indexed Addressing Modes

The scaled indexed addressing modes are similar to the indexed addressing modes with two differences: (1) the scaled indexed addressing modes allow you to combine two registers plus a displacement, and (2) the scaled indexed addressing modes let you multiply the index register by a (scaling) factor of one, two, four, or eight. The allowable forms for these addressing modes are

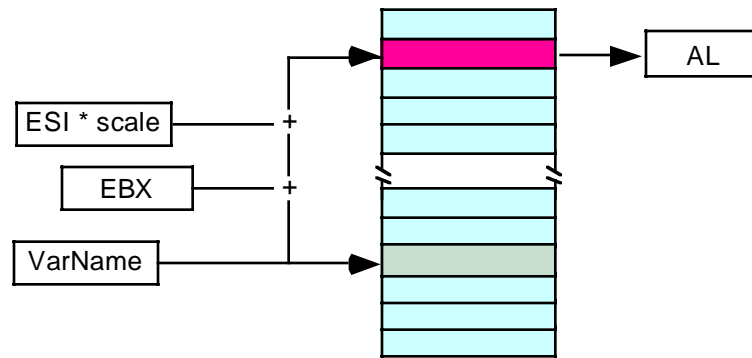
```
VarName[ IndexReg32*scale ]
VarName[ IndexReg32*scale + displacement ]
VarName[ IndexReg32*scale - displacement ]

[ BaseReg32 + IndexReg32*scale ]
[ BaseReg32 + IndexReg32*scale + displacement ]
[ BaseReg32 + IndexReg32*scale - displacement ]

VarName[ BaseReg32 + IndexReg32*scale ]
VarName[ BaseReg32 + IndexReg32*scale + displacement ]
VarName[ BaseReg32 + IndexReg32*scale - displacement ]
```

In these examples, *BaseReg₃₂* represents any general purpose 32-bit register, *IndexReg₃₂* represents any general purpose 32-bit register except ESP, and *scale* must be one of the constants: 1, 2, 4, or 8.

The primary difference between the scaled indexed addressing mode and the indexed addressing mode is the inclusion of the *IndexReg₃₂*scale* component. The effective address computation is extended by adding in the value of this new register after it has been multiplied by the specified scaling factor (see Figure 2.6 for an example involving EBX as the base register and ESI as the index register).



```
mov( VarName[ ebx + esi*scale ], al );
```

Figure 2.6 The Scaled Indexed Addressing Mode

In Figure 2.6, suppose that EBX contains \$100, ESI contains \$20, and *VarName* is at base address \$2000 in memory, then the following instruction:

```
mov( VarName[ ebx + esi*4 + 4 ], al );
```

will move the byte at address \$2184 ($\$1000 + \$100 + \$20*4 + 4$) into the AL register.

The scaled indexed addressing mode is typically used to access elements of arrays whose elements are two, four, or eight bytes each. This addressing mode is also useful for access elements of an array when you have a pointer to the beginning of the array.

Warning: although this addressing mode contains to variable components (the base and index registers), don't get the impression that you use this addressing mode to access elements of a two-dimensional array by loading the two array indices into the two registers. Two-dimensional array access is quite a bit more complicated than this. A later chapter in this text will consider multi-dimensional array access and discuss how to do this.

2.2.2.6 Addressing Mode Wrap-up

Well, believe it or not, you've just learned several hundred addressing modes! That wasn't hard now, was it? If you're wondering where all these modes came from, just consider the fact that the register indirect addressing mode isn't a single addressing mode, but eight different addressing modes (involving the eight different registers). Combinations of registers, constant sizes, and other factors multiply the number of possible addressing modes on the system. In fact, you only need to memorize less than two dozen forms and you've got it made. In practice, you'll use less than half the available addressing modes in any given program (and many addressing modes you may never use at all). So learning all these addressing modes is actually much easier than it sounds.

2.3 Run-Time Memory Organization

An operating system like Linux or Windows tends to put different types of data into different sections (or segments) of main memory. Although it is possible to reconfigure memory to your choice by running the Linker and specify various parameters, by default Windows loads an HLA program into memory using the following basic organization (Linux is similar, though it rearranges some of the sections):

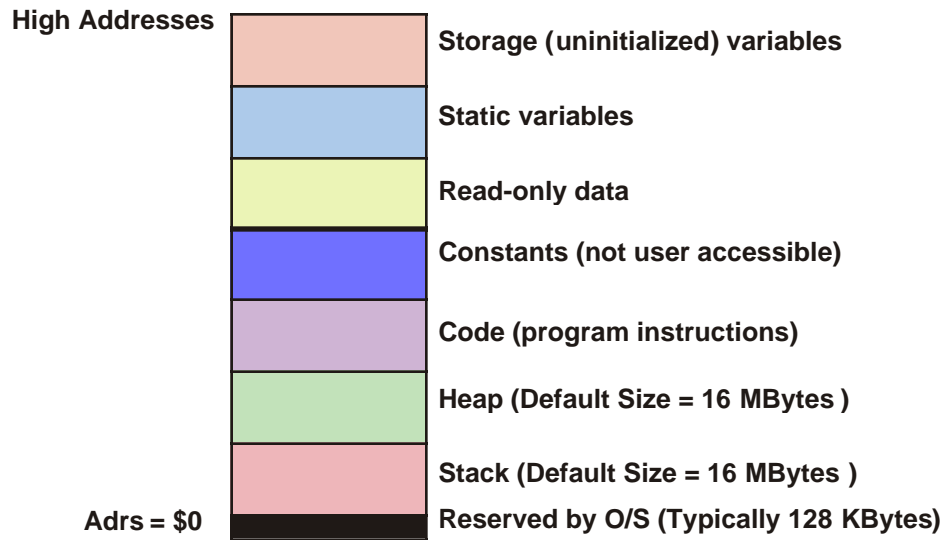


Figure 2.7 Win32 Typical Run-Time Memory Organization

The lowest memory addresses are reserved by the operating system. Generally, your application is not allowed to access data (or execute instructions) at the lowest addresses in memory. One reason the O/S reserves this space is to help trap NULL pointer references. If you attempt to access memory location zero, the operating system will generate a “general protection fault” meaning you’ve accessed a memory location that doesn’t contain valid data. Since programmers often initialize pointers to NULL (zero) to indicate that the pointer is not pointing anywhere, an access of location zero typically means that the programmer has made a mistake and has not properly initialized a pointer to a legal (non-NULL) value. Also note that if you attempt to use one of the 80x86 sixteen-bit addressing modes (HLA doesn’t allow this, but were you to encode the instruction yourself and execute it...) the address will always be in the range 0..\$1FFFE⁴. This will also access a location in the reserved area, generating a fault.

The remaining six areas in the memory map hold different types of data associated with your program. These sections of memory include the stack section, the heap section, the code section, the READONLY section, the STATIC section, and the STORAGE section. Each of these memory sections correspond to some type of data you can create in your HLA programs. The following sections discuss each of these sections in detail.

2.3.1 The Code Section

The code section contains the machine instructions that appear in an HLA program. HLA translates each machine instruction you write into a sequence of one or more byte values. The CPU interprets these byte values as machine instructions during program execution.

By default, when HLA links your program it tells the system that your program can execute instructions out of the code segment and you can read data from the code segment. Note, specifically, that you cannot write data to the code segment. The operating system will generate a general protection fault if you attempt to store any data into the code segment.

4. It’s \$1FFFE, not \$FFFF because you could use the indexed addressing mode with a displacement of \$FFFF along with the value \$FFFF in a 16-bit register.

Remember, machine instructions are nothing more than data bytes. In theory, you could write a program that stores data values into memory and then transfers control to the data it just wrote, thereby producing a program that writes itself as it executes. This possibility produces romantic visions of Artificially Intelligent programs that modify themselves to produce some desired result. In real life, the effect is somewhat less glamorous.

Prior to the popularity of *protected mode operating systems*, like Windows and Linux, a program could overwrite the machine instructions during execution. Most of the time this was caused by defects in a program, not by some super-smart artificial intelligence program. A program would begin writing data to some array and fail to stop once it reached the end of the array, eventually overwriting the executing instructions that make up the program. Far from improving the quality of the code, such a defect usually causes the program to fail spectacularly.

Of course, if a feature is available, someone is bound to take advantage of it. Some programmers have discovered that in some special cases, using *self-modifying code*, that is, a program that modifies its machine instructions during execution, can produce slightly faster or slightly smaller programs. Unfortunately, self-modifying code is very difficult to test and debug. Given the speed of modern processors combined with their instruction set and wide variety of addressing modes, there is almost no reason to use self-modifying code in a modern program. Indeed, protected mode operating systems like Linux and Windows make it difficult for you to write self modifying code.

HLA automatically stores the data associated with your machine code into the code section. In addition to machine instructions, you can also store data into the code section by using the following pseudo-opcodes:

- byte
- word
- dword
- uns8
- uns16
- uns32
- int8
- int16
- int32
- boolean
- char

The syntax for each of these *pseudo-opcodes*⁵ is exemplified by the following BYTE statement:

```
byte comma_separated_list_of_byte_constants ;
```

Here are some examples:

```
boolean    true;
char       'A';
byte       0,1,2;
byte       "Hello", 0
word       0,2;
int8       -5;
uns32      356789, 0;
```

If more than one value appears in the list of values after the pseudo-opcode, HLA emits each successive value to the code stream. So the first *byte* statement above emits three bytes to the code stream, the values zero, one, and two. If a string appears within a byte statement, HLA emits one byte of data for each character in the string. Therefore, the second byte statement above emits six bytes: the characters 'H', 'e', 'l', 'l', and 'o', followed by a zero byte.

5. A pseudo-opcode is a data declaration statement that emits data to the code section, but isn't a true machine instruction (e.g., BYTE is a pseudo-opcode, MOV is a machine instruction).

Keep in mind that the CPU will attempt to treat data you emit to the code stream as machine instructions unless you take special care not to allow the execution of the data. For example, if you write something like the following:

```
mov( 0, ax );
byte 0,1,2,3;
add( bx, cx );
```

Your program will attempt to execute the 0, 1, 2, and 3 byte values as a machine instruction after executing the MOV. Unless you know the machine code for a particular instruction sequence, sticking such data values into the middle of your code will almost always produce unexpected results. More often than not, this will crash your program. Therefore, you should never insert arbitrary data bytes into the middle of an instruction stream unless you know exactly what executing those data values will do in your program⁶.

2.3.2 The Static Sections

In addition to declaring static variables, you can also embed lists of data into the STATIC memory segment. You use the same technique to embed data into your STATIC section that you use to embed data into the code section: you use the *byte*, *word*, *dword*, *uns32*, etc., pseudo-opcodes. Consider the following example:

```
static
  b: byte := 0;
     byte 1,2,3;

  u: uns32 := 1;
     uns32 5,2,10;

  c: char;
     char 'a', 'b', 'c', 'd', 'e', 'f';

  bn: boolean;
     boolean true;
```

Data that HLA writes to the STATIC memory segment using these pseudo-opcodes is written to the segment after the preceding variables. For example, the byte values one, two, and three are emitted to the STATIC section after *b*'s zero byte in the example above. Since there aren't any labels associated with these values, you do not have direct access to these values in your program. The section on address expressions, later in this chapter, will discuss how to access these extra values.

In the examples above, note that the *c* and *bn* variables do not have an (explicit) initial value. However, HLA always initializes variables in the STATIC section to all zero bits, so HLA assigns the NULL character (ASCII code zero) to *c* as its initial value. Likewise, HLA assigns false as the initial value for *bn*. In particular, you should note that your variable declarations in the STATIC section always consume memory, even if you haven't assigned them an initial value. Any data you declare in a pseudo-opcode like BYTE will always follow the actual data associated with the variable declaration.

2.3.3 The Read-Only Data Section

The READONLY data section holds constants, tables, and other data that your program must not change during program execution. You can place read only objects in your program by declaring them in the

6. The main reason for encoding machine code using a data directive like *byte* is to implement machine instructions that HLA does not support (for example, to implement machine instructions added after HLA was written but before HLA could be updated for the new instruction(s)).

READONLY declaration section. The READONLY data declaration section is very similar to the STATIC section with three primary differences:

- The READONLY section begins with the reserved word READONLY rather than STATIC,
- All declarations in the READONLY section must have an initializer, and
- You are not allowed to store data into a READONLY object while the program is running.

Example:

```
readonly
  pi:      real32 := 3.14159;
  e:      real32 := 2.71;
  MaxU16:  uns16  := 65_535;
  MaxI16:  int16  := 32_767;
```

All READONLY object declarations must have an initializer because you cannot initialize the value under program control (since you are not allowed to write data into a READONLY object). The operating system will generate an exception and abort your program if you attempt to write a value to a READONLY object. For all intents and purposes, READONLY objects can be thought of as constants. However, these constants consume memory and other than the fact that you cannot write data to READONLY objects, they behave like, and can be used like, STATIC variables. Since they behave like STATIC objects, you cannot use a READONLY object everywhere a constant is allowed; in particular, READONLY objects are memory objects, so you cannot supply a READONLY object and some other memory object as the operand to an instruction⁷.

The READONLY reserved word allows an alignment parameter, just like the STATIC keyword (See “HLA Support for Data Alignment” on page 146.). You may also place the ALIGN directive in the READONLY section in order to align individual objects on a specific boundary. The following example demonstrates both of these features in the READONLY section:

```
readonly( 8 )
  pi:      real64 := 3.14159265359;
  aChar:  char   := 'a';
  align(4);
  d:      dword  := 4;
```

Note that, also like the STATIC section, you may embed data values in the READONLY section using the BYTE, WORD, DWORD, etc., data declarations, e.g.,

```
readonly
  roArray: byte := 0;
           byte 1, 2, 3, 4, 5;
  qwVal:  dword := 1;
           dword 0;
```

2.3.4 The Storage Section

The READONLY section requires that you initialize all objects you declare. The STATIC section lets you optionally initialize objects (or leave them uninitialized, in which case they have the default initial value of zero). The STORAGE section completes the initialization coverage: you use it to declare variables that are always uninitialized when the program begins running. The STORAGE section begins with the “storage” reserved word and then contains variable declarations that are identical to those appearing in the STATIC section except that you are not allowed to initialize the object. Here is an example:

```
storage
  UunitUns32:      uns32;
```

7. MOV is an exception to this rule since HLA emits special code for memory to memory move operations.

```

i:          int32;
character:  char;
b:          byte;

```

Linux and Windows will initialize all storage objects to zero when they load your program into memory. However, it's probably not a good idea to depend upon this implicit initialization. If you need an object initialized with zero, declare it in a `STATIC` section and explicitly set it to zero.

Variables you declare in the `STORAGE` section may consume less disk space in the executable file for the program. This is because HLA writes out initial values for `READONLY` and `STATIC` objects to the executable file, but uses a compact representation for uninitialized variables you declare in the `STORAGE` section.

Like the `STATIC` and `READONLY` sections, you can supply an alignment parameter after the `STORAGE` keyword and the `ALIGN` directive may appear within the `STORAGE` section (See “HLA Support for Data Alignment” on page 146.). Of course, aligning your data can produce faster access to that data at the expense of a slightly larger `STORAGE` section. The following example demonstrates the use of these two features in the `STORAGE` section:

```

storage( 4 )
  d:    dword;
  b:    byte;
  align(2);
  w:    word;

```

Since the `STORAGE` section does not allow initialized values, you *cannot* put unlabelled values in the `STORAGE` section using the `BYTE`, `WORD`, `DWORD`, etc., data declarations.

2.3.5 The @NOSTORAGE Attribute

The `@NOSTORAGE` attribute lets you declare variables in the static data declaration sections (i.e., `STATIC`, `READONLY`, and `STORAGE`) without actually allocating memory for the variable. The `@NOSTORAGE` option tells HLA to assign the current address in a data declaration section to a variable but not allocate any storage for the object. Therefore, that variable will share the same memory address as the next object appearing in the variable declaration section. Here is the syntax for the `@NOSTORAGE` option:

```

variableName: varType; @nostorage;

```

Note that you follow the type name with “`@nostorage;`” rather than some initial value or just a semicolon. The following code sequence provides an example of using the `@NOSTORAGE` option in the `READONLY` section:

```

readonly
  abcd: dword; nostorage;
        byte 'a', 'b', 'c', 'd';

```

In this example, `abcd` is a double word whose L.O. byte contains 97 ('a'), byte #1 contains 98 ('b'), byte #2 contains 99 ('c'), and the H.O. byte contains 100 ('d'). HLA does not reserve storage for the `abcd` variable, so HLA associates the following four bytes in memory (allocated by the `BYTE` directive) with `abcd`.

Note that the `@NOSTORAGE` attribute is only legal in the `STATIC`, `STORAGE`, and `READONLY` sections. HLA does not allow its use in the `VAR` section.

2.3.6 The Var Section

HLA provides another variable declaration section, the `VAR` section, that you can use to create *automatic* variables. Your program will allocate storage for automatic variables whenever a program unit (i.e., main program or procedure) begins execution, and it will deallocate storage for automatic variables when

that program unit returns to its caller. Of course, any automatic variables you declare in your main program have the same *lifetime*⁸ as all the `STATIC`, `READONLY`, and `STORAGE` objects, so the automatic allocation feature of the `VAR` section is wasted on the main program. In general, you should only use automatic objects in procedures (see the chapter on procedures for details). HLA allows them in your main program's declaration section as a generalization.

Since variables you declare in the `VAR` section are created at run-time, HLA does not allow initializers on variables you declare in this section. So the syntax for the `VAR` section is nearly identical to that for the `STORAGE` section; the only real difference in the syntax between the two is the use of the `VAR` reserved word rather than the `STORAGE` reserved word. The following example illustrates this:

```
var
  vInt:  int32;
  vChar: char;
```

HLA allocates variables you declare in the `VAR` section in the stack segment. HLA does not allocate `VAR` objects at fixed locations within the stack segment; instead, it allocates these variables in an *activation record* associated with the current program unit. The chapter on intermediate procedures will discuss activation records in greater detail, for now it is important only to realize that HLA programs use the `EBP` register as a pointer to the current activation record. Therefore, anytime you access a `var` object, HLA automatically replaces the variable name with “[`EBP`+displacement]”. Displacement is the offset of the object in the activation record. This means that you cannot use the full scaled indexed addressing mode (a base register plus a scaled index register) with `VAR` objects because `VAR` objects already use the `EBP` register as their base register. Although you will not directly use the two register addressing modes often, the fact that the `VAR` section has this limitation is a good reason to avoid using the `VAR` section in your main program.

The `VAR` section supports the `align` parameter and the `ALIGN` directive, like the other declaration sections, however, these `align` directives only guarantee that the alignment within the activation record is on the boundary you specify. If the activation record is not aligned on a reasonable boundary (unlikely, but possible) then the actual variable alignment won't be correct.

2.3.7 Organization of Declaration Sections Within Your Programs

The `STATIC`, `READONLY`, `STORAGE`, and `VAR` sections may appear zero or more times between the `PROGRAM` header and the associated `BEGIN` for the main program. Between these two points in your program, the declaration sections may appear in any order as the following example demonstrates:

```
program demoDeclarations;

static
  i_static:  int32;

var
  i_auto:   int32;

storage
  i_uninit: int32;

readonly
  i_readonly: int32 := 5;

static
  j:        uns32;

var
```

8. The lifetime of a variable is the point from which memory is first allocated to the point the memory is deallocated for that variable.

```

    k:char;

readonly
    i2:uns8 := 9;

storage
    c:char;

storage
    d:dword;

begin demoDeclarations;

    << code goes here >>

end demoDeclarations;

```

In addition to demonstrating that the sections may appear in an arbitrary order, this section also demonstrates that a given declaration section may appear more than once in your program. When multiple declaration sections of the same type (e.g., the three STORAGE sections above) appear in a declaration section of your program, HLA combines them into a single section⁹.

2.4 Address Expressions

In the section on addressing modes (see “The 80x86 Addressing Modes” on page 157) this chapter points out that addressing modes take a couple generic forms, including:

```

    VarName[ Reg32 ]
    VarName[ Reg32 + offset ]
    VarName[ RegNotESP32*Scale ]
    VarName[ Reg32 + RegNotESP32*Scale ]
    VarName[ RegNotESP32*Scale + offset ]
    and
    VarName[ Reg32 + RegNotESP32*Scale + offset ]

```

Another legal form, which isn’t actually a new addressing mode but simply an extension of the displacement-only addressing mode is

```

    VarName[ offset ]

```

This latter example computes its effective address by adding the (constant) offset within the brackets to the specified variable address. For example, the instruction “MOV(Address[3], AL);” loads the AL register with the byte in memory that is three bytes beyond the *Address* object.

9. Remember, though, that HLA combines *static* and *data* declarations into the same memory segment.

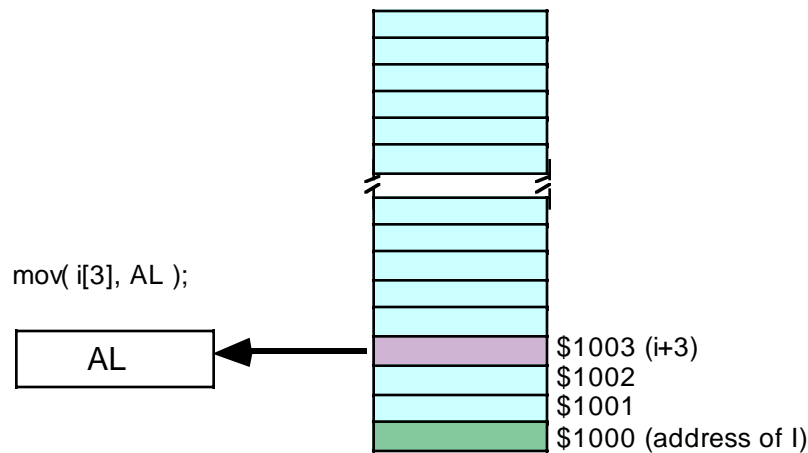


Figure 2.8 Using an Address Expression to Access Data Beyond a Variable

It is extremely important to remember that the *offset* value in these examples must be a constant. If *Index* is an *int32* variable, then “Variable[Index]” is not a legal specification. If you wish to specify an index that varies at run-time, then you must use one of the indexed or scaled indexed addressing modes; that is, any index that changes at run-time must be held in a general purpose 32-bit register.

Another important thing to remember is that the offset in “Address[offset]” is a byte offset. Despite the fact that this syntax is reminiscent of array indexing in a high level language like C/C++ or Pascal, this does not properly index into an array of objects unless *Address* is an array of bytes.

This text will consider an *address expression* to be any legal 80x86 addressing mode that includes a displacement (i.e., variable name) or an offset. In addition to the above forms, the following are also address expressions:

$$\begin{aligned} & [\text{Reg}_{32} + \text{offset}] \\ & [\text{Reg}_{32} + \text{RegNotESP}_{32} * \text{Scale} + \text{offset}] \end{aligned}$$

This text will *not* consider the following to be address expressions since they do not involve a displacement or offset component:

$$\begin{aligned} & [\text{Reg}_{32}] \\ & [\text{Reg}_{32} + \text{RegNotESP}_{32} * \text{Scale}] \end{aligned}$$

Address expressions are special because those instructions containing an address expression always encode a displacement constant as part of the machine instruction. That is, the machine instruction contains some number of bits (usually eight or thirty-two) that hold a numeric constant. That constant is the sum of the displacement (i.e., the address or offset of the variable) plus the offset supplied in the addressing mode. Note that HLA automatically adds these two values together for you (or subtracts the offset if you use the “-” rather than “+” operator in the addressing mode).

Until this point, the offset in all the addressing mode examples has always been a single numeric constant. However, HLA also allows a *constant expression* anywhere an offset is legal. A constant expression consists of one or more constant terms manipulated by operators such as addition, subtraction, multiplication, division, modulo, and a wide variety of other operators. Most address expressions, however, will only involve addition, subtraction, multiplication, and sometimes, division. Consider the following example:

```
mov( X[ 2*4+1 ], al );
```

This instruction will move the byte at address $X+9$ into the AL register.

The value of an address expression is always computed at compile-time, never while the program is running. When HLA encounters the instruction above, it calculates $2*4+1$ on the spot and adds this result to the base address of *X* in memory. HLA encodes this single sum (base address of *X* plus nine) as part of the instruction; HLA does not emit extra instructions to compute this sum for you at run-time (which is good, doing so would be less efficient). Since HLA computes the value of address expressions at compile-time, all components of the expression must be constants since HLA cannot know what the value of a variable will be at run-time while it is compiling the program.

Address expressions are very useful for accessing additional bytes in memory beyond a variable, particularly when you've used the *byte*, *word*, *dword*, etc., statements in a *STATIC*, or *READONLY* section to tack on additional bytes after a data declaration. For example, consider the following program:

```

program adrsExpressions;
#include( "stdlib.hhf" );
static
    i:  int8; @nostorage;
        byte 0, 1, 2, 3;

begin adrsExpressions;

    stdout.put
    (
        "i[0]=", i[0], nl,
        "i[1]=", i[1], nl,
        "i[2]=", i[2], nl,
        "i[3]=", i[3], nl
    );

end adrsExpressions;

```

Program 3.1 Demonstration of Address Expressions

Throughout this chapter and those that follow you will see several additional uses of address expressions.

2.5 Type Coercion

Although HLA is fairly loose when it comes to type checking, HLA does ensure that you specify appropriate operand sizes to an instruction. For example, consider the following (incorrect) program:

```

program hasErrors;
static
    i8:    int8;
    i16:   int16;
    i32:   int32;
begin hasErrors;

    mov( i8,  eax );
    mov( i16, al );
    mov( i32,  ax );

end hasErrors;

```

HLA will generate errors for the three MOV instructions appearing in this program. This is because the operand sizes do not agree. The first instruction attempts to move a byte into EAX, the second instruction attempts to move a word into AL and the third instruction attempts to move a dword into AX. The MOV instruction, of course, requires that its two operands both be the same size.

While this is a good feature in HLA¹⁰, there are times when it gets in the way of the task at hand. For example, consider the following data declaration:

```
static
    byte_values: byte; @nostorage;
                byte    0, 1;

    ...

    mov( byte_values, ax );
```

In this example let's assume that the programmer really wants to load the word starting at address *byte_values* in memory into the AX register because they want to load AL with zero and AH with one using a single instruction. HLA will refuse, claiming there is a type mismatch error (since *byte_values* is a *byte* object and AX is a *word* object). The programmer could break this into two instructions, one to load AL with the byte at address *byte_values* and the other to load AH with the byte at address *byte_values[1]*. Unfortunately, this decomposition makes the program slightly less efficient (which was probably the reason for using the single MOV instruction in the first place). Somehow, it would be nice if we could tell HLA that we know what we're doing and we want to treat the *byte_values* variable as a *word* object. HLA's *type coercion* facilities provide this capability.

Type coercion¹¹ is the process of telling HLA that you want to treat an object as an explicitly specified type, regardless of its actual type. To coerce the type of a variable, you use the following syntax:

```
(type newTypeName addressingMode)
```

The *newTypeName* component is the new type you wish HLA to apply to the memory location specified by *addressingMode*. You may use this coercion operator anywhere a memory address is legal. To correct the previous example, so HLA doesn't complain about type mismatches, you would use the following statement:

```
mov( (type word byte_values), ax );
```

This instruction tells HLA to load the AX register with the word starting at address *byte_values* in memory. Assuming *byte_values* still contains its initial values, this instruction will load zero into AL and one into AH.

Type coercion is necessary when you specify an anonymous variable as the operand to an instruction that modifies memory directly (e.g., NEG, SHL, NOT, etc.). Consider the following statement:

```
not( [ebx] );
```

HLA will generate an error on this instruction because it cannot determine the size of the memory operand. That is, the instruction does not supply sufficient information to determine whether the program should invert the bits in the byte pointed at by EBX, the word pointed at by EBX, or the double word pointed at by EBX. You must use type coercion to explicitly tell HLA the size of the memory operand when using anonymous variables with these types of instructions:

```
not( (type byte [ebx]) );
not( (type word [ebx]) );
not( (type dword [ebx]) );
```

Warning: do not use the type coercion operator unless you know exactly what you are doing and the effect that it has on your program. Beginning assembly language programmers often use type coercion as a

10. After all, if the two operand sizes are different this usually indicates an error in the program.

11. Also called type casting in some languages.

tool to quiet the compiler when it complains about type mismatches without solving the underlying problem. For example, consider the following statement (where *byteVar* is an actual eight-bit variable):

```
mov( eax, (type dword byteVar) );
```

Without the type coercion operator, HLA probably complains about this instruction because it attempts to store a 32-bit register into an eight-bit memory location (assuming *byteVar* is a byte variable). A beginning programmer, wanting their program to compile, may take a short cut and use the type coercion operator as shown in this instruction; this certainly quiets the compiler - it will no longer complain about a type mismatch. So the beginning programmer is happy. But the program is still incorrect, the only difference is that HLA no longer warns you about your error. The type coercion operator does not fix the problem of attempting to store a 32-bit value into an eight-bit memory location - it simply allows the instruction to store a 32-bit value *starting at the address* specified by the eight-bit variable. The program still stores away four bytes, overwriting the three bytes following *byteVar* in memory. This often produces unexpected results including the phantom modification of variables in your program¹². Another, rarer, possibility is for the program to abort with a general protection fault. This can occur if the three bytes following *byteVar* are not allocated in real memory or if those bytes just happen to fall in a read-only segment in memory. The important thing to remember about the type coercion operator is this: "If you can't exactly state the affect this operator has, don't use it."

Also keep in mind that the type coercion operator does not perform any translation of the data in memory. It simply tells the compiler to treat the bits in memory as a different type. It will not automatically sign extend an eight-bit value to 32 bits nor will it convert an integer to a floating point value. It simply tells the compiler to treat the bit pattern that exists in memory as a different type.

2.6 Register Type Coercion

You can also cast a register as a specific type using the type coercion operator. By default, the eight-bit registers are of type *byte*, the 16-bit registers are of type *word*, and the 32-bit registers are of type *dword*. With type coercion, you can cast a register as a different type *as long as the size of the new type agrees with the size of the register*. This is an important restriction that does not apply when applying type coercion to a memory variable.

Most of the time you do not need to coerce a register to a different type. After all, as *byte*, *word*, and *dword* objects, they are already compatible with all one, two, and four byte objects. However, there are a few instances where register type coercion is handy, if not downright necessary. Two examples include boolean expressions in HLA high level language statements (e.g., IF and WHILE) and register I/O in the *stdout.put* and *stdin.get* (and related) statements.

In boolean expressions, *byte*, *word*, and *dword* objects are always treated as unsigned values. Therefore, without type coercion register objects are always treated as unsigned values so the boolean expression in the following IF statement is always false (since there is no unsigned value less than zero):

```
if( eax < 0 ) then

    stdout.put( "EAX is negative!", nl );

endif;
```

You can overcome this limitation by casting EAX as an *int32* value:

```
if( (type int32 eax) < 0 ) then

    stdout.put( "EAX is negative!", nl );

endif;
```

12. If you have a variable immediately following *byteVar* in this example, the MOV instruction will surely overwrite the value of that variable, whether or not you intend this to happen.

In a similar vein, the HLA Standard Library `stdout.put` routine always outputs *byte*, *word*, and *dword* values as hexadecimal numbers. Therefore, if you attempt to print a register, the `stdout.put` routine will print it as a hex value. If you would like to print the value as some other type, you can use register type coercion to achieve this:

```
stdout.put( "AL printed as a char = '", (type char al), "'", nl );
```

The same is true for the `stdin.get` routine. It will always read a hexadecimal value for a register unless you coerce its type to something other than *byte*, *word*, or *dword*.

2.7 The Stack Segment and the Push and Pop Instructions

This chapter mentions that all variables you declare in the VAR section wind up in the stack memory segment (see “The Var Section” on page 169). However, VAR objects are not the only things that wind up in the stack segment in memory; your programs manipulate data in the stack segment in many different ways. This section introduces a set of instructions, the PUSH and POP instructions, that also manipulate data in the stack segment.

The stack segment in memory is where the 80x86 maintains the *stack*. The stack is a dynamic data structure that grows and shrinks according to certain memory needs of the program. The stack also stores important information about program including local variables, subroutine information, and temporary data.

The 80x86 controls its stack via the ESP (stack pointer) register. When your program begins execution, the operating system initializes ESP with the address of the last memory location in the stack memory segment. Data is written to the stack segment by “pushing” data onto the stack and “popping” or “pulling” data off of the stack. Whenever you push data onto the stack, the 80x86 decrements the stack pointer by the size of the data you are pushing and then it copies the data to memory where ESP is then pointing. As a concrete example, consider the 80x86 PUSH instruction:

```
push( reg16 );
push( reg32 );
push( memory16 );
push( memory32 );
pushw( constant );
pushd( constant );
```

These six forms allow you to push *word* or *dword* registers, memory locations, and constants. You should specifically note that you cannot push *byte* values onto the stack.

2.7.1 The Basic PUSH Instruction

The PUSH instruction does the following:

```
ESP := ESP - Size_of_Register_or_Memory_Operand (2 or 4)
[ESP] := Operand's_Value
```

The PUSHW and PUSHHD operand sizes are always two or four bytes, respectively.

Assuming that ESP contains \$00FF_FFE8, then the instruction “PUSH(EAX);” will set ESP to \$00FF_FFE4 and store the current value of EAX into memory location \$00FF_FFE4 as shown in Figure 2.9 and Figure 2.10:

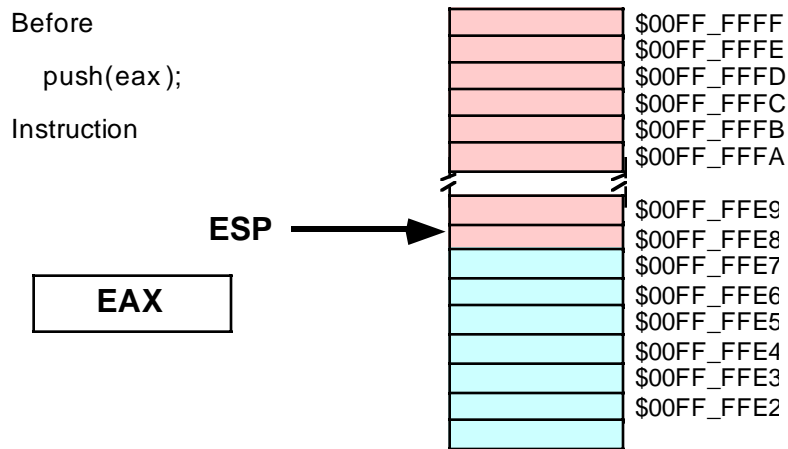


Figure 2.9 Stack Segment Before “PUSH(EAX);” Operation

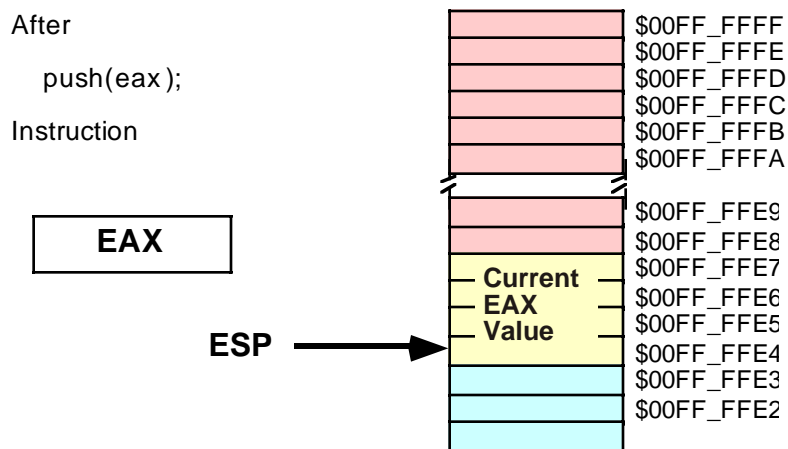


Figure 2.10 Stack Segment After “PUSH(EAX);” Operation

Note that the “PUSH(EAX);” instruction does not affect the value in the EAX register.

Although the 80x86 supports 16-bit push operations, these are intended primarily for use in 16-bit environments such as DOS. For maximum performance, the stack pointer should always be an even multiple of four; indeed, your program may malfunction under Windows or Linux if ESP contains a value that is not a multiple of four and you make an HLA Standard Library or an operating system API call. The only practical reason for pushing less than four bytes at a time on the stack is because you’re building up a double word via two successive word pushes.

2.7.2 The Basic POP Instruction

To retrieve data you’ve pushed onto the stack, you use the POP instruction. The basic POP instruction allows the following different forms:

```
pop( reg16 );  
pop( reg32 );
```

```
pop( memory16 );  
pop( memory32 );
```

Like the PUSH instruction, the POP instruction only supports 16-bit and 32-bit operands; you cannot pop an eight-bit value from the stack. Also like the PUSH instruction, you should avoid popping 16-bit values (unless you do two 16-bit pops in a row) because 16-bit pops may leave the ESP register containing a value that is not an even multiple of four. One major difference between PUSH and POP is that you cannot POP a constant value (which makes sense, because the operand for PUSH is a source operand while the operand for POP is a destination operand).

Formally, here's what the POP instruction does:

```
Operand := [ESP]  
ESP := ESP + Size_of_Operand (2 or 4)
```

As you can see, the POP operation is the converse of the PUSH operation. Note that the POP instruction copies the data from memory location [ESP] before adjusting the value in ESP. See Figure 2.11 and Figure 2.12 for details on this operation:

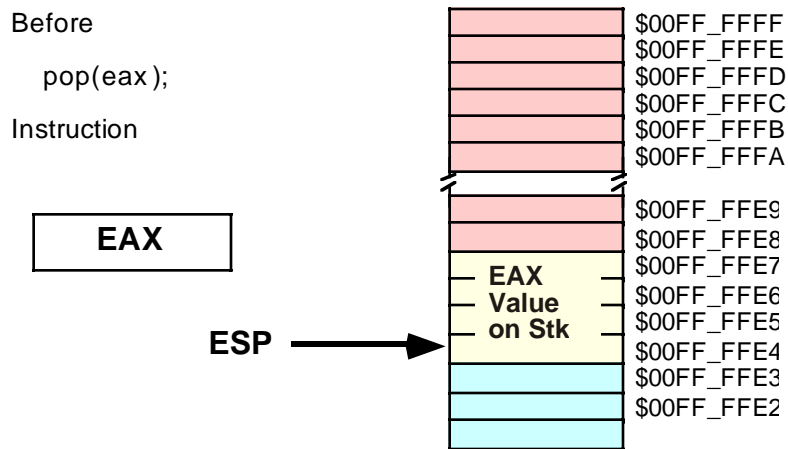


Figure 2.11 Memory Before a “POP(EAX);” Operation

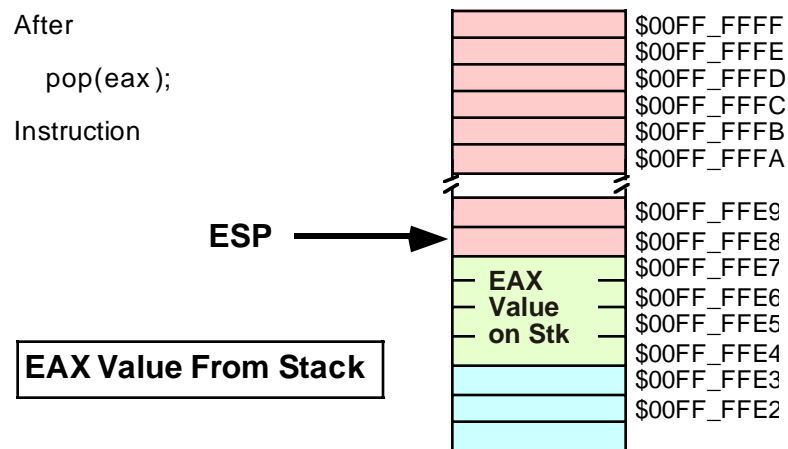


Figure 2.12 Memory After the “POP(EAX);” Instruction

Note that the value popped from the stack is still present in memory. Popping a value does not erase the value in memory, it just adjusts the stack pointer so that it points at the next value above the popped value. However, you should never attempt to access a value you’ve popped off the stack. The next time something is pushed onto the stack, the popped value will be obliterated. Since your code isn’t the only thing that uses the stack (i.e., the operating system uses the stack as do other subroutines), you cannot rely on data remaining in stack memory once you’ve popped it off the stack.

2.7.3 Preserving Registers With the PUSH and POP Instructions

Perhaps the most common use of the PUSH and POP instructions is to save register values during intermediate calculations. A problem with the 80x86 architecture is that it provides very few general purpose registers. Since registers are the best place to hold temporary values, and registers are also needed for the various addressing modes, it is very easy to run out of registers when writing code that performs complex calculations. The PUSH and POP instructions can come to your rescue when this happens.

Consider the following program outline:

```
<< Some sequence of instructions that use the EAX register >>

<< Some sequence of instructions that need to use EAX, for a
    different purpose than the above instructions >>

<< Some sequence of instructions that need the original value in EAX >>
```

The PUSH and POP instructions are perfect for this situation. By inserting a PUSH instruction before the middle sequence and a POP instruction after the middle sequence above, you can preserve the value in EAX across those calculations:

```
<< Some sequence of instructions that use the EAX register >>
push( eax );
<< Some sequence of instructions that need to use EAX, for a
    different purpose than the above instructions >>
pop( eax );
<< Some sequence of instructions that need the original value in EAX >>
```

The PUSH instruction above copies the data computed in the first sequence of instructions onto the stack. Now the middle sequence of instructions can use EAX for any purpose it chooses. After the middle sequence of instructions finishes, the POP instruction restores the value in EAX so the last sequence of instructions can use the original value in EAX.

2.7.4 The Stack is a LIFO Data Structure

You can push more than one value onto the stack without first popping previous values off the stack. However, the stack is a last-in, first-out (LIFO) data structure, so you must be careful how you push and pop multiple values. For example, suppose you want to preserve EAX and EBX across some block of instructions, the following code demonstrates the obvious way to handle this:

```
push( eax );
push( ebx );
<< Code that uses EAX and EBX goes here >>
pop( eax );
pop( ebx );
```

Unfortunately, this code will not work properly! Figures 2.13, 2.14, 2.15, and 2.16 show the problem. Since this code pushes EAX first and EBX second, the stack pointer is left pointing at EBX's value on the stack. When the POP(EAX) instruction comes along, it removes the value that was originally in EBX from the stack and places it in EAX! Likewise, the POP(EBX) instruction pops the value that was originally in EAX into the EBX register. The end result is that this code has managed to swap the values in the registers by popping them in the same order that it pushed them.

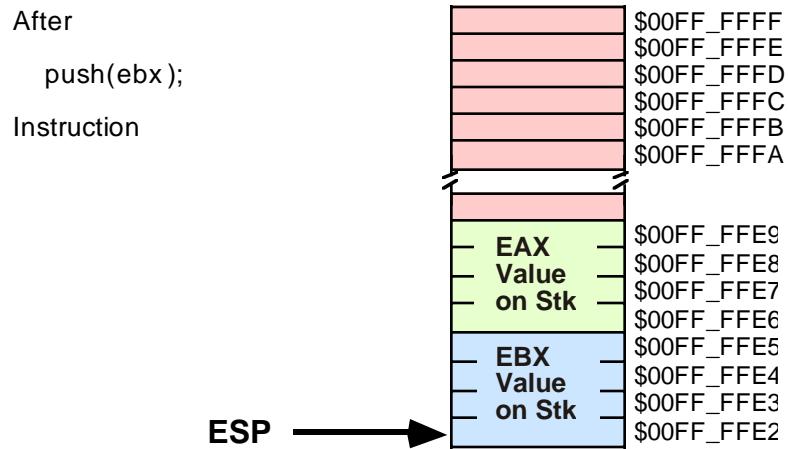


Figure 2.13 Stack After Pushing EAX

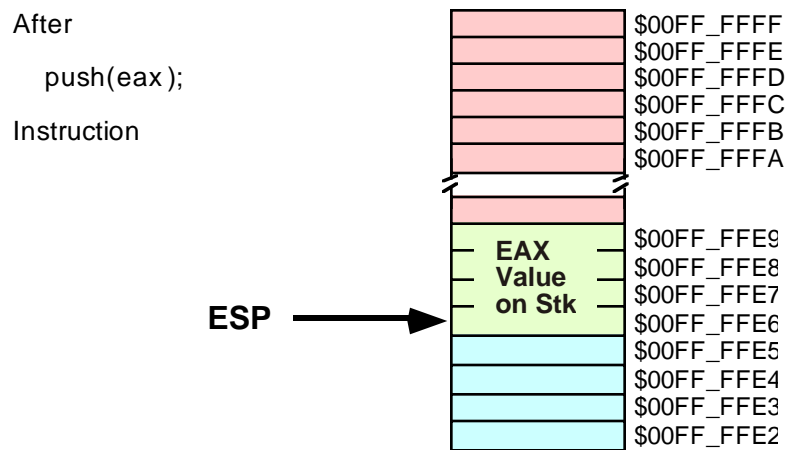


Figure 2.14 Stack After Pushing EBX

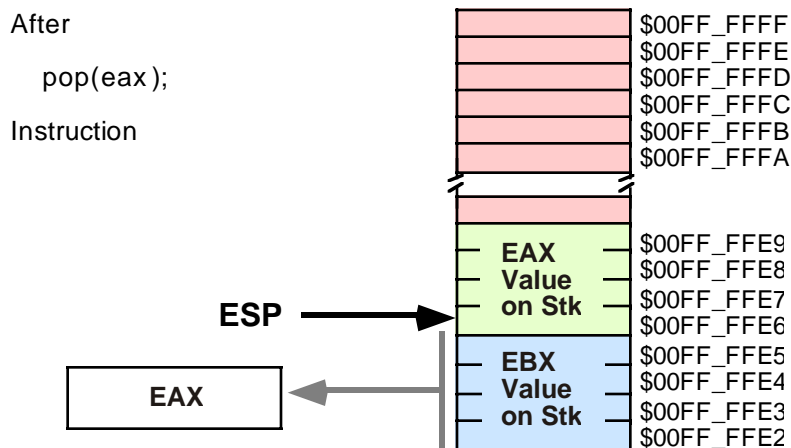


Figure 2.15 Stack After Popping EAX

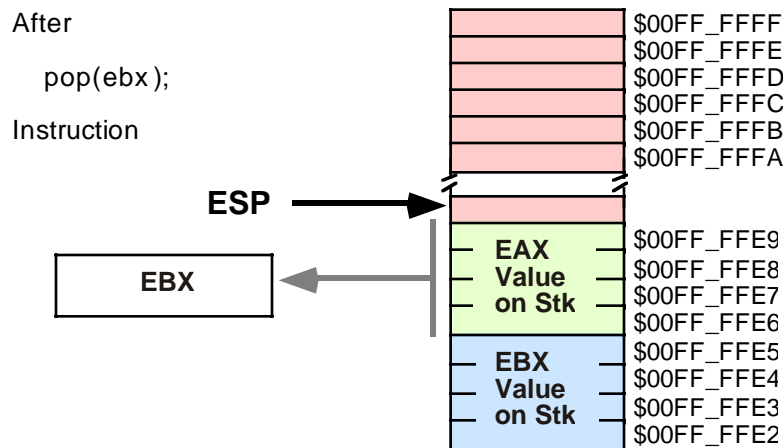


Figure 2.16 Stack After Popping EBX

To rectify this problem, you must note that the stack is a last-in, first-out data structure, so the first thing you must pop is the last thing you've pushed onto the stack. Therefore, you must always observe the following maxim:

- ❑ Always pop values in the reverse order that you push them.

The correction to the previous code is

```
push( eax );
push( ebx );
<< Code that uses EAX and EBX goes here >>
pop( ebx );
pop( eax );
```

Another important maxim to remember is

- ❑ Always pop exactly the same number of bytes that you push.

This generally means that the number of pushes and pops must exactly agree. If you have too few pops, you will leave data on the stack which may confuse the running program¹³; If you have too many pops, you will accidentally remove previously pushed data, often with disastrous results.

A corollary to the maxim above is “Be careful when pushing and popping data within a loop.” Often it is quite easy to put the pushes in a loop and leave the pops outside the loop (or vice versa), creating an inconsistent stack. Remember, it is the execution of the PUSH and POP instructions that matters, not the number of PUSH and POP instructions that appear in your program. At run-time, the number (and order) of the PUSH instructions the program executes must match the number (and reverse order) of the POP instructions.

13. You'll see why when we cover procedures.

2.7.5 Other PUSH and POP Instructions

The 80x86 provides several additional PUSH and POP instructions in addition to the basic instructions described in the previous sections. These instructions include the following:

- PUSHA
- PUSHAD
- PUSHF
- PUSHFD
- POPA
- POPAD
- POPF
- POPFD

The PUSHA instruction pushes all the general-purpose 16-bit registers onto the stack. This instruction is primarily intended for older 16-bit operating systems like DOS. In general, you will have very little need for this instruction. The PUSHA instruction pushes the registers onto the stack in the following order:

```
ax
cx
dx
bx
sp
bp
si
di
```

The PUSHAD instruction pushes all the 32-bit (dword) registers onto the stack. It pushes the registers onto the stack in the following order:

```
eax
ecx
edx
ebx
esp
ebp
esi
edi
```

Since the SP/ESP register is inherently modified by the PUSHA and PUSHAD instructions, you may wonder why Intel bothered to push it at all. It was probably easier in the hardware to go ahead and push SP/ESP rather than make a special case out of it. In any case, these instructions do push SP or ESP so don't worry about it too much - there is nothing you can do about it.

The POPA and POPAD instructions provide the corresponding "pop all" operation to the PUSHA and PUSHAD instructions. This will pop the registers pushed by PUSHA or PUSHAD in the appropriate order (that is, POPA and POPAD will properly restore the register values by popping them in the reverse order that PUSHA or PUSHAD pushed them).

Although the PUSHA/POPA and PUSHAD/POPAD sequences are short and convenient, they are actually slower than the corresponding sequence of PUSH/POP instructions, this is especially true when you consider that you rarely need to push a majority, much less all the registers¹⁴. So if you're looking for maximum speed, you should carefully consider whether to use the PUSH(A/D)/POP(A/D) instructions. This text generally opts for convenience and readability; so it will use the PUSHAD and POPAD instructions without worrying about lost efficiency.

14. For example, it is extremely rare for you to need to push and pop the ESP register with the PUSHAD/POPAD instruction sequence.

The PUSHF, PUSHFD, POPF, and POPFD instructions push and pop the (E)FLAGS register. These instructions allow you to preserve condition code and other flag settings across the execution of some sequence of instructions. Unfortunately, unless you go to a lot of trouble, it is difficult to preserve individual flags. When using the PUSHF(D) and POPF(D) instructions it's an all or nothing proposition - you preserve all the flags when you push them, you restore all the flags when you pop them.

Like the PUSHA and POPA instructions, you should really use the PUSHFD and POPFD instructions to push the full 32-bit version of the EFLAGS register. Although the extra 16-bits you push and pop are essentially ignored when writing applications, you still want to keep the stack aligned by pushing and popping only double words.

2.7.6 Removing Data From the Stack Without Popping It

Once in a while you may discover that you've pushed data onto the stack that you no longer need. Although you could pop the data into an unused register or memory location, there is an easier way to remove unwanted data from the stack - simply adjust the value in the ESP register to skip over the unwanted data on the stack.

Consider the following dilemma:

```

push( eax );
push( ebx );

<< Some code that winds up computing some values we want to keep
    into EAX and EBX >>

if( Calculation_was_performed ) then

    // Whoops, we don't want to pop EAX and EBX!
    // What to do here?

else

    // No calculation, so restore EAX, EBX.

    pop( ebx );
    pop( eax );

endif;

```

Within the THEN section of the IF statement, this code wants to remove the old values of EAX and EBX without otherwise affecting any registers or memory locations. How to do this?

Since the ESP register simply contains the memory address of the item on the top of the stack, we can remove the item from the top of stack by adding the size of that item to the ESP register. In the example above, we want to remove two double word items from the top of stack, so we can easily accomplish this by adding eight to the stack pointer:

```

push( eax );
push( ebx );

<< Some code that winds up computing some values we want to keep
    into EAX and EBX >>

if( Calculation_was_performed ) then

    add( 8, ESP );    // Remove unneeded EAX and EBX values from the stack.

else

```

```

// No calculation, so restore EAX, EBX.

pop( ebx );
pop( eax );

endif;

```

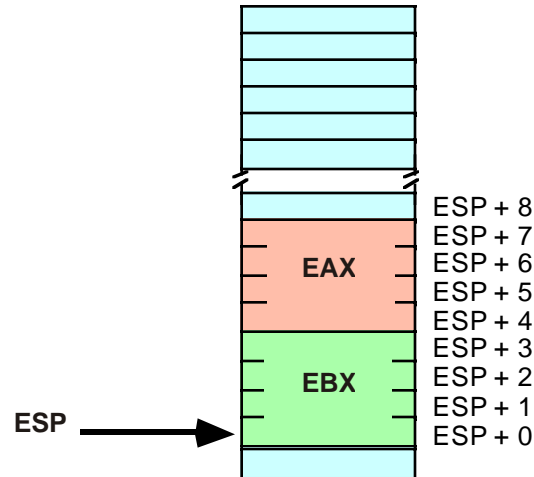


Figure 2.17 Removing Data from the Stack, Before `ADD(8, ESP)`

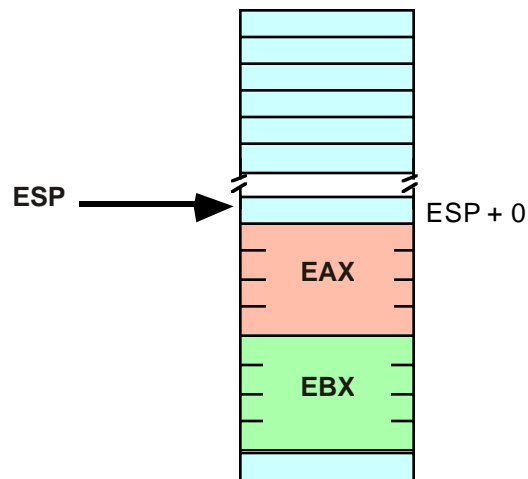


Figure 2.18 Removing Data from the Stack, After `ADD(8, ESP)`;

Effectively, this code pops the data off the stack without moving it anywhere. Also note that this code is faster than two dummy POP instructions because it can remove any number of bytes from the stack with a single ADD instruction.

Warning: remember to keep the stack aligned on a double word boundary. Therefore, you should always add a constant that is an even multiple of four to ESP when removing data from the stack.

2.7.7 Accessing Data You've Pushed on the Stack Without Popping It

Once in a while you will push data onto the stack and you will want to get a copy of that data's value, or perhaps you will want to change that data's value, without actually popping the data off the stack (that is, you wish to pop the data off the stack at a later time). The 80x86 "[reg₃₂ + offset]" addressing mode provides the mechanism for this.

Consider the stack after the execution of the following two instructions (see Figure 2.19):

```
push( eax );
push( ebx );
```

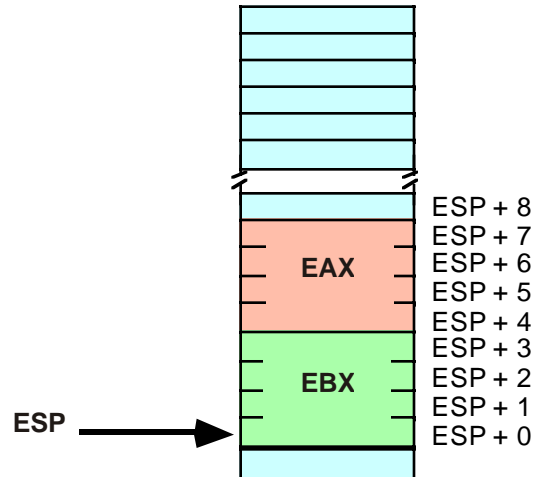


Figure 2.19 Stack After Pushing EAX and EBX

If you wanted to access the original EBX value without removing it from the stack, you could cheat and pop the value and then immediately push it again. Suppose, however, that you wish to access EAX's old value; or some other value even farther up on the stack. Popping all the intermediate values and then pushing them back onto the stack is problematic at best, impossible at worst. However, as you will notice from Figure 2.19, each of the values pushed on the stack is at some offset from the ESP register in memory. Therefore, we can use the "[ESP + offset]" addressing mode to gain direct access to the value we are interested in. In the example above, you can reload EAX with its original value by using the single instruction:

```
mov( [esp+4], eax );
```

This code copies the four bytes starting at memory address ESP+4 into the EAX register. This value just happens to be the value of EAX that was earlier pushed onto the stack. This same technique can be used to access other data values you've pushed onto the stack.

Warning: Don't forget that the offsets of values from ESP into the stack change every time you push or pop data. Abusing this feature can create code that is hard to modify; if you use this feature throughout your code, it will make it difficult to push and pop other data items between the point you first push data onto the stack and the point you decide to access that data again using the "[ESP + offset]" memory addressing mode.

The previous section pointed out how to remove data from the stack by adding a constant to the ESP register. That code example could probably be written more safely as:

```
push( eax );
push( ebx );
```

```

<< Some code that winds up computing some values we want to keep
    into EAX and EBX >>

if( Calculation_was_performed ) then

    // Overwrite saved values on stack with new EAX/EBX values.
    // (so the pops that follow won't change the values in EAX/EBX.)
    mov( eax, [esp+4] );
    mov( ebx, [esp] );

endif;
pop( ebx );
pop( eax );

```

In this code sequence, the calculated result was stored over the top of the values saved on the stack. Later on, when the values are popped off the stack, the program loads these calculated values into EAX and EBX.

2.8 Dynamic Memory Allocation and the Heap Segment

Although static and automatic variables are all simple programs may need, more sophisticated programs need the ability to allocate and deallocate storage dynamically (at run-time) under program control. In the C language, you would use the *malloc* and *free* functions for this purpose. C++ provides the *new* and *delete* operators. Pascal uses *new* and *dispose*. Other languages provide comparable routines. These memory allocation routines share a couple of things in common: they let the programmer request how many bytes of storage to allocate, they return a *pointer* to the newly allocated storage, and they provide a facility for returning the storage to the system so the system can reuse it in a future allocation call. As you've probably guessed, HLA also provides a set of routines in the HLA Standard Library that handle memory allocation and deallocation.

The HLA Standard Library *malloc* and *free* routines handle the memory allocation and deallocation chores (respectively)¹⁵. The *malloc* routine uses the following calling sequence:

```
malloc( Number_of_Bytes_Requested );
```

The single parameter is a *dword* value (an unsigned constant) specifying the number of bytes of storage you are requesting. This procedure allocates storages in the *heap* segment in memory. The HLA *malloc* function locates an unused block of memory of the specified size in the heap segment and marks the block as “in use” so that future calls to *malloc* will not reallocate this same storage. After marking the block as “in use” the *malloc* routine returns a pointer to the first byte of this storage in the EAX register.

For many objects, you will know the number of bytes that you need in order to represent that object in memory. For example, if you wish to allocate storage for an *uns32* variable, you could use the following call to the *malloc* routine:

```
malloc( 4 );
```

Although you can specify a literal constant as this example suggests, it's generally a poor idea to do so when allocating storage for a specific data type. Instead, use the HLA built-in compile-time function *@size* to compute the size of some data type. The *@size* function uses the following syntax:

```
@size( variable_or_type_name )
```

The *@size* function returns an unsigned integer constant that specifies the size of its parameter in bytes. So you should rewrite the previous call to *malloc* as follows:

```
malloc( @size( uns32 ) );
```

15. HLA provides some other memory allocation and deallocation routines as well. See the HLA Standard Library documentation for more details.

This call will properly allocate a sufficient amount of storage for the specified object, regardless of its type. While it is unlikely that the number of bytes required by an *uns32* object will ever change, this is not necessarily true for other data types; so you should always use *@size* rather than a literal constant in these calls.

Upon return from the *malloc* routine, the EAX register contains the address of the storage you have requested (see Figure 2.20):

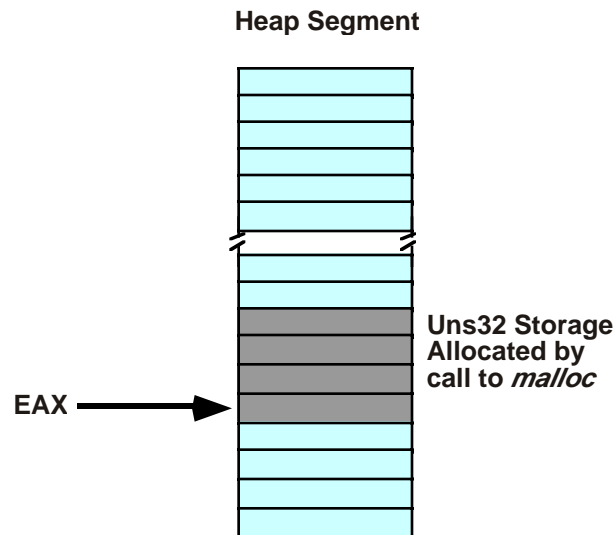


Figure 2.20 Call to Malloc Returns a Pointer in the EAX Register

To access the storage *malloc* allocates you must use a register indirect addressing mode. The following code sequence demonstrates how to assign the value 1234 to the *uns32* variable *malloc* creates:

```
malloc( @size( uns32 ));
mov( 1234, (type uns32 [eax]));
```

Note the use of the type coercion operation. This is necessary in this example because anonymous variables don't have a type associated with them and the constant 1234 could be a *word* or *dword* value. The type coercion operator eliminates the ambiguity.

A call to the *malloc* routine is not guaranteed to succeed. If there isn't a single contiguous block of free memory in the heap segment that is large enough to satisfy the request, then the *malloc* routine will raise an *ex.MemoryAllocationFailure* exception. If you do not provide a TRY..EXCEPTION..ENDTRY handler to deal with this situation, a memory allocation failure will cause your program to abort execution. Since most programs do not allocate massive amounts of dynamic storage using *malloc*, this exception rarely occurs. However, you should never assume that the memory allocation will always occur without error.

When you are done using a value that *malloc* allocates on the heap, you can release the storage (that is, mark it as "no longer in use") by calling the *free* procedure. The *free* routine requires a single parameter that must be an address that was a previous return value of the *malloc* routine (that you have not already freed). The following code fragment demonstrates the nature of the *malloc/free* pairing:

```
malloc( @size( uns32));

<< use the storage pointed at by EAX >>
<< Note: this code must not modify EAX >>

free( eax );
```

This code demonstrates a very important point - in order to properly free the storage that *malloc* allocates, you must preserve the value that *malloc* returns. There are several ways to do this if you need to use EAX

for some other purpose; you could save the pointer value on the stack using PUSH and POP instructions or you could save EAX's value in a variable until you need to free it.

Storage you release is available for reuse by future calls to the *malloc* routine. Like automatic variables you declare in the VAR section, the ability to allocate storage while you need it and then free the storage for other use when you are done with it improves the memory efficiency of your program. By deallocating storage once you are finished with it, your program can reuse that storage for other purposes allowing your program to operate with less memory than it would if you statically allocated storage for the individual objects.

There are several problems that can occur when you use pointers. You should be aware of a few common errors that beginning programmers make when using dynamic storage allocation routines like *malloc* and *free*:

- Mistake #1: Continuing to refer to storage after you free it. Once you return storage to the system via the call to *free*, you should no longer access that storage. Doing so may cause a protection fault or, worse yet, corrupt other data in your program without indicating an error.
- Mistake #2: Calling *free* twice to release a single block of storage. Doing so may accidentally free some other storage that you did not intend to release or, worse yet, it may corrupt the system memory management tables.

A later chapter will discuss some additional problems you will typically encounter when dealing with dynamically allocated storage.

The examples thus far in this section have all allocated storage for a single unsigned 32-bit object. Obviously you can allocate storage for any data type using a call to *malloc* by simply specifying the size of that object as *malloc*'s parameter. It is also possible to allocate storage for a sequence of contiguous objects in memory when calling *malloc*. For example, the following code will allocate storage for a sequence of 8 characters:

```
malloc( @size( char ) * 8 );
```

Note the use of the constant expression to compute the number of bytes required by an eight-character sequence. Since “@size(char)” always returns a constant value (one in this case), the compiler can compute the value of the expression “@size(char) * 8” without generating any extra machine instructions.

Calls to *malloc* always allocate multiple bytes of storage in contiguous memory locations. Hence the former call to *malloc* produces the sequence appearing in Figure 2.21:

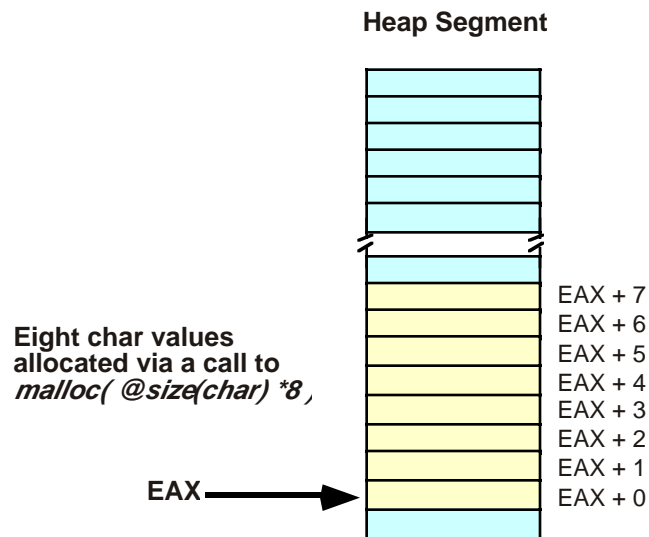


Figure 2.21 Allocating a Sequence of Eight Character Objects Using Malloc

To access these extra character values you use an offset from the base address (contained in EAX upon return from *malloc*). For example, “MOV(CH, [EAX + 2]);” stores the character found in CH into the third byte that *malloc* allocates. You can also use an addressing mode like “[EAX + EBX]” to step through each of the allocated objects under program control. For example, the following code will set all the characters in a block of 128 bytes to the NULL character (#0):

```
malloc( 128 );
for( mov( 0, ebx ); ebx < 128; add( 1, ebx ) ) do

    mov( 0, (type byte [eax+ebx]) );

endfor;
```

The chapter on arrays, later in this text, discusses additional ways to deal with blocks of memory.

2.9 The INC and DEC Instructions

As the example in the last section indicates, indeed, as several examples up to this point have indicated, adding or subtracting one from a register or memory location is a very common operation. In fact, this operation is so common that Intel’s engineer’s included a pair of instructions to perform these specific operations: the INC (increment) and DEC (decrement) instructions.

The INC and DEC instructions use the following syntax:

```
inc( mem/reg );
dec( mem/reg );
```

The single operand can be any legal eight-bit, 16-bit, or 32-bit register or memory operand. The INC instruction will add one to the specified operand, the DEC instruction will subtract one from the specified operand.

These two instructions are slightly more efficient (they are smaller) than the corresponding ADD or SUB instructions. There is also one slight difference between these two instructions and the corresponding ADD or SUB instructions: they do not affect the carry flag.

As an example of the INC instruction, consider the example from the previous section, recoded to use INC rather than ADD:

```

malloc( 128 );
for( mov( 0, ebx ); ebx < 128; inc( ebx ) ) do

    mov( 0, (type byte [eax+ebx]) );

endfor;

```

2.10 Obtaining the Address of a Memory Object

In the section “The Register Indirect Addressing Modes” on page 159 this chapter discusses how to use the address-of operator, “&”, to take the address of a static variable¹⁶. Unfortunately, you cannot use the address-of operator to take the address of an automatic variable (one you declare in the VAR section), you cannot use it to compute the address of an anonymous variable, nor can you use this operator to take the address of a memory reference that uses an indexed or scaled indexed addressing mode (even if a static variable is part of the address expression). You may only use the address-of operator to take the address of a static variable that uses the displacement-only memory addressing mode. Often, you will need to take the address of other memory objects as well; fortunately, the 80x86 provides the *load effective address* instruction, LEA, to give you this capability.

The LEA instruction uses the following syntax¹⁷:

```
lea( reg32, Memory_operand );
```

The first operand must be a 32-bit register, the second operand can be any legal memory reference using any valid memory addressing mode. This instruction will load the address of the specified memory location into the register. This instruction does not modify the value of the memory operand in any way, nor does it reference that value in memory.

Once you load the effective address of a memory location into a 32-bit general purpose register, you can use the register indirect, indexed, or scaled indexed addressing modes to access the data at the specified memory address. For example, consider the following code:

```

static
b:byte; @nostorage;
    byte 7, 0, 6, 1, 5, 2, 4, 3;
    .
    .
    .
lea( ebx, b );
for( mov( 0, ecx ); ecx < 8; inc( ecx ) ) do

    stdout.put( "[ebx+ecx]=", (type byte [ebx+ecx]), nl );

endwhile;

```

This code steps through each of the eight bytes following the *b* label in the STATIC section and prints their values. Note the use of the “[ebx+ecx]” addressing mode. The EBX register holds the base address of the list (that is, the address of the first item in the list) and ECX contains the byte index into the list.

```
program testCls;
```

16. A static variable is one that you declare in the *static*, *readonly*, *storage*, or *data* sections of your program.

17. Actually, the lea instruction allows the operands to appear in either order since there is no ambiguity. However, the standard syntax is to specify the register as the first operand and the memory location as the second operand.

```

#include( "stdlib.hhf" );
begin testCls;

    // Throw up some text to prove that
    // this program really clears the screen:

    stdout.put
    (
        nl,
        "HLA console.cls() Test Routine", nl
        "-----", nl
        nl
        "This routine will clear the screen and move the cursor to (0,0),", nl
        "then it will print a short message and quit", nl
        nl
        "Press the Enter key to continue:"
    );

    // Make the user hit Enter to continue. This is so that they
    // can see that the screen is not blank.

    stdin.readLine();

    // Okay, clear the screen and print a simple message:

    console.cls();
    stdout.put( "The screen was cleared.", nl );

end testCls;
program testGotoxy;
#include( "stdlib.hhf" );

var
    x:int16;
    y:int16;

begin testGotoxy;

    // Throw up some text to prove that
    // this program really clears the screen:

    stdout.put
    (
        nl,
        "HLA console.gotoxy() Test Routine", nl,
        "-----", nl,
        nl,
        "This routine will clear the screen then demonstrate the use", nl,
        "of the gotoxy routine to position the cursor at various", nl,
        "points on the screen.",nl,
        nl,
        "Press the Enter key to continue:"
    );

    // Make the user hit Enter to continue. This is so that they
    // can control when they see the effect of console.gotoxy.

    stdin.readLine();

    // Okay, clear the screen:

```

```

console.cls();

// Now demonstrate the gotoxy routine:

console.gotoxy( 5,10 );
stdout.put( "(5,10)" );

console.gotoxy( 10, 5 );
stdout.put( "(10,5)" );

mov( 20, x );
for( mov( 0,y ); y<20; inc(y)) do

    console.gotoxy( y, x );
    stdout.put( "(, x, ",", y, ")" );
    inc( x );

endfor;

end testGotoxy;
program testGetxy;
#include( "stdlib.hhf" );

var
    x:uns32;
    y:uns32;

begin testGetxy;

    // Begin by getting the current cursor position

    console.getX();
    mov( eax, x );

    console.getY();
    mov( eax, y );

    // Clear the screen and print a banner message:

    console.cls();

    stdout.put
    (
        nl,
        "HLA console.GetX() and console.GetY() Test Routine", nl,
        "-----", nl,
        nl,
        "This routine will clear the screen then demonstrate the use", nl,
        "of the GetX and GetY routines to reposition the cursor", nl,
        "to its original location on the screen.",nl,
        nl,
        "Press the Enter key to continue:"
    );

    // Make the user hit Enter to continue. This is so that they
    // can control when they see the effect of console.gotoxy.

    stdin.readLn();

```

```

// Now demonstrate the GetX and GetY routines by calling
// the gotoxy routine to move the cursor back to its original
// position.

console.gotoxy( (type uns16 y), (type uns16 x) );
stdout.put( "*<- Cursor was originally here.", nl );

end testGetxy;
program testSetOutputAttr;
#include( "stdlib.hhf" );

var
    x:uns32;
    y:uns32;

begin testSetOutputAttr;

    // Clear the screen and print a banner message:

    console.cls();

    console.setOutputAttr( win.fgnd_LightRed | win.bgnd_Black );
    stdout.put
    (
        nl,
        "HLA console.setOutputAttr Test Routine", nl,
        "-----", nl,
        nl,
        "Press the Enter key to continue:"
    );

    // Make the user hit Enter to continue. This is so that they
    // can control when they see the effect of console.gotoxy.

    stdin.readLine();

    console.setOutputAttr( win.fgnd_Yellow | win.bgnd_Blue );
    stdout.put
    (
        "          ", nl
        " In blue and yellow ", nl,
        "          ", nl,
        " Press Enter to continue ", nl,
        "          ", nl,
        nl
    );
    stdin.readLine();

    // Note: set the attributes back to black and white when
    // the program exits so the console window doesn't continue
    // displaying text in Blue and Yellow.

    console.setOutputAttr( win.fgnd_White | win.bgnd_Black );

end testSetOutputAttr;
program testFillRect;
#include( "stdlib.hhf" );

var

```

```

x:uns32;
y:uns32;

begin testFillRect;

    console.setOutputAttr( win.fgnd_LightRed | win.bgnd_Black );
    stdout.put
    (
        nl,
        "HLA console.fillRect Test Routine", nl,
        "-----", nl,
        nl,
        "Press the Enter key to continue:"
    );

    // Make the user hit Enter to continue.

    stdin.readLine();
    console.cls();

    // Test outputting rectangular blocks of color.
    // Note that the blocks are always filled with spaces,
    // so there is no need to specify a foreground color.

    console.fillRect( 2, 50, 5, 55, ' ', win.bgnd_Black );
    console.fillRect( 6, 50, 9, 55, ' ', win.bgnd_Green );
    console.fillRect( 10, 50, 13, 55, ' ', win.bgnd_Cyan );
    console.fillRect( 14, 50, 17, 55, ' ', win.bgnd_Red );
    console.fillRect( 18, 50, 21, 55, ' ', win.bgnd_Magenta );

    console.fillRect( 2, 60, 5, 65, ' ', win.bgnd_Brown );
    console.fillRect( 6, 60, 9, 65, ' ', win.bgnd_LightGray );
    console.fillRect( 10, 60, 13, 65, ' ', win.bgnd_DarkGray );
    console.fillRect( 14, 60, 17, 65, ' ', win.bgnd_LightBlue );
    console.fillRect( 18, 60, 21, 65, ' ', win.bgnd_LightGreen );

    console.fillRect( 2, 70, 5, 75, ' ', win.bgnd_LightCyan );
    console.fillRect( 6, 70, 9, 75, ' ', win.bgnd_LightRed );
    console.fillRect( 10, 70, 13, 75, ' ', win.bgnd_LightMagenta );
    console.fillRect( 14, 70, 17, 75, ' ', win.bgnd_Yellow );
    console.fillRect( 18, 70, 21, 75, ' ', win.bgnd_White );

    // Note: set the attributes back to black and white when
    // the program exits so the console window doesn't continue
    // displaying text in Blue and Yellow.

    console.setOutputAttr( win.fgnd_White | win.bgnd_Black );

end testFillRect;
program testPutsx;
#include( "stdlib.hhf" );

var
    x:uns32;
    y:uns32;

begin testPutsx;

    // Clear the screen and print a banner message:

    console.cls();

```

```

// Note that console.puts always defaults to black and white text.
// The following setOutputAttr call proves this.

console.setOutputAttr( win.fgnd_LightRed | win.bgnd_Black );

// Display the text in black and white:

console.puts
(
    10,
    10,
    "HLA console.setOutputAttr Test Routine"
);
console.puts
(
    11,
    10,
    "-----"
);
console.puts
(
    13,
    10,
    "Press the Enter key to continue:"
);

// Make the user hit Enter to continue.

stdin.readLine();

// Demonstrate the console.putsx routine.
// Note that the colors set by putsx are
// "local" to this call. Hence, the current
// output attribute colors will not be affected
// by this call.

console.putsx
(
    15,
    15,
    win.bgnd_White | win.fgnd_Blue,
    35,
    "Putsx at (15, 15) of length 35....."
);

console.putsx
(
    16,
    15,
    win.bgnd_White | win.fgnd_Red,
    40,
    "1234567890123456789012345678901234567890"
);

// Since the following is a stdout call, the text
// will use the current output attribute, which
// is the red/black attributes set at the beginning
// of this program.

```

```

console.gotoxy( 23, 0 );
stdout.put( "Press enter to continue:" );
stdin.readLn();

// Note: set the attributes back to black and white when
// the program exits.

console.setOutputAttr( win.fgnd_White | win.bgnd_Black );
console.cls();

end testPutsx;

```

2.11 Putting It All Together

This chapter discussed the 80x86 address modes and other related topics. It began by discussing the 80x86's register, displacement-only (direct), register indirect, and indexed addressing modes. A good knowledge of these addressing modes and their uses is essential if you want to write good assembly language programs. Although this chapter does not delve deeply into the use of each of these addressing modes, it does present their syntax and a few simple examples of each (later chapters will expand on how you use each of these addressing modes).

After discussing addressing modes, this chapter described how HLA and the operating system organizes your code and data in memory. At this point this chapter also discussed the HLA `STATIC`, `READONLY`, `STORAGE`, and `VAR` data declaration sections. The alignment of data in memory can affect the performance of your programs; therefore, when discussing this topic, this chapter also described how to properly align objects in memory to obtain the fastest executing code.

One special section of memory is the 80x86 stack. In addition to briefly discussing the stack, this chapter also described how to use the stack to save temporary values using the `PUSH` and `POP` instructions (and several variations on these instructions).

To a running program, a variable is really nothing more than a simple address in memory. In an HLA source file, however, you may specify the address and type of an object in memory using powerful address expressions and type coercion operators. This chapter discusses the syntax for these expressions and operators and gives several examples of why you would want to use them.

This chapter concludes by discussing two modules in the HLA Standard Library: the dynamic memory allocation routines (*malloc* and *free*).

Logic circuits are the basis for modern digital computer systems. To appreciate how computer systems operate you will need to understand digital logic and boolean algebra.

This chapter provides only a basic introduction to boolean algebra. That subject alone is often the subject of an entire textbook. This chapter concentrates on those subjects that support other chapters in this text.

Chapter Overview

Boolean logic forms the basis for computation in modern binary computer systems. You can represent any algorithm, or any electronic computer circuit, using a system of boolean equations. This chapter provides a brief introduction to boolean algebra, truth tables, canonical representation, of boolean functions, boolean function simplification, logic design, and combinatorial and sequential circuits.

This material is especially important to those who want to design electronic circuits or write software that controls electronic circuits. Even if you never plan to design hardware or write software than controls hardware, the introduction to boolean algebra this chapter provides is still important since you can use such knowledge to optimize certain complex conditional expressions within IF, WHILE, and other conditional statements.

The section on minimizing (optimizing) logic functions uses *Veitch Diagrams* or *Karnaugh Maps*. The optimizing techniques this chapter uses reduce the number of *terms* in a boolean function. You should realize that many people consider this optimization technique obsolete because reducing the number of terms in an equation is not as important as it once was. This chapter uses the mapping method as an example of boolean function optimization, not as a technique one would regularly employ. If you are interested in circuit design and optimization, you will need to consult a text on logic design for better techniques.

3.1 Boolean Algebra

Boolean algebra is a deductive mathematical system closed over the values zero and one (false and true). A *binary operator* \downarrow defined over this set of values accepts a pair of boolean inputs and produces a single boolean value. For example, the boolean AND operator accepts two boolean inputs and produces a single boolean output (the logical AND of the two inputs).

For any given algebra system, there are some initial assumptions, or *postulates*, that the system follows. You can deduce additional rules, theorems, and other properties of the system from this basic set of postulates. Boolean algebra systems often employ the following postulates:

☞ *☐Closure*. The boolean system is *closed* with respect to a binary operator if for every pair of boolean values, it produces a boolean result. For example, logical AND is closed in the boolean system because it accepts only boolean operands and produces only boolean results.

☞ *☐Commutativity*. A binary operator \downarrow is said to be commutative if $A \downarrow B = B \downarrow A$ for all possible boolean values A and B.

☞ *☐Associativity*. A binary operator \downarrow is said to be associative if

$$\downarrow (A \downarrow B) \downarrow C = A \downarrow (B \downarrow C)$$

∅

∅ for all boolean values A, B, and C.

∅ *Distribution*. Two binary operators ∅ and % are distributive if

∅ $A ∅ (B \% C) = (A ∅ B) \% (A ∅ C)$

∅

∅ for all boolean values A, B, and C.

∅ *Identity*. A boolean value I is said to be the *identity element* with respect to some binary operator ∅ if $A ∅ I = A$ for all boolean values A.

∅ *Inverse*. A boolean value I is said to be the *inverse element* with respect to some binary operator ∅ if $A ∅ I = B$ and $B ≠ A$ (i.e., B is the opposite value of A in a boolean system) for all boolean values A and B.

For our purposes, we will base boolean algebra on the following set of operators and values:

The two possible values in the boolean system are zero and one. Often we will call these values false and true (respectively).

The symbol ∅ represents the logical AND operation; e.g., $A ∅ B$ is the result of logically ANDing the boolean values A and B. When using single letter variable names, this text will drop the ∅ symbol; Therefore, AB also represents the logical AND of the variables A and B (we will also call this the product of A and B).

The symbol + represents the logical OR operation ; e.g., $A + B$ is the result of logically ORing the boolean values A and B. (We will also call this the sum of A and B.)

Logical complement, negation, or not, is a unary operator. This text will use the (') symbol to denote logical negation. For example, A' denotes the logical NOT of A.

If several different operators appear in a single boolean expression, the result of the expression depends on the *precedence* of the operators. We ll use the following precedences (from highest to lowest) for the boolean operators: parenthesis, logical NOT, logical AND, then logical OR. The logical AND and OR operators are *left associative*. If two operators with the same precedence are adjacent, you must evaluate them from left to right. The logical NOT operation is right associative, although it would produce the same result using left or right associativity since it is a unary operator.

We will also use the following set of postulates:

P1 Boolean algebra is closed under the AND, OR, and NOT operations.

P2 The identity element with respect to ∅ is one and + is zero. There is no identity element with respect to logical NOT.

P3 The ∅ and + operators are commutative.

P4 ∅ and + are distributive with respect to one another. That is, $A ∅ (B + C) = (A ∅ B) + (A ∅ C)$ and $A + (B ∅ C) = (A + B) ∅ (A + C)$.

P5 For every value A there exists a value A' such that $A ∅ A' = 0$ and $A + A' = 1$. This value is the logical complement (or NOT) of A.

P6 ∅ and + are both associative. That is, $(A ∅ B) ∅ C = A ∅ (B ∅ C)$ and $(A + B) + C = A + (B + C)$.

You can prove all other theorems in boolean algebra using these postulates. This text will not go into the formal proofs of these theorems, however, it is a good idea to familiarize yourself with some important theorems in boolean algebra. A sampling includes:

- Th1: $A + A = A$
 Th2: $A \bar{A} = A$
 Th3: $A + 0 = A$
 Th4: $A \bar{1} = A$
 Th5: $A \bar{0} = 0$
 Th6: $A + 1 = 1$
 Th7: $(A + B) = A \bar{B}$
 Th8: $(A \bar{B}) = A + B$
 Th9: $A + A\bar{B} = A$
 Th10: $A \bar{(A + B)} = A$
 Th11: $A + AB = A+B$
 Th12: $A \bar{(A + B)} = AB$
 Th13: $AB + AB = A$
 Th14: $(A + B) \bar{(A + B)} = A$
 Th15: $A + A = 1$
 Th16: $A \bar{A} = 0$

Theorems seven and eight above are known as *DeMorgan's Theorems* after the mathematician who discovered them.

The theorems above appear in pairs. Each pair (e.g., Th1 & Th2, Th3 & Th4, etc.) form a *dual*. An important principle in the boolean algebra system is that of *duality*. Any valid expression you can create using the postulates and theorems of boolean algebra remains valid if you interchange the operators and constants appearing in the expression. Specifically, if you exchange the $\bar{}$ and $+$ operators and swap the 0 and 1 values in an expression, you will wind up with an expression that obeys all the rules of boolean algebra. *This does not mean the dual expression computes the same values*, it only means that both expressions are legal in the boolean algebra system. Therefore, this is an easy way to generate a second theorem for any fact you prove in the boolean algebra system.

Although we will not be proving any theorems for the sake of boolean algebra in this text, we will use these theorems to show that two boolean equations are identical. This is an important operation when attempting to produce *canonical representations* of a boolean expression or when simplifying a boolean expression.

3.2 Boolean Functions and Truth Tables

A boolean *expression* is a sequence of zeros, ones, and *literals* separated by boolean operators. A literal is a primed (negated) or unprimed variable name. For our purposes, all variable names will be a single alphabetic character. A boolean function is a specific boolean expression; we will generally give boolean functions the name F with a possible subscript. For example, consider the following boolean:

$$F_0 = AB+C$$

This function computes the logical AND of A and B and then logically ORs this result with C. If $A=1$, $B=0$, and $C=1$, then F_0 returns the value one ($1\bar{0} + 1 = 1$).

Another way to represent a boolean function is via a *truth table*. A previous chapter (see Logical Operations on Bits on page 65) used truth tables to represent the AND and OR functions. Those truth tables took the forms:

Table 13: AND Truth Table

AND	0	1
0	0	0
1	0	1

Table 14: OR Truth Table

OR	0	1
0	0	1
1	1	1

For binary operators and two input variables, this form of a truth table is very natural and convenient. However, reconsider the boolean function F_0 above. That function has *three* input variables, not two. Therefore, one cannot use the truth table format given above. Fortunately, it is still very easy to construct truth tables for three or more variables. The following example shows one way to do this for functions of three or four variables:

$F = AB + C$		BA			
		00	01	10	11
C	0	0	0	0	1
	1	1	1	1	1

$F = AB + CD$		BA			
		00	01	10	11
DC	00	0	0	0	1
	01	0	0	0	1
	10	0	0	0	1
	11	1	1	1	1

In the truth tables above, the four columns represent the four possible combinations of zeros and ones for A & B (B is the H.O. or leftmost bit, A is the L.O. or rightmost bit). Likewise the four rows in the second truth table above represent the four possible combinations of zeros and ones for the C and D variables. As before, D is the H.O. bit and C is the L.O. bit.

The following table shows another way to represent truth tables. This form has two advantages over the forms above — it is easier to fill in the table and it provides a compact representation for two or more functions.

C	B	A	F = ABC	F = AB + C	F = A+BC
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	1	0
1	0	1	0	1	1
1	1	0	0	1	1
1	1	1	1	1	1

Note that the truth table above provides the values for three separate functions of three variables.

Although you can create an infinite variety of boolean functions, they are not all unique. For example, $F=A$ and $F=AA$ are two different functions. By theorem two, however, it is easy to show that these two functions are equivalent, that is, they produce exactly the same outputs for all input combinations. If you fix the number of input variables, there are a finite number of unique boolean functions possible. For example, there are only 16 unique boolean functions with two inputs and there are only 256 possible boolean functions of three input variables. Given n input variables, there are 2^{2^n} (two raised to the two raised to the n^{th} power)¹ unique boolean functions of those n input values. For two input variables, $2^{2^2} = 2^4$ or 16 different functions. With three input variables there are $2^{2^3} = 2^8$ or 256 possible functions. Four input variables create 2^{2^4} or 2^{16} , or 65,536 different unique boolean functions.

When dealing with only 16 boolean functions, it is easy enough to name each function. The following table lists the 16 possible boolean functions of two input variables along with some common names for those functions:

Function #	Description
0	Zero or Clear. Always returns zero regardless of A and B input values.
1	Logical NOR ($\text{NOT}(A \text{ OR } B)) = (A+B)'$
2	Inhibition = AB' (A, not B). Also equivalent to $A > B$ or $B < A$.
3	NOT B. Ignores A and returns B'.
4	Inhibition = BA' (B, not A). Also equivalent to $B > A$ or $A < B$.
5	NOT A. Returns A' and ignores B
6	Exclusive-or (XOR) = $A \oplus B$. Also equivalent to $A \neq B$.
7	Logical NAND ($\text{NOT}(A \text{ AND } B)) = (A \cdot B)'$
8	Logical AND = $A \cdot B$. Returns A AND B.

1. In this context, the operator ****** means exponentiation.

Function #	Description
9	Equivalence = (A = B). Also known as exclusive-NOR (not exclusive-or).
10	Copy A. Returns the value of A and ignores B's value.
11	Implication, B implies A, or A + B'. (if B then A). Also equivalent to B >= A.
12	Copy B. Returns the value of B and ignores A's value.
13	Implication, A implies B, or B + A' (if A then B). Also equivalent to A >= B.
14	Logical OR = A+B. Returns A OR B.
15	One or Set. Always returns one regardless of A and B input values.

Beyond two input variables there are too many functions to provide specific names. Therefore, we will refer to the function's number rather than the function's name. For example, F_8 denotes the logical AND of A and B for a two-input function and F_{14} is the logical OR operation. Of course, the only problem is to determine a function's number. For example, given the function of three variables $F=AB+C$, what is the corresponding function number? This number is easy to compute by looking at the truth table for the function. If we treat the values for A, B, and C as bits in a binary number with C being the H.O. bit and A being the L.O. bit, they produce the binary numbers in the range zero through seven. Associated with each of these binary strings is a zero or one function result. If we construct a binary value by placing the function result in the bit position specified by A, B, and C, the resulting binary number is that function's number. Consider the truth table for $F=AB+C$:

```
CBA:  7  6  5  4  3  2  1  0
F=AB+C:1 1  1  1  1  0  0  0
```

If we treat the function values for F as a binary number, this produces the value F_8_{16} or 248_{10} . We will usually denote function numbers in decimal.

This also provides the insight into why there are 2^{2^n} different functions of n variables: if you have n input variables, there are 2^n bits in function's number. If you have m bits, there are 2^m different values. Therefore, for n input variables there are $m=2^n$ possible bits and 2^m or 2^{2^n} possible functions.

3.3 Algebraic Manipulation of Boolean Expressions

You can transform one boolean expression into an equivalent expression by applying the postulates and theorems of boolean algebra. This is important if you want to convert a given expression to a *canonical form* (a standardized form) or if you want to minimize the number of literals (primed or unprimed variables) or terms in an expression. Minimizing terms and expressions can be important because electrical circuits often consist of individual components that implement each term or literal for a given expression. Minimizing the expression allows the designer to use fewer electrical components and, therefore, can reduce the cost of the system.

Unfortunately, there are no fixed rules you can apply to optimize a given expression. Much like constructing mathematical proofs, an individual's ability to easily do these transformations is usually a function of experience. Nevertheless, a few examples can show the possibilities:

$$\begin{aligned}
 ab + ab' + a'b &= a(b+b') + a'b && \text{By P4} \\
 &= a \cdot 1 + a'b && \text{By P5}
 \end{aligned}$$

$$\begin{aligned}
&= a + a' b && \text{By Th4} \\
&= a + b && \text{By Th11}
\end{aligned}$$

$$\begin{aligned}
(a' b + a' b' + b')' &= (a' (b+b') + b')' && \text{By P4} \\
&= (a' \cdot 1 + b')' && \text{By P5} \\
&= (a' + b') && \text{By Th4} \\
&= ((ab)')' && \text{By Th8} \\
&= ab && \text{By definition of not}
\end{aligned}$$

$$\begin{aligned}
b(a+c) + ab' + bc' + c &= ba + bc + ab' + bc' + c && \text{By P4} \\
&= a(b+b') + b(c + c') + c && \text{By P4} \\
&= a \cdot 1 + b \cdot 1 + c && \text{By P5} \\
&= a + b + c && \text{By Th4}
\end{aligned}$$

Although these examples all use algebraic transformations to simplify a boolean expression, we can also use algebraic operations for other purposes. For example, the next section describes a canonical form for boolean expressions. We can use algebraic manipulation to produce canonical forms even though the canonical forms are rarely optimal.

3.4 Canonical Forms

Since there are a finite number of boolean functions of n input variables, yet an infinite number of possible logic expressions you can construct with those n input values, clearly there are an infinite number of logic expressions that are equivalent (i.e., they produce the same result given the same inputs). To help eliminate possible confusion, logic designers generally specify a boolean function using a *canonical*, or standardized, form. For any given boolean function there exists a unique canonical form. This eliminates some confusion when dealing with boolean functions.

Actually, there are several different canonical forms. We will discuss only two here and employ only the first of the two. The first is the so-called *sum of minterms* and the second is the *product of maxterms*. Using the duality principle, it is very easy to convert between these two.

A *term* is a variable or a product (logical AND) of several different literals. For example, if you have two variables, A and B, there are eight possible terms: A, B, A', B', A'B', A'B, AB', and AB. For three variables we have 26 different terms: A, B, C, A', B', C', A'B', A'B, AB', AB, A'C', A'C, AC', AC, B'C', B'C, BC', BC, A'B'C', AB'C', A'BC', ABC', A'B'C, AB'C, A'BC, and ABC. As you can see, as the number of variables increases, the number of terms increases dramatically. A *minterm* is a product containing exactly n literals. For example, the minterms for two variables are A'B', AB', A'B, and AB. Likewise, the minterms for three variables A, B, and C are A'B'C', AB'C', A'BC', ABC', A'B'C, AB'C, A'BC, and ABC. In general, there are 2^n minterms for n variables. The set of possible minterms is very easy to generate since they correspond to the sequence of binary numbers:

Binary Equivalent (CBA)	Minterm
000	A'B'C'
001	AB'C'
010	A'BC'
011	ABC'
100	A'B'C
101	AB'C
110	A'BC
111	ABC

We can specify *any* boolean function using a sum (logical OR) of minterms. Given $F_{248}=AB+C$ the equivalent canonical form is $ABC+A'BC+AB'C+A'B'C+ABC'$. Algebraically, we can show that these two are equivalent as follows:

$$\begin{aligned}
 ABC+A'BC+AB'C+A'B'C+ABC' &= BC(A+A') + B'C(A+A') + ABC' \quad \text{By P4} \\
 &= BC \cdot 1 + B'C \cdot 1 + ABC' \quad \text{By Th15} \\
 &= C(B+B') + ABC' \quad \text{By P4} \\
 &= C + ABC' \quad \text{By Th15 \& Th4} \\
 &= C + AB \quad \text{By Th11}
 \end{aligned}$$

Obviously, the canonical form is not the optimal form. On the other hand, there is a big advantage to the sum of minterms canonical form: it is very easy to generate the truth table for a function from this canonical form. Furthermore, it is also very easy to generate the logic equation from the truth table.

To build the truth table from the canonical form, simply convert each minterm into a binary value by substituting a 1 for unprimed variables and a 0 for primed variables. Then place a 1 in the corresponding position (specified by the binary minterm value) in the truth table:

1) Convert minterms to binary equivalents:

$$\begin{aligned}
 F_{248} &= CBA + CBA + CB A + CB A + C BA \\
 &= 111 + 110 + 101 + 100 + 011
 \end{aligned}$$

2) Substitute a one in the truth table for each entry above:

C	B	A	F = AB+C
0	0	0	
0	0	1	
0	1	0	
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Finally, put zeros in all the entries that you did not fill with ones in the first step above:

C	B	A	F = AB+C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Going in the other direction, generating a logic function from a truth table, is almost as easy. First, locate all the entries in the truth table with a one. In the table above, these are the last five entries. The number of table entries containing ones determines the number of minterms in the canonical equation. To generate the individual minterms, substitute A, B, or C for ones and A', B', or C' for zeros in the truth table above. Then compute the sum of these items. In the example above, F_{248} contains one for CBA = 111, 110, 101, 100, and 011. Therefore, $F_{248} = CBA + CBA' + CB'A + CB'A' + C'AB$. The first term, CBA, comes from the last entry in the table above. C, B, and A all contain ones so we generate the minterm CBA (or ABC, if you prefer). The second to last entry contains 110 for CBA, so we generate the minterm CBA'. Likewise, 101 produces CB'A; 100 produces CB'A', and 011 produces C'BA. Of course, the logical OR and logical AND operations are both commutative, so we can rearrange the terms within the minterms as we please and we can rearrange the minterms within the sum as we see fit. This process works equally well for any number of variables. Consider the function $F_{53504} = ABCD + A'BCD + A'B'CD + A'B'C'D$. Placing ones in the appropriate positions in the truth table generates the following:

D	C	B	A	$F = ABCD + A'BCD + A'B'CD + A'B'C'D$
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	1
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	1
1	1	0	1	
1	1	1	0	1
1	1	1	1	1

The remaining elements in this truth table all contain zero.

Perhaps the easiest way to generate the canonical form of a boolean function is to first generate the truth table for that function and then build the canonical form from the truth table. We'll use this technique, for example, when converting between the two canonical forms this chapter presents. However, it is also a simple matter to generate the sum of minterms form algebraically. By using the distributive law and theorem 15 ($A + A' = 1$) makes this task easy. Consider $F_{248} = AB + C$. This function contains two terms, AB and C , but they are not minterms. Minterms contain each of the possible variables in a primed or unprimed form. We can convert the first term to a sum of minterms as follows:

$$\begin{aligned}
 AB &= AB \cdot 1 && \text{By Th4} \\
 &= AB \cdot (C + C') && \text{By Th 15} \\
 &= ABC + ABC' && \text{By distributive law} \\
 &= CBA + C'BA && \text{By associative law}
 \end{aligned}$$

Similarly, we can convert the second term in F_{248} to a sum of minterms as follows:

$$\begin{aligned}
 C &= C \cdot 1 && \text{By Th4} \\
 &= C \cdot (A + A') && \text{By Th15} \\
 &= CA + CA' && \text{By distributive law} \\
 &= CA \cdot 1 + CA' \cdot 1 && \text{By Th4} \\
 &= CA \cdot (B + B') + CA' \cdot (B + B') && \text{By Th15} \\
 &= CAB + CAB' + CA'B + CA'B' && \text{By distributive law} \\
 &= CBA + CBA' + CB'A + CB'A' && \text{By associative law}
 \end{aligned}$$

The last step (rearranging the terms) in these two conversions is optional. To obtain the final canonical form for F_{248} we need only sum the results from these two conversions:

$$\begin{aligned} F_{248} &= (CBA + C'BA) + (CBA + CBA' + CB'A + CB'A') \\ &= CBA + CBA' + CB'A + CB'A' + C'BA \end{aligned}$$

Another way to generate a canonical form is to use *products of maxterms*. A maxterm is the sum (logical OR) of all input variables, primed or unprimed. For example, consider the following logic function G of three variables:

$$G = (A+B+C) \cdot (A'+B+C) \cdot (A+B'+C)$$

Like the sum of minterms form, there is exactly one product of maxterms for each possible logic function. Of course, for every product of maxterms there is an equivalent sum of minterms form. In fact, the function G , above, is equivalent to

$$F_{248} = CBA + CBA' + CB'A + CB'A' + C'BA = AB + C.$$

Generating a truth table from the product of maxterms is no more difficult than building it from the sum of minterms. You use the duality principle to accomplish this. Remember, the duality principle says to swap AND for OR and zeros for ones (and vice versa). Therefore, to build the truth table, you would first swap primed and non-primed literals. In G above, this would yield:

$$G = (A + B + C) \cdot (A + B + C) \cdot (A + B + C)$$

The next step is to swap the logical OR and logical AND operators. This produces

$$G = ABC + ABC + ABC$$

Finally, you need to swap all zeros and ones. This means that you store *zeros* into the truth table for each of the above entries and then fill in the rest of the truth table with ones. This will place a zero in entries zero, one, and two in the truth table. Filling the remaining entries with ones produces F_{248} .

You can easily convert between these two canonical forms by generating the truth table for one form and working backwards from the truth table to produce the other form. For example, consider the function of two variables, $F_7 = A + B$. The sum of minterms form is $F_7 = A'B + AB' + AB$. The truth table takes the form:

Table 15: F_7 (OR) Truth Table for Two Variables

F_7	A	B
0	0	0
0	1	0
1	0	1
1	1	1

Working backwards to get the product of maxterms, we locate all entries that have a zero result. This is the entry with A and B equal to zero. This gives us the first step of $G=A'B'$. However, we still need to invert all the vari-

ables to obtain $G=AB$. By the duality principle we need to swap the logical OR and logical AND operators obtaining $G=A+B$. This is the canonical *product of maxterms* form.

Since working with the product of maxterms is a little messier than working with sums of minterms, this text will generally use the sum of minterms form. Furthermore, the sum of minterms form is more common in boolean logic work. However, you will encounter both forms when studying logic design.

3.5 Simplification of Boolean Functions

Since there are an infinite variety of boolean functions of n variables, but only a finite number of unique boolean functions of those n variables, you might wonder if there is some method that will simplify a given boolean function to produce the optimal form. Of course, you can always use algebraic transformations to produce the optimal form, but using heuristics does not guarantee an optimal transformation. There are, however, two methods that *will* reduce a given boolean function to its optimal form: the map method and the prime implicants method. In this text we will only cover the mapping method, see any text on logic design for other methods.

Since for any logic function some optimal form must exist, you may wonder why we don't use the optimal form for the canonical form. There are two reasons. First, there may be several optimal forms. They are not guaranteed to be unique. Second, it is easy to convert between the canonical and truth table forms.

Using the map method to optimize boolean functions is practical only for functions of two, three, or four variables. With care, you can use it for functions of five or six variables, but the map method is cumbersome to use at that point. For more than six variables, attempting map simplifications by hand would not be wise².

The first step in using the map method is to build a two-dimensional truth table for the function (see Figure 3.1)

2. However, it's probably quite reasonable to write a *program* that uses the map method for seven or more variables.

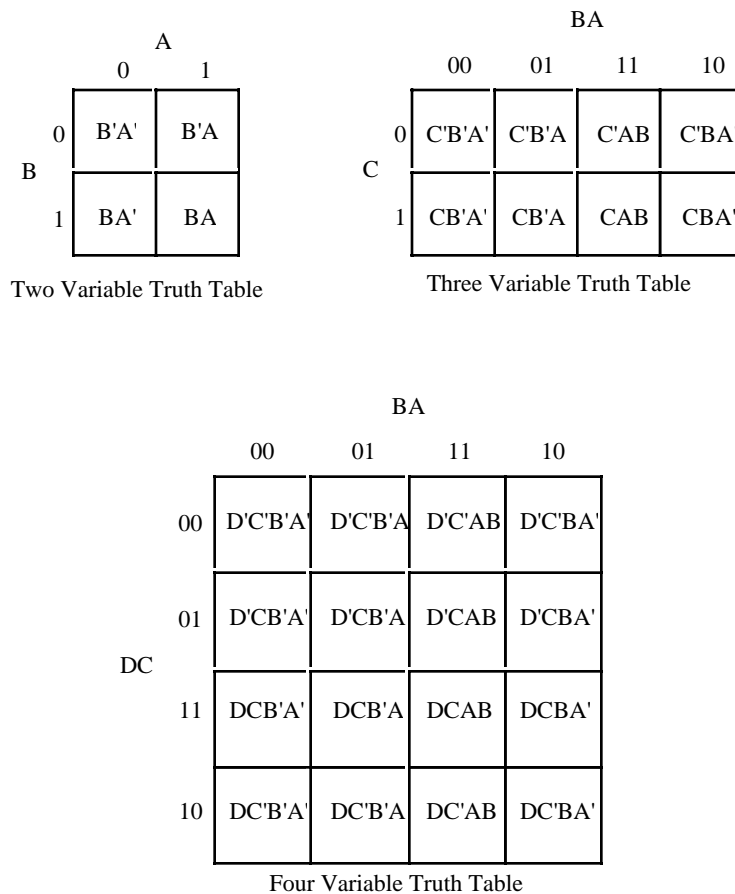


Figure 3.1 Two, Three, and Four Dimensional Truth Tables

Warning: Take a careful look at these truth tables. They do not use the same forms appearing earlier in this chapter. In particular, the progression of the values is 00, 01, 11, 10, not 00, 01, 10, 11. This is very important! If you organize the truth tables in a binary sequence, the mapping optimization method will not work properly. We will call this a *truth map* to distinguish it from the standard truth table.

Assuming your boolean function is in canonical form (sum of minterms), insert ones for each of the truth map entries corresponding to a minterm in the function. Place zeros everywhere else. For example, consider the function of three variables $F=C'B'A + C'BA' + C'BA + CB'A' + CB'A + CBA' + CBA$. Figure 3.2 shows the truth map for this function.

		BA			
		00	01	11	10
C	0	0	1	1	1
	1	1	1	1	1

$$F = C'B'A + C'BA' + C'BA + CB'A' + CB'A + CBA' + CBA.$$

Figure 3.2 A Simple Truth Map

The next step is to draw rectangles around rectangular groups of ones. The rectangles you enclose must have sides whose lengths are powers of two. For functions of three variables, the rectangles can have sides whose lengths are one, two, and four. The set of rectangles you draw must surround all cells containing ones in the truth map. The trick is to draw all possible rectangles unless a rectangle would be completely enclosed within another. In the truth map in Figure 3.3 there are three such rectangles (see Figure 3.3)

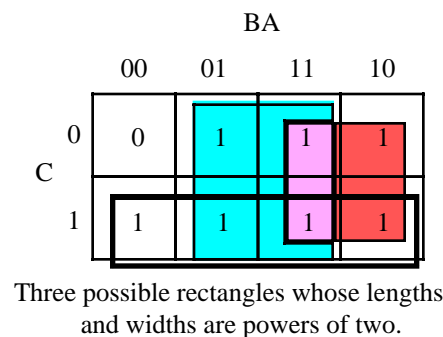


Figure 3.3 Surrounding Rectangular Groups of Ones in a Truth Map

Each rectangle represents a term in the simplified boolean function. Therefore, the simplified boolean function will contain only three terms. You build each term using the process of elimination. You eliminate any variables whose primed and unprimed form both appear within the rectangle. Consider the long skinny rectangle above that is sitting in the row where $C=1$. This rectangle contains both A and B in primed and unprimed form. Therefore, we can eliminate A and B from the term. Since the rectangle sits in the $C=1$ region, this rectangle represents the single literal C .

Now consider the blue square above. This rectangle includes C, C', B, B' and A . Therefore, it represents the single term A . Likewise, the red square above contains C, C', A, A' and B . Therefore, it represents the single term B .

The final, optimal, function is the sum (logical OR) of the terms represented by the three squares. Therefore, $F = A + B + C$. You do not have to consider the remaining squares containing zeros.

When enclosing groups of ones in the truth map, you must consider the fact that a truth map forms a *torus* (i.e., a doughnut shape). The right edge of the map *wraps around* to the left edge (and vice-versa). Likewise, the top edge *wraps around* to the bottom edge. This introduces additional possibilities when surrounding groups of

ones in a map. Consider the boolean function $F=C'B'A' + C'BA' + CB'A' + CBA'$. Figure 3.4 shows the truth map for this function.

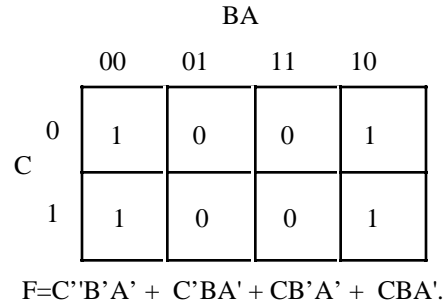


Figure 3.4 Truth Map for $F=C'B'A' + C'BA' + CB'A' + CBA'$

At first glance, you would think that there are two possible rectangles here as Figure 3.5 shows.

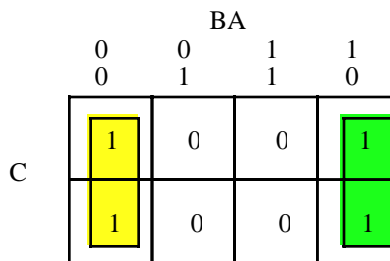


Figure 3.5 First Attempt at Surrounding Rectangles Formed by Ones

However, because the truth map is a continuous object with the right side and left sides connected, we can form a single, square rectangle, as Figure 3.6 shows.

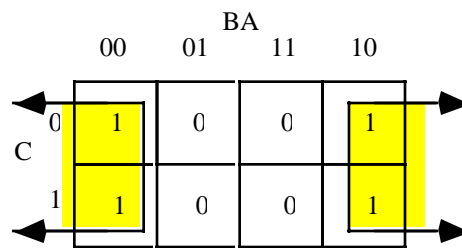


Figure 3.6 Correct Rectangle for the Function

So what? Why do we care if we have one rectangle or two in the truth map? The answer is because the larger the rectangles are, the more terms they will eliminate. The fewer rectangles that we have, the fewer terms will appear in the final boolean function. For example, the former example with two rectangles generates a function with two terms. The first rectangle (on the left) eliminates the C variable, leaving $A'B'$ as its term. The second rectangle, on the right, also eliminates the C variable, leaving the term BA' . Therefore, this truth map would produce the equation $F=A'B' + A'B$. We know this is not optimal, see Th 13. Now consider the second truth map above. Here we have a single rectangle so our boolean function will only have a single

term. Obviously this is more optimal than an equation with two terms. Since this rectangle includes both C and C' and also B and B', the only term left is A'. This boolean function, therefore, reduces to $F=A'$.

There are only two cases that the truth map method cannot handle properly: a truth map that contains all zeros or a truth map that contains all ones. These two cases correspond to the boolean functions $F=0$ and $F=1$ (that is, the function number is 2^{n-1}), respectively. These functions are easy to generate by inspection of the truth map.

An important thing you must keep in mind when optimizing boolean functions using the mapping method is that you always want to pick the largest rectangles whose sides' lengths are a power of two. You must do this even for overlapping rectangles (unless one rectangle encloses another). Consider the boolean function $F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA$. This produces the truth map appearing in Figure 3.7.

		BA			
		00	01	11	10
C	0	1	0	1	1
	1	1	0	1	1

Figure 3.7 Truth Map for $F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA$

The initial temptation is to create one of the sets of rectangles found in Figure 3.8. However, the correct mapping appears in Figure 3.9

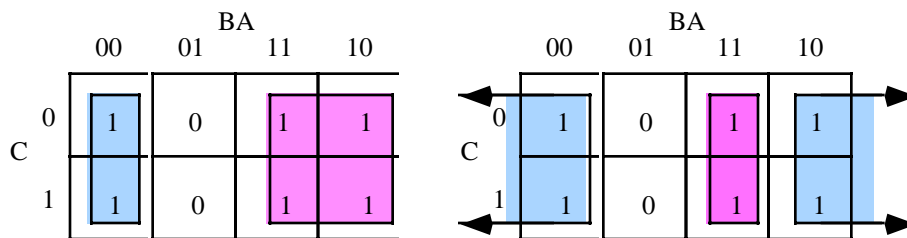


Figure 3.8 Obvious Choices for Rectangles

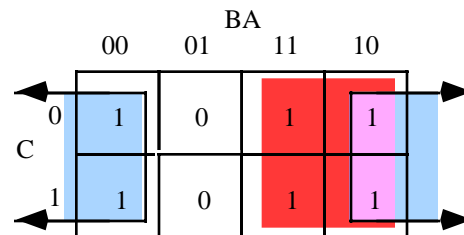


Figure 3.9 Correct Set of Rectangles for $F = C'B'A' + C'BA' + CB'A' + C'AB + CBA' + CBA$

All three mappings will produce a boolean function with two terms. However, the first two will produce the expressions $F = B + A'B'$ and $F = AB + A'$. The third form produces $F = B + A'$. Obviously, this last form is better optimized than the other two forms (see theorems 11 and 12).

For functions of three variables, the size of the rectangle determines the number of terms it represents:

- A rectangle enclosing a single square represents a minterm. The associated term will have three literals (assuming we're working with functions of three variables).
- A rectangle surrounding two squares containing ones represents a term containing two literals.
- A rectangle surrounding four squares containing ones represents a term containing a single literal.
- A rectangle surrounding eight squares represents the function $F = 1$.

Truth maps you create for functions of four variables are even trickier. This is because there are lots of places rectangles can hide from you along the edges. Figure 3.10 shows some possible places rectangles can hide.

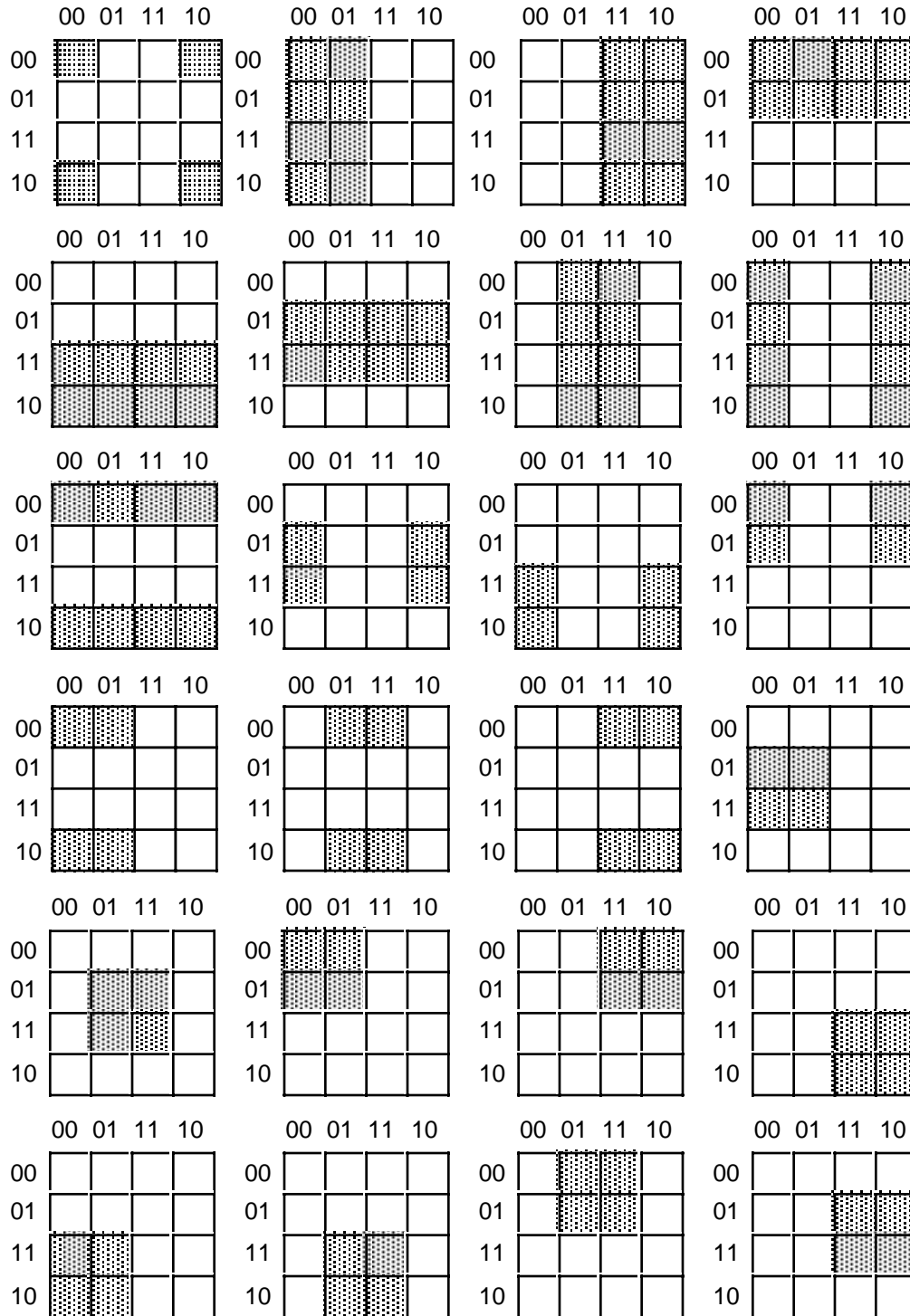


Figure 3.10 Partial Pattern List for 4x4 Truth Map

This list of patterns doesn't even begin to cover all of them! For example, these diagrams show none of the 1x2 rectangles. You must exercise care when working with four variable maps to ensure you select the largest possible rectangles, especially when overlap occurs. This is particularly important with you have a rectangle next to an edge of the truth map.

As with functions of three variables, the size of the rectangle in a four variable truth map controls the number of terms it represents:

- A rectangle enclosing a single square represents a minterm. The associated term will have four literals.
- A rectangle surrounding two squares containing ones represents a term containing three literals.
- A rectangle surrounding four squares containing ones represents a term containing two literals.
- A rectangle surrounding eight squares containing ones represents a term containing a single literal.
- A rectangle surrounding sixteen squares represents the function $F=1$.

This last example demonstrates an optimization of a function containing four variables. The function is $F = D'C'B'A' + D'C'B'A + D'C'BA + D'CB'A + D'CB'A + DCBA + DCB'A + DCBA + DC'B'A' + DC'BA'$, the truth map appears in Figure 3.11.

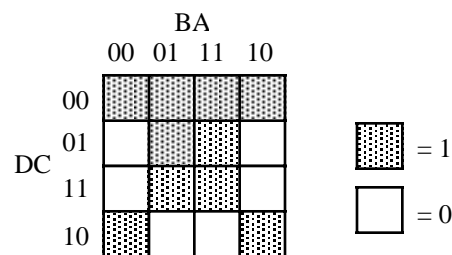


Figure 3.11 Truth Map for $F = D'C'B'A' + D'C'B'A + D'C'BA + D'CB'A + D'CB'A + DCBA + DCB'A + DCBA + DC'B'A' + DC'BA'$

Here are two possible sets of maximal rectangles for this function, each producing three terms (see Figure 3.12). Both functions are equivalent; both are as optimal as you can get³. Either will suffice for our purposes.

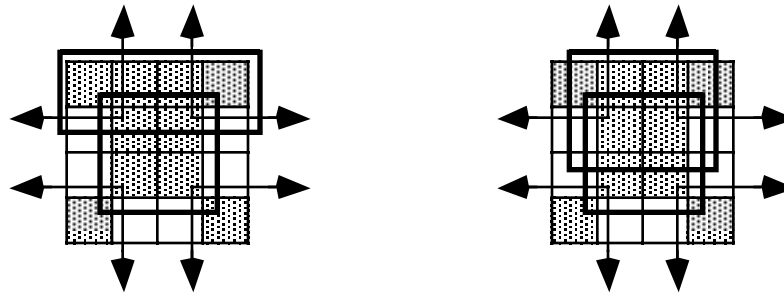


Figure 3.12 Two Combinations of Surrounded Values Yielding Three Terms

First, let's consider the term represented by the rectangle formed by the four corners. This rectangle contains $B, B', D,$ and D' ; so we can eliminate those terms. The remaining terms contained within these rectangles are C' and A' , so this rectangle represents the term $C'A'$.

3. Remember, there is no guarantee that there is a unique optimal solution.

The second rectangle, common to both maps in Figure 3.12, is the rectangle formed by the middle four squares. This rectangle includes the terms $A, B, B', C, D,$ and D' . Eliminating $B, B', D,$ and D' (since both primed and unprimed terms exist), we obtain CA as the term for this rectangle.

The map on the left in Figure 3.12 has a third term represented by the top row. This term includes the variables A, A', B, B', C' and D' . Since it contains $A, A', B,$ and B' , we can eliminate these terms. This leaves the term $C'D'$. Therefore, the function represented by the map on the left is $F=C'A' + CA + C'D'$.

The map on the right in Figure 3.12 has a third term represented by the top/middle four squares. This rectangle subsumes the variables $A, B, B', C, C',$ and D' . We can eliminate $B, B', C,$ and C' since both primed and unprimed versions appear, this leaves the term AD' . Therefore, the function represented by the function on the right is $F=C'A' + CA + AD'$.

Since both expressions are equivalent, contain the same number of terms, and the same number of operators, either form is equivalent. Unless there is another reason for choosing one over the other, you can use either form.

3.6 What Does This Have To Do With Computers, Anyway?

Although there is a tenuous relationship between boolean functions and boolean expressions in programming languages like C or Pascal, it is fair to wonder why we're spending so much time on this material. However, the relationship between boolean logic and computer systems is much stronger than it first appears. There is a one-to-one relationship between boolean functions and electronic circuits. Electrical engineers who design CPUs and other computer related circuits need to be intimately familiar with this stuff. Even if you never intend to design your own electronic circuits, understanding this relationship is important if you want to make the most of any computer system.

3.6.1 Correspondence Between Electronic Circuits and Boolean Functions

There is a one-to-one correspondence between an electrical circuits and boolean functions. For any boolean function you can design an electronic circuit and vice versa. Since boolean functions only require the AND, OR, and NOT boolean operators⁴, we can construct any electronic circuit using these operations exclusively. The boolean AND, OR, and NOT functions correspond to the following electronic circuits, the AND, OR, and inverter (NOT) gates (see Figure 3.13).

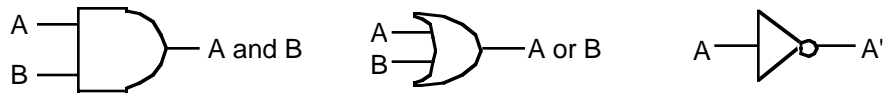


Figure 3.13 AND, OR, and Inverter (NOT) Gates

One interesting fact is that you only need a single gate type to implement *any* electronic circuit. This gate is the NAND gate, shown in Figure 3.14.

4. We know this is true because these are the only operators that appear within canonical forms.

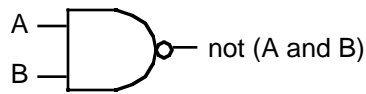


Figure 3.14 The NAND Gate

To prove that we can construct any boolean function using only NAND gates, we need only show how to build an inverter (NOT), an AND gate, and an OR gate from a NAND (since we can create any boolean function using only AND, NOT, and OR). Building an inverter is easy, just connect the two inputs together (see Figure 3.15).

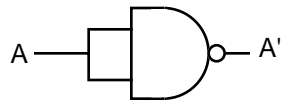


Figure 3.15 Inverter Built from a NAND Gate

Once we can build an inverter, building an AND gate is easy – just invert the output of a NAND gate. After all, NOT (NOT (A AND B)) is equivalent to A AND B (see Figure 3.16). Of course, this takes two NAND gates to construct a single AND gate, but no one said that circuits constructed only with NAND gates would be optimal, only that it is possible.

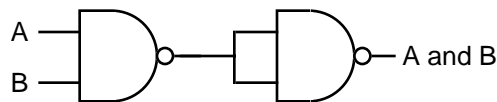


Figure 3.16 Constructing an AND Gate From Two NAND Gates

The remaining gate we need to synthesize is the logical-OR gate. We can easily construct an OR gate from NAND gates by applying DeMorgan's theorems.

$(A \text{ or } B)'$	$= A' \text{ and } B'$	DeMorgan's Theorem.
$A \text{ or } B$	$= (A' \text{ and } B')'$	Invert both sides of the equation.
$A \text{ or } B$	$= A' \text{ nand } B'$	Definition of NAND operation.

By applying these transformations, you get the circuit in Figure 3.17.

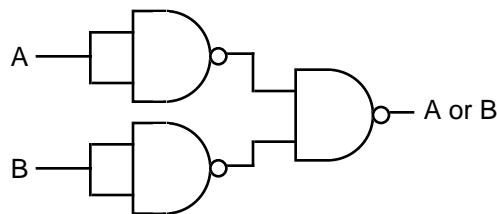


Figure 3.17 Constructing an OR Gate from NAND Gates

Now you might be wondering why we would even bother with this. After all, why not just use logical AND, OR, and inverter gates directly? There are two reasons for this. First, NAND gates are generally less expensive to build than other gates.

Second, it is also much easier to build up complex integrated circuits from the same basic building blocks than it is to construct an integrated circuit using different basic gates.

Note, by the way, that it is possible to construct any logic circuit using only NOR gates⁵. The correspondence between NAND and NOR logic is orthogonal to the correspondence between the two canonical forms appearing in this chapter (sum of minterms vs. product of maxterms). While NOR logic is useful for many circuits, most electronic designs use NAND logic.

3.6.2 Combinatorial Circuits

A combinatorial circuit is a system containing basic boolean operations (AND, OR, NOT), some inputs, and a set of outputs. Since each output corresponds to an individual logic function, a combinatorial circuit often implements several different boolean functions. It is very important that you remember this fact – each output represents a different boolean function.

A computer's CPU is built up from various combinatorial circuits. For example, you can implement an addition circuit using boolean functions. Suppose you have two one-bit numbers, A and B. You can produce the one-bit sum and the one-bit carry of this addition using the two boolean functions:

$$\begin{aligned} S &= AB' + A'B && \text{Sum of A and B.} \\ C &= AB && \text{Carry from addition of A and B.} \end{aligned}$$

These two boolean functions implement a *half-adder*. Electrical engineers call it a half adder because it adds two bits together but cannot add in a carry from a previous operation. A *full adder* adds three one-bit inputs (two bits plus a carry from a previous addition) and produces two outputs: the sum and the carry. The two logic equations for a full adder are

$$\begin{aligned} S &= A'B'C_{in} + A'BC_{in}' + AB'C_{in}' + ABC_{in} \\ C_{out} &= AB + AC_{in} + BC_{in} \end{aligned}$$

Although these logic equations only produce a single bit result (ignoring the carry), it is easy to construct an n-bit sum by combining adder circuits (see Figure 3.18). So, as this example clearly illustrates, we can use logic functions to implement arithmetic and boolean operations.

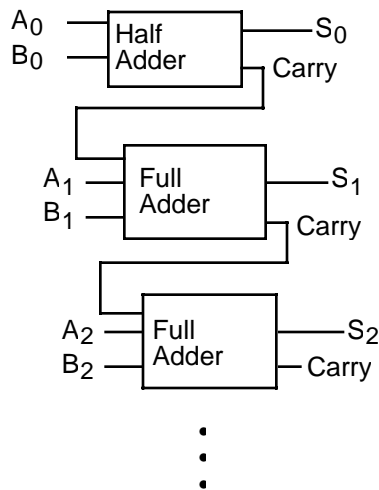


Figure 3.18 Building an N-Bit Adder Using Half and Full Adders

Another common combinatorial circuit is the *seven-segment decoder*. This is a combinatorial circuit that accepts four inputs and determines which of the segments on a seven-segment LED display should be on (logic one) or off (logic zero). Since a seven segment display contains seven output values (one for each segment), there will be seven logic functions associ-

5. NOR is NOT (A OR B).

ated with the display (segment zero through segment six). See Figure 3.19 for the segment assignments. Figure 3.20 shows the segment assignments for each of the ten decimal values.

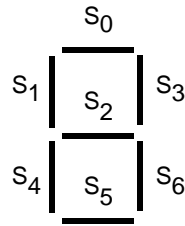


Figure 3.19 Seven Segment Display

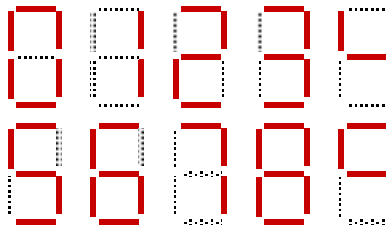


Figure 3.20 Seven Segment Values for “0” Through “9”

The four inputs to each of these seven boolean functions are the four bits from a binary number in the range 0..9. Let D be the H.O. bit of this number and A be the L.O. bit of this number. Each logic function should produce a one (segment on) for a given input if that particular segment should be illuminated. For example S_4 (segment four) should be on for binary values 0000, 0010, 0110, and 1000. For each value that illuminates a segment, you will have one minterm in the logic equation:

$$S_4 = D'C'B'A' + D'C'BA' + D'CBA' + DC'B'A'.$$

S_0 , as a second example, is on for values zero, two, three, five, six, seven, eight, and nine. Therefore, the logic function for S_0 is

$$S_0 = D'C'B'A' + D'C'BA' + D'C'BA + D'CB'A + D'CBA' + D'CBA + DC'B'A' + DC'B'A$$

You can generate the other five logic functions in a similar fashion.

Decoder circuits are among the more important circuits in computer system design. They provide the ability to recognize (or ‘decode’) a string of bits. One very common use for a decoder is memory expansion. For example, suppose a system designer wishes to install four (identical) 256 MByte memory modules in a system to bring the total to one gigabyte of RAM. These 256 MByte memory modules have 28 address lines ($A_0..A_{27}$) assuming each memory module is eight bits wide ($2^{28} \times 8$ bits is 256 MBytes)⁶. Unfortunately, if the system designer hooked up those four memory modules to the CPU’s address bus they would all respond to the same addresses on the bus. Pandemonium would result. To correct this problem, we need to select each memory module when a different set of addresses appear on the address bus. By adding a chip enable line to each of the memory modules and using a two-input, four-output decoder circuit, we can easily do this. See Figure 3.21 for the details.

6. Actually, most memory modules are wider than eight bits, so a real 256 MByte memory module will have fewer than 28 address lines, but we will ignore this technicality in this example.

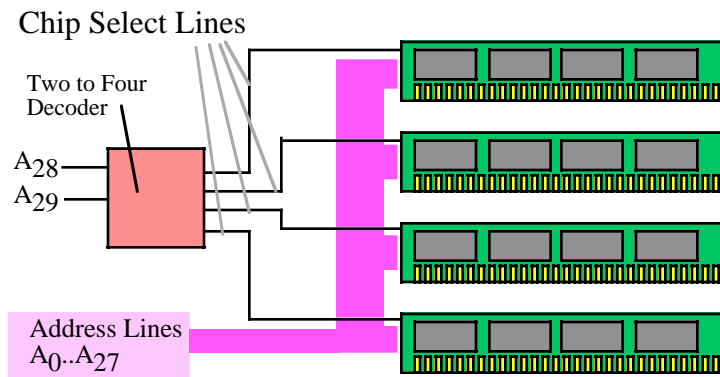


Figure 3.21 Adding Four 256 MByte Memory Modules to a System

The two-line to four-line decoder circuit in Figure 3.21 actually incorporates four different logic functions, one function for each of the outputs. Assume the inputs are A and B ($A=A_{28}$ and $B=A_{29}$) then the four output functions have the following (simple) equations:

$$\begin{aligned} Q_0 &= A' B' \\ Q_1 &= A B' \\ Q_2 &= A' B \\ Q_3 &= A B \end{aligned}$$

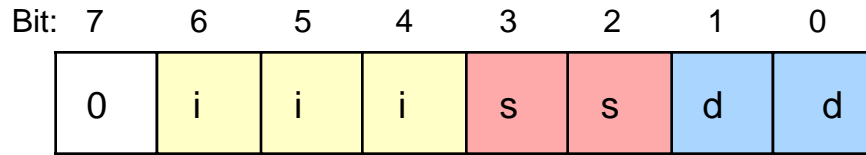
Following standard electronic circuit notation, these equations use “Q” to denote an output (electronic designers use “Q” for output rather than “O” because “Q” looks somewhat like an “O” and is more easily differentiated from zero). Also note that most circuit designers use *active low logic* for decoders and chip enables. This means that they enable a circuit with a low input value (zero) and disable the circuit with a high input value (one). Likewise, the output lines of a decoder chip are normally high and go low when the inputs select a given output line. This means that the equations above really need to be inverted for real-world examples. We’ll ignore this issue here and use positive (or active high) logic⁷.

Another big use for decoding circuits is to decode a byte in memory that represents a machine instruction in order to activate the corresponding circuitry to perform whatever tasks the instruction requires. We’ll cover this subject in much greater depth in a later chapter, but a simple example at this point will provide another solid example for using decoders.

Most modern (Von Neumann) computer systems represent machine instructions via values in memory. To execute an instruction the CPU fetches a value from memory, decodes that value, and then does the appropriate activity the instruction specifies. Obviously, the CPU uses decoding circuitry to decode the instruction. To see how this is done, let’s create a very simple CPU with a very simple instruction set. Figure 3.22 provides the instruction format (that is, it specifies all the numeric codes) for our simple CPU.

7. Electronic circuits often use active low logic because the circuits that employ them typically require fewer transistors to implement.

Instruction (opcode) Format:



iii	ss & dd
000 = MOV	00 = EAX
001 = ADD	01 = EBX
010 = SUB	10 = ECX
011 = MUL	11 = EDX
100 = DIV	
101 = AND	
110 = OR	
111 = XOR	

Figure 3.22 Instruction (opcode) Format for a Very Simple CPU

To determine the eight-bit operation code (opcode) for a given instruction, the first thing you do is choose the instruction you want to encode. Let's pick "MOV(EAX, EBX);" as our simple example. To convert this instruction to its numeric equivalent we must first look up the value for MOV in the iii table above; the corresponding value is 000. Therefore, we must substitute 000 for iii in the opcode byte.

Second, we consider our source operand. The source operand is EAX, whose encoding in the source operand table (ss & dd) is 00. Therefore, we substitute 00 for ss in the instruction opcode.

Next, we need to convert the destination operand to its numeric equivalent. Once again, we look up the value for this operand in the ss & dd table. The destination operand is EBX and its value is 01. So we substitute 01 for dd in our opcode byte. Assembling these three fields into the opcode byte (a packed data type), we obtain the following bit value: %00000001. Therefore, the numeric value \$1 is the value for the "MOV(EAX, EBX);" instruction (see Figure 3.23).

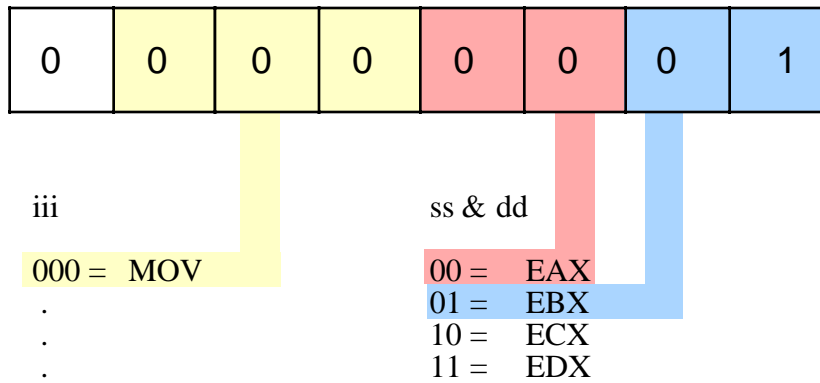
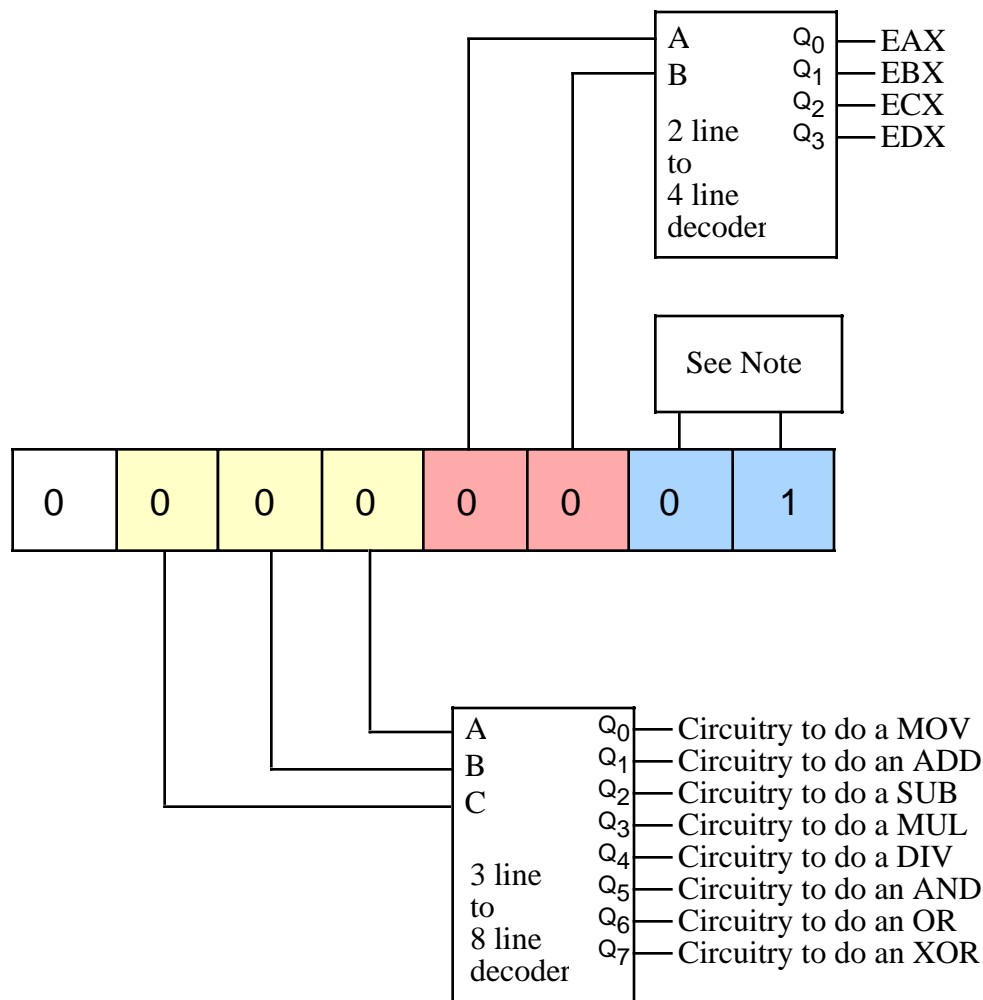


Figure 3.23 Encoding the MOV(EAX, EBX); Instruction

As another example, consider the “AND(EDX, ECX);” instruction. For this instruction the iii field is %101, the ss field is %11, and the dd field is %10. This yields the opcode %01011110 or \$5E. You may easily create other opcodes for our simple instruction set using this same technique.

Warning: please do not come to the conclusion that these encodings apply to the 80x86 instruction set. The encodings in this examples are highly simplified in order to demonstrate instruction decoding. They do not correspond to any real-life CPU, and they especially don’t apply to the x86 family.

In these past few examples we were actually *encoding* the instructions. Of course, the real purpose of this exercise is to discover how the CPU can use a decoder circuit to decode these instructions and execute them at run time. A typical set of decoder circuits for this might look like that in Figure 3.24:



Note: the circuitry attached to the destination register bits is identical to the circuitry for the source register bits.

Figure 3.24 Decoding Simple Machine Instructions

Notice how this circuit uses three separate decoders to decode the individual fields of the opcode. This is much less complex than creating a seven-line to 128-line decoder to decode each individual opcode. Of course, all that the circuit above will do is tell you which instruction and what operands a given opcode specifies. To actually execute this instruction you must supply additional circuitry to select the source and destination operands from an array of registers and act accordingly upon those operands. Such circuitry is beyond the scope of this chapter, so we'll save the juicy details for later.

Combinatorial circuits are the basis for many components of a basic computer system. You can construct circuits for addition, subtraction, comparison, multiplication, division, and many other operations using combinatorial logic.

3.6.3 Sequential and Clocked Logic

One major problem with combinatorial logic is that it is *memoryless*. In theory, all logic function outputs depend only on the current inputs. Any change in the input values is immediately reflected in the outputs⁸. Unfortunately, computers need the ability to *remember* the results of past computations. This is the domain of sequential or clocked logic.

A *memory cell* is an electronic circuit that remembers an input value after the removal of that input value. The most basic memory unit is the *set/reset flip-flop*. You can construct an *SR flip-flop* using two NAND gates, as shown in Figure 3.25.

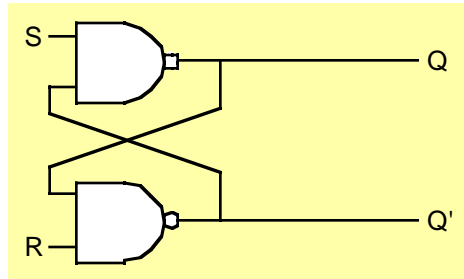


Figure 3.25 Set/Reset Flip Flop Constructed from NAND Gates

The S and R inputs are normally high. If you *temporarily* set the S input to zero and then bring it back to one (*toggle* the S input), this forces the Q output to one. Likewise, if you toggle the R input from one to zero back to one, this sets the Q output to zero. The Q' input is generally the inverse of the Q output.

Note that if both S and R are one, then the Q output depends upon Q. That is, whatever Q happens to be, the top NAND gate continues to output that value. If Q was originally one, then there are two ones as inputs to the bottom flip-flop (Q and R). This produces an output of zero (Q'). Therefore, the two inputs to the top NAND gate are zero and one. This produces the value one as an output (matching the original value for Q).

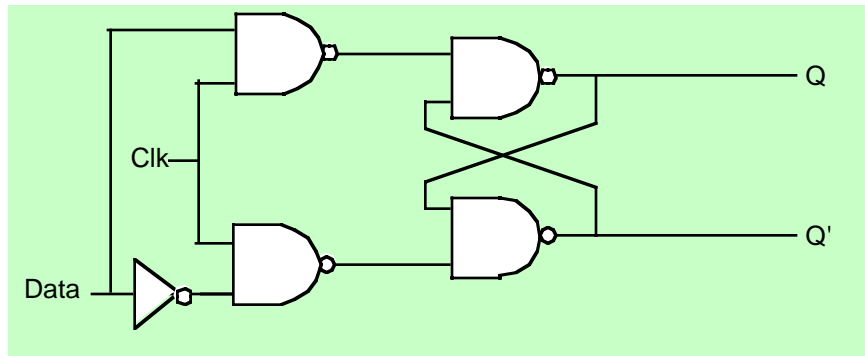
If the original value for Q was zero, then the inputs to the bottom NAND gate are Q=0 and R=1. Therefore, the output of this NAND gate is one. The inputs to the top NAND gate, therefore, are S=1 and Q'=1. This produces a zero output, the original value of Q.

Suppose Q is zero, S is zero and R is one. This sets the two inputs to the top flip-flop to one and zero, forcing the output (Q) to one. Returning S to the high state does not change the output at all. You can obtain this same result if Q is one, S is zero, and R is one. Again, this produces an output value of one. This value remains one even when S switches from zero to one. Therefore, toggling the S input from one to zero and then back to one produces a one on the output (i.e., *sets* the flip-flop). The same idea applies to the R input, except it forces the Q output to zero rather than to one.

There is one catch to this circuit. It does not operate properly if you set both the S and R inputs to zero simultaneously. This forces both the Q and Q' outputs to one (which is logically inconsistent). Whichever input remains zero the longest determines the final state of the flip-flop. A flip-flop operating in this mode is said to be *unstable*.

The only problem with the S/R flip-flop is that you must use separate inputs to remember a zero or a one value. A memory cell would be more valuable to us if we could specify the data value to remember on one input and provide a *clock input* to *latch* the input value. This type of flip-flop, the D flip-flop (for *data*) uses the circuit in Figure 3.26.

8. In practice, there is a short *propagation delay* between a change in the inputs and the corresponding outputs in any electronic implementation of a boolean function.



Assuming you fix the Q and Q' outputs to either 0/1 or 1/0, sending a *clock pulse* that goes from zero to one back to zero will copy the D input to the Q output. It will also copy D' to Q'. The exercises at the end of this topic section will expect you to describe this operation in detail, so study this diagram carefully.

Although remembering a single bit is often important, in most computer systems you will want to remember a group of bits. You can remember a sequence of bits by combining several D flip-flops in parallel. Concatenating flip-flops to store an n-bit value forms a *register*. The electronic schematic in Figure 3.27 shows how to build an eight-bit register from a set of D flip-flops.

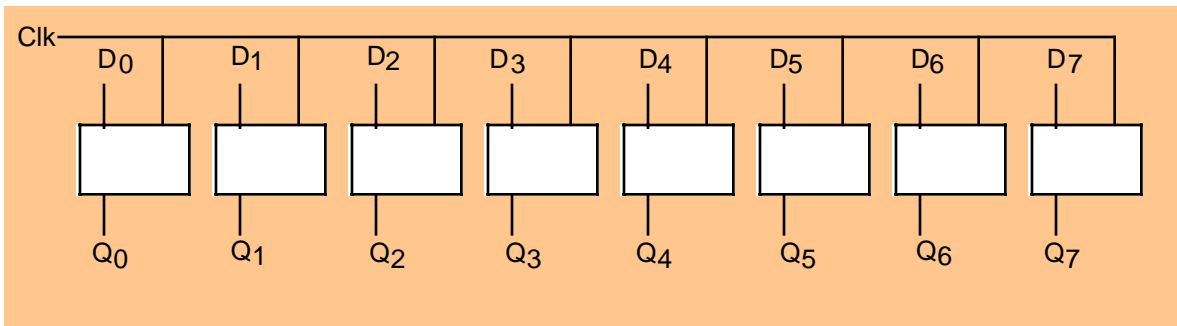


Figure 3.27 An Eight-bit Register Implemented with Eight D Flip-flops

Note that the eight D flip-flops use a common clock line. This diagram does not show the Q' outputs on the flip-flops since they are rarely required in a register.

D flip-flops are useful for building many sequential circuits above and beyond simple registers. For example, you can build a *shift register* that shifts the bits one position to the left on each clock pulse. A four-bit shift register appears in Figure 3.28.

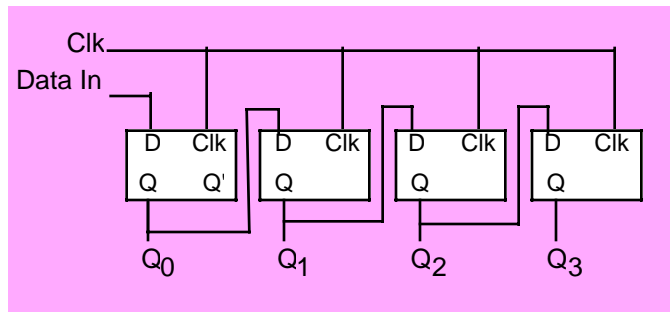


Figure 3.28 A Four-bit Shift Register Built from D Flip-flops

You can even build a *counter*, that counts the number of times the clock toggles from one to zero and back to one using flip-flops. The circuit in Figure 3.29 implements a four bit counter using D flip-flops.

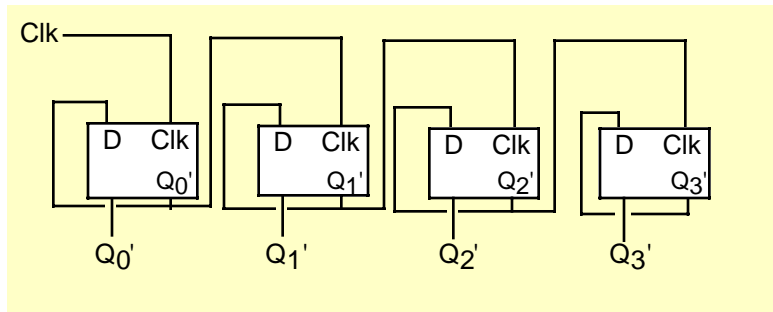


Figure 3.29 Four-bit Counter Built from D Flip-flops

Surprisingly, you can build an entire CPU with combinatorial circuits and only a few additional sequential circuits beyond these. For example, you can build a simple state machine known as a sequencer by combining a counter and a decoder as shown in Figure 3.30. For each cycle of the clock this sequencer activates one of its output lines. Those lines, in turn, may control other circuitry. By “firing” these circuits on each of the 16 output lines of the decoder, we can control the order in which these 16 different circuits accomplish their tasks. This is a fundamental need in a CPU since we often need to control the sequence of various operations (for example, it wouldn’t be a good thing if the “ADD(EAX, EBX);” instruction stored the result into EBX before fetching the source operand from EAX (or EBX). A simple sequencer such as this one can tell the CPU when to fetch the first operand, when to fetch the second operand, when to add them together, and when to store the result away. But we’re getting a little ahead of ourselves, we’ll discuss this in greater detail in a later chapter.

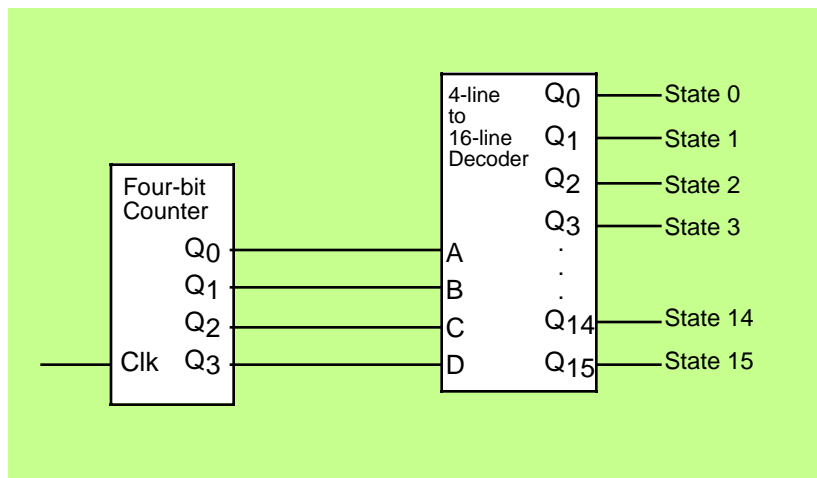


Figure 3.30 A Simple 16-State Sequencer

3.7 Okay, What Does It Have To Do With Programming, Then?

Once you have registers, counters, and shift registers, you can build *state machines*. The implementation of an algorithm in hardware using state machines is well beyond the scope of this text. However, one important point must be made with respect to such circuitry – *any algorithm you can implement in software you can also implement directly in hardware*. This suggests that boolean logic is the basis for computation on all modern computer systems. Any program you can write, you can specify as a sequence of boolean equations.

Of course, it is much easier to specify a solution to a programming problem using languages like Pascal, C, or even assembly language than it is to specify the solution using boolean equations. Therefore, it is unlikely that you would ever implement an entire program using a set of state machines and other logic circuitry. Nevertheless, there are times when a hardware implementation is better. A hardware solution can be one, two, three, or more *orders of magnitude* faster than an equivalent software solution. Therefore, some time critical operations may require a hardware solution.

A more interesting fact is that the converse of the above statement is also true. Not only can you implement all software functions in hardware, but it is also possible to *implement all hardware functions in software*. This is an important revelation because many operations you would normally implement in hardware are *much cheaper* to implement using software on a microprocessor. Indeed, this is a primary use of *assembly language* in modern systems – to inexpensively replace a complex electronic circuit. It is often possible to replace many tens or hundreds of dollars of electronic components with a single \$5 microcomputer chip. The whole field of *embedded systems* deals with this very problem. Embedded systems are computer systems embedded in other products. For example, most microwave ovens, TV sets, video games, CD players, and other consumer devices contain one or more complete computer systems whose sole purpose is to replace a complex hardware design. Engineers use computers for this purpose because they are *less expensive* and *easier to design with* than traditional electronic circuitry.

You can easily design software that reads switches (input variables) and turns on motors, LEDs or lights, locks or unlocks a door, etc. (output functions). To write such software, you will need an understanding of boolean functions and how to implement such functions in software.

Of course, there is one other reason for studying boolean functions, even if you never intend to write software intended for an embedded system or write software that manipulates real-world devices. Many high level languages process boolean expressions (e.g., those expressions that control an IF statement or WHILE loop). By applying transformations like DeMorgan's theorems or a mapping optimization it is often possible to improve the performance of high level language code. Therefore, studying boolean functions *is* important even if you never intend to design an electronic circuit. It can help you write better code in a traditional programming language.

For example, suppose you have the following statement in Pascal:

```
if ((x=y) and (a <> b)) or ((x=y) and (c <= d)) then SomeStmt;
```

You can use the distributive law to simplify this to:

```
if ((x=y) and ((a <> b) or (c <= d))) then SomeStmt;
```

Likewise, we can use DeMorgan's theorem to reduce

```
while (not((a=b) and (c=d))) do Something;
```

to

```
while (a <> b) or (c <> d) do Something;
```

So as you can see, understanding a little boolean algebra can actually help you write better software.

3.8 Putting It All Together

A good understanding of boolean algebra and digital design is absolutely necessary for anyone who wants to understand the internal operation of a CPU. As an added bonus, programmers who understand digital design can write better assembly language (and high level language) programs. This chapter provides a basic introduction to boolean algebra and digital circuit design. Although a detailed knowledge of this material isn't necessary if you simply want to write assembly language programs, this knowledge will help explain why Intel chose to implement instructions in certain ways; questions that will undoubtedly arise as we begin to look at the low-level implementation of the CPU.

This chapter is not, by any means, a complete treatment of this subject. If you're interested in learning more about boolean algebra and digital circuit design, there are dozens and dozens of texts on this subject available. Since this is a text on assembly language programming, we cannot afford to spend additional time on this subject; please see one of these other texts for more information.

4.1 Chapter Overview

This chapter discusses history of the 80x86 CPU family and the major improvements occurring along the line. The historical background will help you better understand the design compromises they made as well as understand the legacy issues surrounding the CPU's design. This chapter also discusses the major advances in computer architecture that Intel employed while improving the x86¹.

4.2 The History of the 80x86 CPU Family

Intel developed and delivered the first commercially viable microprocessor way back in the early 1970s: the 4004 and 4040 devices. These four-bit microprocessors, intended for use in calculators, had very little power. Nevertheless, they demonstrated the future potential of the microprocessor — an entire CPU on a single piece of silicon². Intel rapidly followed their four-bit offerings with their 8008 and 8080 eight-bit CPUs. A small outfit in Santa Fe, New Mexico, incorporated the 8080 CPU into a box they called the Altair 8800. Although this was not the world's first "personal computer" (there were some limited distribution machines built around the 8008 prior to this), the Altair was the device that sparked the imaginations of hobbyists the world over and the personal computer revolution was born.

Intel soon had competition from Motorola, MOS Technology, and an upstart company formed by disgruntled Intel employees, Zilog. To compete, Intel produced the 8085 microprocessor. To the software engineer, the 8085 was essentially the same as the 8080. However, the 8085 had lots of hardware improvements that made it easier to design into a circuit. Unfortunately, from a software perspective the other manufacturer's offerings were better. Motorola's 6800 series was easier to program, MOS Technology's 65xx family was easier to program and very inexpensive, and Zilog's Z80 chip was upwards compatible with the 8080 with lots of additional instructions and other features. By 1978 most personal computers were using the 6502 or Z80 chips, not the Intel offerings.

Sometime between 1976 and 1978 Intel decided that they needed to leap-frog the competition and produce a 16-bit microprocessor that offered substantially more power than their competitor's eight-bit offerings. This initiative led to the design of the 8086 microprocessor. The 8086 microprocessor was not the world's first 16-bit microprocessor (there were some oddball 16-bit microprocessors prior to this point) but it was certainly the highest performance single-chip 16-bit microprocessor when it was first introduced.

During the design timeframe of the 8086 memory was very expensive. Sixteen Kilobytes of RAM was selling above \$200 at the time. One problem with a 16-bit CPU is that programs tend to consume more memory than their counterparts on an eight-bit CPU. Intel, ever cognizant of the fact that designers would reject their CPU if the total system cost was too high, made a special effort to design an instruction set that had a high memory density (that is, packed as many instructions into as little RAM as possible). Intel achieved their design goal and programs written for the 8086 were comparable in size to code running on eight-bit microprocessors. However, those design decisions still haunt us today as you'll soon see.

-
1. Note that Intel wasn't the inventor of most of these new technological advances. They simply duplicated research long since commercially employed by mainframe designers.
 2. Prior to this point, commercial computer systems used multiple semiconductor devices to implement the CPU.

At the time Intel designed the 8086 CPU the average lifetime of a CPU was only a couple of years. Their experiences with the 4004, 4040, 8008, 8080, and 8085 taught them that designers would quickly ditch the old technology in favor of the new technology as long as the new stuff was radically better. So Intel designed the 8086 assuming that whatever compromises they made in order to achieve a high instruction density would be fixed in newer chips. Based on their experience, this was a reasonable assumption.

Intel's competitors were not standing still. Zilog created their own 16-bit processor that they called the Z8000, Motorola created the 68000, their own 16-bit processor, and National Semiconductor introduced the 16032 device (later to be renamed the 32016). The designers of these chips had different design goals than Intel. Primarily, they were more interested in providing a reasonable instruction set for programmers even if their code density wasn't anywhere near as high as the 8086. The Motorola and National offers even provided 32-bit integer registers, making programming the chips even easier. All in all, these chips were much better (from a software development standpoint) than the Intel chip.

Intel wasn't resting on its laurels with the 8086. Immediately after the release of the 8086 they created an eight-bit version, the 8088. The purpose of this chip was to reduce system cost (since a minimal system could get by with half the memory chips and cheaper peripherals since the 8088 had an eight-bit data bus). In the very early 1980s, Intel also began work on their intended successor to the 8086 — the iAPX432 CPU. Intel fully expected the 8086 and 8088 to die away and that system designers who were creating general purpose computer systems would choose the 432 chip instead.

Then a major event occurred that would forever change history: in 1980 a small group at IBM got the go-ahead to create a "personal computer" along the likes of the Apple II and TRS-80 computers (the most popular PCs at the time). IBM's engineers probably evaluated lots of different CPUs and system designs. Ultimately, they settled on the 8088 chip. Most likely they chose this chip because they could create a minimal system with only 16 Kilobytes of RAM and a set of cheap eight-bit peripheral devices. So Intel's design goals of creating CPUs that worked well in low-cost systems landed them a very big "design win" from IBM.

Intel was still hard at work on the (ill-fated) iAPX432 project, but a funny thing happened — IBM PCs started selling far better than anyone had ever dreamed. As the popularity of the IBM PCs increased (and as people began "cloning" the PC), lots of software developers began writing software for the 8088 (and 8086) CPU, mostly in assembly language. In the meantime, Intel was pushing their iAPX432 with the Ada programming language (which was supposed to be the next big thing after Pascal, a popular language at the time). Unfortunately for Intel, no one was interested in the 432. Their PC software, written mostly in assembly language wouldn't run on the 432 and the 432 was notoriously slow. It took a while, but the iAPX432 project eventually died off completely and remains a black spot on Intel's record to this day.

Intel wasn't sitting pretty on the 8086 and 8088 CPUs, however. In the late 1970s and early 1980s they developed the 80186 and 80188 CPUs. These CPUs, unlike their previous CPU offerings, were fully upwards compatible with the 8086 and 8088 CPUs. In the past, whenever Intel produced a new CPU it did not necessarily run the programs written for the previous processors. For example, the 8086 did not run 8080 software and the 8080 did not run 4040 software. Intel, recognizing that there was a tremendous investment in 8086 software, decided to create an upgrade to the 8086 that was superior (both in terms of hardware capability and with respect to the software it would execute). Although the 80186 did not find its way into many PCs, it was a very popular chip in embedded applications (i.e., non-computer devices that use a CPU to control their functions). Indeed, variants of the 80186 are in common use even today.

The unexpected popularity of the IBM PC created a problem for Intel. This popularity obliterated the assumption that designers would be willing to switch to a better chip when such a chip arrived, even if it meant rewriting their software. Unfortunately, IBM and tens of thousands of software developers weren't willing to do this to make life easy for Intel. They wanted to stick with the 8086 software they'd written but they also wanted something a little better than the 8086. If they were going to be forced into jumping ship to a new CPU, the Motorola, Zilog, and National offerings were starting to look pretty good. So Intel did something that saved their

bacon and has infuriated computer architects ever since: they started creating upwards compatible CPUs that continued to execute programs written for previous members of their growing CPU family while adding new features.

As noted earlier, memory was very expensive when Intel first designed the 8086 CPU. At that time, computer systems with a megabyte of memory usually cost megabucks. Intel was expecting a typical computer system employing the 8086 to have somewhere between 4 Kilobytes and 64 Kilobytes of memory. So when they designed in a one megabyte limitation, they figured no one would ever install that much memory in a system. Of course, by 1983 people were still using 8086 and 8088 CPUs in their systems and memory prices had dropped to the point where it was very common to install 640 Kilobytes of memory on a PC (the IBM PC design effectively limited the amount of RAM to 640 Kilobytes even though the 8086 was capable of addressing one megabyte). By this time software developers were starting to write more sophisticated programs and users were starting to use these programs in more sophisticated ways. The bottom line was that everyone was bumping up against the one megabyte limit of the 8086. Despite the investment in existing software, Intel was about to lose their cash cow if they didn't do something about the memory addressing limitations of their 8086 family (the 68000 and 32016 CPUs could address up to 16 Megabytes at the time and many system designers [e.g., Apple] were defecting to these other chips). So Intel introduced the 80286 which was a big improvement over the previous CPUs. The 80286 added lots of new instructions to make programming a whole lot easier and they added a new "protected" mode of operation that allowed access to as much as 16 megabytes of memory. They also improved the internal operation of the CPU and bumped up the clock frequency so that the 80286 ran about 10 times faster than the 8088 in IBM PC systems.

IBM introduced the 80286 in their IBM PC/AT (AT = "advanced technology"). This change proved enormously popular. PC/AT clones based on the 80286 started appearing everywhere and Intel's financial future was assured.

Realizing that the 80x86 (x = "", "1", or "2") family was a big money maker, Intel immediately began the process of designing new chips that continued to execute the old code while improving performance and adding new features. Intel was still playing catch-up with their competitors in the CPU arena with respect to features, but they were definitely the king of the hill with respect to CPUs installed in PCs. One significant difference between Intel's chips and many of their competitors was that their competitors (notably Motorola and National) had a 32-bit internal architecture while the 80x86 family was stuck at 16-bits. Again, concerned that people would eventually switch to the 32-bit devices their competitors offered, Intel upgraded the 80x86 family to 32 bits by adding the 80386 to the product line.

The 80386 was truly a remarkable chip. It maintained almost complete compatibility with the previous 16-bit CPUs while fixing most of the real complaints people had with those older chips. In addition to supporting 32-bit computing, the 80386 also bumped up the maximum addressability to four gigabytes as well as solving some problems with the "segmented" organization of the previous chips (a big complaint by software developers at the time). The 80386 also represented the most radical change to ever occur in the 80x86 family. Intel more than doubled the total number of instructions, added new memory management facilities, added hardware debugging support for software, and introduced many other features. Continuing the trend they set with the 80286, the 80386 executed instructions faster than previous generation chips, even when running at the same clock speed plus the new chip ran at a higher clock speed than the previous generation chips. Therefore, it ran existing 8088 and 80286 programs faster than on these older chips. Unfortunately, while people adopted the new chip for its higher performance, they didn't write new software to take advantage of the chip's new features. But more on that in a moment.

Although the 80386 represented the most radical change in the 80x86 architecture from the programmer's view, Intel wasn't done wringing all the performance out of the x86 family. By the time the 80386 appeared, computer architects were making a big noise about the so-called RISC (Reduced Instruction Set Computer) CPUs. While there were several advantages to these new RISC chips, a important advantage of these chips is

that they purported to execute one instruction every clock cycle. The 80386 instructions required a wildly varying number of cycles to execute ranging from a few cycles per instruction to well over a hundred. Although comparing RISC processors directly with the 80386 was dangerous (because many 80386 instructions actually did the work of two or more RISC instructions), there was a general perception that, at the same clock speed, the 80386 was slower since it executed fewer instructions in a given amount of time.

The 80486 CPU introduced two major advances in the x86 design. First, the 80486 integrated the floating point unit (or FPU) directly onto the CPU die. Prior to this point Intel supplied a separate, external, chip to provide floating point calculations (these were the 8087, 80287, and 80387 devices). By incorporating the FPU with the CPU, Intel was able to speed up floating point operations and provide this capability at a lower cost (at least on systems that required floating point arithmetic). The second major architectural advance was the use of *pipelined instruction execution*. This feature (which we will discuss in detail a little later in this chapter) allowed Intel to overlap the execution of two or more instructions. The end result of pipelining is that they effectively reduced the number of cycles each instruction required for execution. With pipelining, many of the simpler instructions had an aggregate throughput of one instruction per clock cycle (under ideal conditions) so the 80486 was able to compete with RISC chips in terms of clocks per instruction cycle.

While Intel was busy adding pipelining to their x86 family, the companies building RISC CPUs weren't standing still. To create ever faster and faster CPU offerings, RISC designers began creating *superscalar* CPUs that could actually execute more than one instruction per clock cycle. Once again, Intel's CPUs were perceived as following the leaders in terms of CPU performance. Another problem with Intel's CPU is that the integrated FPU, though faster than the earlier models, was significantly slower than the FPUs on the RISC chips. As a result, those designing high-end engineering workstations (that typically require good floating point hardware support) began using the RISC chips because they were faster than Intel's offerings.

From the programmer's perspective, there was very little difference between an 80386 with an 80387 FPU and an 80486 CPU. There were only a handful of new instructions (most of which had very little utility in standard applications) and not much in the way of other architectural features that software could use. The 80486, from the software engineer's point of view, was just a really fast 80386/80387 combination.

So Intel went back to their CAD³ tools and began work on their next CPU. This new CPU featured a superscalar design with vastly improved floating point performance. Finally, Intel was closing in on the performance of the RISC chips. Like the 80486 before it, this new CPU added only a small number of new instructions and most of those were intended for use by operating systems, not application software.

Intel did not designate this new chip the 80586. Instead, they called it the *Pentium* "Processor"⁴. The reason they discontinued referring to processors by number and started naming them was because of confusion in the marketplace. Intel was not the only company producing x86 compatible CPUs. AMD, Cyrix, and a host of others were also building and selling these chips in direct competition with Intel. Until the 80486 came along, the internal design of the CPUs were relatively simple and even small companies could faithfully reproduce the functionality of Intel's CPUs. The 80486 was a different story altogether. This chip was quite complex and taxed the design capabilities of the smaller companies. Some companies, like AMD, actually licensed Intel's design and they were able to produce chips that were compatible with Intel's (since they were, effectively, Intel's chips). Other companies attempted to create their own version of the 80486 and fell short of the goal. Perhaps they didn't integrate an FPU or the new instructions on the 80486. Many didn't support pipelining. Some chips lacked other features found on the 80486. In fact, most of the (non-Intel) chips were really 80386 devices with some very slight improvements. Nevertheless, they called these chips 80486 CPUs.

3. Computer aided design.

4. Pentium Processor is a registered trademark of Intel Corporation. For legal reasons Intel could not trademark the name Pentium by itself, hence the full name of the CPU is the "Pentium Processor".

This created massive confusion in the marketplace. Prior to this, if you'd purchased a computer with an 80386 chip you knew the capabilities of the CPU. All 80386 chips were equivalent. However, when the 80486 came along and you purchased a computer system with an 80486, you didn't know if you were getting an actual 80486 or a remarked 80386 CPU. To counter this, Intel began their enormously successful "Intel Inside" campaign to let people know that there was a difference between Intel CPUs and CPUs from other vendors. This marketing campaign was so successful that people began specifying Intel CPUs even though some other vendor's chips (i.e., AMD) were completely compatible.

Not wanting to repeat this problem with the 80586 generation, Intel ditched the numeric designation of their chips. They created the term "Pentium Processor" to describe their new CPU so they could trademark the name and prevent other manufacturers from using the same designation for their chip. Initially, of course, savvy computer users griped about Intel's strong-arm tactics but the average user benefited quite a bit from Intel's marketing strategy. Other manufacturers release their own 80586 chips (some even used the "586" designation), but they couldn't use the Pentium Processor name on their parts so when someone purchased a system with a Pentium in it, they knew it was going to have all the capabilities of Intel's chip since it had to be Intel's chip. This was a good thing because most of the other 586 class chips that people produced at that time were not as powerful as the Pentium.

The Pentium cemented Intel's position as champ of the personal computer. It had near RISC performance and ran tons of existing software. Only the Apple Macintosh and high-end UNIX workstations and servers went the RISC route. Together, these other machines comprised less than 10% of the total desktop computer market.

Intel still was not satisfied. They wanted to control the server market as well. So they developed the Pentium Pro CPU. The Pentium Pro had a couple of features that made it ideal for servers. Intel improved the 32-bit performance of the CPU (at the expense of its 16-bit performance), they added better support for multiprocessing to allow multiple CPUs in a system (high-end servers usually have two or more processors), and they added a handful of new instructions to improve the performance of certain instruction sequences on the pipelined architecture. Unfortunately, most application software written at the time of the Pentium Pro's release was 16-bit software which actually ran slower on the Pentium Pro than it did on a Pentium at equivalent clock frequencies. So although the Pentium Pro did wind up in a few server machines, it was never as popular as the other chips in the Intel line.

The Pentium Pro had another big strike against it: shortly after the introduction of the Pentium Pro, Intel's engineers introduced an upgrade to the standard Pentium chip, the MMX (multimedia extension) instruction set. These new instructions (nearly 60 in all) gave the Pentium additional power to handle computer video and audio applications. These extensions became popular overnight, putting the last nail in the Pentium Pro's coffin. The Pentium Pro was slower than the standard Pentium chip and slower than high-end RISC chips, so it didn't see much use.

Intel corrected the 16-bit performance in the Pentium Pro, added the MMX extensions and called the result the Pentium II⁵. The Pentium II demonstrated an interesting point. Computers had reached a point where they were powerful enough for most people's everyday activities. Prior to the introduction of the Pentium II, Intel (and most industry pundits) had assumed that people would always want more power out of their computer systems. Even if they didn't need the machines to run faster, surely the software developers would write larger (and slower) systems requiring more and more CPU power. The Pentium II proved this idea wrong. The average user needed email, word processing, Internet access, multimedia support, simple graphics editing capabilities, and a spreadsheet now and then. Most of these applications, at least as home users employed them, were fast enough on existing CPUs. The applications that were slow (e.g., Internet access) were generally beyond the control of the CPU (i.e., the modem was the bottleneck not the CPU). As a result, when Intel introduced their pricey Pen-

5. Interestingly enough, by the time the Pentium II appeared, the 16-bit efficiency was no longer a factor since most software was written as 32-bit code.

tium II CPUs, they discovered that system manufacturers started buying other people's x86 chips because they were far less expensive and quite suitable for their customer's applications. This nearly stunned Intel since it contradicted their experience up to that point.

Realizing that the competition was capturing the low-end market and stealing sales away, Intel devised a low-cost (lower performance) version of the Pentium II that they named *Celeron*⁶. The initial Celerons consisted of a Pentium II CPU without the on-board level two cache. Without the cache, the chip ran only a little bit better than half the speed of the Pentium II part. Nevertheless, the performance was comparable to other low-cost parts so Intel's fortunes improved once more.

While designing the low-end Celeron, Intel had not lost sight of the fact that they wanted to capture a chunk of the high-end workstation and server market as well. So they created a third version of the Pentium II, the Xeon Processor with improved cache and the capability of multiprocessor more than two CPUs. The Pentium II supports a two CPU multiprocessor system but it isn't easy to expand it beyond this number; the Xeon processor corrected this limitation. With the introduction of the Xeon processor (plus special versions of Unix and Windows NT), Intel finally started to make some serious inroads into the server and high-end workstation markets.

You can probably imagine what followed the Pentium II. Yep, the Pentium III. The Pentium III introduced the SIMD (pronounced SIM-DEE) extensions to the instruction set. These new instructions provided high performance floating point operations for certain types of computations that allow the Pentium III to compete with high-end RISC CPUs. The Pentium III also introduced another handful of integer instructions to aid certain applications.

With the introduction of the Pentium III, nearly all serious claims about RISC chips offering better performance were fading away. In fact, for most applications, the Intel chips were actually faster than the RISC chips available at the time. Next, of course, Intel introduced the Pentium IV chip (it was running at 2 GHz as this was being written, a much higher clock frequency than its RISC contemporaries). An interesting issue concerning the Pentium IV is that it does not execute code faster than the Pentium III when running at the same clock frequency (it runs slower, in fact). The Pentium IV makes up for this problem by executing at a much higher clock frequency than is possible with the Pentium III. One would think that Intel would soon own it all. Surely by the time of the Pentium V, the RISC competition wouldn't be a factor anymore.

There is one problem with this theory: even Intel is admitting that they've pushed the x86 architecture about as far as they can. For nearly 20 years, computer architects have blasted Intel's architecture as being gross and bloated having to support code written for the 8086 processor way back in 1978. Indeed, Intel's design decisions (like high instruction density) that seemed so important in 1978 are holding back the CPU today. So-called "clean" designs, that don't have to support legacy applications, allow CPU designers to create high-performance CPUs with far less effort than Intel's. Worse, those decisions Intel made in the 1976-1978 time frame are beginning to catch up with them and will eventually stall further development of the CPU. Computer architects have been warning everyone about this problem for twenty years; it is a testament to Intel's design effort (and willingness to put money into R&D) that they've taken the CPU as far as they have.

The biggest problem on the horizon is that most RISC manufacturers are now extending their architectures to 64-bits. This has two important impacts on computer systems. First, arithmetic calculations will be somewhat faster as will many internal operations and second, the CPUs will be able to directly address more than four gigabytes of main memory. This last factor is probably the most important for server and workstation systems. Already, high-end servers have more than four gigabytes installed. In the future, the ability to address more than four gigabytes of physical RAM will become essential for servers and high-end workstations. As the price of a gigabyte or more of memory drops below \$100, you'll see low-end personal computers with more than four gigabytes installed. To effectively handle this kind of memory, Intel will need a 64-bit processor to compete with the RISC chips.

6. The term "Celeron Processor" is also an Intel trademark.

Perhaps Intel has seen the light and decided it's time to give up on the x86 architecture. Towards the middle to end of the 1990s Intel announced that they were going to create a partnership with Hewlett-Packard to create a new 64-bit processor based around HP's PA-RISC architecture. This new 64-bit chip would execute x86 code in a special "emulation" mode and run native 64-bit code using a new instruction set. It's too early to tell if Intel will be successful with this strategy, but there are some major risks (pardon the pun) with this approach. The first such CPUs (just becoming available as this is being written) run 32-bit code far slower than the Pentium III and IV chips. Not only does the emulation of the x86 instruction set slow things down, but the clock speeds of the early CPUs are half the speed of the Pentium IVs. This is roughly the same situation Intel had with the Pentium Pro running 16-bit code slower than the Pentium. Second, the 64-bit CPUs (the IA64 family) rely heavily on compiler technology and are using a commercially untested architecture. This is similar to the situation with the iAPX432 project that failed quite miserably. Hopefully Intel knows what they're doing and ten years from now we'll all be using IA64 processors and wondering why anyone ever stuck with the IA32. On the other hand, hopefully Intel has a back-up plan in case the IA64 initiative fails.

Intel is betting that people will move to the IA64 when they need 64-bit computing capabilities. AMD, on the other hand, is betting that people would rather have a 64-bit x86 processor. Although the details are sketchy, AMD has announced that they will extend the x86 architecture to 64 bits in much the same way that Intel extended the 8086 and 80286 to 32-bits with the introduction of the 80386 microprocessor. Only time will tell if Intel or AMD (or both) are successful with their visions.

Processor	Date of Introduction	Transistors on Chip	Maximum MIPS at Introduction ^a	Maximum Clock Frequency at Introduction ^b	On-chip Cache Memory	Maximum Addressable Memory
8086	1978	29K	0.8	8 MHz		1 MB
80286	1982	134K	2.7	12.5 MHz		16 MB
80386	1985	275K	6	20 MHz		4 GB
80486	1989	1.2M	20	25 MHz ^c	8K Level 1	4 GB
Pentium	1993	3.1M	100	60MHz	16K Level 1	4 GB
Pentium Pro	1995	5.5M	440	200 MHz	16K Level 1, 256K/512K Level 2	64 GB
Pentium II	1997	7M	466	266 MHz	32K Level 1, 256/512K Level 2	64 GB
Pentium III	1999	8.2M	1,000	500 MHz	32K Level 1, 512K Level 2	64 GB

- a. By the introduction of the next generation this value was usually higher.
 - b. Maximum clock frequency at introduction was very limited sampling. Usually, the chips were available at the next lower clock frequency in Intel's scale. Also note that by the introduction of the next generation this value was usually much higher.
 - c. Shortly after the introduction of the 25MHz 80486, Intel began using "Clock doubling" techniques to run the CPU twice as fast internally as the external clock. Hence, a 50 MHz 80486 DX2 chip was really running at 25 MHz externally and 50 MHz internally. Most chips after the 80486 employ a different internal clock frequency compared to the external (or "bus") frequency.
-

4.3 A History of Software Development for the x86

A section on the history of software development may seem unusual in a chapter on CPU Architecture. However, the 80x86 architecture is inexorably tied to the development of the software for this platform. Many architectural design decisions were a direct result of ensuring compatibility with existing software. So to fully understand the architecture, you must know a little bit about the history of the software that runs on the chip.

From the date of the very first working sample of the 8086 microprocessor to the latest and greatest IA-64 CPU, Intel has had an important goal: as much as possible, ensure compatibility with software written for previous generations of the processor. This mantra existed even on the first 8086, before there was a previous generation of the family. For the very first member of the family, Intel chose to include a modicum of compatibility with their previous eight-bit microprocessor, the 8085. The 8086 was not capable of running 8085 software, but Intel designed the 8086 instruction set to provide almost a one for one mapping of 8085 instructions to 8086 instructions. This allowed 8085 software developers to easily translate their existing assembly language programs to the 8086 with very little effort (in fact, software translators were available that did about 85% of the work for these developers).

Intel did not provide *object code compatibility*⁷ with the 8085 instruction set because the design of the 8085 instruction set did not allow the expansion Intel needed for the 8086. Since there was very little software running on the 8085 that needed to run on the 8086, Intel felt that making the software developers responsible for this translation was a reasonable thing to do.

When Intel introduced the 8086 in 1978, the majority of the world's 8085 (and Z80) software was written in Microsoft's BASIC running under Digital Research's CP/M operating system. Therefore, to "port" the majority of business software (such that it existed at the time) to the 8086 really only required two things: porting the CP/M operating system (which was less than eight kilobytes long) and Microsoft's BASIC (most versions were around 16 kilobytes at the time). Porting such small programs may have seemed like a bit of work to developers of that era, but such porting is trivial compared with the situation that exists today. Anyway, as Intel expected, both Microsoft and Digital Research ported their products to the 8086 in short order so it was possible for a large percentage of the 8085 software to run on 8086 within about a year of the 8086's introduction.

Unfortunately, there was no great rush by computer hobbyists (the computer users of that era) to switch to the 8086. About this time the Radio Shack TRS-80 and the Apple II microcomputer systems were battling for supremacy of the home computer market and no one was really making computer systems utilizing the 8086 that appealed to the mass market. Intel wasn't doing poorly with the 8086; its market share, when you compared it with the other microprocessors, was probably better than most. However, the situation certainly wasn't like it is today (circa 2001) where the 80x86 CPU family owns 85% of the general purpose computer market.

7. That is, the ability to run 8085 machine code directly.

The 8086 CPU, and its smaller sibling, the eight-bit 8088, was happily raking in its portion of the microprocessor market and Intel naturally assumed that it was time to start working on a 32-bit processor to replace the 8086 in much the same way that the 8086 replaced the eight-bit 8085. As noted earlier, this new processor was the ill-fated iAPX 432 system. The iAPX 432 was such a dismal failure that Intel might not have survived had it not been for a big stroke of luck — IBM decided to use the 8088 microprocessor in their personal computer system.

To most computer historians, there were two watershed events in the history of the personal computer. The first was the introduction of the Visicalc spreadsheet program on the Apple II personal computer system. This single program demonstrated that there was a real reason for owning a computer beyond the nerdy "gee, I've got my own computer" excuse. Visicalc quickly (and, alas, briefly) made Apple Computer the largest PC company around. The second big event in the history of personal computers was, of course, the introduction of the IBM PC. The fact that IBM, a "real" computer company, would begin building PCs legitimized the market. Up to that point, businesses tended to ignore PCs and treated them as toys that nerdy engineers liked to play with. The introduction of the IBM PC caused a lot of businesses to take notice of these new devices. Not only did they take notice, but they liked what they saw. Although IBM cannot make the claim that they started the PC revolution, they certainly can take credit for giving it a big jumpstart early on in its life.

Once people began buying lots of PCs, it was only natural that people would start writing and selling software for these machines. The introduction of the IBM PC greatly expanded the marketplace for computer systems. Keep in mind that at the time of the IBM PC's introduction, most computer systems had only sold tens of thousands of units. The more popular models, like the TRS-80 and Apple II had only sold hundreds of thousands of units. Indeed, it wasn't until a couple of years after the introduction of the IBM PC that the first computer system sold one million units; and that was a Commodore 64 system, not the IBM PC.

For a brief period, the introduction of the IBM PC was a godsend to most of the other computer manufacturers. The original IBM PC was underpowered and quite a bit more expensive than its counterparts. For example, a dual-floppy disk drive PC with 64 Kilobytes of memory and a monochrome display sold for \$3,000. A comparable Apple II system with a color display sold for under \$2,000. The original IBM PC with its 4.77 MHz 8088 processor (that's four-point-seven-seven, not four hundred seventy-seven!) was only about two to three times as fast as the Apple II with its paltry 1 MHz eight-bit 6502 processor. The fact that most Apple II software was written by expert assembly language programmers while most (early) IBM software was written in a high level language (often interpreted) or by inexperienced 8086 assembly language programmers narrowed the gap even more.

Nonetheless, software development on PCs accelerated. The wide range of different (and incompatible) systems made software development somewhat risky. Those who did not have an emotional attachment to one particular company (and didn't have the resources to develop for more than one platform) generally decided to go with IBM's PC when developing their software.

One problem with the 8086's architecture was beginning to show through by 1983 (remember, this is five years after Intel introduced the 8086). The *segmented memory architecture* that allowed them to extend their 16-bit addressing scheme to 20 bits (allowing the 8086 to address a megabyte of memory) was being attacked on two fronts. First, this segmented addressing scheme was difficult to use in a program, especially if that program needed to access more than 64 kilobytes of data or, worse yet, needed to access a single data structure that was larger than 64K long. By 1983 software had reached the level of sophistication that most programs were using this much memory and many needed large data structures. The software community as a whole began to grumble and complain about this segmented memory architecture and what a stupid thing it was.

The second problem with Intel's segmented architecture is that it only supported a maximum of a one megabyte address space. Worse, the design of the IBM PC effectively limited the amount of RAM the system could have to 640 kilobytes. This limitation was also beginning to create problems for more sophisticated programs

running on the PC. Once again, the software development community grumbled and complained about Intel's segmented architecture and the limitations it imposed upon their software.

About the time people began complaining about Intel's architecture, Intel began running an ad campaign bragging about how great their chip was. They quoted top executives at companies like Visicorp (the outfit selling Visicalc) who claimed that the segmented architecture was great. They also made a big deal about the fact that over a billion dollars worth of software had been written for their chip. This was all marketing hype, of course. Their chip was not particularly special. Indeed, the 8086's contemporaries (Z8000, 68000, and 16032) were architecturally superior. However, Intel was quite right about one thing — people had written a lot of software for the 8086 and most of the really good stuff was written in 8086 assembly language and could not be easily ported to the other processors. Worse, the software that people were writing for the 8086 was starting to get large; making it even more difficult to port it to the other chips. As a result, software developers were becoming locked into using the 8086 CPU.

About this time, Intel undoubtedly realized that they were getting locked into the 80x86 architecture, as well. The iAPX 432 project was on its death bed. People were no more interested in the iAPX 432 than they were the other processors (in fact, they were less interested). So Intel decided to do the only reasonable thing — extend the 8086 family so they could continue to make more money off their cash cow.

The first real extension to the 8086 family that found its way into general purpose PCs was the 80286 that appeared in 1982. This CPU answered the second complaint by adding the ability to address up to 16 MBytes of RAM (a formidable amount in 1982). Unfortunately, it did not extend the segment size beyond 64 kilobytes. In 1985 Intel introduced the 80386 microprocessor. This chip answered most of the complaints about the x86 family, and then some, but people still complained about these problems for nearly ten years after the introduction of the 80386.

Intel was suffering at the hands of Microsoft and the installed base of existing PCs. When IBM introduced the floppy disk drive for the IBM PC they didn't choose an operating system to ship with it. Instead, they offered their customers a choice of the widely available operating systems at the time. Of course, Digital Research had ported CP/M to the PC, UCSD/Softech had ported UCSD Pascal (a very popular language/operating system at the time) to the PC, and Microsoft had quickly purchased a CP/M knock-off named QD DOS (for Quick and Dirty DOS) from Seattle Microsystems, relabelled it "MS-DOS", and offered this as well. CP/M-86 cost somewhere in the vicinity of \$595. UCSD Pascal was selling for something like \$795. MS-DOS was selling for \$50. Guess which one sold more copies! Within a year, almost no one ran CP/M or UCSD Pascal on PCs. Microsoft and MS-DOS (also called IBM DOS) ruled the PC.

MS-DOS v1.0 lived up to its "quick and dirty" heritage. Working furiously, Microsoft's engineers added lots of new features (many taken from the UNIX operating system and shell program) and MS-DOS v2.0 appeared shortly thereafter. Although still crude, MS-DOS v2.0 was a substantial improvement and people started writing tons of software for it.

Unfortunately, MS-DOS, even in its final version, wasn't the best operating system design. In particular, it left all but rudimentary control of the hardware to the application programmer. It provided a file system so application writers didn't have to deal with the disk drive and it provided mediocre support for keyboard input and character display. It provided nearly useless support for other devices. As a result, most application programmers (and most high level languages) bypassed MS-DOS device control and used MS-DOS primarily as a file system module.

In addition to poor device management, MS-DOS provided nearly non-existent memory management. For all intents and purposes, once MS-DOS started a program running, it was that program's responsibility to manage the system's resources. Not only did this create extra work for application programmers, but it was one of the main reasons most software could not take advantage of the new features Intel was adding to their microprocessors.

When Intel introduced the 80286 and, later, the 80386, the only way to take advantage of their extra addressing capabilities and the larger segments of the 80386 was to operate in a so-called *protected mode*. Unfortunately, neither MS-DOS nor most applications (that managed memory themselves) were capable of operating in protected mode without substantial change (actually, it would have been easy to modify MS-DOS to use protected mode, but it would have broken all the existing software that ran under MS-DOS; Microsoft, like Intel, couldn't afford to alienate the software developers in this manner).

Even if Microsoft could magically make MS-DOS run under protected mode, they couldn't afford to do so. When Intel introduced the 80386 microprocessor it was a very expensive device (the chip itself cost over \$1,000 at initial introduction). Although the 80286 had been out for three years, systems built around the 8088 were still extremely popular (since they were much lower cost than systems using the 80386). Software developers had a choice: they could solve their memory addressing problems and use the new features of the 80386 chip but limit their market to the few who had 80386 systems, or they could continue to suffer with the 64K segment limitation imposed by the 8088 and MS-DOS and be able to sell their software to millions of users who owned one of the earlier machines. The marketing departments of these companies ruled the day, all software was written to run on plain 8088 boxes so that it had a larger market. It wasn't until 1995, when Microsoft introduced Windows 95 that people finally felt they could abandon processors earlier than the 80386. The end result was the people were still complaining about the Intel architecture and its 64K segment limitation ten years after Intel had corrected the problem. The concept of upwards compatibility was clearly a double-edged sword in this case.

Segmentation had developed such a bad name over the years that Microsoft abandoned the use of segments in their 32-bit versions of Windows (95, 98, NT, 2000, ME, etc.). In a couple of respects, this was a real shame because Intel finally did segmentation right (or, at least, pretty good) in the 80386 and later processors. By not allowing the use of segmentation in Win32 programs Microsoft limited the use of this powerful feature. They also limited their users to a maximum address space of 4GB (the Pentium Pro and later processors were capable of addressing 64GB of physical memory). Considering that many applications are starting to push the 4GB barrier, this limitation on Microsoft's part was ill-considered. Nevertheless, the "flat" memory model that Microsoft employs is easier to write software for, undoubtedly a big part of their decision not to use segmentation.

The introduction of Windows NT, that actually ran on CPUs other than Intel's, must have given Intel a major scare. Fortunately for Intel, NT was an abysmal failure on non-Intel architectures like the Alpha and the PowerPC. On the other hand, the new Windows architecture does make it easier to move existing applications to 64-bit processors like the IA-64; so maybe WinNT's flexibility will work to Intel's advantage after all.

The 8086 software legacy has both advanced and retarded the 80x86 architecture. On the one hand, had software developers not written so much software for the 80x86, Intel would have abandoned the family in favor of something better a long time ago (not an altogether bad thing, in many people's opinions). On the other hand, however, the general acceptance of the 80386 and later processors was greatly delayed by the fact that software developers were writing software for the installed base of processors.

Around 1996, two types of software actually accelerated the design and acceptance of Intel's newer processors: multimedia software and games. When Intel introduced the MMX extensions to the 80x86 instruction set, software developers ignored the installed base and immediately began writing software to take advantage of these new instructions. This change of heart took place because the MMX instructions allowed developers to do things they hadn't been able to do before - not simply run faster, but run fast enough to display actual video and quickly render 3D images. Combined with a change in pricing policy by Intel on new processor technology, the public quickly accepted these new systems.

Hard-core gamers, multimedia artists, and others quickly grabbed new machines and software as it became available. More often than not, each new generation of software would only run on the latest hardware, forcing these individuals to upgrade their equipment far more rapidly than ever before.

Intel, sensing an opportunity here, began developing CPUs with additional instruction targeted at specific applications. For example, the Pentium III introduced the SIMD (pronounced SIM-DEE) instructions that did

for floating point calculations what the MMX instructions did for integer calculations. Intel also hired lots of software engineers and began funding research into topic areas like speech recognition and (visual) pattern recognition in order to drive the new technologies that would require the new instructions their Pentium IV and later processors would offer. As this is being written, Intel is busy developing new uses for their specialized instructions so that system designers and software developers continue to use the 80x86 (and, perhaps, IA-64) family chips.

However, this discussion of fancy instruction sets is getting way ahead of the game. Let's take a long step back to the original 8086 chip and take a look at how system designers put a CPU together.

4.4 Basic CPU Design

A fair question to ask at this point is How exactly does a CPU perform assigned chores? This is accomplished by giving the CPU a fixed set of commands, or instructions, to work on. Keep in mind that CPU designers construct these processors using logic gates to execute these instructions. To keep the number of logic gates reasonably small, CPU designers must necessarily restrict the number and complexity of the commands the CPU recognizes. This small set of commands is the CPU's instruction set.

Programs in early (pre-Von Neumann) computer systems were often hard-wired into the circuitry. That is, the computer's wiring determined what problem the computer would solve. One had to rewire the circuitry in order to change the program. A very difficult task. The next advance in computer design was the programmable computer system, one that allowed a computer programmer to easily rewire the computer system using a sequence of sockets and plug wires. A computer program consisted of a set of rows of holes (sockets), each row representing one operation during the execution of the program. The programmer could select one of several instructions by plugging a wire into the particular socket for the desired instruction (see Figure 4.1).

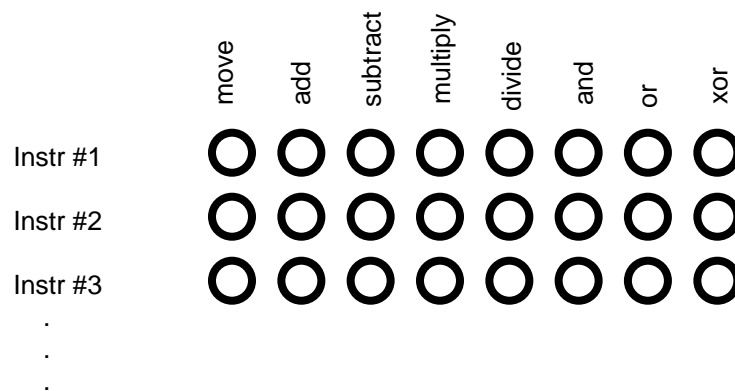


Figure 4.1 Patch Panel Programming

Of course, a major difficulty with this scheme is that the number of possible instructions is severely limited by the number of sockets one could physically place on each row. However, CPU designers quickly discovered that with a small amount of additional logic circuitry, they could reduce the number of sockets required from n holes for n instructions to $\log_2(n)$ holes for n instructions. They did this by assigning a numeric code to each instruction and then encode that instruction as a binary number using $\log_2(n)$ holes (see Figure 4.2).

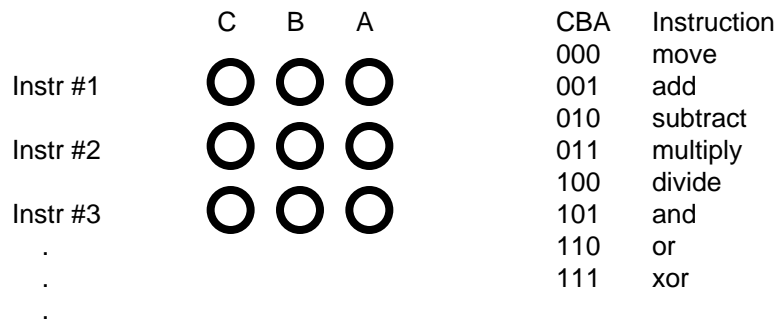


Figure 4.2 Encoding Instructions

This addition requires eight logic functions to decode the A, B, and C bits from the patch panel, but the extra circuitry is well worth the cost because it reduces the number of sockets that must be repeated for each instruction (this circuitry, by the way, is nothing more than a single three-line to eight-line decoder).

Of course, many CPU instructions are not stand-alone. For example, the move instruction is a command that moves data from one location in the computer to another (e.g., from one register to another). Therefore, the move instruction requires two operands: a source operand and a destination operand. The CPU's designer usually encodes these source and destination operands as part of the machine instruction, certain sockets correspond to the source operand and certain sockets correspond to the destination operand. Figure 4.3 shows one possible combination of sockets to handle this. The move instruction would move data from the source register to the destination register, the add instruction would add the value of the source register to the destination register, etc.

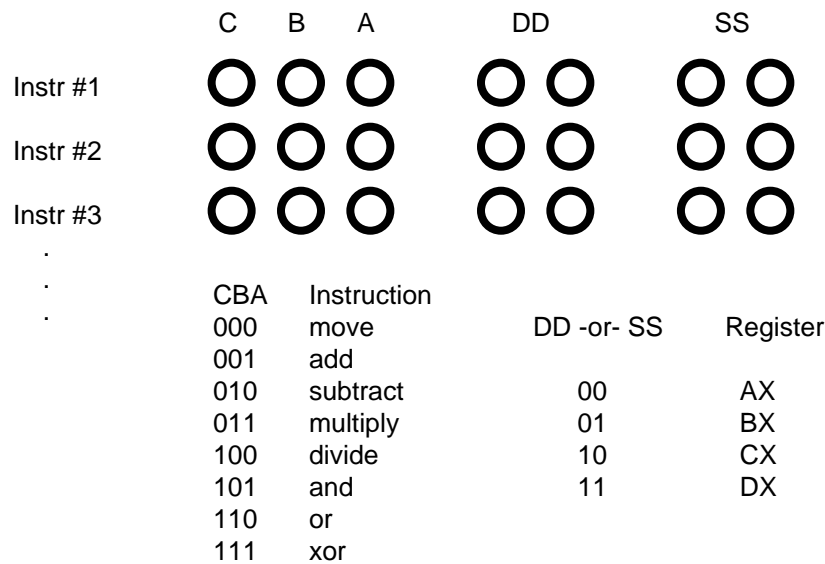


Figure 4.3 Encoding Instructions with Source and Destination Fields

One of the primary advances in computer design that the VNA provides is the concept of a stored program. One big problem with the patch panel programming method is that the number of program steps (machine instructions) is limited by the number of rows of sockets available on the machine. John Von Neumann and oth-

ers recognized a relationship between the sockets on the patch panel and bits in memory; they figured they could store the binary equivalents of a machine program in main memory and fetch each program from memory, load it into a special decoding register that connected directly to the instruction decoding circuitry of the CPU.

The trick, of course, was to add yet more circuitry to the CPU. This circuitry, the control unit (CU), fetches instruction codes (also known as operation codes or opcodes) from memory and moves them to the instruction decoding register. The control unit contains a special register, the instruction pointer that contains the address of an executable instruction. The control unit fetches this instruction's opcode from memory and places it in the decoding register for execution. After executing the instruction, the control unit increments the instruction pointer and fetches the next instruction from memory for execution, and so on.

When designing an instruction set, the CPU's designers generally choose opcodes that are a multiple of eight bits long so the CPU can easily fetch complete instructions from memory. The goal of the CPU's designer is to assign an appropriate number of bits to the instruction class field (move, add, subtract, etc.) and to the operand fields. Choosing more bits for the instruction field lets you have more instructions, choosing additional bits for the operand fields lets you select a larger number of operands (e.g., memory locations or registers). There are additional complications. Some instructions have only one operand or, perhaps, they don't have any operands at all. Rather than waste the bits associated with these fields, the CPU designers often reuse these fields to encode additional opcodes, once again with some additional circuitry. The Intel 80x86 CPU family takes this to an extreme with instructions ranging from one to almost 15 bytes long⁸.

4.5 Decoding and Executing Instructions: Random Logic Versus Microcode

Once the control unit fetches an instruction from memory, you may wonder "exactly how does the CPU execute this instruction?" In traditional CPU design there have been two common approaches: hardwired logic and emulation. The 80x86 family uses both of these techniques.

A hardwired, or *random logic*⁹, approach uses decoders, latches, counters, and other logic devices to move data around and operate on that data. The microcode approach uses a very fast but simple internal processor that uses the CPU's opcodes as an index into a table of operations (the *microcode*) and executes a sequence of *microinstructions* that do the work of the *macroinstruction* (i.e., the CPU instruction) they are emulating.

The random logic approach has the advantage that it is possible to devise faster CPUs if typical CPU speeds are faster than typical memory speeds (a situation that has been true for quite some time). The drawback to random logic is that it is difficult to design CPUs with large and complex instruction sets using a random logic approach. The logic to execute the instructions winds up requiring large percentage of the chip's real estate and it becomes difficult to properly lay out the logic so that related circuits are close to one another in the two-dimensional space of the chip,

CPUs based on microcode contain a small, very fast, execution unit that fetches instructions from the microcode bank (which is really nothing more than fast ROM on the CPU chip). This microcode executes one microinstruction per clock cycle and a sequence of microinstructions decode the instruction, fetch its operands, move the operands to appropriate functional units that do whatever calculations are necessary, store away necessary results, and then update appropriate registers and flags in anticipation of the next instruction.

8. Though this is, by no means, the most complex instruction set. The VAX, for example, has instructions up to 150 bytes long!

9. There is actually nothing random about this logic at all. This design technique gets its name from the fact that if you view a photomicrograph of a CPU die that uses microcode, the microcode section looks very regular; the same photograph of a CPU that utilizes random logic contains no such easily discernable patterns.

The microcode approach may appear to be substantially slower than the random logic approach because of all the steps involved. Actually, this isn't necessarily true. Keep in mind that with a random logic approach to instruction execution, part of the random logic is often a sequencer that steps through several states (one state per clock cycle). Whether you use your clock cycles executing microinstructions or stepping through a random logic state machine, you're still burning up clock cycles.

One advantage of microcode is that it makes better reuse of existing silicon on the CPU. Many CPU instructions (macroinstructions) execute some of the same microinstructions as many other instructions. This allows the CPU designer to use microcode subroutines to implement many common operations, thus saving silicon on the CPU. While it is certainly possible to share circuitry in a random logic device, this is often difficult if two circuits could otherwise share some logic but are across the chip from one another.

Another advantage of microcode is that it lets you create some very complex instructions that consist of several different operations. This provides programmers (especially assembly language programmers) with the ability to do more work with fewer instructions in their programs. In theory, this lets them write faster programs since they now execute half as many instructions, each doing twice the work of a simpler instruction set (the 80x86 MMX instruction set extension is a good example of this theory in action, although the MMX instructions do not use a microcode implementation).

Microcode does suffer from one disadvantage compared to random logic: the speed of the processor is tied to the speed of the internal microcode execution unit. Although the "microengine" itself is usually quite fast, the microengine must fetch its instruction from the microcode ROM. Therefore, if memory technology is slower than the execution logic, the microcode ROM will slow the microengine down because the system will have to introduce wait states into the microcode ROM access. Actually, microengines generally don't support the use of wait states, so this means that the microengine will have to run at the same speed as the microcode ROM. This effectively limits the speed at which the microengine, and therefore the CPU, can run.

Which approach is better for CPU design? That depends entirely on the current state of memory technology. If memory technology is faster than CPU technology, then the microcode approach tends to make more sense. If memory technology is slower than CPU technology, then random logic tends to produce the faster CPUs.

When Intel first began designing the 8086 CPU sometime between 1976 and 1978, memory technology was faster so they used microcode. Today, CPU technology is much faster than memory technology, so random logic CPUs tend to be faster. Most modern (non-x86) processors use random logic. The 80x86 family uses a combination of these technologies to improve performance while maintaining compatibility with the complex instruction set that relied on microcode way back in 1978.

4.6 RISC vs. CISC vs. VLIW

In the 1970s, CPU designers were busy extending their instruction sets to make their chips easier to program. It was very common to find a CPU designer poring over the assembly output of some high level language compiler searching for common two and three instruction sequences the compiler would emit. The designer would then create a single instruction that did the work of this two or three instruction sequence, the compiler writer would modify the compiler to use this new instruction, and a recompilation of the program would, presumably, produce a faster and shorter program than before.

Digital Equipment Corporation (now part of Compaq Computer who is looking at merging with Hewlett Packard as this is being written) raised this process to a new level in their VAX minicomputer series. It is not surprising, therefore, that many research papers appearing in the 1980s would commonly use the VAX as an example of what not to do.

The problem is, these designers lost track of what they were trying to do, or to use the old cliché, they couldn't see the forest for the trees. They assumed that there were making their processors faster by executing a

single instruction that previously required two or more. They also assumed that they were making the programs smaller, for exactly the same reason. They also assumed that they were making the processors easier to program because programmers (or compilers) could write a single instruction instead of using multiple instructions. In many cases, they assumed wrong.

In the early 80 s, researchers at IBM and several institutions like Stanford and UC Berkeley challenged the assumptions of these designers. They wrote several papers showing how complex instructions on the VAX mini-computer could actually be done faster (and sometimes in less space) using a sequence of simpler instructions. As a result, most compiler writers did not use the fancy new instructions on the VAX (nor did assembly language programmers). Some might argue that having an unused instruction doesn't hurt anything, but these researchers argued otherwise. They claimed that any unnecessary instructions required additional logic to implement and as the complexity of the logic grows it becomes more and more difficult to produce a high clock speed CPU.

This research led to the development of the RISC, or Reduced Instruction Set Computer, CPU. The basic idea behind RISC was to go in the opposite direction of the VAX. Decide what the smallest reasonable instruction set could be and implement that. By throwing out all the complex instructions, RISC CPU designers could use random logic rather than microcode (by this time, CPU speeds were outpacing memory speeds). Rather than making an individual instruction more complex, they could move the complexity to the system level and add many on-chip features to improve the overall system performance (like caches, pipelines, and other advanced mainframe features of the time). Thus, the great "RISC vs. CISC¹⁰" debate was born.

Before commenting further on the result of this debate, you should realize that RISC actually means "(Reduced Instruction) Set Computer," not "Reduced (Instruction Set) Computer." That is, the goal of RISC was to reduce the complexity of individual instructions, not necessarily reduce the number of instructions a RISC CPU supports. It was often the case that RISC CPUs had fewer instructions than their CISC counterparts, but this was not a precondition for calling a CPU a RISC device. Many RISC CPUs had more instructions than some of their CISC contemporaries, depending on how you count instructions.

First, there is no debate about one thing: if you have two CPUs, one RISC and one CISC and they both run at the same clock frequency and they execute the same average number of instructions per clock cycle, CISC is the clear winner. Since CISC processors do more work with each instruction, if the two CPUs execute the same number of instructions in the same amount of time, the CISC processor usually gets more work done.

However, RISC performance claims were based on the fact that RISC's simpler design would allow the CPU designers to reduce the overall complexity of the chip, thereby allowing it to run at a higher clock frequency. Further, with a little added complexity, they could easily execute more instructions per clock cycle, on the average, than their CISC contemporaries.

One drawback to RISC CPUs is that their code density was much lower than CISC CPUs. Although memory devices were dropping in price and the need to keep programs small was decreasing, low code density requires larger caches to maintain the same number of instructions in the cache. Further, since memory speeds were not keeping up with CPU speeds, the larger instruction sizes found on the RISC CPUs meant that the system spent more time bringing in those instructions from memory to cache since they could transfer fewer instructions per bus transaction. For many years, CPU architects argued to and fro about whether RISC or CISC was the better approach. With one big footnote, the RISC approach has generally won the argument. Most of the popular CISC systems, e.g., the VAX, the Z8000, the 16032/32016, and the 68000, have quietly faded away to be replaced by the likes of the PowerPC, the MIPS CPUs, the Alpha, and the SPARC. The one footnote here is, of course, the 80x86 family. Intel has proven that if you really want to keep extending a CISC architecture, and you're willing to throw a lot of money at it, you can extend it far beyond what anyone ever expected. As of late 2001/early 2002 the 80x86 is the raw performance leader. The CPU runs at a higher clock frequency than the competing RISC

10.CISC stands for Complex Instruction Set Computer and defines those CPUs that were popular at the time like the VAX and the 80x86.

chips; it executes fairly close to the same number of instructions per clock cycle as the competing RISC chips; it has about the same "average instruction size to cache size" ratio as the RISC chips; and it is a CISC, so many of the instructions do more work than their RISC equivalents. So overall, the 80x86 is, on the average, faster than contemporary RISC chips¹¹.

To achieve this raw performance advantage, the 80x86 has borrowed heavily from RISC research. Intel has divided the instruction set into a set of simple instructions that Intel calls the "RISC core" and the remaining, complex instructions. The complex instructions do not execute as rapidly as the RISC core instructions. In fact, it is often the case that the task of a complex instruction can be accomplished faster using multiple RISC core instructions. Intel supports the complex instructions to provide full compatibility with older software, but compiler writers and assembly language programmers tend to avoid the use of these instructions. Note that Intel moves instructions between these two sets over time. As Intel improves the processor they tend to speed up some of the slower, complex, instructions. Therefore, it is not possible to give a static list of instructions you should avoid; instead, you will have to refer to Intel's documentation for the specific processor you use.

Later Pentium processors do not use an interpretive engine and microcode like the earlier 80x86 processors. Instead, the Pentium core processors execute a set of "micro-operations" (or "micro-ops"). The Pentium processors translate the 80x86 instruction set into a sequence of micro-ops on the fly. The RISC core instructions typically generate a single micro-op while the CISC instructions generate a sequence of two or more micro-ops. For the purposes of determining the performance of a section of code, we can treat each micro-op as a single instruction. Therefore, the CISC instructions are really nothing more than "macro-instructions" that the CPU automatically translates into a sequence of simpler instructions. This is the reason the complex instructions take longer to execute.

Unfortunately, as the x86 nears its 25th birthday, it's clear (to Intel, at least) that it's been pushed to its limits. This is why Intel is working with HP to base their IA-64 architecture on the PA-RISC instruction set. The IA-64 architecture is an interesting blend. On the one hand, it (supposedly) supports object-code compatibility with the previous generation x86 family (though at reduced performance levels). Obviously, it's a RISC architecture since it was originally based on Hewlett-Packard's PA-RISC (PA=Precision Architecture) design. However, Intel and HP have extended on the RISC design by using another technology: Very Long Instruction Word (VLIW) computing. The idea behind VLIW computing is to use a very long opcode that handle multiple operations in parallel. In some respects, this is similar to CISC computing since a single VLIW "instruction" can do some very complex things. However, unlike CISC instructions, a VLIW instruction word can actually complete several independent tasks simultaneously. Effectively, this allows the CPU to execute some number of instructions in parallel.

Intel's VLIW approach is risky. To succeed, they are depending on compiler technology that doesn't yet exist. They made this same mistake with the iAPX 432. It remains to be seen whether history is about to repeat itself or if Intel has a winner on their hands.

4.7 Instruction Execution, Step-By-Step

To understand the problems with developing an efficient CPU, let's consider four representative 80x86 instructions: MOV, ADD, LOOP, and JNZ (jump if not zero). These instructions will allow us to explore many of the issues facing the x86 CPU designer.

You've seen the MOV and ADD instructions in previous chapters so there is no need to review them here. The LOOP and JNZ instructions are new, so it's probably a good idea to explain what they do before proceeding. Both of these instructions

11. Note, by the way, that this doesn't imply that 80x86 systems are faster than computer systems built around RISC chips. Many RISC systems gain additional speed by supporting multiple processors better than the x86 or by having faster bus throughput. This is one reason, for example, why Internet companies select Sun equipment for their web servers.

are *conditional jump* instructions. A conditional jump instruction tests some condition and jumps to some other instruction in memory if the condition is true and they fall through to the next instruction if the condition is false. This is basically the opposite of HLA's IF statement (which falls through if the condition is true and jumps to the else section if the condition is false). The JNZ (jump if not zero) instruction tests the CPU's zero flag and transfers control to some target location if the zero flag contains zero; JNZ falls through to the next instruction if the zero flag contains one. The program specifies the target instruction to jump to by specifying the distance from the JNZ instruction to the target instruction as a small signed integer (for our purposes here, we'll assume that the distance is within the range ± 128 bytes so the instruction uses a single byte to specify the distance to the target location).

The last instruction of interest to us here is the LOOP instruction. The LOOP instruction decrements the value of the ECX register and transfers control to a target instruction within ± 128 bytes if ECX does not contain zero (after the decrement). This is a good example of a CISC instruction since it does multiple operations: (1) it subtracts one from ECX and then it (2) does a conditional jump if ECX does not contain zero. That is, LOOP is equivalent to the following two 80x86 instructions¹²:

```
loop SomeLabel;
```

-is roughly equivalent to-

```
dec( ecx );  
jnz SomeLabel;
```

Note that *SomeLabel* specifies the address of the target instruction that must be within about ± 128 bytes of the LOOP or JNZ instructions above. The LOOP instruction is a good example of a complex (vs. RISC core) instruction on the Pentium processors. It is actually faster to execute a DEC and a JNZ instruction¹³ than it is to execute a LOOP instruction. In this section we will not concern ourselves with this issue; we will assume that the LOOP instruction operates as though it were a RISC core instruction.

The 80x86 CPUs do not execute instructions in a single clock cycle. For example, the MOV instruction (which is relatively simple) could use the following execution steps¹⁴:

- Fetch the instruction byte from memory.
- Update the EIP register to point at the next byte.
- Decode the instruction to see what it does.
- If required, fetch a 16-bit instruction operand from memory.
- If required, update EIP to point beyond the operand.
- If required, compute the address of the operand (e.g., EBX+disp) .
- Fetch the operand.
- Store the fetched value into the destination register

If we allocate one clock cycle for each of the above steps, an instruction could take as many as eight clock cycles to complete (note that three of the steps above are optional, depending on the MOV instruction's addressing mode, so a simple MOV instruction could complete in as few as five clock cycles).

The ADD instruction is a little more complex. Here's a typical set of operations the ADD(reg, reg) instruction must complete:

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.
- Get the value of the source operand and send it to the ALU.
- Fetch the value of the destination operand (a register) and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the first register operand.
- Update the flags register with the result of the addition operation.

If the source operand is a memory location, the operation is slightly more complicated:

12.This sequence is not exactly equivalent to LOOP since this sequence affects the flags while LOOP does not.

13.Actually, you'll see a little later that there is a *decrement* instruction you can use to subtract one from ECX. The decrement instruction is better because it is shorter.

14.It is not possible to state exactly what steps each CPU requires since many CPUs are different from one another.

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.
- If required, fetch a displacement for use in the effective address calculation
- If required, update EIP to point beyond the displacement value.
- Get the value of the source operand from memory and send it to the ALU.
- Fetch the value of the destination operand (a register) and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the register operand.
- Update the flags register with the result of the addition operation.

ADD(const, memory) is the messiest of all, this code sequence looks something like the following:

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.
- If required, fetch a displacement for use in the effective address calculation
- If required, update EIP to point beyond the displacement value.
- Fetch the constant value from memory and send it to the ALU.
- Update EIP to point beyond the constant's value (at the next instruction in memory).
- Get the value of the source operand from memory and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the memory operand.
- Update the flags register with the result of the addition operation.

Note that there are other forms of the ADD instruction requiring their own special processing. These are just representative examples. As you see in these examples, the ADD instruction could take as many as ten steps (or cycles) to complete. Note that this is one advantage of a RISC design. Most RISC design have only one or two forms of the ADD instruction (that add registers together and, perhaps, that add constants to registers). Since register to register adds are often the fastest (and constant to register adds are probably the second fastest), the RISC CPUs force you to use the fastest forms of these instructions.

The JNZ instruction might use the following sequence of steps:

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.
- Fetch a displacement byte to determine the jump distance send this to the ALU
- Update EIP to point at the next byte.
- Test the zero flag to see if it is clear.
- If the zero flag was clear, copy the EIP register to the ALU.
- If the zero flag was clear, instruct the ALU to add the displacement and EIP register values.
- If the zero flag was clear, copy the result of the addition above back to the EIP register.

Notice how the JNZ instruction requires fewer steps if the jump is not taken. This is very typical for conditional jump instructions. If each step above corresponds to one clock cycle, the JNZ instruction would take six or nine clock cycles, depending on whether the branch is taken. Because the 80x86 JNZ instruction does not allow different types of operands, there is only one sequence of steps needed for this application.

The 80x86 LOOP instruction might use an execution sequence like the following:

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.
- Fetch the value of the ECX register and send it to the ALU.
- Instruct the ALU to decrement the value.
- Send the result back to the ECX register. Set a special internal flag if this value is non-zero.
- Fetch a displacement byte to determine the jump distance send this to the ALU
- Update EIP to point at the next byte.
- Test the special flag to see if ECX was non-zero.
- If the flag was set, copy the EIP register to the ALU.
- If the flag was set, instruct the ALU to add the displacement and EIP register values.

- If the flag was set, copy the result of the addition above back to the EIP register.

Although a given 80x86 CPU might not execute the steps for the instructions above, they all execute some sequence of operations. Each operation requires a finite amount of time to execute (generally, one clock cycle per operation or *stage* as we usually refer to each of the above steps). Obviously, the more steps needed for an instruction, the slower it will run. This is why complex instructions generally run slower than simple instructions, complex instructions usually have lots of execution stages.

4.8 Parallelism – the Key to Faster Processors

An early goal of the RISC processors was to execute one instruction per clock cycle, on the average. However, even if a RISC instruction is simplified, the actual execution of the instruction still requires multiple steps. So how could they achieve this goal? And how do later members the 80x86 family with their complex instruction sets also achieve this goal? The answer is parallelism.

Consider the following steps for a MOV(reg, reg) instruction:

- Fetch the instruction byte from memory.
- Update the EIP register to point at the next byte.
- Decode the instruction to see what it does.
- Fetch the source register.
- Store the fetched value into the destination register

There are five stages in the execution of this instruction with certain dependencies between each stage. For example, the CPU must fetch the instruction byte from memory before it updates EIP to point at the next byte in memory. Likewise, the CPU must decode the instruction before it can fetch the source register (since it doesn't know it needs to fetch a source register until it decodes the instruction). As a final example, the CPU must fetch the source register before it can store the fetched value in the destination register.

Most of the stages in the execution of this MOV instruction are *serial*. That is, the CPU must execute one stage before proceeding to the next. The one exception is the "Update EIP" step. Although this stage must follow the first stage, none of the following stages in the instruction depend upon this step. Therefore, this could be the third, fourth, or fifth step in the calculation and it wouldn't affect the outcome of the instruction. Further, we could execute this step concurrently with any of the other steps and it wouldn't affect the operation of the MOV instruction, e.g.,

- Fetch the instruction byte from memory.
- Decode the instruction to see what it does.
- Fetch the source register and update the EIP register to point at the next byte.
- Store the fetched value into the destination register

By doing two of the stages in parallel, we can reduce the execution time of this instruction by one clock cycle. Although the remaining stages in the "mov(reg, reg);" instruction must remain serialized (that is, they must take place in exactly this order), other forms of the MOV instruction offer similar opportunities to overlapped portions of their execution to save some cycles. For example, consider the "mov([ebx+disp], eax);" instruction:

- Fetch the instruction byte from memory.
- Update the EIP register to point at the next byte.
- Decode the instruction to see what it does.
- Fetch a displacement operand from memory.
- Update EIP to point beyond the displacement.
- Compute the address of the operand (e.g., EBX+disp) .
- Fetch the operand.
- Store the fetched value into the destination register

Once again there is the opportunity to overlap the execution of several stages in this instruction, for example:

- Fetch the instruction byte from memory.
- Decode the instruction to see what it does and update the EIP register to point at the next byte.
- Fetch a displacement operand from memory.
- Compute the address of the operand (e.g., EBX+disp) and update EIP to point beyond the displacement..
- Fetch the operand.

- Store the fetched value into the destination register

In this example, we reduced the number of execution steps from eight to six by overlapping the update of EIP with two other operations.

As a last example, consider the "add(const, [ebx+disp]);" instruction (the instruction with the largest number of steps we've considered thus far). It's non-overlapped execution looks like this:

- Fetch the instruction byte from memory.
- Update EIP to point at the next byte.
- Decode the instruction.
- Fetch a displacement for use in the effective address calculation
- Update EIP to point beyond the displacement value.
- Fetch the constant value from memory and send it to the ALU.
- Compute the address of the memory operand (EBX+disp).
- Get the value of the source operand from memory and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the memory operand.
- Update the flags register with the result of the addition operation.
- Update EIP to point beyond the constant's value (at the next instruction in memory).

We can overlap at least three steps in this instruction by noting that certain stages don't depend on the result of their immediate predecessor

- Fetch the instruction byte from memory.
- Decode the instruction and update EIP to point at the next byte.
- Fetch a displacement for use in the effective address calculation
- Update EIP to point beyond the displacement value.
- Fetch the constant value from memory and send it to the ALU.
- Compute the address of the memory operand (EBX+disp).
- Get the value of the source operand from memory and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the memory operand and update the flags register with the result of the addition operation and update EIP to point beyond the constant's value.

Note that we could not merge one of the "Update EIP" operations because the previous stage and following stages of the instruction both use the value of EIP before and after the update.

Unlike the MOV instruction, the steps in the ADD instruction above are not all dependent upon the previous stage in the instruction's execution. For example, the sequence above fetches the constant from memory and then computes the effective address (EBX+disp) of the memory operand. Neither operation depends upon the other, so we could easily swap their positions above to yield the following:

- Fetch the instruction byte from memory.
- Decode the instruction and update EIP to point at the next byte.
- Fetch a displacement for use in the effective address calculation
- Update EIP to point beyond the displacement value.
- Compute the address of the memory operand (EBX+disp).
- Fetch the constant value from memory and send it to the ALU.
- Get the value of the source operand from memory and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the memory operand and update the flags register with the result of the addition operation and update EIP to point beyond the constant's value.

This doesn't save any steps, but it does reduce some dependencies between certain stages and their immediate predecessors, allowing additional parallel operation. For example, we can now merge the "Update EIP" operation with the effective address calculation:

- Fetch the instruction byte from memory.
- Decode the instruction and update EIP to point at the next byte.
- Fetch a displacement for use in the effective address calculation

- Compute the address of the memory operand (EBX+disp) and update EIP to point beyond the displacement value.
- Fetch the constant value from memory and send it to the ALU.
- Get the value of the source operand from memory and send it to the ALU.
- Instruct the ALU to add the values.
- Store the result back into the memory operand and update the flags register with the result of the addition operation and update EIP to point beyond the constant's value.

Although it might seem possible to fetch the constant and the memory operand in the same step (since their values do not depend upon one another), the CPU can't actually do this (yet!) because it has only a single data bus. Since both of these values are coming from memory, we can't bring them into the CPU during the same step because the CPU uses the data bus to fetch these two values. In the next section you'll see how we can overcome this problem.

By overlapping various stages in the execution of these instructions we've been able to substantially reduce the number of steps (i.e., clock cycles) that the instructions need to complete execution. This process of executing various steps of the instruction in parallel with other steps is a major key to improving CPU performance without cranking up the clock speed on the chip. In this section we've seen how to speed up the execution of an instruction by doing many of the internal operations of that instruction in parallel. However, there's only so much to be gained from this approach. In this approach, the instructions themselves are still serialized (one instruction completes before the next instruction begins execution). Starting with the next section we'll start to see how to overlap the execution of adjacent instructions in order to save additional cycles.

4.8.1 The Prefetch Queue – Using Unused Bus Cycles

The key to improving the speed of a processor is to perform operations in parallel. If we were able to do two operations on each clock cycle, the CPU would execute instructions twice as fast when running at the same clock speed. However, simply deciding to execute two operations per clock cycle is not so easy. Many steps in the execution of an instruction share *functional units* in the CPU (functional units are groups of logic that perform a common operation, e.g., the ALU and the CU). A functional unit is only capable of one operation at a time. Therefore, you cannot do two operations that use the same functional unit concurrently (e.g., incrementing the EIP register and adding two values together). Another difficulty with doing certain operations concurrently is that one operation may depend on the other's result. For example, the two steps of the ADD instruction that involve adding two values and then storing their sum. You cannot store the sum into a register until after you've computed the sum. There are also some other resources the CPU cannot share between steps in an instruction. For example, there is only one data bus; the CPU cannot fetch an instruction opcode at the same time it is trying to store some data to memory. The trick in designing a CPU that executes several steps in parallel is to arrange those steps to reduce conflicts or add additional logic so the two (or more) operations can occur simultaneously by executing in different functional units.

Consider again the steps the MOV(mem/reg, reg) instruction requires:

- Fetch the instruction byte from memory.
- Update the EIP register to point at the next byte.
- Decode the instruction to see what it does.
- If required, fetch a displacement operand from memory.
- If required, update EIP to point beyond the displacement.
- Compute the address of the operand, if required (i.e., EBX+xxxx) .
- Fetch the operand.
- Store the fetched value into the destination register

The first operation uses the value of the EIP register (so we cannot overlap incrementing EIP with it) and it uses the bus to fetch the instruction opcode from memory. Every step that follows this one depends upon the opcode it fetches from memory, so it is unlikely we will be able to overlap the execution of this step with any other.

The second and third operations do not share any functional units, nor does decoding an opcode depend upon the value of the EIP register. Therefore, we can easily modify the control unit so that it increments the EIP register at the same time it decodes the instruction. This will shave one cycle off the execution of the MOV instruction.

The third and fourth operations above (decoding and optionally fetching the displacement operand) do not look like they can be done in parallel since you must decode the instruction to determine if it the CPU needs to fetch an operand from memory. However, we could design the CPU to go ahead and fetch the operand anyway, so that it's available if we need it. There is one problem with this idea, though, we must have the address of the operand to fetch (the value in the EIP register) and if we

must wait until we are done incrementing the EIP register before fetching this operand. If we are incrementing EIP at the same time we're decoding the instruction, we will have to wait until the next cycle to fetch this operand.

Since the next three steps are optional, there are several possible instruction sequences at this point:

- #1 (step 4, step 5, step 6, and step 7) — e.g., `MOV([ebx+1000], eax)`
- #2 (step 4, step 5, and step 7) — e.g., `MOV(disp, eax)` -- assume displacement address is 1000
- #3 (step 6 and step 7) — e.g., `MOV([ebx], eax)`
- #4 (step 7) — e.g., `MOV(ebx, eax)`

In the sequences above, step seven always relies on the previous steps in the sequence. Therefore, step seven cannot execute in parallel with any of the other steps. Step six also relies upon step four. Step five cannot execute in parallel with step four since step four uses the value in the EIP register, however, step five can execute in parallel with any other step. Therefore, we can shave one cycle off the first two sequences above as follows:

- #1 (step 4, step 5/6, and step 7)
- #2 (step 4, step 5/7)
- #3 (step 6 and step 7)
- #4 (step 7)

Of course, there is no way to overlap the execution of steps seven and eight in the MOV instruction since it must surely fetch the value before storing it away. By combining these steps, we obtain the following steps for the MOV instruction:

- Fetch the instruction byte from memory.
- Decode the instruction and update ip
- If required, fetch a displacement operand from memory.
- Compute the address of the operand, if required (i.e., `ebx+xxxx`).
- Fetch the operand, if required update EIP to point beyond `xxxx`.
- Store the fetched value into the destination register

By adding a small amount of logic to the CPU, we've shaved one or two cycles off the execution of the MOV instruction. This simple optimization works with most of the other instructions as well.

Consider what happens with the MOV instruction above executes on a CPU with a 32-bit data bus. If the MOV instruction fetches an eight-bit displacement from memory, the CPU may actually wind up fetching the following three bytes after the displacement along with the displacement value (since the 32-bit data bus lets us fetch four bytes in a single bus cycle). The second byte on the data bus is actually the opcode of the next instruction. If we could save this opcode until the execution of the next instruction, we could shave a cycle of its execution time since it would not have to fetch the opcode byte. Furthermore, since the instruction decoder is idle while the CPU is executing the MOV instruction, we can actually decode the next instruction while the current instruction is executing, thereby shaving yet another cycle off the execution of the next instruction. This, effectively, overlaps a portion of the MOV instruction with the beginning of the execution of the next instruction, allowing additional parallelism.

Can we improve on this? The answer is yes. Note that during the execution of the MOV instruction the CPU is not accessing memory on every clock cycle. For example, while storing the data into the destination register the bus is idle. During time periods when the bus is idle we can pre-fetch instruction opcodes and operands and save these values for executing the next instruction.

The hardware to do this is the prefetch queue. Figure 4.4 shows the internal organization of a CPU with a prefetch queue. The Bus Interface Unit, as its name implies, is responsible for controlling access to the address and data buses. Whenever some component inside the CPU wishes to access main memory, it sends this request to the bus interface unit (or BIU) that acts as a "traffic cop" and handles simultaneous requests for bus access by different modules (e.g., the execution unit and the prefetch queue).

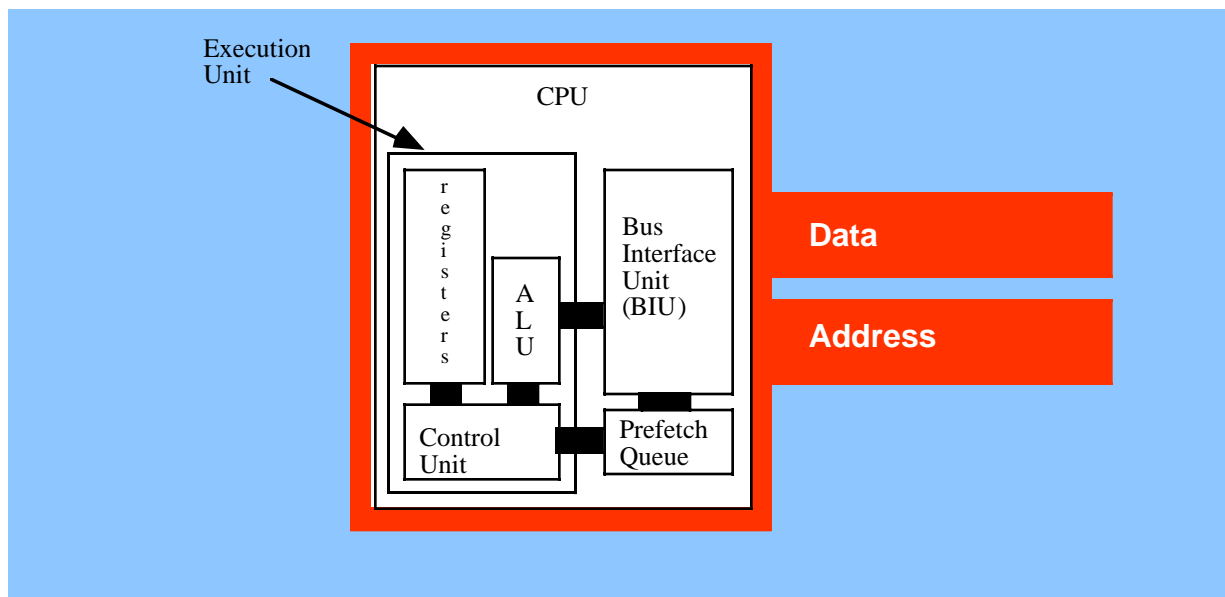


Figure 4.4 CPU Design with a Prefetch Queue

Whenever the execution unit is not using the Bus Interface Unit, the BIU can fetch additional bytes from the instruction stream. Whenever the CPU needs an instruction or operand byte, it grabs the next available byte from the prefetch queue. Since the BIU grabs four bytes at a time from memory (assuming a 32-bit data bus) and it generally consumes fewer than four bytes per clock cycle, any bytes the CPU would normally fetch from the instruction stream will already be sitting in the prefetch queue.

Note, however, that we're not guaranteed that all instructions and operands will be sitting in the prefetch queue when we need them. For example, consider the "JNZ Label;" instruction, if it transfers control to *Label*, will invalidate the contents of the prefetch queue. If this instruction appears at locations 400 and 401 in memory (it is a two-byte instruction), the prefetch queue will contain the bytes at addresses 402, 403, 404, 405, 406, 407, etc. If the target address of the JNZ instruction is 480, the bytes at addresses 402, 403, 404, etc., won't do us any good. So the system has to pause for a moment to fetch the double word at address 480 before it can go on.

Another improvement we can make is to overlap instruction decoding with the last step of the previous instruction. After the CPU processes the operand, the next available byte in the prefetch queue is an opcode, and the CPU can decode it in anticipation of its execution. Of course, if the current instruction modifies the EIP register then any time spent decoding the next instruction goes to waste, but since this occurs in parallel with other operations, it does not slow down the system (though it does require extra circuitry to do this).

The instruction execution sequence now assumes that the following events occur in the background:

CPU Prefetch Events:

- If the prefetch queue is not full (generally it can hold between eight and thirty-two bytes, depending on the processor) and the BIU is idle on the current clock cycle, fetch the next double word from memory at the address in EIP at the beginning of the clock cycle¹⁵.
- If the instruction decoder is idle and the current instruction does not require an instruction operand, begin decoding the opcode at the front of the prefetch queue (if present), otherwise begin decoding the byte beyond the current operand in the prefetch queue (if present). If the desired byte is not in the prefetch queue, do not execute this event.

15.This operation fetches only a byte if ip contains an odd value.

Now let's reconsider our "mov(reg, reg);" instruction from the previous section. With the addition of the prefetch queue and the bus interface unit, fetching and decoding opcode bytes, as well as updating the EIP register, takes place in parallel with the previous instruction. Without the BIU and the prefetch queue, the "mov(reg, reg);" requires the following steps:

- Fetch the instruction byte from memory.
- Decode the instruction to see what it does.
- Fetch the source register and update the EIP register to point at the next byte.
- Store the fetched value into the destination register

However, now that we can overlap the instruction fetch and decode with the previous instruction, we now get the following steps:

- Fetch and Decode Instruction - overlapped with previous instruction
- Fetch the source register and update the EIP register to point at the next byte.
- Store the fetched value into the destination register

The instruction execution timings make a few optimistic assumptions, namely that the opcode is already present in the prefetch queue and that the CPU has already decoded it. If either case is not true, additional cycles will be necessary so the system can fetch the opcode from memory and/or decode the instruction.

Because they invalidate the prefetch queue, jump and conditional jump instructions (when actually taken) are much slower than other instructions. This is because the CPU cannot overlap fetching and decoding the opcode for the next instruction with the execution of the jump instruction since the opcode is (probably) not in the prefetch queue. Therefore, it may take several cycles after the execution of one of these instructions for the prefetch queue to recover and the CPU is decoding opcodes in parallel with the execution of previous instructions. This has one very important implication to your programs: if you want to write fast code, make sure to avoid jumping around in your program as much as possible.

Note that the conditional jump instructions only invalidate the prefetch queue if they actually make the jump. If the condition is false, they fall through to the next instruction and continue to use the values in the prefetch queue as well as any pre-decoded instruction opcodes. Therefore, if you can determine, while writing the program, which condition is most likely (e.g., less than vs. not less than), you should arrange your program so that the most common case falls through and conditional jump rather than take the branch.

Instruction size (in bytes) can also affect the performance of the prefetch queue. The longer the instruction, the faster the CPU will empty the prefetch queue. Instructions involving constants and memory operands tend to be the largest. If you place a string of these in a row, the CPU may wind up having to wait because it is removing instructions from the prefetch queue faster than the BIU is copying data to the prefetch queue. Therefore, you should attempt to use shorter instructions whenever possible since they will improve the performance of the prefetch queue.

Usually, including the prefetch queue improves performance. That's why Intel provides the prefetch queue on many models of the 80x86 family, from the 8088 on up. On these processors, the BIU is constantly fetching data for the prefetch queue whenever the program is not actively reading or writing data.

Prefetch queues work best when you have a wide data bus. The 8086 processor runs much faster than the 8088 because it can keep the prefetch queue full with fewer bus accesses. Don't forget, the CPU needs to use the bus for other purposes than fetching opcodes, displacements, and immediate constants. Instructions that access memory compete with the prefetch queue for access to the bus (and, therefore, have priority). If you have a sequence of instructions that all access memory, the prefetch queue may become empty if there are only a few bus cycles available for filling the prefetch queue during the execution of these instructions. Of course, once the prefetch queue is empty, the CPU must wait for the BIU to fetch new opcodes from memory, slowing the program.

A wider data bus allows the BIU to pull in more prefetch queue data in the few bus cycles available for this purpose, so it is less likely the prefetch queue will ever empty out with a wider data bus. Executing shorter instructions also helps keep the prefetch queue full. The reason is that the prefetch queue has time to refill itself with the shorter instructions. Moral of the story: when programming a processor with a prefetch queue, always use the shortest instructions possible to accomplish a given task.

4.8.2 Pipelining – Overlapping the Execution of Multiple Instructions

Executing instructions in parallel using a bus interface unit and an execution unit is a special case of pipelining. The 80x86 family, starting with the 80486, incorporates pipelining to improve performance. With just a few exceptions, we'll see that pipelining allows us to execute one instruction per clock cycle.

The advantage of the prefetch queue was that it let the CPU overlap instruction fetching and decoding with instruction execution. That is, while one instruction is executing, the BIU is fetching and decoding the next instruction. Assuming you're willing to add hardware, you can execute almost all operations in parallel. That is the idea behind pipelining.

4.8.2.1 A Typical Pipeline

Consider the steps necessary to do a generic operation:

- Fetch opcode.
- Decode opcode and (in parallel) prefetch a possible displacement or constant operand (or both)
- Compute complex addressing mode (e.g., [ebx+xxxx]), if applicable.
- Fetch the source value from memory (if a memory operand) and the destination register value (if applicable).
- Compute the result.
- Store result into destination register.

Assuming you're willing to pay for some extra silicon, you can build a little "mini-processor" to handle each of the above steps. The organization would look something like Figure 4.5.

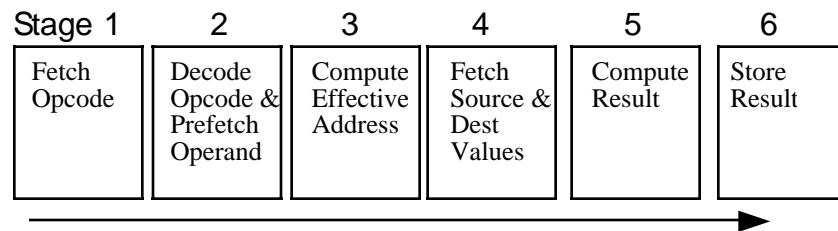


Figure 4.5 A Pipelined Implementation of Instruction Execution

Note how we've combined some stages from the previous section. For example, in stage four of Figure 4.5 the CPU fetches the source and destination operands in the same step. You can do this by putting multiple data paths inside the CPU (e.g., from the registers to the ALU) and ensuring that no two operands ever compete for simultaneous use of the data bus (i.e., no memory-to-memory operations).

If you design a separate piece of hardware for each stage in the pipeline above, almost all these steps can take place in parallel. Of course, you cannot fetch and decode the opcode for more than one instruction at the same time, but you can fetch one opcode while decoding the previous instruction. If you have an n-stage pipeline, you will usually have n instructions executing concurrently.

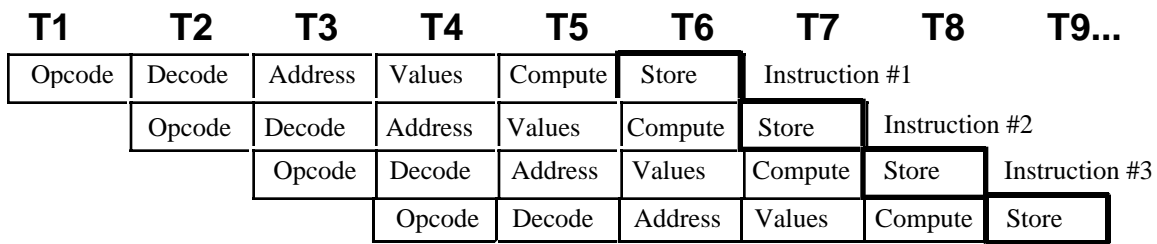


Figure 4.6 Instruction Execution in a Pipeline

Figure 4.6 shows pipelining in operation. T1, T2, T3, etc., represent consecutive “ticks” of the system clock. At T=T1 the CPU fetches the opcode byte for the first instruction.

At T=T2, the CPU begins decoding the opcode for the first instruction. In parallel, it fetches a block of bytes from the prefetch queue in the event the instruction has an operand. Since the first instruction no longer needs the opcode fetching circuitry, the CPU instructs it to fetch the opcode of the second instruction in parallel with the decoding of the first instruction. Note there is a minor conflict here. The CPU is attempting to fetch the next byte from the prefetch queue for use as an operand, at the same time it is fetching operand data from the prefetch queue for use as an opcode. How can it do both at once? You’ll see the solution in a few moments.

At T=T3 the CPU computes an operand address for the first instruction, if any. The CPU does nothing on the first instruction if it does not use an addressing mode requiring such computation. During T3, the CPU also decodes the opcode of the second instruction and fetches any necessary operand. Finally the CPU also fetches the opcode for the third instruction. With each advancing tick of the clock, another step in the execution of each instruction in the pipeline completes, and the CPU fetches yet another instruction from memory.

This process continues until at T=T6 the CPU completes the execution of the first instruction, computes the result for the second, etc., and, finally, fetches the opcode for the sixth instruction in the pipeline. The important thing to see is that after T=T5 the CPU completes an instruction on every clock cycle. Once the CPU fills the pipeline, it completes one instruction on each cycle. Note that this is true even if there are complex addressing modes to be computed, memory operands to fetch, or other operations which use cycles on a non-pipelined processor. All you need to do is add more stages to the pipeline, and you can still effectively process each instruction in one clock cycle.

A bit earlier you saw a small conflict in the pipeline organization. At T=T2, for example, the CPU is attempting to prefetch a block of bytes for an operand and at the same time it is trying to fetch the next opcode byte. Until the CPU decodes the first instruction it doesn’t know how many operands the instruction requires nor does it know their length. However, the CPU needs to know this information to determine the length of the instruction so it knows what byte to fetch as the opcode of the next instruction. So how can the pipeline fetch an instruction opcode in parallel with an address operand?

One solution is to disallow this. If an instruction has an address or constant operand, we simply delay the start of the next instruction (this is known as a *hazard* as you shall soon see). Unfortunately, many instructions have these additional operands, so this approach will have a substantial negative impact on the execution speed of the CPU.

The second solution is to throw (a lot) more hardware at the problem. Operand and constant sizes usually come in one, two, and four-byte lengths. Therefore, if we actually fetch three bytes from memory, at offsets one, three, and five, beyond the current opcode we are decoding, we know that one of these bytes will probably contain the opcode of the next instruction. Once we are through decoding the current instruction we know how long it will be and, therefore, we know the offset of the next opcode. We can use a simple data selector circuit to choose which of the three opcode bytes we want to use.

In actual practice, we have to select the next opcode byte from more than three candidates because 80x86 instructions take many different lengths. For example, an instruction that moves a 32-bit constant to a memory location can be ten or more bytes long. And there are instruction lengths for nearly every value between one and fifteen bytes. Also, some opcodes on the 80x86 are longer than one byte, so the CPU may have to fetch multiple bytes in order to properly decode the current instruction. Nevertheless, by throwing more hardware at the problem we can decode the current opcode at the same time we’re fetching the next.

4.8.2.2 Stalls in a Pipeline

Unfortunately, the scenario presented in the previous section is a little too simplistic. There are two drawbacks to that simple pipeline: bus contention among instructions and non-sequential program execution. Both problems may increase the average execution time of the instructions in the pipeline.

Bus contention occurs whenever an instruction needs to access some item in memory. For example, if a "mov(reg, mem);" instruction needs to store data in memory and a "mov(mem, reg);" instruction is reading data from memory, contention for the address and data bus may develop since the CPU will be trying to simultaneously fetch data and write data in memory.

One simplistic way to handle bus contention is through a *pipeline stall*. The CPU, when faced with contention for the bus, gives priority to the instruction furthest along in the pipeline. The CPU suspends fetching opcodes until the current instruction fetches (or stores) its operand. This causes the new instruction in the pipeline to take two cycles to execute rather than one (see Figure 4.7).

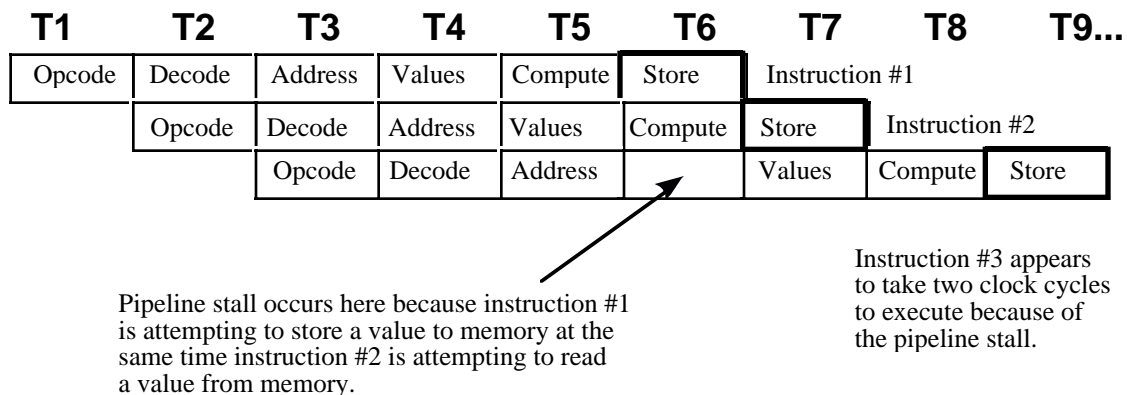


Figure 4.7 A Pipeline Stall

This example is but one case of bus contention. There are many others. For example, as noted earlier, fetching instruction operands requires access to the prefetch queue at the same time the CPU needs to fetch an opcode. Given the simple scheme above, it's unlikely that most instructions would execute at one clock per instruction (CPI).

Fortunately, the intelligent use of a cache system can eliminate many pipeline stalls like the ones discussed above. The next section on caching will describe how this is done. However, it is not always possible, even with a cache, to avoid stalling the pipeline. What you cannot fix in hardware, you can take care of with software. If you avoid using memory, you can reduce bus contention and your programs will execute faster. Likewise, using shorter instructions also reduces bus contention and the possibility of a pipeline stall.

What happens when an instruction *modifies* the EIP register? This, of course, implies that the next set of instructions to execute do not immediately follow the instruction that modifies EIP. By the time the instruction

```
JNZ Label;
```

completes execution (assuming the zero flag is clear so the branch is taken), we've already started five other instructions and we're only one clock cycle away from the completion of the first of these. Obviously, the CPU must not execute those instructions or it will compute improper results.

The only reasonable solution is to *flush* the entire pipeline and begin fetching opcodes anew. However, doing so causes a severe execution time penalty. It will take six clock cycles (the length of the pipeline in our examples) before the next instruction completes execution. Clearly, you should avoid the use of instructions which interrupt the sequential execution of a program. This also shows another problem – pipeline length. The longer the pipeline is, the more you can accomplish per cycle in the system. However, lengthening a pipeline may slow a program if it jumps around quite a bit. Unfortunately, you cannot control the number of stages in the pipeline¹⁶. You can, however, control the number of transfer instructions which appear in your programs. Obviously you should keep these to a minimum in a pipelined system.

4.8.3 Instruction Caches – Providing Multiple Paths to Memory

System designers can resolve many problems with bus contention through the intelligent use of the prefetch queue and the cache memory subsystem. They can design the prefetch queue to buffer up data from the instruction stream, and they can design the cache with separate data and code areas. Both techniques can improve system performance by eliminating some conflicts for the bus.

The prefetch queue simply acts as a buffer between the instruction stream in memory and the opcode fetching circuitry. The prefetch queue works well when the CPU isn't constantly accessing memory. When the CPU isn't accessing memory, the BIU can fetch additional instruction opcodes for the prefetch queue. Alas, the pipelined 80x86 CPUs are constantly accessing memory since they fetch an opcode byte on every clock cycle. Therefore, the prefetch queue cannot take advantage of any "dead" bus cycles to fetch additional opcode bytes – there aren't any "dead" bus cycles. However, the prefetch queue is still valuable for a very simple reason: the BIU fetches multiple bytes on each memory access and most instructions are shorter. Without the prefetch queue, the system would have to explicitly fetch each opcode, even if the BIU had already "accidentally" fetched the opcode along with the previous instruction. With the prefetch queue, however, the system will not refetch any opcodes. It fetches them once and saves them for use by the opcode fetch unit.

For example, if you execute two one-byte instructions in a row, the BIU can fetch both opcodes in one memory cycle, freeing up the bus for other operations. The CPU can use these available bus cycles to fetch additional opcodes or to deal with other memory accesses.

Of course, not all instructions are one byte long. The 80x86 has a large number of different instruction sizes. If you execute several large instructions in a row, you're going to run slower. Once again we return to that same rule: *the fastest programs are the ones which use the shortest instructions*. If you can use shorter instructions to accomplish some task, do so.

Suppose, for a moment, that the CPU has two separate memory spaces, one for instructions and one for data, each with their own bus. This is called the *Harvard Architecture* since the first such machine was built at Harvard. On a Harvard machine there would be no contention for the bus. The BIU could continue to fetch opcodes on the instruction bus while accessing memory on the data/memory bus (see Figure 4.8),

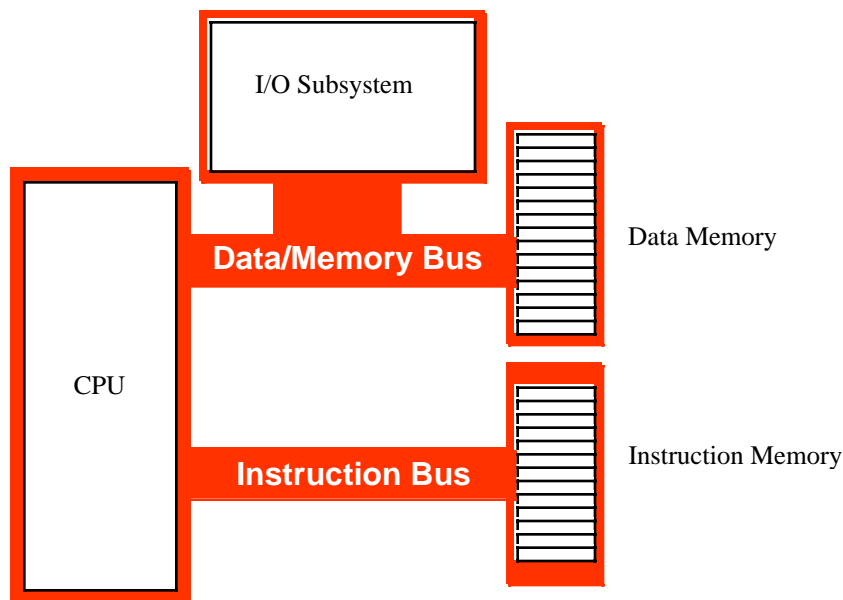


Figure 4.8 A Typical Harvard Machine

In the real world, there are very few true Harvard machines. The extra pins needed on the processor to support two physically separate busses increase the cost of the processor and introduce many other engineering problems. However, micropro-

16. Note, by the way, that the number of stages in an instruction pipeline varies among CPUs.

cessor designers have discovered that they can obtain many benefits of the Harvard architecture with few of the disadvantages by using separate on-chip caches for data and instructions. Advanced CPUs use an internal Harvard architecture and an external Von Neumann architecture. Figure 4.9 shows the structure of the 80x86 with separate data and instruction caches.

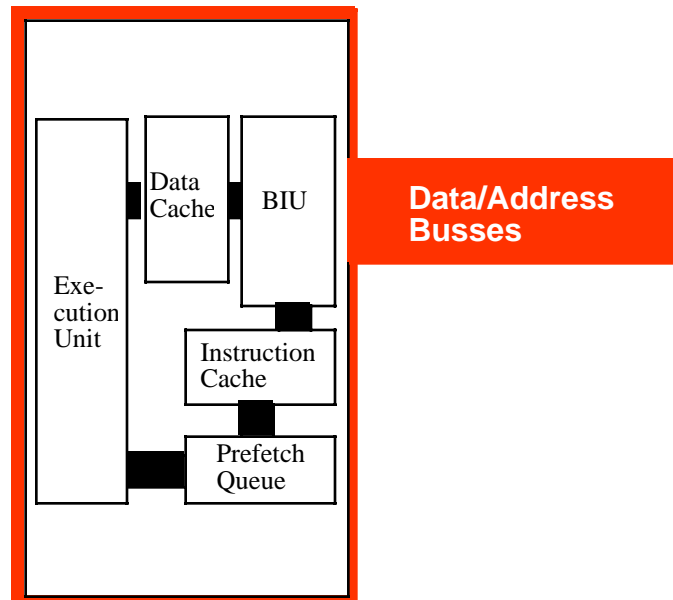


Figure 4.9 Using Separate Code and Data Caches

Each path inside the CPU represents an independent bus. Data can flow on all paths concurrently. This means that the prefetch queue can be pulling instruction opcodes from the instruction cache while the execution unit is writing data to the data cache. Now the BIU only fetches opcodes from memory whenever it cannot locate them in the instruction cache. Likewise, the data cache buffers memory. The CPU uses the data/address bus only when reading a value which is not in the cache or when flushing data back to main memory.

Although you cannot control the presence, size, or type of cache on a CPU, as an assembly language programmer you must be aware of how the cache operates to write the best programs. On-chip level one instruction caches are generally quite small (8,192 bytes on the 80486, for example). Therefore, the shorter your instructions, the more of them will fit in the cache (getting tired of “shorter instructions” yet?). The more instructions you have in the cache, the less often bus contention will occur. Likewise, using registers to hold temporary results places less strain on the data cache so it doesn’t need to flush data to memory or retrieve data from memory quite so often. *Use the registers wherever possible!*

4.8.4 Hazards

There is another problem with using a pipeline: the data hazard. Let’s look at the execution profile for the following instruction sequence:

```
mov( Somevar, ebx );  
mov( [ebx], eax );
```

When these two instructions execute, the pipeline will look something like shown in Figure 4.10:

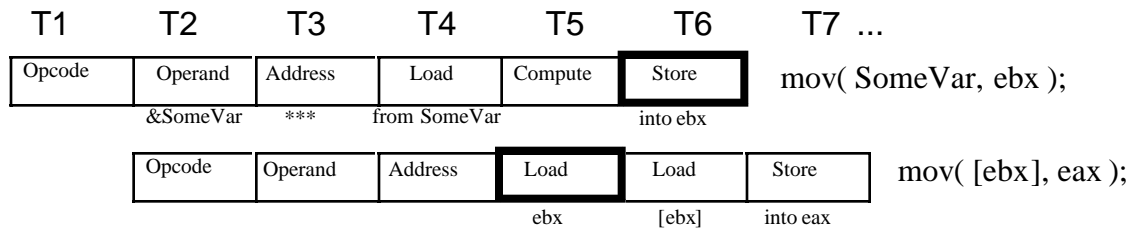


Figure 4.10 A Data Hazard

Note a major problem here. These two instructions fetch the 32 bit value whose address appears at location *&SomeVar* in memory. *But this sequence of instructions won't work properly!* Unfortunately, the second instruction has already used the value in EBX before the first instruction loads the contents of memory location *&SomeVar* (T4 & T6 in the diagram above).

CISC processors, like the 80x86, handle hazards automatically¹⁷. However, they will stall the pipeline to synchronize the two instructions. The actual execution would look something like shown in Figure 4.11.

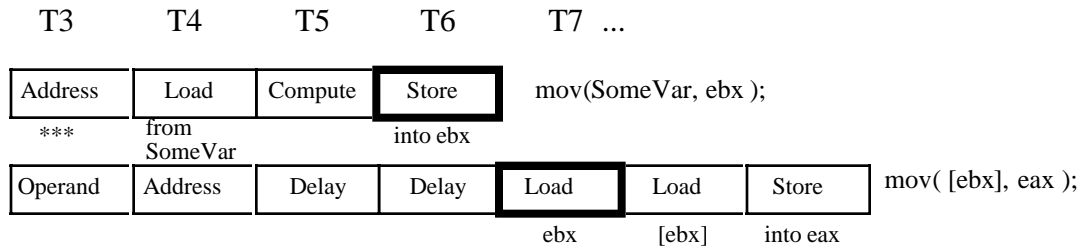


Figure 4.11 How the 80x86 Handles a Data Hazard

By delaying the second instruction two clock cycles, the CPU guarantees that the load instruction will load EAX from the proper address. Unfortunately, the second load instruction now executes in three clock cycles rather than one. However, requiring two extra clock cycles is better than producing incorrect results. Fortunately, you can reduce the impact of hazards on execution speed within your software.

Note that the data hazard occurs when the source operand of one instruction was a destination operand of a previous instruction. There is nothing wrong with loading EBX from *SomeVar* and then loading EAX from [EBX], *unless they occur one right after the other*. Suppose the code sequence had been:

```

mov( 2000, ecx );
mov( SomeVar, ebx );
mov( [ebx], eax );

```

We could reduce the effect of the hazard that exists in this code sequence by simply *rearranging the instructions*. Let's do that and obtain the following:

```

mov( SomeVar, ebx );
mov( 2000, ecx );
mov( [ebx], eax );

```

Now the "mov([ebx], eax);" instruction requires only one additional clock cycle rather than two. By inserting yet another instruction between the "mov(SomeVar, ebx);" and the "mov([ebx], eax);" instructions you can eliminate the effects of the hazard altogether¹⁸.

17. Some RISC chips do not. If you tried this sequence on certain RISC chips you would get an incorrect answer.

On a pipelined processor, the order of instructions in a program may dramatically affect the performance of that program. Always look for possible hazards in your instruction sequences. Eliminate them wherever possible by rearranging the instructions.

In addition to data hazards, there are also *control hazards*. We've actually discussed control hazards already, although we did not refer to them by that name. A control hazard occurs whenever the CPU branches to some new location in memory and the CPU has to flush the following instructions following the branch that are in various stages of execution.

4.8.5 Superscalar Operation— Executing Instructions in Parallel

With the pipelined architecture we could achieve, at best, execution times of one CPI (clock per instruction). Is it possible to execute instructions faster than this? At first glance you might think, "Of course not, we can do at most one operation per clock cycle. So there is no way we can execute more than one instruction per clock cycle." Keep in mind however, that a single instruction is *not* a single operation. In the examples presented earlier each instruction has taken between six and eight operations to complete. By adding seven or eight separate units to the CPU, we could effectively execute these eight operations in one clock cycle, yielding one CPI. If we add more hardware and execute, say, 16 operations at once, can we achieve 0.5 CPI? The answer is a qualified "yes." A CPU including this additional hardware is a *superscalar* CPU and can execute more than one instruction during a single clock cycle. The 80x86 family began supporting superscalar execution with the introduction of the Pentium processor.

A superscalar CPU has, essentially, several execution units (see Figure 4.12). If it encounters two or more instructions in the instruction stream (i.e., the prefetch queue) which can execute independently, it will do so.

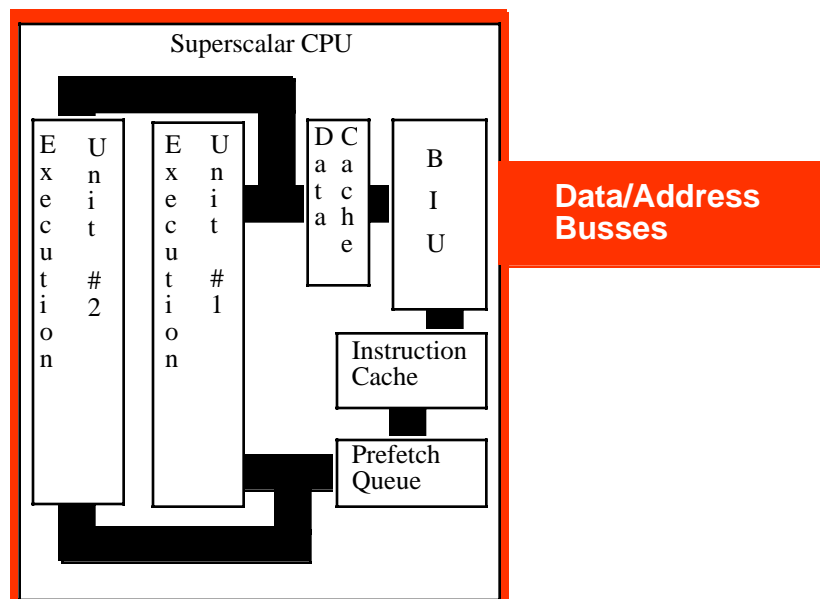


Figure 4.12 A CPU that Supports Superscalar Operation

There are a couple of advantages to going superscalar. Suppose you have the following instructions in the instruction stream:

```
mov( 1000, eax );  
mov( 2000, ebx );
```

18. Of course, any instruction you insert at this point must *not* modify the values in the `eax` and `ebx` registers. Also note that the examples in this section are contrived to demonstrate pipeline stalls. Actual 80x86 CPUs have additional circuitry to help reduce the effect of pipeline stalls on the execution time.

If there are no other problems or hazards in the surrounding code, and all six bytes for these two instructions are currently in the prefetch queue, there is no reason why the CPU cannot fetch and execute both instructions in parallel. All it takes is extra silicon on the CPU chip to implement two execution units.

Besides speeding up independent instructions, a superscalar CPU can also speed up program sequences that have hazards. One limitation of superscalar CPU is that once a hazard occurs, the offending instruction will completely stall the pipeline. Every instruction which follows will also have to wait for the CPU to synchronize the execution of the instructions. With a superscalar CPU, however, instructions following the hazard may continue execution through the pipeline as long as they don't have hazards of their own. This alleviates (though does not eliminate) some of the need for careful instruction scheduling.

As an assembly language programmer, the way you write software for a superscalar CPU can dramatically affect its performance. First and foremost is that rule you're probably sick of by now: *use short instructions*. The shorter your instructions are, the more instructions the CPU can fetch in a single operation and, therefore, the more likely the CPU will execute faster than one CPI. Most superscalar CPUs do not completely duplicate the execution unit. There might be multiple ALUs, floating point units, etc. This means that certain instruction sequences can execute very quickly while others won't. You have to study the exact composition of your CPU to decide which instruction sequences produce the best performance.

4.8.6 Out of Order Execution

In a standard superscalar CPU it is the programmer's (or compiler's) responsibility to schedule (arrange) the instructions to avoid hazards and pipeline stalls. Fancier CPUs can actually remove some of this burden and improve performance by automatically rescheduling instructions while the program executes. To understand how this is possible, consider the following instruction sequence:

```
mov( SomeVar, ebx );  
mov( [ebx], eax );  
mov( 2000, ecx );
```

A data hazard exists between the first and second instructions above. The second instruction must delay until the first instruction completes execution. This introduces a pipeline stall and increases the running time of the program. Typically, the stall affects every instruction that follows. However, note that the third instruction's execution does not depend on the result from either of the first two instructions. Therefore, there is no reason to stall the execution of the "mov(2000, ecx);" instruction. It may continue executing while the second instruction waits for the first to complete. This technique, appearing in later members of the Pentium line, is called "out of order execution" because the CPU completes the execution of some instruction prior to the execution of previous instructions appearing in the code stream.

Clearly, the CPU may only execute instruction out of sequence if doing so produces exactly the same results as in-order execution. While there are a lot of little technical issues that make this problem a little more difficult than it seems, with enough engineering effort it is quite possible to implement this feature.

Although you might think that this extra effort is not worth it (why not make it the programmer's or compiler's responsibility to schedule the instructions) there are some situations where out of order execution will improve performance that static scheduling could not handle.

4.8.7 Register Renaming

One problem that hampers the effectiveness of superscalar operation on the 80x86 CPU is the 80x86's limited number of general purpose registers. Suppose, for example, that the CPU had four different pipelines and, therefore, was capable of executing four instructions simultaneously. Actually achieving four instructions per clock cycle would be very difficult because most instructions (that can execute simultaneously with other instructions) operate on two register operands. For four instructions to execute concurrently, you'd need four separate destination registers and four source registers (and the two sets of registers must be disjoint, that is, a destination register for one instruction cannot be the source of another). CPUs that have lots of registers can handle this task quite easily, but the limited register set of the 80x86 makes this difficult. Fortunately, there is a way to alleviate part of the problem: through *register renaming*.

Register renaming is a sneaky way to give a CPU more registers than it actually has. Programmers will not have direct access to these extra registers, but the CPU can use these additional register to prevent hazards in certain cases. For example, consider the following short instruction sequence:

```
mov( 0, eax );
mov( eax, i );
mov( 50, eax );
mov( eax, j );
```

Clearly a data hazard exists between the first and second instructions and, likewise, a data hazard exists between the third and fourth instructions in this sequence. Out of order execution in a superscalar CPU would normally allow the first and third instructions to execute concurrently and then the second and fourth instructions could also execute concurrently. However, a data hazard, of sorts, also exists between the first and third instructions since they use the same register. The programmer could have easily solved this problem by using a different register (say EBX) for the third and fourth instructions. However, let's assume that the programmer was unable to do this because the other registers are all holding important values. Is this sequence doomed to executing in four cycles on a superscalar CPU that should only require two?

One advanced trick a CPU can employ is to create a bank of registers for each of the general purpose registers on the CPU. That is, rather than having a single EAX register, the CPU could support an array of EAX registers; let's call these registers EAX[0], EAX[1], EAX[2], etc. Similarly, you could have an array of each of the registers, so we could also have EBX[0]..EBX[n], ECX[0]..ECX[n], etc. Now the instruction set does not give the programmer the ability to select one of these specific register array elements for a given instruction, but the CPU can automatically choose a different register array element if doing so would not change the overall computation and doing so could speed up the execution of the program. For example, consider the following sequence (with register array elements automatically chosen by the CPU):

```
mov( 0, eax[0] );
mov( eax[0], i );
mov( 50, eax[1] );
mov( eax[1], j );
```

Since EAX[0] and EAX[1] are different registers, the CPU can execute the first and third instructions concurrently. Likewise, the CPU can execute the second and fourth instructions concurrently.

The code above provides an example of *register renaming*. Dynamically, the CPU automatically selects one of several different elements from a register array in order to prevent data hazards. Although this is a simple example, and different CPUs implement register renaming in many different ways, this example does demonstrate how the CPU can improve performance in certain instances through the use of this technique.

4.8.8 Very Long Instruction Word Architecture (VLIW)

Superscalar operation attempts to schedule, in hardware, the execution of multiple instructions simultaneously. Another technique that Intel is using in their IA-64 architecture is the use of very long instruction words, or VLIW. In a VLIW computer system, the CPU fetches a large block of bytes (41 in the case of the IA-64 Itanium CPU) and decodes and executes this block all at once. This block of bytes usually contains two or more instructions (three in the case of the IA-64). VLIW computing requires the programmer or compiler to properly schedule the instructions in each block (so there are no hazards or other conflicts), but if properly scheduled, the CPU can execute three or more instructions per clock cycle.

The Intel IA-64 Architecture is not the only computer system to employ a VLIW architecture. Transmeta's Crusoe processor family also uses a VLIW architecture. The Crusoe processor is different than the IA-64 architecture insofar as it does not support native execution of IA-32 instructions. Instead, the Crusoe processor dynamically translates 80x86 instructions to Crusoe's VLIW instructions. This "code morphing" technology results in code running about 50% slower than native code, though the Crusoe processor has other advantages.

We will not consider VLIW computing any further since the IA-32 architecture does not support it. But keep this architectural advance in mind if you move towards the IA-64 family or the Crusoe family.

4.8.9 Parallel Processing

Most of the techniques for improving CPU performance via architectural advances involve the parallel (overlapped) execution of instructions. Most of the techniques of this chapter are transparent to the programmer. That is, the programmer does not have to do anything special to take minimal advantage of the parallel operation of pipelines and superscalar operations. True, if programmers are aware of the underlying architecture they can write code that runs even faster, but these architectural advances often improve performance even if programmers do not write special code to take advantage of them.

The only problem with this approach (attempting to dynamically parallelize an inherently sequential program) is that there is only so much you can do to parallelize a program that requires sequential execution for proper operation (which covers most programs). To truly produce a parallel program, the programmer must specifically write parallel code; of course, this does require architectural support from the CPU. This section and the next touches on the types of support a CPU can provide.

Typical CPUs use what is known as the SISD model: *Single Instruction, Single Data*. This means that the CPU executes one instruction at a time that operates on a single piece of data¹⁹. Two common parallel models are the so-called SIMD (*Single Instruction, Multiple Data*) and MIMD (*Multiple Instruction, Multiple Data*) models. As it turns out, x86 systems can support both of these parallel execution models.

In the SIMD model, the CPU executes a single instruction stream, just like the standard SISD model. However, the CPU executes the specified operation on multiple pieces of data concurrently rather than a single data object. For example, consider the 80x86 ADD instruction. This is a SISD instruction that operates on (that is, produces) a single piece of data; true, the instruction fetches values from two source operands and stores a sum into a destination operand but the end result is that the ADD instruction will only produce a single sum. An SIMD version of ADD, on the other hand, would compute the sum of several values simultaneously. The Pentium III's MMX and SIMD instruction extensions operate in exactly this fashion. With an MMX instruction, for example, you can add up to eight separate pairs of values with the execution of a single instruction. The aptly named SIMD instruction extensions operate in a similar fashion.

Note that SIMD instructions are only useful in specialized situations. Unless you have an algorithm that can take advantage of SIMD instructions, they're not that useful. Fortunately, high-speed 3-D graphics and multimedia applications benefit greatly from these SIMD (and MMX) instructions, so their inclusion in the 80x86 CPU offers a huge performance boost for these important applications.

The MIMD model uses multiple instructions, operating on multiple pieces of data (usually one instruction per data object, though one of these instructions could also operate on multiple data items). These multiple instructions execute independently of one another. Therefore, it's very rare that a single program (or, more specifically, a single thread of execution) would use the MIMD model. However, if you have a multiprogramming environment with multiple programs attempting to execute concurrently in memory, the MIMD model does allow each of those programs to execute their own code stream concurrently. This type of parallel system is usually called a multiprocessor system. Multiprocessor systems are the subject of the next section.

The common computation models are SISD, SIMD, and MIMD. If you're wondering if there is a MISD model (Multiple Instruction, Single Data) the answer is no. Such an architecture doesn't really make sense.

4.8.10 Multiprocessing

Pipelining, superscalar operation, out of order execution, and VLIW design are techniques CPU designers use in order to execute several operations in parallel. These techniques support *fine-grained parallelism*²⁰ and are useful for speeding up adjacent instructions in a computer system. If adding more functional units increases parallelism (and, therefore, speeds up the system), you might wonder what would happen if you added two CPUs to the system. This technique, known as *multiprocessing*, can improve system performance, though not as uniformly as other techniques. As noted in the previous section, a multiprocessor system uses the MIMD parallel execution model.

The techniques we've considered to this point don't require special programming to realize a performance increase. True, if you do pay attention you will get better performance; but no special programming is necessary to activate these features. Multiprocessing, on the other hand, doesn't help a program one bit unless that program was specifically written to use multi-

19. We will ignore the parallelism provided by pipelining and superscalar operation in this discussion.

20. For our purposes, fine-grained parallelism means that we are executing adjacent program instructions in parallel.

processor (or runs under an O/S specifically written to support multiprocessing). If you build a system with two CPUs, those CPUs cannot trade off executing alternate instructions within a program. In fact, it is very expensive (timewise) to switch the execution of a program from one processor to another. Therefore, multiprocessor systems are really only effective in a system that execute multiple programs concurrently (i.e., a multitasking system)²¹. To differentiate this type of parallelism from that afforded by pipelining and superscalar operation, we'll call this kind of parallelism *coarse-grained parallelism*.

Adding multiple processors to a system is not as simple as wiring the processor to the motherboard. A big problem with multiple processors is the *cache coherency* problem. To understand this problem, consider two separate programs running on separate processors in a multiprocessor system. Suppose also that these two processor communicate with one another by writing to a block of shared physical memory. Unfortunately, when CPU #1 writes to this block of addresses the CPU caches the data up and might not actually write the data to physical memory for some time. Simultaneously, CPU #2 might be attempting to read this block of shared memory but winds up reading the data out of its local cache rather than the data that CPU #1 wrote to the block of shared memory (assuming the data made it out of CPU #1's local cache). In order for these two functions to operate properly, the two CPU's must communicate writes to common memory addresses in cache between themselves. This is a very complex and involved process.

Currently, the Pentium III and IV processors directly support cache updates between two CPUs in a system. Intel also builds a more expensive processor, the XEON, that supports more than two CPUs in a system. However, one area where the RISC CPUs have a big advantage over Intel is in the support for multiple processors in a system. While Intel systems reach a point of diminishing returns at about 16 processors, Sun SPARC and other RISC processors easily support 64-CPU systems (with more arriving, it seems, every day). This is why large databases and large web server systems tend to use expensive UNIX-based RISC systems rather than x86 systems.

4.9 Putting It All Together

The performance of modern CPUs is intrinsically tied to the architecture of that CPU. Over the past half century there have been many major advances in CPU design that have dramatically improved performance. Although the clock frequency has improved by over four orders of magnitude during this time period, other improvements have added one or two orders of magnitude improvement as well. Over the 80x86's lifetime, performance has improved 10,000-fold.

Unfortunately, the 80x86 family has just about pushed the limits of what it can achieve by extending the architecture. Perhaps another order of magnitude is possible, but Intel is reaching the point of diminishing returns. Having realized this, Intel has chosen to implement a new architecture using VLIW for their IA-64 family. Only time will prove whether their approach is the correct one, but most people believe that the IA-32 has reached the end of its lifetime. On the other hand, people have been announcing the death of the IA-32 for the past decade, so we'll see what happens now.

21. Technically, it only needs to execute multiple threads concurrently, but we'll ignore this distinction here since the technical distinction between threads and programs/processes is beyond the scope of this chapter.

5.1 Chapter Overview

This chapter discusses the low-level implementation of the 80x86 instruction set. It describes how the Intel engineers decided to encode the instructions in a numeric format (suitable for storage in memory) and it discusses the trade-offs they had to make when designing the CPU. This chapter also presents a historical background of the design effort so you can better understand the compromises they had to make.

5.2 The Importance of the Design of the Instruction Set

In this chapter we will be exploring one of the most interesting and important aspects of CPU design: the design of the CPU's instruction set. The instruction set architecture (or ISA) is one of the most important design issues that a CPU designer must get right from the start. Features like caches, pipelining, superscalar implementation, etc., can all be grafted on to a CPU design long after the original design is obsolete. However, it is very difficult to change the instructions a CPU executes once the CPU is in production and people are writing software that uses those instructions. Therefore, one must carefully choose the instructions for a CPU.

You might be tempted to take the "kitchen sink" approach to instruction set design¹ and include as many instructions as you can dream up in your instruction set. This approach fails for several reasons we'll discuss in the following paragraphs. Instruction set design is the epitome of compromise management. Good CPU design is the process of selecting what to throw out rather than what to leave in. It's easy enough to say "let's include everything." The hard part is deciding what to leave out once you realize you can't put everything on the chip.

Nasty reality #1: Silicon real estate. The first problem with "putting it all on the chip" is that each feature requires some number of transistors on the CPU's silicon die. CPU designers work with a "silicon budget" and are given a finite number of transistors to work with. This means that there aren't enough transistors to support "putting all the features" on a CPU. The original 8086 processor, for example, had a transistor budget of less than 30,000 transistors. The Pentium III processor had a budget of over eight million transistors. These two budgets reflect the differences in semiconductor technology in 1978 vs. 1998.

Nasty reality #2: Cost. Although it is possible to use millions of transistors on a CPU today, the more transistors you use the more expensive the CPU. Pentium IV processors, for example, cost hundreds of dollars (circa 2002). A CPU with only 30,000 transistors (also circa 2002) would cost only a few dollars. For low-cost systems it may be more important to shave some features and use fewer transistors, thus lowering the CPU's cost.

Nasty reality #3: Expandability. One problem with the "kitchen sink" approach is that it's very difficult to anticipate all the features people will want. For example, Intel's MMX and SIMD instruction enhancements were added to make multimedia programming more practical on the Pentium processor. Back in 1978 very few people could have possibly anticipated the need for these instructions.

Nasty reality #4: Legacy Support. This is almost the opposite of expandability. Often it is the case that an instruction the CPU designer feels is important turns out to be less useful than anticipated. For example, the LOOP instruction on the 80x86 CPU sees very little use in modern high-performance programs. The 80x86 ENTER instruction is another good example. When designing a CPU using the "kitchen sink" approach, it is often common to discover that programs almost never use some of the available instructions. Unfortunately, you cannot easily remove instructions in later versions of a processor because this will break some existing programs

1. As in "Everything, including the kitchen sink."

that use those instructions. Generally, once you add an instruction you have to support it forever in the instruction set. Unless very few programs use the instruction (and you're willing to let them break) or you can automatically simulate the instruction in software, removing instructions is a very difficult thing to do.

Nasty reality #4: Complexity. The popularity of a new processor is easily measured by how much software people write for that processor. Most CPU designs die a quick death because no one writes software specific to that CPU. Therefore, a CPU designer must consider the assembly programmers and compiler writers who will be using the chip upon introduction. While a "kitchen sink" approach might seem to appeal to such programmers, the truth is no one wants to learn an overly complex system. If your CPU does everything under the sun, this might appeal to someone who is already familiar with the CPU. However, pity the poor soul who doesn't know the chip and has to learn it all at once.

These problems with the "kitchen sink" approach all have a common solution: design a simple instruction set to begin with and leave room for later expansion. This is one of the main reasons the 80x86 has proven to be so popular and long-lived. Intel started with a relatively simple CPU and figured out how to extend the instruction set over the years to accommodate new features.

5.3 Basic Instruction Design Goals

In a typical Von Neumann architecture CPU, the computer encodes CPU instructions as numeric values and stores these numeric values in memory. The encoding of these instructions is one of the major tasks in instruction set design and requires very careful thought.

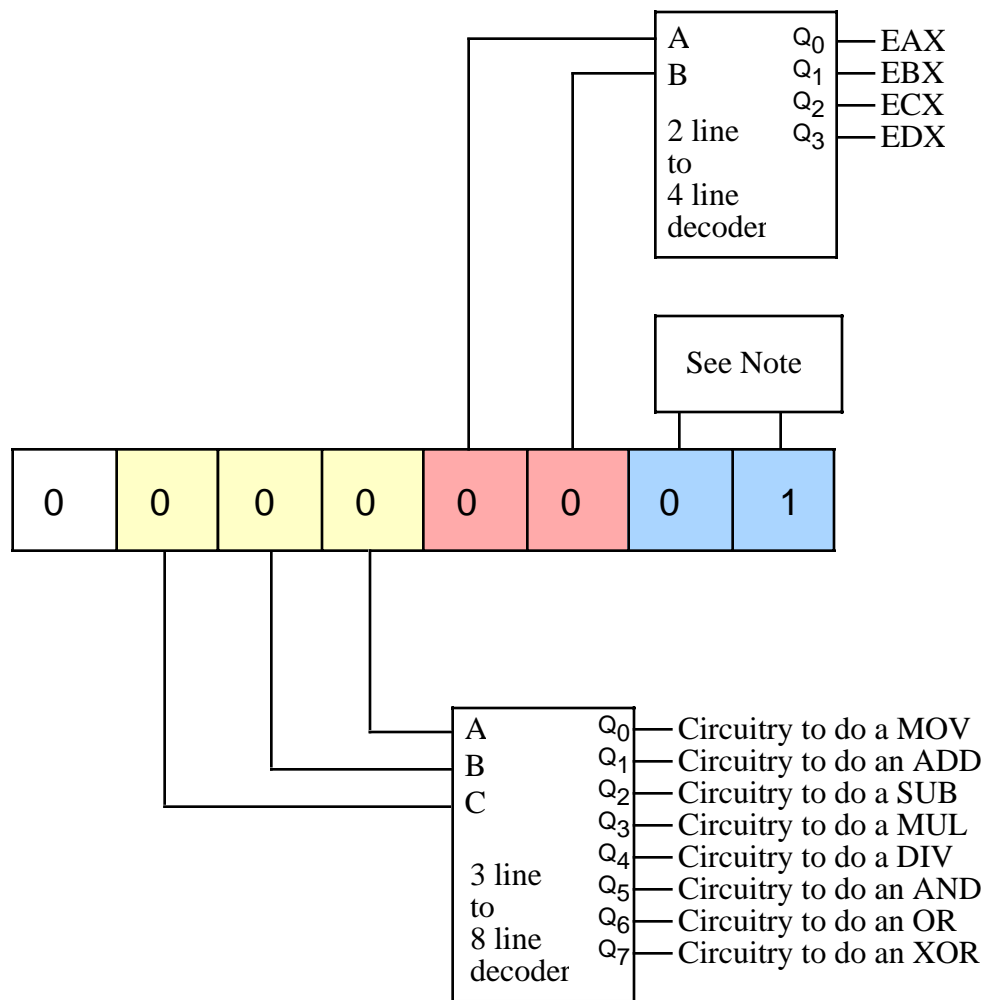
To encode an instruction we must pick a unique numeric opcode value for each instruction (clearly, two different instructions cannot share the same numeric value or the CPU will not be able to differentiate them when it attempts to decode the opcode value). With an n -bit number, there are 2^n different possible opcodes, so to encode m instructions you will need an opcode that is at least $\log_2(m)$ bits long.

Encoding opcodes is a little more involved than assigning a unique numeric value to each instruction. Remember, we have to use actual hardware (i.e., decoder circuits) to figure out what each instruction does and command the rest of the hardware to do the specified task. Suppose you have a seven-bit opcode. With an opcode of this size we could encode 128 different instructions. To decode each instruction individually requires a seven-line to 128-line decoder – an expensive piece of circuitry. Assuming our instructions contain certain patterns, we can reduce the hardware by replacing this large decoder with three smaller decoders.

If you have 128 truly unique instructions, there's little you can do other than to decode each instruction individually. However, in most architectures the instructions are not completely independent of one another. For example, on the 80x86 CPUs the opcodes for "mov(eax, ebx);" and "mov(ecx, edx);" are different (because these are different instructions) but these instructions are not unrelated. They both move data from one register to another. In fact, the only difference between them is the source and destination operands. This suggests that we could encode instructions like MOV with a sub-opcode and encode the operands using other strings of bits within the opcode.

For example, if we really have only eight instructions, each instruction has two operands, and each operand can be one of four different values, then we can encode the opcode as three packed fields containing three, two, and two bits (see Figure 5.1). This encoding only requires the use of three simple decoders to completely determine what instruction the CPU should execute. While this is a bit of a trivial case, it does demonstrate one very important facet of instruction set design – it is important to make opcodes easy to decode and the easiest way to do this is to break up the opcode into several different bit fields, each field contributing part of the information necessary to execute the full instruction. The smaller these bit fields, the easier it will be for the hardware to decode and execute them².

2. Not to mention faster and less expensive.



Note: the circuitry attached to the destination register bits is identical to the circuitry for the source register bits.

Figure 5.1 Separating an Opcode into Separate Fields to Ease Decoding

Although Intel probably went a little overboard with the design of the original 8086 instruction set, an important design goal is to keep instruction sizes within a reasonable range. CPUs with unnecessarily long instructions consume extra memory for their programs. This tends to create more cache misses and, therefore, hurts the overall performance of the CPU. Therefore, we would like our instructions to be as compact as possible so our programs code uses as little memory as possible.

It would seem that if we are encoding 2^n different instructions using n bits, there would be very little leeway in choosing the size of the instruction. It's going to take n bits to encode those 2^n instructions, you can't do it with any fewer. You may, of course, use more than n bits; and believe it or not, that's the secret to reducing the size of a typical program on the CPU.

Before discussing how to use longer instructions to generate shorter programs, a short digression is necessary. The first thing to note is that we generally cannot choose an arbitrary number of bits for our opcode length. Assuming that our CPU is capable of reading bytes from memory, the opcode will probably have to be some

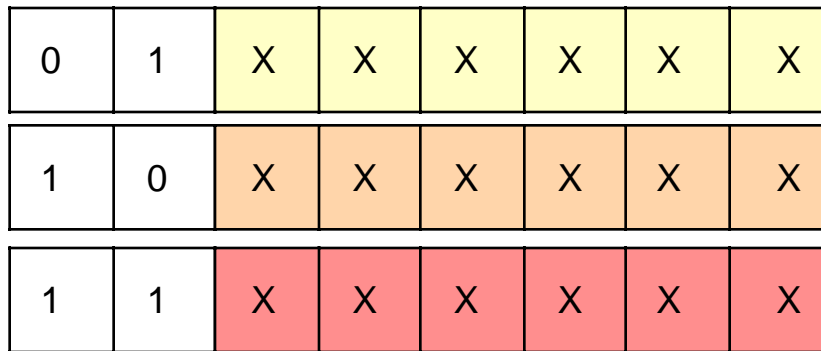
even multiple of eight bits long. If the CPU is not capable of reading bytes from memory (e.g., most RISC CPUs only read memory in 32 or 64 bit chunks) then the opcode is going to be the same size as the smallest object the CPU can read from memory at one time (e.g., 32 bits on a typical RISC chip). Any attempt to shrink the opcode size below this data bus enforced lower limit is futile. Since we're discussing the 80x86 architecture in this text, we'll work with opcodes that must be an even multiple of eight bits long.

Another point to consider here is the size of an instruction's operands. Some CPU designers (specifically, RISC designers) include all operands in their opcode. Other CPU designers (typically CISC designers) do not count operands like immediate constants or address displacements as part of the opcode (though they do usually count register operand encodings as part of the opcode). We will take the CISC approach here and not count immediate constant or address displacement values as part of the actual opcode.

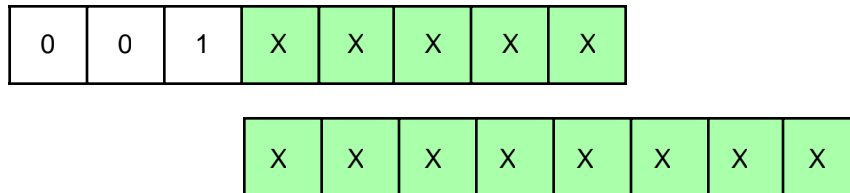
With an eight-bit opcode you can only encode 256 different instructions. Even if we don't count the instruction's operands as part of the opcode, having only 256 different instructions is somewhat limiting. It's not that you can't build a CPU with an eight-bit opcode, most of the eight-bit processors predating the 8086 had eight-bit opcodes, it's just that modern processors tend to have far more than 256 different instructions. The next step up is a two-byte opcode. With a two-byte opcode we can have up to 65,536 different instructions (which is probably enough) but our instructions have doubled in size (not counting the operands, of course).

If reducing the instruction size is an important design goal³ we can employ some techniques from data compression theory to reduce the average size of our instructions. The basic idea is this: first we analyze programs written for our CPU (or a CPU similar to ours if no one has written any programs for our CPU) and count the number of occurrences of each opcode in a large number of typical applications. We then create a sorted list of these opcodes from most-frequently-used to least-frequently-used. Then we attempt to design our instruction set using one-byte opcodes for the most-frequently-used instructions, two-byte opcodes for the next set of most-frequently-used instructions, and three (or more) byte opcodes for the rarely used instructions. Although our maximum instruction size is now three or more bytes, most of the actual instructions appearing in a program will use one or two byte opcodes, so the average opcode length will be somewhere between one and two bytes (let's call it 1.5 bytes) and a typical program will be shorter than had we chosen a two byte opcode for all instructions (see Figure 5.2).

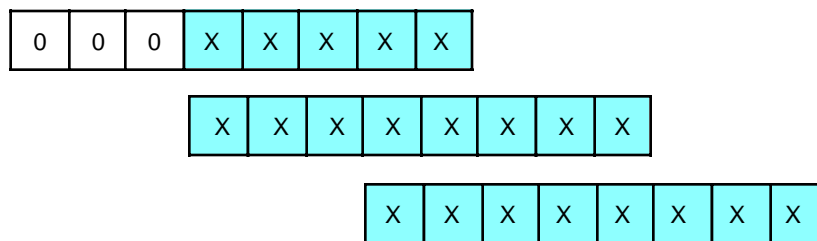
3. To many CPU designers it is not; however, since this was a design goal for the 8086 we'll follow this path.



If the H.O. two bits of the first opcode byte are not both zero, then the whole opcode is one byte long and the remaining six bits let us encode 64 one-byte instructions. Since there are a total of three opcode bytes of these form, we can encode up to 192 different one-byte instructions.



If the H.O. three bits of our first opcode byte contain %001, then the opcode is two bytes long and the remaining 13 bits let us encode 8192 different instructions.



If the H.O. three bits of our first opcode byte contain all zeros, then the opcode is three bytes long and the remaining 21 bits let us encode two million (2^{21}) different instructions.

Figure 5.2 Encoding Instructions Using a Variable-Length Opcode

Although using variable-length instructions allows us to create smaller programs, it comes at a price. First of all, decoding the instructions is a bit more complicated. Before decoding an instruction field, the CPU must first decode the instruction's size. This extra step consumes time and may affect the overall performance of the CPU (by introducing delays in the decoding step and, thereby, limiting the maximum clock frequency of the CPU).

Another problem with variable length instructions is that it makes decoding multiple instructions in a pipeline quite difficult (since we cannot trivially determine the instruction boundaries in the prefetch queue). These reasons, along with some others, is why most popular RISC architectures avoid variable-sized instructions. However, for our purpose, we'll go with a variable length approach since saving memory is an admirable goal.

Before actually choosing the instructions you want to implement in your CPU, now would be a good time to plan for the future. Undoubtedly, you will discover the need for new instructions at some point in the future, so reserving some opcodes specifically for that purpose is a real good idea. If you were using the instruction encoding appearing in Figure 5.2 for your opcode format, it might not be a bad idea to reserve one block of 64 one-byte opcodes, half (4,096) of the two-byte instructions, and half (1,048,576) of the three-byte opcodes for future use. In particular, giving up 64 of the very valuable one-byte opcodes may seem extravagant, but history suggests that such foresight is rewarded.

The next step is to choose the instructions you want to implement. Note that although we've reserved nearly half the instructions for future expansion, we don't actually have to implement instructions for all the remaining opcodes. We can choose to leave a good number of these instructions unimplemented (and effectively reserve them for the future as well). The right approach is not to see how quickly we can use up all the opcodes, but rather to ensure that we have a consistent and complete instruction set given the compromises we have to live with (e.g., silicon limitations). The main point to keep in mind here is that it's much easier to add an instruction later than it is to remove an instruction later. So for the first go-around, it's generally better to go with a simpler design rather than a more complex design.

The first step is to choose some generic instruction types. For a first attempt, you should limit the instructions to some well-known and common instructions. The best place to look for help in choosing these instructions is the instruction sets of other processors. For example, most processors you find will have instructions like the following:

- Data movement instructions (e.g., MOV)

- Arithmetic and logical instructions (e.g., ADD, SUB, AND, OR, NOT)

- Comparison instructions

- A set of conditional jump instructions (generally used after the compare instructions)

- Input/Output instructions

- Other miscellaneous instructions

Your goal as the designer of the CPU's initial instruction set is to choose a reasonable set of instructions that will allow programmers to efficiently write programs (using as few instructions as possible) without adding so many instructions you exceed your silicon budget or violate other system compromises. This is a very strategic decision, one that CPU designers should base on careful research, experimentation, and simulation. The job of the CPU designer is not to create the best instruction set, but to create an instruction set that is optimal given all the constraints.

Once you've decided which instructions you want to include in your (initial) instruction set, the next step is to assign opcodes for them. The first step is to group your instructions into sets by common characteristics of those instructions. For example, an ADD instruction is probably going to support the exact same set of operands as the SUB instruction. So it makes sense to put these two instructions into the same group. On the other hand, the NOT instruction generally requires only a single operand⁴ as does a NEG instruction. So you'd probably put these two instructions in the same group but a different group than ADD and SUB.

4. Assuming this operation treats its single operand as both a source and destination operand, a common way of handling this instruction.

Once you've grouped all your instructions, the next step is to encode them. A typical encoding will use some bits to select the group the instruction falls into, it will use some bits to select a particular instruction from that group, and it will use some bits to determine the types of operands the instruction allows (e.g., registers, memory locations, and constants). The number of bits needed to encode all this information may have a direct impact on the instruction's size, regardless of the frequency of the instruction. For example, if you need two bits to select a group, four bits to select an instruction within that group, and six bits to specify the instruction's operand types, you're not going to fit this instruction into an eight-bit opcode. On the other hand, if all you really want to do is push one of eight different registers onto the stack, you can use four bits to select the PUSH instruction and three bits to select the register (assuming the encoding in Figure 5.2 the eighth and H.O. bit would have to contain zero).

Encoding operands is always a problem because many instructions allow a large number of operands. For example, the generic 80x86 MOV instruction requires a two-byte opcode⁵. However, Intel noticed that the "mov(disp, eax);" and "mov(eax, disp);" instructions occurred very frequently. So they created a special one-byte version of this instruction to reduce its size and, therefore, the size of those programs that use this instruction frequently. Note that Intel did not remove the two-byte versions of these instructions. They have two different instructions that will store EAX into memory or load EAX from memory. A compiler or assembler would always emit the shorter of the two instructions when given an option of two or more instructions that wind up doing exactly the same thing.

Notice an important trade-off Intel made with the MOV instruction. They gave up an extra opcode in order to provide a shorter version of one of the MOV instructions. Actually, Intel used this trick all over the place to create shorter and easier to decode instructions. Back in 1978 this was a good compromise (reducing the total number of possible instructions while also reducing the program size). Today, a CPU designer would probably want to use those redundant opcodes for a different purpose, however, Intel's decision was reasonable at the time (given the high cost of memory in 1978).

To further this discussion, we need to work with an example. So the next section will go through the process of designing a very simple instruction set as a means of demonstrating this process.

The Y86 Hypothetical Processor

Because of enhancements made to the 80x86 processor family over the years, Intel's design goals in 1978, and advances in computer architecture occurring over the years, the encoding of 80x86 instructions is very complex and somewhat illogical. Therefore, the 80x86 is not a good candidate for an example architecture when discussing how to design and encode an instruction set. However, since this is a text about 80x86 assembly language programming, attempting to present the encoding for some simpler real-world processor doesn't make sense. Therefore, we will discuss instruction set design in two stages: first, we will develop a simple (trivial) instruction set for a hypothetical processor that is a small subset of the 80x86, then we will expand our discussion to the full 80x86 instruction set. Our hypothetical processor is not a true 80x86 CPU, so we will call it the Y86 processor to avoid any accidental association with the Intel x86 family.

The Y86 processor is a *very* stripped down version of the x86 CPUs. First of all, the Y86 only supports one operand size — 16 bits. This simplification frees us from having to encode the size of the operand as part of the opcode (thereby reducing the total number of opcodes we will need). Another simplification is that the Y86 processor only supports four 16-bit registers: AX, BX, CX, and DX. This lets us encode register operands with only two bits (versus the three bits the 80x86 family requires to encode eight registers). Finally, the Y86 processors only support a 16-bit address bus with a maximum of 65,536 bytes of addressable memory. These simplifications, plus a very limited instruction set will allow us to encode all Y86 instructions using a single byte opcode and a two-byte displacement/offset (if needed).

5. Actually, Intel claims it's a one-byte opcode plus a one-byte "mod-reg-r/m" byte. For our purposes, we'll treat the mod-reg-r/m byte as part of the opcode.

The Y86 CPU provides 20 instructions. Seven of these instructions have two operands, eight of these instructions have a single operand, and five instructions have no operands at all. The instructions are MOV (two forms), ADD, SUB, CMP, AND, OR, NOT, JE, JNE, JB, JBE, JA, JAE, JMP, BRK, IRET, HALT, GET, and PUT. The following paragraphs describe how each of these work.

The MOV instruction is actually two instruction classes merged into the same instruction. The two forms of the mov instruction take the following forms:

```
mov( reg/memory/constant, reg );
mov( reg, memory );
```

where *reg* is any of AX, BX, CX, or DX; *constant* is a numeric constant (using hexadecimal notation), and *memory* is an operand specifying a memory location. The next section describes the possible forms the memory operand can take. The *reg/memory/constant* operand tells you that this particular operand may be a register, memory location, or a constant.

The arithmetic and logical instructions take the following forms:

```
add( reg/memory/constant, reg );
sub( reg/memory/constant, reg );
cmp( reg/memory/constant, reg );
and( reg/memory/constant, reg );
or( reg/memory/constant, reg );

not( reg/memory );
```

Note: the NOT instruction appears separately because it is in a different class than the other arithmetic instructions (since it supports only a single operand).

The ADD instruction adds the value of the first operand to the second (register) operand, leaving the sum in the second (register) operand. The SUB instruction subtracts the value of the first operand from the second, leaving the difference in the second operand. The CMP instruction compares the first operand against the second and saves the result of this comparison for use with one of the conditional jump instructions (described in a moment). The AND and OR instructions compute the corresponding bitwise logical operation on the two operands and store the result into the first operand. The NOT instruction inverts the bits in the single memory or register operand.

The *control transfer instructions* interrupt the sequential execution of instructions in memory and transfer control to some other point in memory either unconditionally, or after testing the result of the previous CMP instruction. These instructions include the following:

```
ja  dest; -- Jump if above (i.e., greater than)
jae dest; -- Jump if above or equal (i.e., greater than or equal)
jb  dest; -- Jump if below (i.e., less than)
jbe dest; -- Jump if below or equal (i.e., less than or equal)
je  dest; -- Jump if equal
jne dest; -- Jump if not equal
```

`jmp dest; -- Unconditional jump`

`iret; -- Return from an interrupt`

The first six instructions let you check the result of the previous `CMP` instruction for greater than, greater or equal, less than, less or equal, equality, or inequality⁶. For example, if you compare the `AX` and `BX` registers with a `"cmp(ax, bx);"` instruction and execute the `JA` instruction, the Y86 CPU will jump to the specified destination location if `AX` was greater than `BX`. If `AX` was not greater than `BX`, control will fall through to the next instruction in the program.

The `JMP` instruction unconditionally transfers control to the instruction at the destination address. The `IRET` instruction returns control from an interrupt service routine, which we will discuss later.

The `GET` and `PUT` instructions let you read and write integer values. `GET` will stop and prompt the user for a hexadecimal value and then store that value into the `AX` register. `PUT` displays (in hexadecimal) the value of the `AX` register.

The remaining instructions do not require any operands, they are `HALT` and `BRK`. `HALT` terminates program execution and `BRK` stops the program in a state that it can be restarted.

The Y86 processors require a unique opcode for every different instruction, not just the instruction classes. Although `mov(bx, ax);` and `mov(cx, ax);` are both in the same class, they must have different opcodes if the CPU is to differentiate them. However, before looking at all the possible opcodes, perhaps it would be a good idea to learn about all the possible operands for these instructions.

5.3.1 Addressing Modes on the Y86

The Y86 instructions use five different operand types: registers, constants, and three memory addressing schemes. Each form is called an *addressing mode*. The Y86 processor supports the *register* addressing mode⁷, the *immediate* addressing mode, the *indirect* addressing mode, the *indexed* addressing mode, and the *direct* addressing mode. The following paragraphs explain each of these modes.

Register operands are the easiest to understand. Consider the following forms of the `MOV` instruction:

`mov(ax, ax);`

`mov(bx, ax);`

`mov(cx, ax);`

`mov(dx, ax);`

The first instruction accomplishes absolutely nothing. It copies the value from the `AX` register back into the `AX` register. The remaining three instructions copy the values of `BX`, `CX` and `DX` into `AX`. Note that these instructions leave `BX`, `CX`, and `DX` unchanged. The second operand (the destination) is not limited to `AX`; you can move values to any of these registers.

Constants are also pretty easy to deal with. Consider the following instructions:

`mov(25, ax);`

6. The Y86 processor only performs *unsigned* comparisons.

7. Technically, registers do not have an address, but we apply the term *addressing mode* to registers nonetheless.

```
mov( 195, bx );  
mov( 2056, cx );  
mov( 1000, dx );
```

These instructions are all pretty straightforward; they load their respective registers with the specified hexadecimal constant⁸.

There are three addressing modes which deal with accessing data in memory. The following instructions demonstrate the use of these addressing modes:

```
mov( [1000], ax );  
mov( [bx], ax );  
mov( [1000+bx], ax );
```

The first instruction above uses the direct addressing mode to load AX with the 16 bit value stored in memory starting at location \$1000.

The "mov([bx], ax);" instruction loads AX from the memory location specified by the contents of the bx register. This is an indirect addressing mode. Rather than using the value in BX, this instruction accesses to the memory location whose address appears in BX. Note that the following two instructions:

```
mov( 1000, bx );  
mov( [bx], ax );
```

are equivalent to the single instruction:

```
mov( [1000], ax );
```

Of course, the second sequence is preferable. However, there are many cases where the use of indirection is faster, shorter, and better. We ll see some examples of this a little later.

The last memory addressing mode is the *indexed* addressing mode. An example of this memory addressing mode is

```
mov( [1000+bx], ax );
```

This instruction adds the contents of BX with \$1000 to produce the address of the memory value to fetch. This instruction is useful for accessing elements of arrays, records, and other data structures.

5.3.2 Encoding Y86 Instructions

Although we could arbitrarily assign opcodes to each of the Y86 instructions, keep in mind that a real CPU uses logic circuitry to decode the opcodes and act appropriately on them. A typical CPU opcode uses a certain number of bits in the opcode to denote the instruction class (e.g., MOV, ADD, SUB), and a certain number of bits to encode each of the operands.

8. All numeric constants in Y86 assembly language are given in hexadecimal. The "\$" prefix is not necessary.

A typical Y86 instruction takes the form shown in Figure 5.3. The basic instruction is either one or three bytes long. The instruction opcode consists of a single byte that contains three fields. The first field, the H.O. three bits, defines the instruction. This provides eight combinations. As you may recall, there are 20 different instructions; we cannot encode 20 instructions with three bits, so we'll have to pull some tricks to handle the other instructions. As you can see in Figure 5.3, the basic opcode encodes the MOV instructions (two instructions, one where the *rr* field specifies the destination, one where the *mmm* field specifies the destination), and the ADD, SUB, CMP, AND, and OR instructions. There is one additional instruction field: *special*. The special instruction class provides a mechanism that allows us to expand the number of available instruction classes, we will return to this expansion opcode shortly.

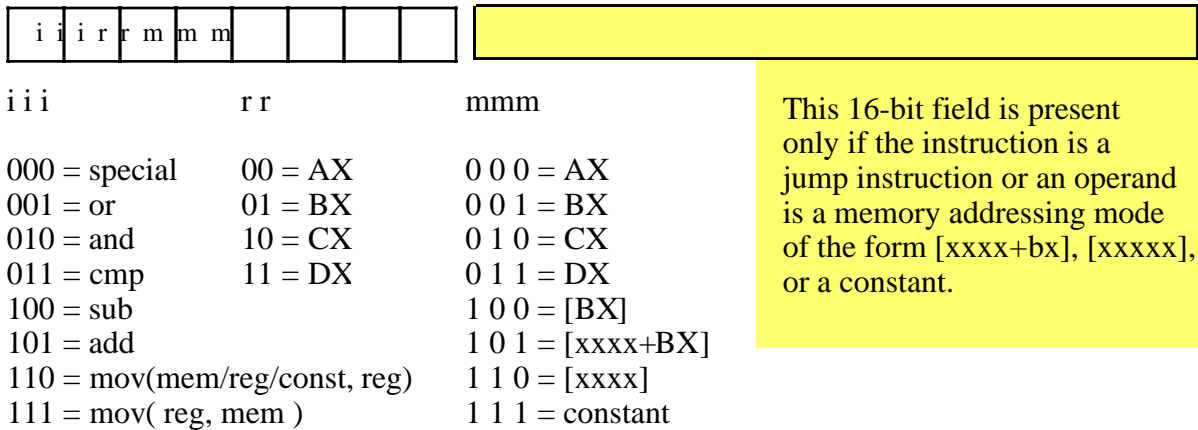


Figure 5.3 Basic Y86 Instruction Encoding

To determine a particular instruction's opcode, you need only select the appropriate bits for the *iii*, *rr*, and *mmm* fields. The *rr* field contains the destination register (except for the MOV instruction whose *iii* field is %111) and the *mmm* field encodes the source operand. For example, to encode the "mov(bx, ax);" instruction you would select *iii*=110 ("mov(reg, reg);"), *rr*=00 (AX), and *mmm*=001 (BX). This produces the one-byte instruction %11000001 or \$C0.

Some Y86 instructions require more than one byte. For example, the instruction "mov([1000], ax);" loads the AX register from memory location \$1000. The encoding for the opcode is %11000110 or \$C6. However, the encoding for the "mov([2000], ax);" instruction's opcode is also \$C6. Clearly these two instructions do different things, one loads the AX register from memory location \$1000 while the other loads the AX register from memory location \$2000. To encode an address for the [xxxx] or [xxxx+bx] addressing modes, or to encode the constant for the immediate addressing mode, you must follow the opcode with the 16-bit address or constant, with the L.O. byte immediately following the opcode in memory and the H.O. byte after that. So the three byte encoding for "mov([1000], ax);" would be \$C6, \$00, \$10 and the three byte encoding for "mov([2000], ax);" would be \$C6, \$00, \$20.

The *special* opcode allows the x86 CPU to expand the set of available instructions. This opcode handles several zero and one-operand instructions as shown in Figure 5.4 and Figure 5.5.

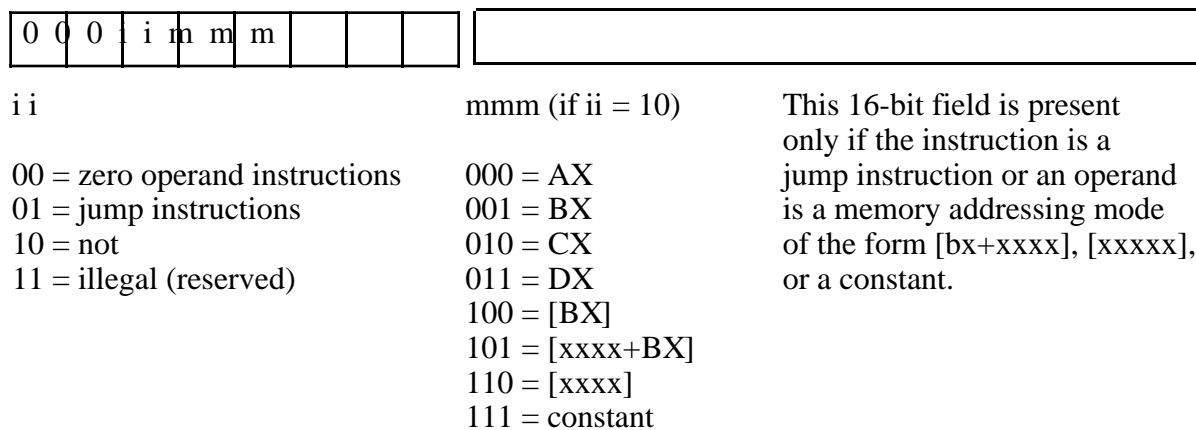


Figure 5.4 Single Operand Instruction Encodings

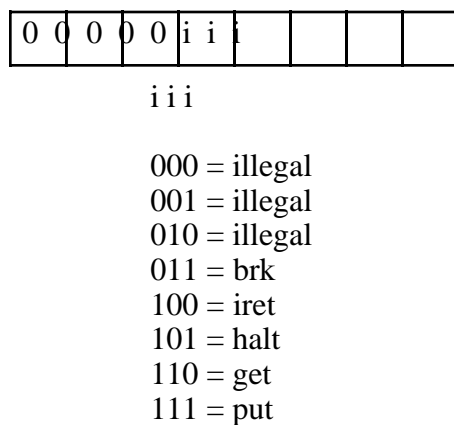


Figure 5.5 Zero Operand Instruction Encodings

There are four one-operand instruction classes. The first encoding (00) further expands the instruction set with a set of zero-operand instructions (see Figure 5.5). The second opcode is also an expansion opcode that provides all the Y86 *jump* instructions (see Figure 5.6). The third opcode is the NOT instruction. This is the bitwise logical not operation that inverts all the bits in the destination register or memory operand. The fourth single-operand opcode is currently unassigned. Any attempt to execute this opcode will halt the processor with an illegal instruction error. CPU designers often reserve unassigned opcodes like this one to extend the instruction set at a future date (as Intel did when moving from the 80286 processor to the 80386).

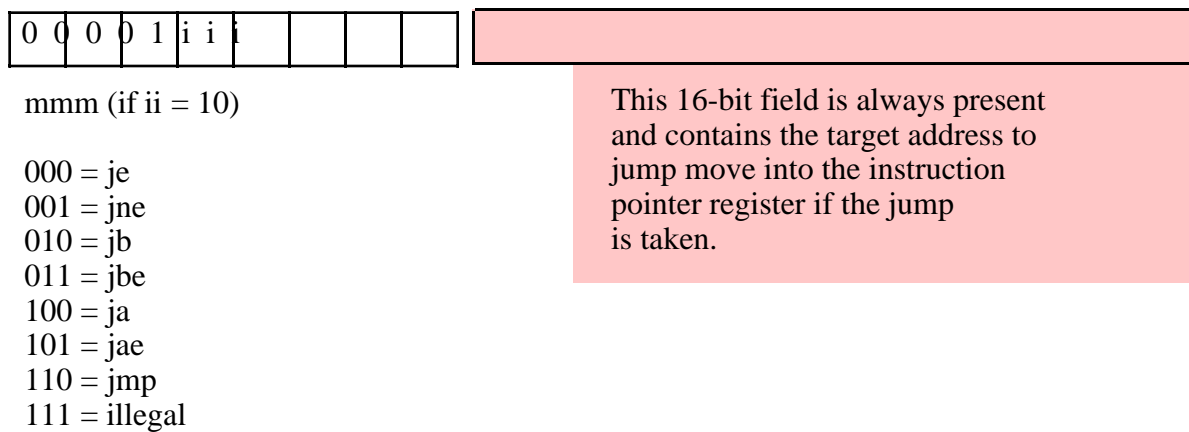


Figure 5.6 Jump Instruction Encodings

There are seven jump instructions in the x86 instruction set. They all take the following form:

`jxx address;`

The JMP instruction copies the 16-bit value (address) following the opcode into the IP register. Therefore, the CPU will fetch the next instruction from this target address; effectively, the program jumps from the point of the JMP instruction to the instruction at the target address.

The JMP instruction is an example of an unconditional jump instruction. It always transfers control to the target address. The remaining six instructions are conditional jump instructions. They test some condition and jump if the condition is true; they fall through to the next instruction if the condition is false. These six instructions, JA, JAE, JB, JBE, JE, and JNE let you test for greater than, greater than or equal, less than, less than or equal, equality, and inequality. You would normally execute these instructions immediately after a CMP instruction since it sets the less than and equality flags that the conditional jump instructions test. Note that there are eight possible jump opcodes, but the x86 uses only seven of them. The eighth opcode is another illegal opcode.

The last group of instructions, the zero operand instructions, appear in Figure 5.5. Three of these instructions are illegal instruction opcodes. The BRK (break) instruction pauses the CPU until the user manually restarts it. This is useful for pausing a program during execution to observe results. The IRET (interrupt return) instruction returns control from an interrupt service routine. We will discuss interrupt service routines later. The HALT program terminates program execution. The GET instruction reads a hexadecimal value from the user and returns this value in the AX register; the PUT instruction outputs the value in the AX register.

5.3.3 Hand Encoding Instructions

Keep in mind that the Y86 processor fetches instructions as bit patterns from memory. It decodes and executes those bit patterns. The processor does not execute instructions of the form "mov(ax, bx);" (that is, a string of characters that are readable by humans). Instead, it executes the bit pattern \$C1 from memory. Instructions like "mov(ax, bx);" and "add(5, cx);" are human-readable representations of these instructions that we must first convert into *machine code* (that is, the binary representation of the instruction that the machine actually executes). In this section we will explore how to manually accomplish this task.

The first step is to choose an instruction to convert into machine code. We'll start with a very simple example, the "add(cx, dx);" instruction. Once you've chosen the instruction, you look up the instruction in one of the figures of the previous section. The ADD instruction is in the first group (see Figure 5.3) and has an *iii* field of %101. The source operand is CX, so the *mmm* field is %010 and the destination operand is DX so the *rr* field is %11. Merging these bits produces the opcode %10111010 or \$BA.

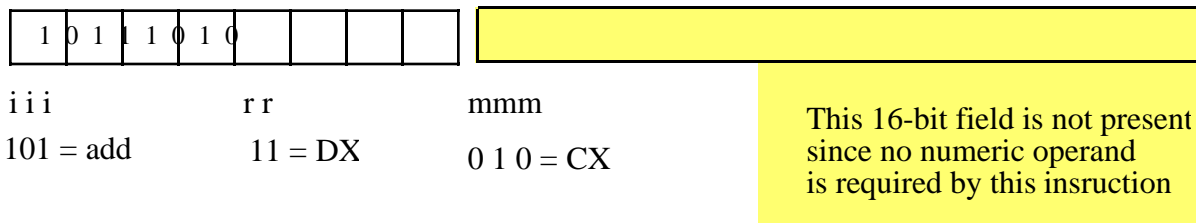


Figure 5.7 Encoding ADD(cx, dx);

Now consider the "add(5, ax);" instruction. Since this instruction has an immediate source operand, the *mmm* field will be %111. The destination register operand is AX (%00) so the full opcode becomes \$10100111 or \$A7. Note, however, that this does not complete the encoding of the instruction. We also have to include the 16-bit constant \$0005 as part of the instruction. The binary encoding of the constant must immediately follow the opcode in memory, so the sequence of bytes in memory (from lowest address to highest address) is \$A7, \$05, \$00. Note that the L.O. byte of the constant follows the opcode and the H.O. byte of the constant follows the L.O. byte. This sequence appears backwards because the bytes are arranged in order of increasing memory address and the H.O. byte of a constant always appears in the highest memory address.

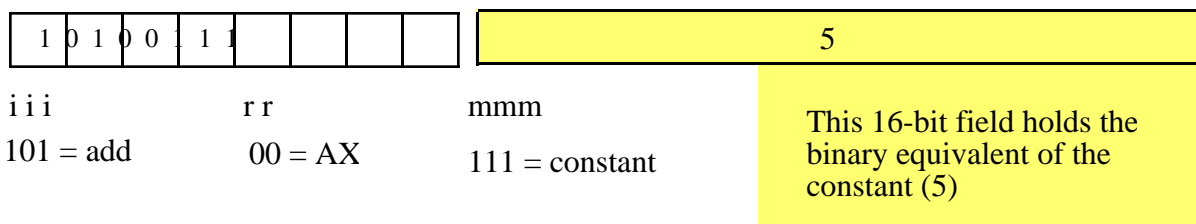


Figure 5.8 Encoding ADD(5, ax);

The "add([2ff+bx], cx);" instruction also contains a 16-bit constant associated with the instruction's encoding — the displacement portion of the indexed addressing mode. To encode this instruction we use the following field values: *iii*=%101, *rr*=%10, and *mmm*=%101. This produces the opcode byte %10110101 or \$B5. The complete instruction also requires the constant \$2FF so the full instruction is the three-byte sequence \$B5, \$FF, \$02.

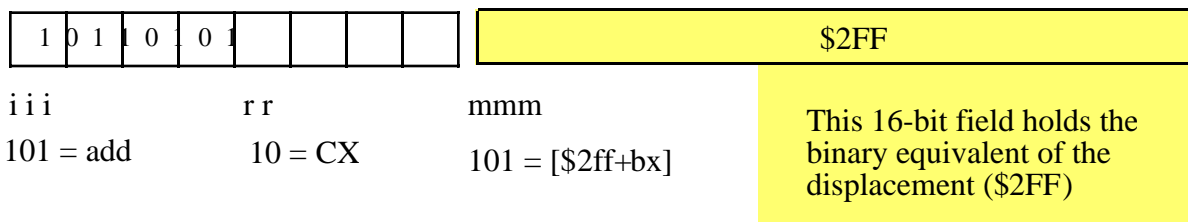


Figure 5.9 Encoding ADD([\$2ff+bx], cx);

Now consider the "add([1000], ax);" instruction. This instruction adds the 16-bit contents of memory locations \$1000 and \$1001 to the value in the AX register. Once again, *iii*=%101 for the ADD instruction. The destination register is AX so *rr*=%00. Finally, the addressing mode is the displacement-only addressing mode, so *mmm*=%110. This forms the opcode %10100110 or \$A6. The instruction is three bytes long since it must encode the displacement (address) of the memory location in the two bytes following the opcode. Therefore, the complete three-byte sequence is \$A6, \$00, \$10.

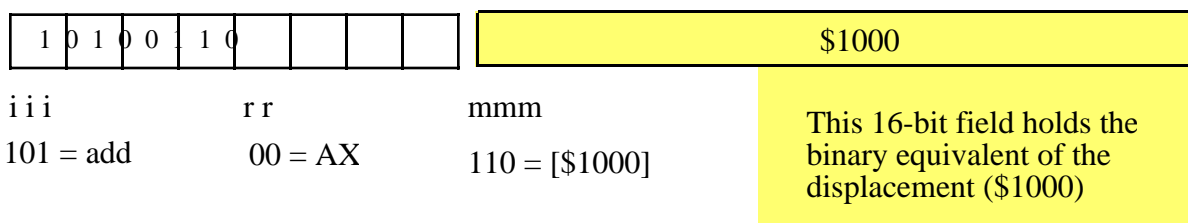


Figure 5.10 Encoding ADD([1000], ax);

The last addressing mode to consider is the register indirect addressing mode, [bx]. The "add([bx], bx);" instruction uses the following encoded values: *mmm*=%101, *rr*=%01 (bx), and *mmm*=%100 ([bx]). Since the value in the BX register completely specifies the memory address, there is no need for a displacement field. Hence, this instruction is only one byte long.

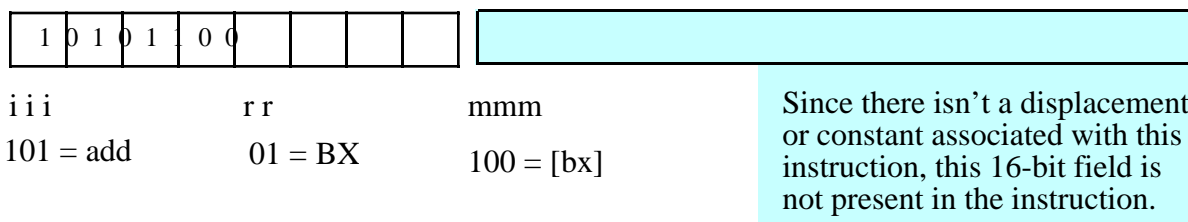


Figure 5.11 Encoding the ADD([bx], bx); Instruction

You use a similar approach to encode the SUB, CMP, AND, and OR instructions as you do the ADD instruction. The only difference is that you use different values for the *iii* field in the opcode.

The MOV instruction is special because there are two forms of the MOV instruction. You encode the first form ($iii=\%110$) exactly as you do the ADD instruction. This form copies a constant or data from memory or a register (the mmm field) into a destination register (the rr field).

The second form of the MOV instruction ($iii=\%111$) copies data from a source register (rr) to a destination memory location (that the mmm field specifies). In this form of the MOV instruction, the source/destination meanings of the rr and mmm fields are reversed so that rr is the source field and mmm is the destination field. Another difference is that the mmm field may only contain the values $\%100$ ($[bx]$), $\%101$ ($[disp+bx]$), and $\%110$ ($[disp]$). The destination values cannot be $\%000.. \%011$ (registers) or $\%111$ (constant). These latter five encodings are illegal (the register destination instructions are handled by the other MOV instruction and storing data into a constant doesn't make any sense).

The Y86 processor supports a single instruction with a single memory/register operand — the NOT instruction. The NOT instruction has the syntax: "not(reg);" or "not(mem);" where mem represents one of the memory addressing modes ($[bx]$, $[disp+bx]$, or $[disp]$). Note that you may not specify a constant as the operand of the NOT instruction.

Since the NOT instruction has only a single operand, it only uses the mmm field to encode this operand. The rr field, combined with the iii field, selects the NOT instruction ($iii=\%000$ and $rr=\%10$). Whenever the iii field contains zero this tells the CPU that special decoding is necessary for the instruction. In this case, the rr field specifies whether we have the NOT instruction or one of the other specially decoded instructions.

To encode an instruction like "not(ax);" you would simply specify $\%000$ for iii and $\%10$ for the rr fields. Then you would encode the mmm field the same way you would encode this field for the ADD instruction. Since $mmm=\%000$ for AX, the encoding of "not(ax);" would be $\%00010000$ or \$10.

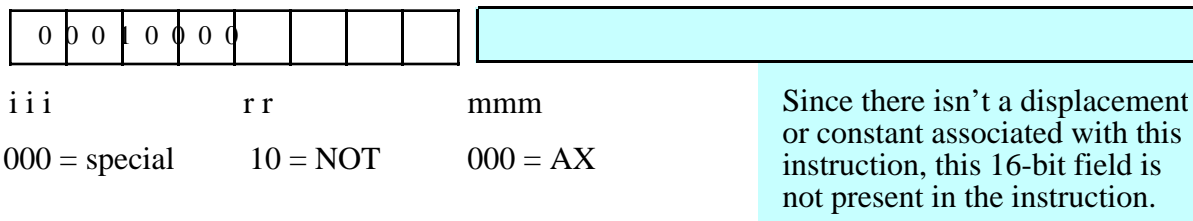


Figure 5.12 Encoding the NOT(ax); Instruction

The NOT instruction does not allow an immediate (constant) operand, hence the opcode $\%00010111$ (\$17) is an illegal opcode.

The Y86 conditional jump instructions also use a special encoding. These instructions are always three bytes long. The first byte (the opcode) specifies which conditional jump instruction to execute and the next two bytes specify where the CPU transfers if the condition is met. There are seven different Y86 jump instructions, six conditional jumps and one unconditional jump. These instructions set $mmm=\%000$, $rr=\%01$, and use the mmm field to select one of the seven possible jumps; the eighth possible opcode is an illegal opcode (see Figure 5.6). Encoding these instructions is relatively straight-forward. Once you pick the instruction you want to encode, you've determined the opcode (since there is a single opcode for each instruction). The opcode values fall in the range \$08..\$0E (\$0F is the illegal opcode).

The only field that requires some thought is the 16-bit operand that follows the opcode. This field holds the address of the target instruction to which the (un)conditional jump transfers if the condition is true (e.g., JE transfers control to this address if the previous CMP instruction found that its two operands were equal). To properly encode this field you must know the address of the opcode byte of the target instruction. If you've already con-

verted the instruction to binary form and stored it into memory, this isn't a problem; just specify the address of that instruction as the operand of the condition jump. On the other hand, if you haven't yet written, converted, and placed that instruction into memory, knowing its address would seem to require a bit of divination. Fortunately, you can figure out the target address by computing the lengths of all the instructions between the current jump instruction you're encoding and the target instruction. Unfortunately, this is an arduous task. The best solution is to write all your instructions down on paper, compute their lengths (which is easy, all instructions are one or three bytes long depending on the presence of a 16-bit operand), and then assign an appropriate address to each instruction. Once you've done this (and, assuming you haven't made any mistakes) you'll know the starting address for each instruction and you can fill in target address operands in your (un)conditional jump instructions as you encode them. Fortunately, there is a better way to do this, as you'll see in the next section.

The last group of instructions, the zero operand instructions, are the easiest to encode. Since they have no operands they are always one byte long and the instruction uniquely specifies the opcode for the instruction. These instructions always have *iii*=%000, *rr*=%00, and *mmm* specifies the particular instruction opcode (see Figure 5.5). Note that the Y86 CPU leaves three of these instructions undefined (so we can use these opcodes for future expansion).

5.3.4 Using an Assembler to Encode Instructions

Of course, hand coding machine language programs as demonstrated in the previous section is impractical for all but the smallest programs. Certainly you haven't had to do anything like this when writing HLA programs. The HLA compiler lets you create a text file containing human readable forms of the instructions. You might wonder why we can write such code for the 80x86 but not for the Y86. The answer is to use an assembler or compiler for the Y86. The job of an assembler/compiler is to read a text file containing human readable text and translate that text into the binary encoded representation for the corresponding machine language program.

An assembler or compiler is nothing special. It's just another program that executes on your computer system. The only thing special about an assembler or compiler is that it translates programs from one form (source code) to another (machine code). A typical Y86 assembler, for example, would read lines of text with each line containing a Y86 instruction, it would *parse*⁹ each statement and then write the binary equivalent of each instruction to memory or to a file for later execution.

Assemblers have two big advantages over coding in machine code. First, they automatically translate strings like "ADD(ax, bx);" and "MOV(ax, [1000]);" to their corresponding binary form. Second, and probably even more important, assemblers let you attach labels to statements and refer to those labels within jump instructions; this means that you don't have to know the target address of an instruction in order to specify that instruction as the target of a jump or conditional jump instruction. Windows users have access to a very simple Y86 assembler¹⁰ that lets you specify up to 26 labels in a program (using the symbols A..Z). To attach a label to a statement, you simply preface the instruction with the label and a colon, e.g.,

```
L:mov( 0, ax );
```

To transfer control to a statement with a label attached to it, you simply specify the label name as the operand of the jump instruction, e.g.,

```
    jmp L;
```

9. "Parse" means to figure out the meaning of the statement.

10. This program is written with Borland's Delphi and was not ported to Linux by the time this was written.

The assembler will compute the address of the label and fill in the address for you whenever you specify the label as the operand of a jump or conditional jump instruction. The assembler can do this even if it hasn't yet encountered the label in the program's source file (i.e., the label is attached to a later instruction in the source file). Most assemblers accomplish this magic by making two passes over the source file. During the first pass the assembler determines the starting address of each symbol and stores this information in a simple database called the *symbol table*. The assembler does not emit any machine code during this first pass. Then the assembler makes a second pass over the source file and actually emits the machine code. During this second pass it looks up all label references in the symbol table and uses the information it retrieves from this database to fill in the operand fields of the instructions that refer to some symbol.

5.3.5 Extending the Y86 Instruction Set

The Y86 CPU is a trivial CPU, suitable only for demonstrating how to encode machine instructions. However, like any good CPU the Y86 design does provide the capability for expansion. So if you wanted to improve the CPU by adding new instructions, the ability to accomplish this exists in the instruction set.

There are two standard ways to increase the number of instructions in a CPU's instruction set. Both mechanisms require the presence of undefined (or illegal) opcodes on the CPU. Since the Y86 CPU has several of these, we can expand the instruction set.

The first method is to directly use the undefined opcodes to define new instructions. This works best when there are undefined bit patterns within an opcode group and the new instruction you want to add falls into that same group. For example, the opcode `%00011mmm` falls into the same group as the NOT instruction. If you decided that you really needed a NEG (negate, take the two's complement) instruction, using this particular opcode for this purpose makes a lot of sense because you'd probably expect the NEG instruction to use the same syntax (and, therefore, decoding) as the NOT instruction.

Likewise, if you want to add a zero-operand instruction to the instruction set, there are three undefined zero-operand instructions that you could use for this purpose. You'd just appropriate one of these opcodes and assign your instruction to it.

Unfortunately, the Y86 CPU doesn't have that many illegal opcodes open. For example, if you wanted to add the SHL, SHR, ROL, and ROR instructions (shift and rotate left and right) as single-operand instructions, there is insufficient space in the single-operand instruction opcodes to add these instructions (there is currently only one open opcode you could use). Likewise, there are no two-operand opcodes open, so if you wanted to add an XOR instruction or some other two-operand instruction, you'd be out of luck.

A common way to handle this dilemma (one the Intel designers have employed) is to use a prefix opcode byte. This opcode expansion scheme uses one of the undefined opcodes as an opcode prefix byte. Whenever the CPU encounters a prefix byte in memory, it reads and decodes the next byte in memory as the actual opcode. However, it does not treat this second byte as it would any other opcode. Instead, this second opcode byte uses a completely different encoding scheme and, therefore, lets you specify as many new instructions as you can encode in that byte (or bytes, if you prefer). For example, the opcode `$FF` is illegal (it corresponds to a "mov(dx, const);" instruction) so we can use this byte as a special prefix byte to further expand the instruction set¹¹.

11. We could also have used values `$F7`, `$EF`, and `$E7` since they also correspond to an attempt to store a register into a constant. However, `$FF` is easier to decode. On the other hand, if you need even more prefix bytes for instruction expansion, you can use these three values as well.

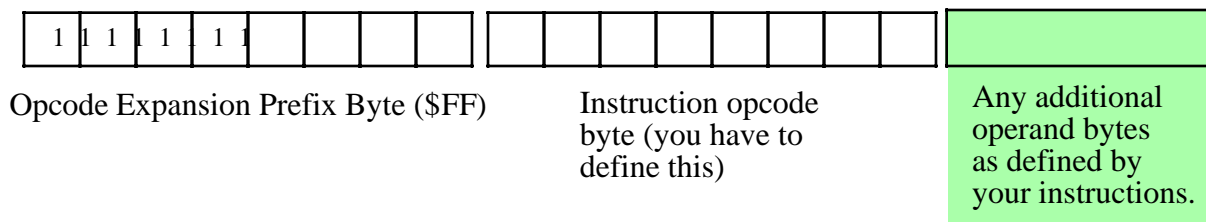


Figure 5.13 Using a Prefix Byte to Extend the Instruction Set

5.4 Encoding 80x86 Instructions

The Y86 processor is simple to understand, easy to hand encode instructions for it, and a great vehicle for learning how to assign opcodes. It is also a purely hypothetical device intended only as a teaching tool. Therefore, you can now forget all about the Y86, it has served its purpose. Now it is time to take a look at the actual machine instruction format for the 80x86 CPU family.

They don't call the 80x86 CPU a *Complex Instruction Set Computer* for nothing. Although more complex instruction encodings do exist, no one is going to challenge the assertion that the 80x86 has a complex instruction encoding. The generic 80x86 instruction takes the form shown in Figure 5.14. Although this diagram seems to imply that instructions can be up to 16 bytes long, in actuality the 80x86 will not allow instructions greater than 15 bytes in length.



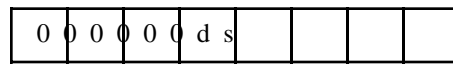
Figure 5.14 80x86 Instruction Encoding

The prefix bytes are not the "opcode expansion prefix" that the previous sections in this chapter discussed. Instead, these are special bytes to modify the behavior of existing instructions (rather than define new instructions). We'll take a look at a couple of these prefix bytes in a little bit, others we'll leave for discussion in later chapters. The 80x86 certainly supports more than four prefix values, however, an instruction may have a maximum of four prefix bytes attached to it. Also note that the behavior of many prefix bytes are mutually exclusive and the results are undefined if you put a pair of mutually exclusive prefix bytes in front of an instruction.

The 80x86 supports two basic opcode sizes: a standard one-byte opcode and a two-byte opcode consisting of a \$0F opcode expansion prefix byte and a second byte specifying the actual instruction. One way to view these opcode bytes is as an eight-bit extension of the *iii* field in the Y86 encoding. This provides for up to 512 different instruction classes (although the 80x86 does not yet use them all). In reality, various instruction classes use certain bits in this opcode for decidedly non-instruction-class purposes. For example, consider the ADD instruction opcode. It takes the form shown in Figure 5.15.

Note that bit number zero specifies the size of the operands the ADD instruction operates upon. If this field contains zero then the operands are eight bit registers and memory locations. If this bit contains one then the operands are either 16-bits or 32-bits. Under 32-bit operating systems the default is 32-bit operands if this field contains a one. To specify a 16-bit operand (under Windows or Linux) you must insert a special "operand-size prefix byte" in front of the instruction.

Bit number one specifies the direction of the transfer. If this bit is zero, then the destination operand is a memory location (e.g., "add(al, [ebx]);" If this bit is one, then the destination operand is a register (e.g., "add([ebx], al);" You'll soon see that this direction bit creates a problem that results in one instruction have two different possible opcodes.



ADD opcode.

d = 0 if adding from register to memory.

d = 1 if adding from memory to register.

s = 0 if adding eight-bit operands.

s = 1 if adding 16-bit or 32-bit operands

Figure 5.15 80x86 ADD Opcode

5.4.1 Encoding Instruction Operands

The "mod-reg-r/m" byte (in Figure 5.14) specifies a basic addressing mode. This byte contains the following fields:

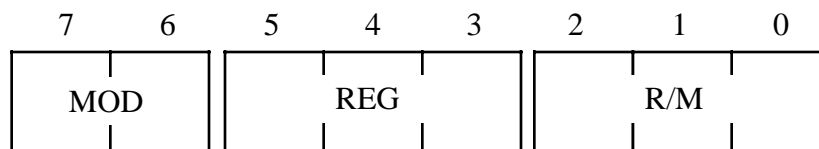


Figure 5.16 MOD-REG-R/M Byte

The REG field specifies an 80x86 register. Depending on the instruction, this can be either the source or the destination operand. Many instructions have the "d" (direction) field in their opcode to choose whether this operand is the source (d=0) or the destination (d=1) operand. This field is encoded using the bit patterns found in the following table:

REG Value	Register if data size is eight bits	Register if data size is 16-bits	Register if data size is 32 bits
%000	al	ax	eax
%001	cl	cx	ecx
%010	dl	dx	edx
%011	bl	bx	ebx
%100	ah	sp	esp
%101	ch	bp	ebp
%110	dh	si	esi

REG Value	Register if data size is eight bits	Register if data size is 16-bits	Register if data size is 32 bits
%111	bh	di	edi

For certain (single operand) instructions, the REG field may contain an opcode extension rather than a register value (the R/M field will specify the operand in this case).

The MOD and R/M fields combine to specify the other operand in a two-operand instruction (or the only operand in a single-operand instruction like NOT or NEG). Remember, the "d" bit in the opcode determines which operand is the source and which is the destination. The MOD and R/M fields together specify the following addressing modes:

MOD	Meaning
%00	Register indirect addressing mode or SIB with no displacement (when R/M=%100) or Displacement only addressing mode (when R/M=%101).
%01	One-byte signed displacement follows addressing mode byte(s).
%10	Four-byte signed displacement follows addressing mode byte(s).
%11	Register addressing mode.

MOD	R/M	Addressing Mode
%00	%000	[eax]
%01	%000	[eax+disp ₈]
%10	%000	[eax+disp ₃₂]
%11	%000	register (al/ax/eax) ^a

MOD	R/M	Addressing Mode
%00	%001	[ecx]
%01	%001	[ecx+disp ₈]
%10	%001	[ecx+disp ₃₂]
%11	%001	register (cl/cx/ecx)
%00	%010	[edx]
%01	%010	[edx+disp ₈]
%10	%010	[edx+disp ₃₂]
%11	%010	register (dl/dx/edx)
%00	%011	[ebx]
%01	%011	[ebx+disp ₈]
%10	%011	[ebx+disp ₃₂]
%11	%011	register (bl/bx/ebx)
%00	%100	SIB Mode
%01	%100	SIB + disp ₈ Mode
%10	%100	SIB + disp ₃₂ Mode
%11	%100	register (ah/sp/esp)
%00	%101	Displacement Only Mode (32-bit displacement)
%01	%101	[ebp+disp ₈]
%10	%101	[ebp+disp ₃₂]
%11	%101	register (ch/bp/ebp)
%00	%110	[esi]
%01	%110	[esi+disp ₈]
%10	%110	[esi+disp ₃₂]
%11	%110	register (dh/si/esi)
%00	%111	[edi]
%01	%111	[edi+disp ₈]
%10	%111	[edi+disp ₃₂]

MOD	R/M	Addressing Mode
%11	%111	register (bh/di/edi)

- a. The size bit in the opcode specifies eight or 32-bit register size. To select a 16-bit register requires a prefix byte.

There are a couple of interesting things to note about this table. First of all, note that there are two forms of the [reg+disp] addressing modes: one form with an eight-bit displacement and one form with a 32-bit displacement. Addressing modes whose displacement falls in the range -128..+127 require only a single byte displacement after the opcode; hence these instructions will be shorter (and sometimes faster) than instructions whose displacement value is outside this range. It turns out that many offsets are within this range, so the assembler/compiler can generate shorter instructions for a large percentage of the instructions.

The second thing to note is that there is no [ebp] addressing mode. If you look in the table above where this addressing mode logically belongs, you'll find that its slot is occupied by the 32-bit displacement only addressing mode. The basic encoding scheme for addressing modes didn't allow for a displacement only addressing mode, so Intel "stole" the encoding for [ebp] and used that for the displacement only mode. Fortunately, anything you can do with the [ebp] addressing mode you can do with the [ebp+disp₈] addressing mode by setting the eight-bit displacement to zero. True, the instruction is a little bit longer, but the capabilities are still there. Intel (wisely) chose to replace this addressing mode because they anticipated that programmers would use this addressing mode less often than the other register indirect addressing modes (for reasons you'll discover in a later chapter).

Another thing you'll notice missing from this table are addressing modes of the form [ebx+edx*4], the so-called scaled indexed addressing modes. You'll also notice that the table is missing addressing modes of the form [esp], [esp+disp₈], and [esp+disp₃₂]. In the slots where you would normally expect these addressing modes you'll find the SIB (scaled index byte) modes. If these values appear in the MOD and R/M fields then the addressing mode is a scaled indexed addressing mode with a second byte (the SIB byte) following the MOD-REG-R/M byte that specifies the registers to use (note that the MOD field still specifies the displacement size of zero, one, or four bytes). The following diagram shows the layout of this SIB byte and the following tables explain the values for each field.

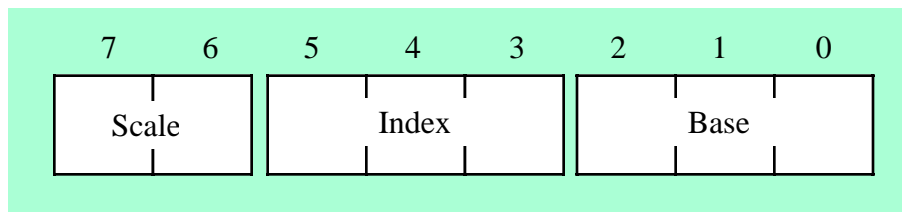


Figure 5.17 SIB (Scaled Index Byte) Layout

Scale Value	Index*Scale Value
%00	Index*1
%01	Index*2

Scale Value	Index*Scale Value
% 10	Index*4
% 11	Index*8

Index	Register
%000	EAX
%001	ECX
%010	EDX
%011	EBX
% 100	Illegal
% 101	EBP
% 110	ESI
% 111	EDI

Base	Register
%000	EAX
%001	ECX
%010	EDX
%011	EBX
% 100	ESP
% 101	Displacement-only if MOD = %00, EBP if MOD = %01 or %10
% 110	ESI
% 111	EDI

The MOD-REG-R/M and SIB bytes are complex and convoluted, no question about that. The reason these addressing mode bytes are so convoluted is because Intel reused their 16-bit addressing circuitry in the 32-bit mode rather than simply abandoning the 16-bit format in the 32-bit mode. There are good hardware reasons for this, but the end result is a complex scheme for specifying addressing modes.

Part of the reason the addressing scheme is so convoluted is because of the special cases for the SIB and displacement-only modes. You will note that the intuitive encoding of the MOD-REG-R/M byte does not allow for a displacement-only mode. Intel added a quick kludge to the addressing scheme replacing the [EBP] addressing mode with the displacement-only mode. Programmers who actually want to use the [EBP] addressing mode have to use [EBP+0] instead. Semantically, this mode produces the same result but the instruction is one byte longer since it requires a displacement byte containing zero.

You will also note that if the REG field of the MOD-REG-R/M byte contains %100 and MOD does not contain %11 then the addressing mode is an "SIB" mode rather than the expected [ESP], [ESP+disp₈], or [ESP+disp₃₂] mode. The SIB mode is used when an addressing mode uses one of the scaled indexed registers, i.e., one of the following addressing modes:

[reg₃₂+eax*n] MOD = %00
 [reg₃₂+ebx*n] Note: n = 1, 2, 4, or 8.
 [reg₃₂+ecx*n]
 [reg₃₂+edx*n]
 [reg₃₂+ebp*n]
 [reg₃₂+esi*n]
 [reg₃₂+edi*n]

[disp+reg₈+eax*n] MOD = %01
 [disp+reg₈+ebx*n]
 [disp+reg₈+ecx*n]
 [disp+reg₈+edx*n]
 [disp+reg₈+ebp*n]
 [disp+reg₈+esi*n]
 [disp+reg₈+edi*n]

[disp+reg₃₂+eax*n] MOD = %10
 [disp+reg₃₂+ebx*n]
 [disp+reg₃₂+ecx*n]
 [disp+reg₃₂+edx*n]
 [disp+reg₃₂+ebp*n]
 [disp+reg₃₂+esi*n]
 [disp+reg₃₂+edi*n]

[disp+eax*n] MOD = %00 and BASE field contains %101
 [disp+ebx*n]
 [disp+ecx*n]

[disp+edx*n]
 [disp+ebp*n]
 [disp+esi*n]
 [disp+edi*n]

In each of these addressing modes, the MOD field of the MOD-REG-R/M byte specifies the size of the displacement (zero, one, or four bytes). This is indicated via the modes "SIB Mode," "SIB + disp₈ Mode," and "SIB + disp₃₂ Mode." The Base and Index fields of the SIB byte select the base and index registers, respectively. Note that this addressing mode does not allow the use of the ESP register as an index register. Presumably, Intel left this particular mode undefined to provide the ability to extend the addressing modes in a future version of the CPU (although extending the addressing mode sequence to three bytes seems a bit extreme).

Like the MOD-REG-R/M encoding, the SIB format redefines the [EBP+index*scale] mode as a displacement plus index mode. Once again, if you really need this addressing mode, you will have to use a single byte displacement value containing zero to achieve the same result.

5.4.2 Encoding the ADD Instruction: Some Examples

To figure out how to encode an instruction using this complex scheme, some examples are warranted. So let's take a look at how to encode the 80x86 ADD instruction using various addressing modes. The ADD opcode is \$00, \$01, \$02, or \$03, depending on the direction and size bits in the opcode (see Figure 5.15). The following figures each describe how to encode various forms of the ADD instruction using different addressing modes.

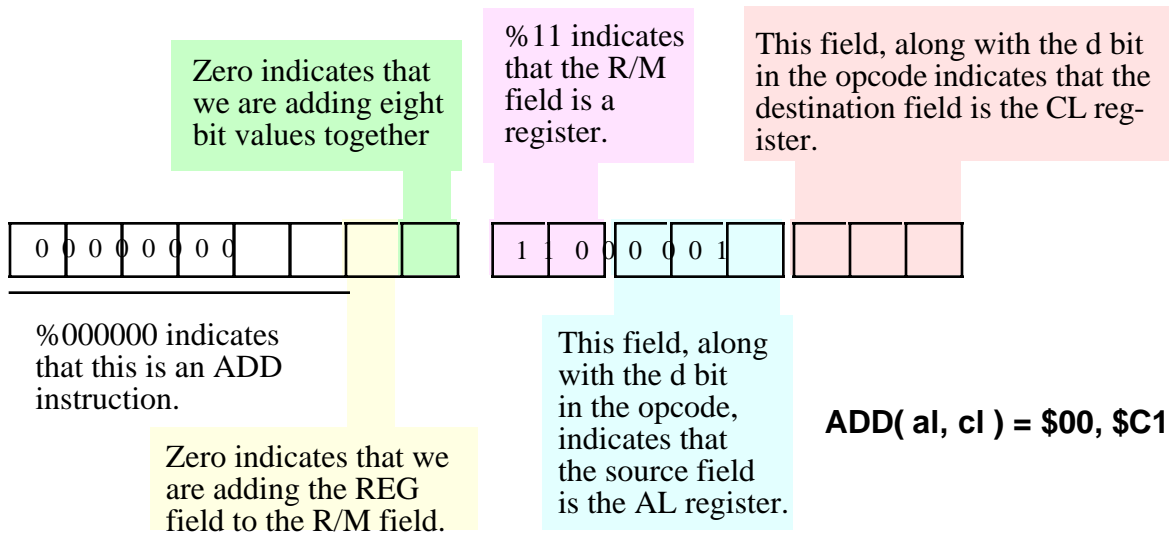


Figure 5.18 Encoding the ADD(al, cl); Instruction

There is an interesting side effect of the operation of the direction bit and the MOD-REG-R/M organization: some instructions have two different opcodes (and both are legal). For example, we could encode the "add(al, cl);" instruction from Figure 5.18 as \$02, \$C8 by reversing the AL and CL registers in the REG and R/M fields and then setting the *d* bit in the opcode (bit #1). This issue applies to instructions with two register operands.

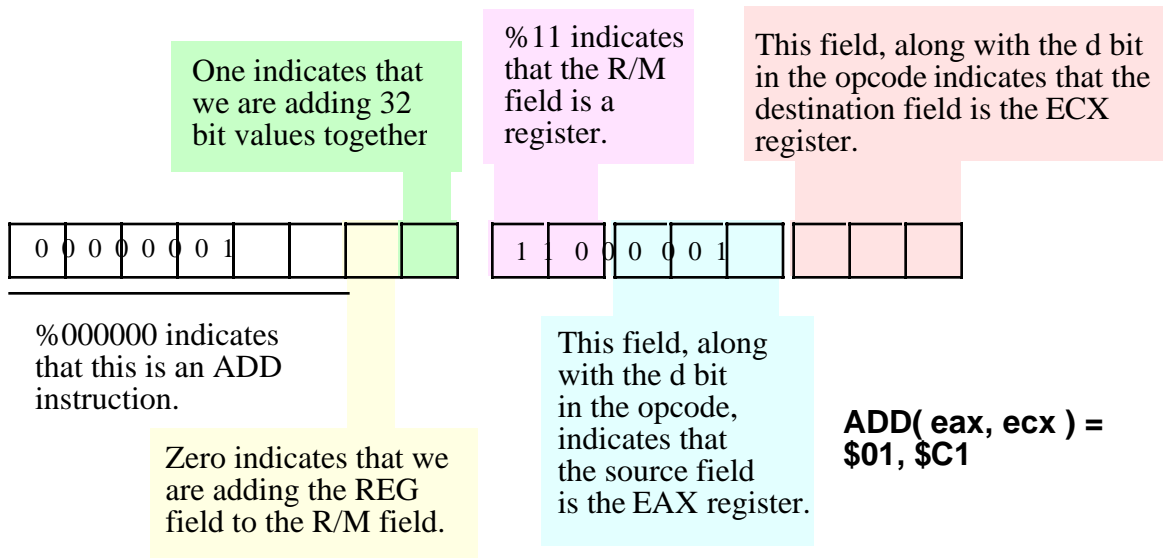
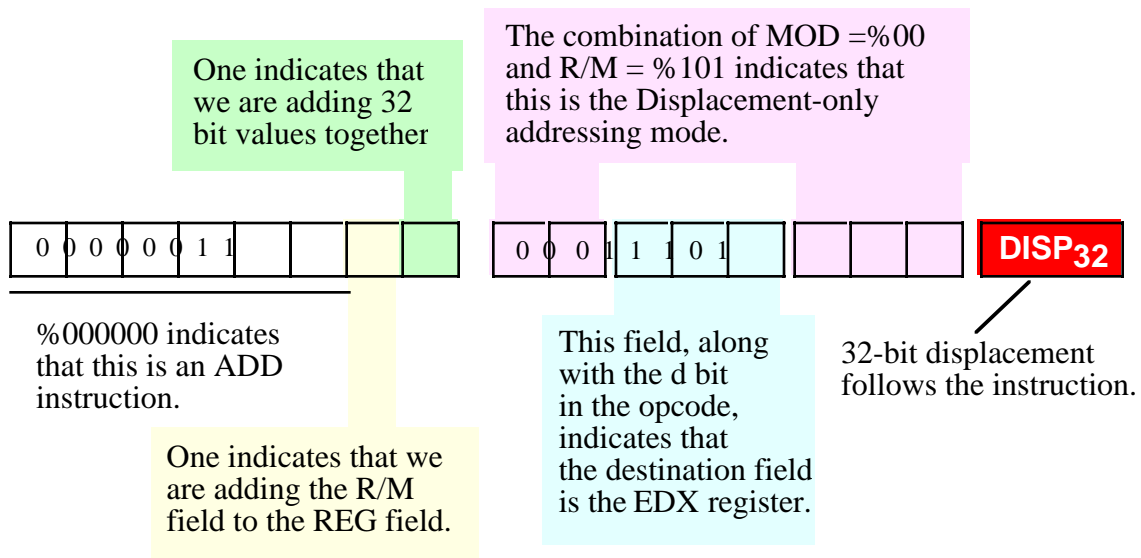


Figure 5.19 Encoding the ADD(eax, ecx); instruction

Note that we can also encode "add(eax, ecx);" using the bytes \$03, \$C8.



ADD(disp, edx) = \$03, \$1D, \$ww, \$xx, \$yy, \$zz

Note: \$ww, \$xx, \$yy, \$zz represent the four displacement byte values with \$ww being the L.O. byte and \$zz being the H.O. byte.

Figure 5.20 Encoding the ADD(disp, edx); Instruction

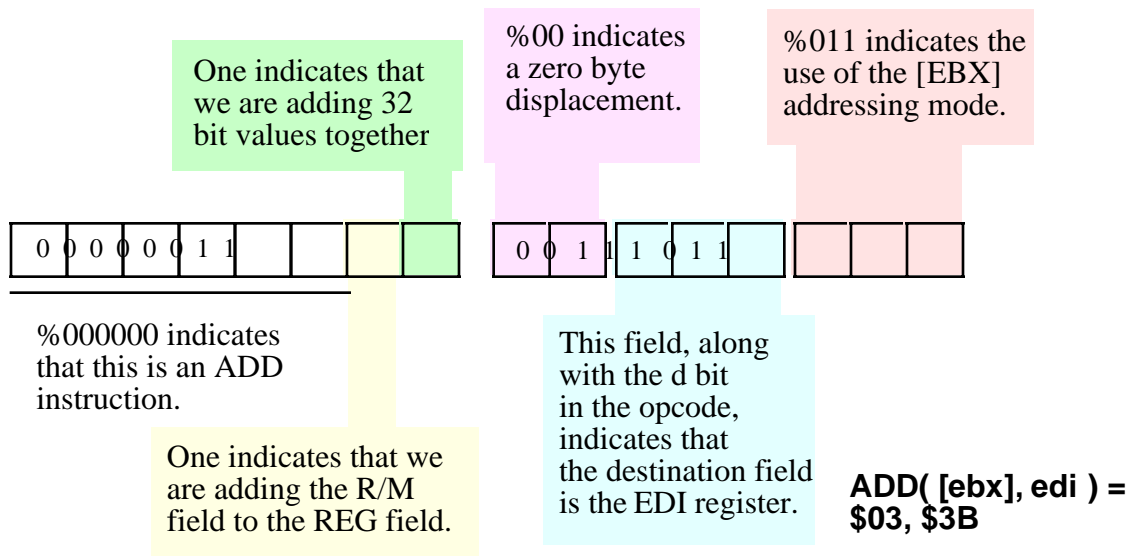


Figure 5.21 Encoding the ADD([ebx], edi); Instruction

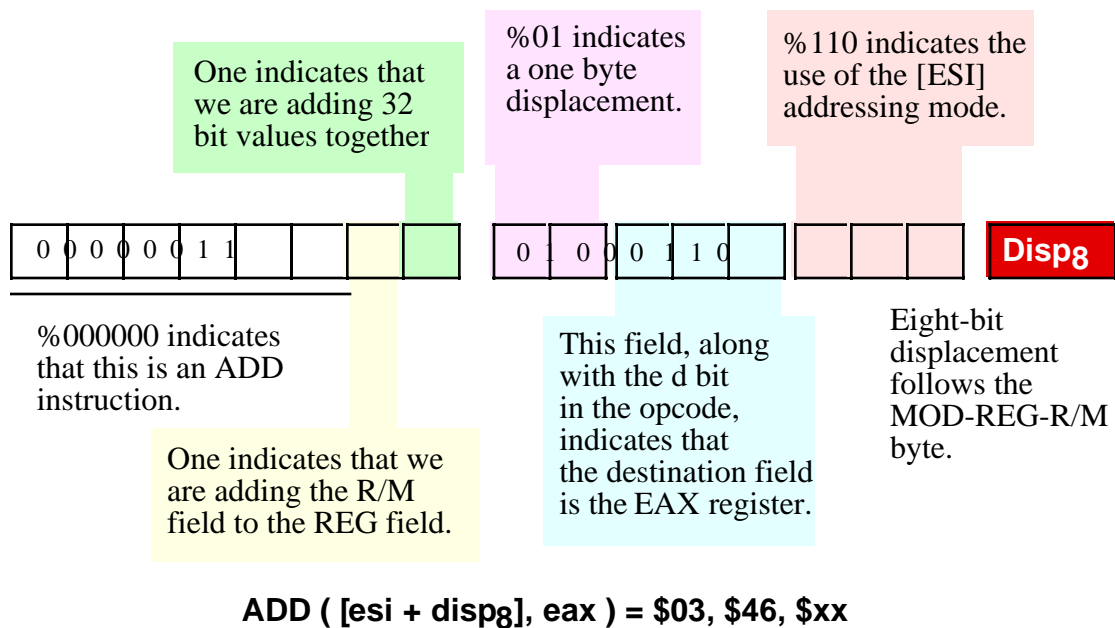
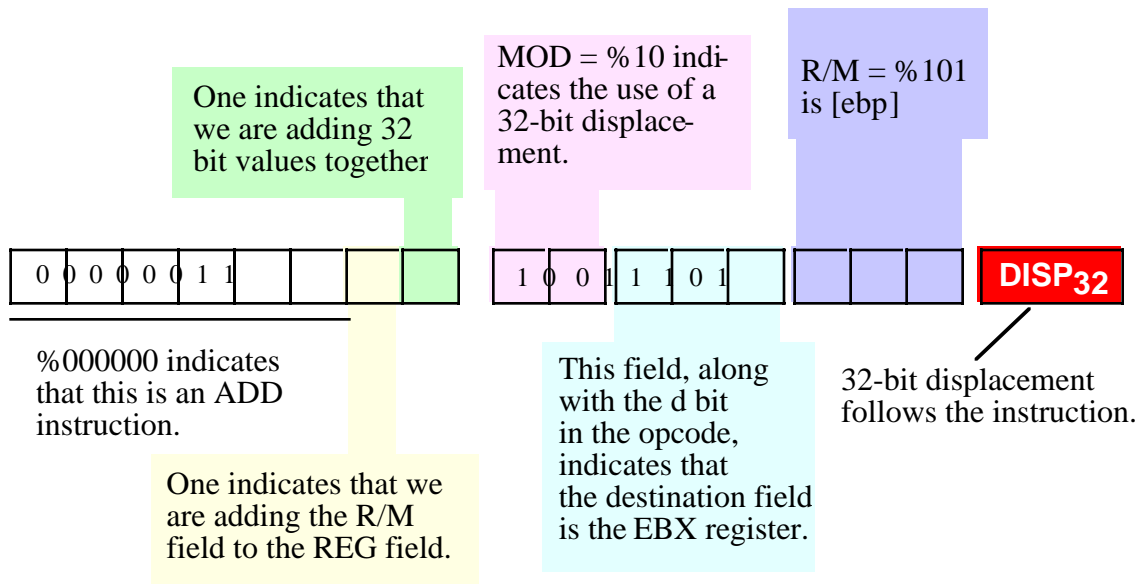


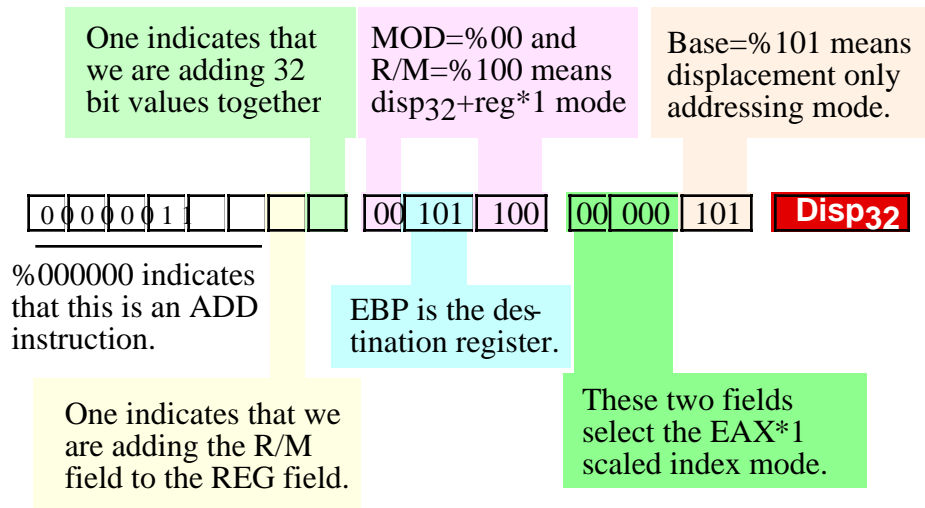
Figure 5.22 Encoding the ADD([esi+disp₈], eax); Instruction



ADD([ebp+disp₃₂], ebx) = \$03, \$9D, \$ww, \$xx, \$yy, \$zz

Note: \$ww, \$xx, \$yy, \$zz represent the four displacement byte values with \$ww being the L.O. byte and \$zz being the H.O. byte.

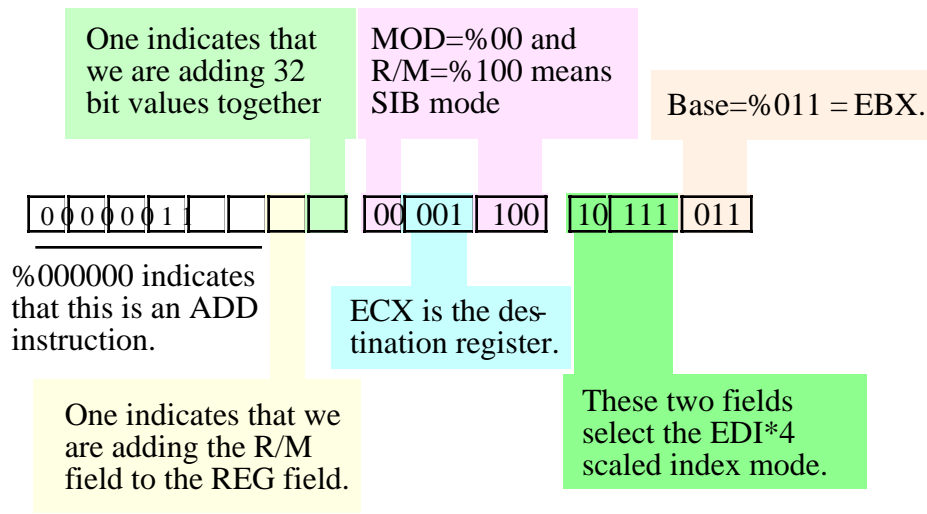
Figure 5.23 Encoding the ADD ([ebp+disp₃₂], ebx); Instruction



ADD ([disp₃₂ + eax*1], ebp) = \$03, \$2C, \$05, \$ww, \$xx, \$yy, \$zz

Note: \$ww, \$xx, \$yy, \$zz represent the four displacement byte values with \$ww being the L.O. byte and \$zz being the H.O. byte.

Figure 5.24 Encoding the ADD([disp₃₂ +eax*1], ebp); Instruction



ADD ([ebx+ edi*4], ecx) = \$03, \$0C, \$BB

Figure 5.25 Encoding the ADD([ebx + edi * 4], ecx); Instruction

5.4.3 Encoding Immediate Operands

You may have noticed that the MOD-REG-R/M and SIB bytes don't contain any bit combinations you can use to specify an immediate operand. The 80x86 uses a completely different opcode to specify an immediate operand. Figure 5.26 shows the basic encoding for an ADD immediate instruction.

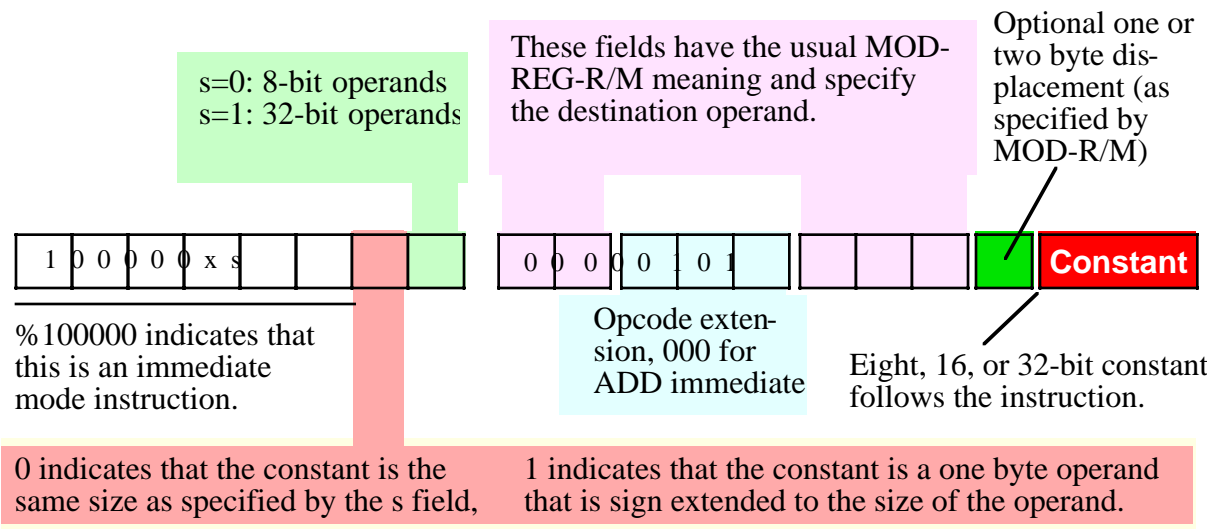


Figure 5.26 Encoding an ADD Immediate Instruction

There are three major differences between the encoding of the ADD immediate and the standard ADD instruction. First, and most important, the opcode has a one in the H.O. bit position. This tells the CPU that the instruction has an immediate

constant. This individual change, however, does not tell the CPU that it must execute an ADD instruction, as you'll see momentarily.

The second difference is that there is no direction bit in the opcode. This makes sense because you cannot specify a constant as a destination operand. Therefore, the destination operand is always the location the MOD and R/M bits specify in the MOD-REG-R/M field.

In place of the direction bit, the opcode has a sign extension (*x*) bit. For eight-bit operands, the CPU ignores this bit. For 16-bit and 32-bit operands, this bit specifies the size of the constant following the ADD instruction. If this bit contains zero then the constant is the same size as the operand (i.e., 16 or 32 bits). If this bit contains one then the constant is a signed eight-bit value and the CPU sign extends this value to the appropriate size before adding it to the operand. This little trick often makes programs quite a bit shorter because one commonly adds small valued constants to 16 or 32 bit operands.

The third difference between the ADD immediate and the standard ADD instruction is the meaning of the REG field in the MOD-REG-R/M byte. Since the instruction implies that the source operand is a constant and the MOD-R/M fields specify the destination operand, the instruction does not need to use the REG field to specify an operand. Instead, the 80x86 CPU uses these three bits as an opcode extension. For the ADD immediate instruction these three bits must contain zero (other bit patterns would correspond to a different instruction).

Note that when adding a constant to a memory location, the displacement (if any) associated with the memory location immediately precedes the immediate (constant) data in the opcode sequence.

5.4.4 Encoding Eight, Sixteen, and Thirty-Two Bit Operands

When Intel designed the 8086 they used one bit (*s*) to select between eight and sixteen bit integer operand sizes in the opcode. Later, when they extended the 80x86 architecture to 32 bits with the introduction of the 80386, they had a problem, with this single bit they could only encode two sizes but they needed to encode three (8, 16, and 32 bits). To solve this problem, they used a *operand size prefix byte*.

Intel studied their instruction set and came to the conclusion that in a 32-bit environment, programs were more likely to use eight-bit and 32-bit operands far more often than 16-bit operands. So Intel decided to let the size bit (*s*) in the opcode select between eight and thirty-two bit operands, as the previous sections describe. Although modern 32-bit programs don't use 16-bit operands that often, they do need them now and then. To allow for 16-bit operands, Intel lets you prefix a 32-bit instruction with the operand size prefix byte, whose value is \$66. This prefix byte tells the CPU to operand on 16-bit data rather than 32-bit data.

You do not have to explicitly put an operand size prefix byte in front of your 16-bit instructions; the assembler will take care of this automatically for you whenever you use a 16-bit operand in an instruction. However, do keep in mind that whenever you use a 16-bit operand in a 32-bit program, the instruction is longer (by one byte) because of the prefix value. Therefore, you should be careful about using 16-bit instructions if size (and to a lesser extent, speed) are important because these instructions are longer (and may be slower because of their effect on the cache).

5.4.5 Alternate Encodings for Instructions

As noted earlier in this chapter, one of Intel's primary design goals for the 80x86 was to create an instruction set to allow programmers to write very short programs in order to save precious (at the time) memory. One way they did this was to create alternate encodings of some very commonly used instructions. These alternate instructions were shorter than the standard counterparts and Intel hoped that programmers would make extensive use of these instructions, thus creating shorter programs.

A good example of these alternate instructions are the "add(constant, accumulator);" instructions (the accumulator is AL, AX, or EAX). The 80x86 provides a single byte opcode for "add(constant, al);" and "add(constant, eax);" (the opcodes are \$04 and \$05, respectively). With a one-byte opcode and no MOD-REG-R/M byte, these instructions are one byte shorter than their standard ADD immediate counterparts. Note that the "add(constant, ax);" instruction requires an operand size prefix (as does the standard "add(constant, ax);" instruction, so it's opcode is effectively two bytes if you count the prefix byte. This, however, is still one byte shorter than the corresponding standard ADD immediate.

You do not have to specify anything special to use these instructions. Any decent assembler will automatically choose the shortest possible instruction it can use when translating your source code into machine code. However, you should note that Intel only provides alternate encodings for the accumulator registers. Therefore, if you have a choice of several instructions to

use and the accumulator registers are among these choices, the AL/AX/EAX registers almost always make the best bet. This is a good reason why you should take some time and scan through the encodings of the 80x86 instructions some time. By familiarizing yourself with the instruction encodings, you'll know which instructions have special (and, therefore, shorter) encodings.

5.5 Putting It All Together

Designing an instruction set that can stand the test of time is a true intellectual challenge. An engineer must balance several compromises when choosing an instruction set and assigning opcodes for the instructions. The Intel 80x86 instruction set is a classic example of a kludge that people are currently using for purposes the original designers never intended. However, the 80x86 is also a marvelous testament to the ingenuity of Intel's engineers who were faced with the difficult task of extending the CPU in ways it was never intended. The end result, though functional, is extremely complex. Clearly, no one designing a CPU (from scratch) today would choose the encoding that Intel's engineers are using. Nevertheless, the 80x86 CPU does demonstrate that careful planning (or just plain luck) does give the designer the ability to extend the CPU far beyond its original design.

Historically, an important fact we've learned from the 80x86 family is that it's very poor planning to assume that your CPU will last only a short time period and that users will replace the chip and their software when something better comes along. Software developers usually don't have a problem adapting to a new architecture when they write new software (assuming financial incentive to do so), but they are very resistant to moving existing software from one platform to another. This is the primary reason the Intel 80x86 platform remains popular to this day.

Choosing which instructions you want to incorporate into the initial design of a new CPU is a difficult task. You must balance the desire to provide lots of useful instructions with the silicon budget and you must also be careful not to include lots of irrelevant instructions that programmers wind up ignoring for one reason or another. Remember, all future versions of the CPU will probably have to support all the instructions in the initial instruction set, so it's better to err on the side of supplying too few instructions rather than too many. Remember, you can always expand the instruction set in a later version of the chip.

Hand in hand with selecting the optimal instruction set is allowing for easy future expansion of the chip. You must leave some undefined opcodes available so you can easily expand the instruction set later on. However, you must balance the number of undefined opcodes with the number of initial instructions and the size of your opcodes. For efficiency reasons, we want the opcodes to be as short as possible. We also need a reasonable set of instructions in the initial instruction set. A reasonable instruction set may consume most of the legal bit patterns in small opcode. So a hard decision has to be made: reduce the number of instructions in the initial instruction set, increase the size of the opcode, or rely on an opcode prefix byte (which makes the newer instructions (you add later) longer. There is no easy answer to this problem, as the CPU designer, you must carefully weigh these choices during the initial CPU design. Unfortunately, you can't easily change your mind later on.

Most CPUs (Von Neumann architecture) use a binary encoding of instructions and fetch these instructions from memory. This chapter introduces the concept of binary instruction encoding via the hypothetical "Y86" processor. This is a trivial (and not very practical) CPU design that makes it easy to demonstrate how to choose opcodes for a simple instruction set, encode operands, and leave room for future expansion. Some of the more interesting features the Y86 demonstrates includes the fact that an opcode often contains subfields and we usually group instructions by the number of types of operands they support. The Y86 encoding also demonstrates how to use special opcodes to differentiate one group of instructions from another and to provide undefined (illegal) opcodes that we can use for future expansion.

The Y86 CPU is purely hypothetical and useful only as an educational tool. After exploring the design of a simple instruction set with the Y86, this chapter began to discuss the encoding of instructions on the 80x86 platform. While the full 80x86 instruction set is far too complex to discuss this early in this text (i.e., there are lots of instructions we still have to discuss later in this text), this chapter was able to discuss basic instruction encoding using the ADD instruction as an example. Note that this chapter only touches on the 80x86 instruction encoding scheme. For a full discussion of 80x86 encoding, see the appendices in this text and the Intel 80x86 documentation.

Memory Architecture

Chapter Six

6.1 Chapter Overview

This chapter discusses the memory hierarchy – the different types and performance levels of memory found on a typical 80x86 computer system. Many programmers tend to view memory as this big nebulous block of storage that holds values for future use. From a semantic point of view, this is a reasonable view. However, from a performance point of view there are many different kinds of memory and using the wrong one or using one form improperly can have a dramatically negative impact on the performance of a program. This chapter discusses the memory hierarchy and how to best use it within your programs.

6.2 The Memory Hierarchy

Most modern programs can benefit greatly from a large amount of very fast memory. A physical reality, however, is that as a memory device gets larger, it tends to get slower. For example, cache memories (see “Cache Memory” on page 153) are very fast but are also small and expensive. Main memory is inexpensive and large, but is slow (requiring wait states, see “Wait States” on page 151). The memory hierarchy is a mechanism of comparing the cost and performance of the various places we can store data and instructions. Figure 6.1 provides a look at one possible form of the memory hierarchy.

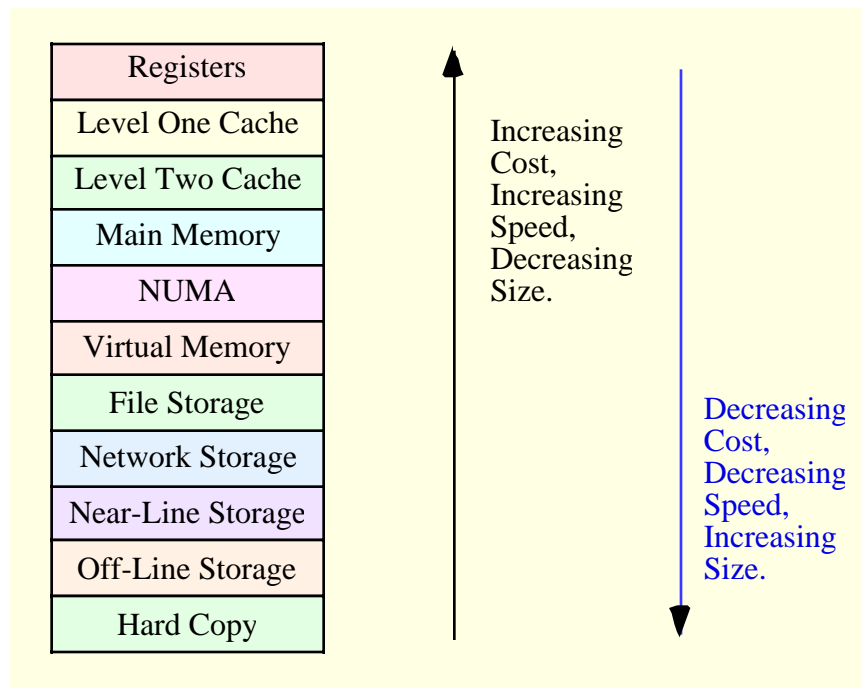


Figure 6.1 The Memory Hierarchy

At the top level of the memory hierarchy are the CPU’s general purpose registers. The registers provide the fastest access to data possible on the 80x86 CPU. The register file is also the smallest memory object in the memory hierarchy (with just eight general purpose registers available). By virtue of the fact that it is virtually impossible to add more registers to the 80x86, registers are also the most expensive memory locations.

Note that we can include FPU, MMX, SIMD, and other CPU registers in this class as well. These additional registers do not change the fact that there are a very limited number of registers and the cost per byte is quite high (figuring the cost of the CPU divided by the number of bytes of register available).

Working our way down, the Level One Cache system is the next highest performance subsystem in the memory hierarchy. On the 80x86 CPUs, the Level One Cache is provided on-chip by Intel and cannot be expanded. The size is usually quite small (typically between 4Kbytes and 32Kbytes), though much larger than the registers available on the CPU chip. Although the Level One Cache size is fixed on the CPU and you cannot expand it, the cost per byte of cache memory is much lower than that of the registers because the cache contains far more storage than is available in all the combined registers.

The Level Two Cache is present on some CPUs, on other CPUs it is the system designer's task to incorporate this cache (if it is present at all). For example, most Pentium II, III, and IV CPUs have a level two cache as part of the CPU package, but many of Intel's Celeron chips do not¹. The Level Two Cache is generally much larger than the level one cache (e.g., 256 or 512KBytes versus 16 Kilobytes). On CPUs where Intel includes the Level Two Cache as part of the CPU package, the cache is not expandable. It is still lower cost than the Level One Cache because we amortize the cost of the CPU across all the bytes in the Level Two Cache. On systems where the Level Two Cache is external, many system designers let the end user select the cache size and upgrade the size. For economic reasons, external caches are actually more expensive than caches that are part of the CPU package, but the cost per bit at the transistor level is still equivalent to the in-package caches.

Below the Level Two Cache system in the memory hierarchy falls the main memory subsystem. This is the general-purpose, relatively low-cost memory found in most computer systems. Typically, this is DRAM or some similar inexpensive memory technology.

Below main memory is the NUMA category. NUMA, which stands for NonUniform Memory Access is a bit of a misnomer here. NUMA means that different types of memory have different access times. Therefore, the term NUMA is fairly descriptive of the entire memory hierarchy. In Figure 6.1a, however, we'll use the term NUMA to describe blocks of memory that are electronically similar to main memory but for one reason or another operate significantly slower than main memory. A good example is the memory on a video display card. Access to memory on video display cards is often much slower than access to main memory. Other peripheral devices that provide a block of shared memory between the CPU and the peripheral probably have similar access times as this video card example. Another example of NUMA includes certain slower memory technologies like Flash Memory that have significant slower access and transfers times than standard semiconductor RAM. We'll use the term NUMA in this chapter to describe these blocks of memory that look like main memory but run at slower speeds.

Most modern computer systems implement a Virtual Memory scheme that lets them simulate main memory using storage on a disk drive. While disks are significantly slower than main memory, the cost per bit is also significantly lower. Therefore, it is far less expensive (by three orders of magnitude) to keep some data on magnetic storage rather than in main memory. A Virtual Memory subsystem is responsible for transparently copying data between the disk and main memory as needed by a program.

File Storage also uses disk media to store program data. However, it is the program's responsibility to store and retrieve file data. In many instances, this is a bit slower than using Virtual Memory, hence the lower position in the memory hierarchy².

Below File Storage in the memory hierarchy comes Network Storage. At this level a program is keeping data on a different system that connects the program's system via a network. With Network Storage you can implement Virtual Memory, File Storage, and a system known as Distributed Shared Memory (where processes running on different computer systems share data in a common block of memory and communicate changes to that block across the network).

Virtual Memory, File Storage, and Network Storage are examples of so-called *on-line memory subsystems*. Memory access via these mechanism is slower than main memory access, but when a program

1. Note, by the way, that the level two cache on the Pentium CPUs is typically not on the same chip as the CPU. Instead, Intel packages a separate chip inside the box housing the Pentium CPU and wires this second chip (containing the level two cache) directly to the Pentium CPU inside the package.

2. Note, however, that in some degenerate cases Virtual Memory can be much slower than file access.

requests data from one of these memory devices, the device is ready and able to respond to the request as quickly as is physically possible. This is not true for the remaining levels in the memory hierarchy.

The Near-Line and Off-Line Storage subsystems are not immediately ready to respond to a program's request for data. An Off-Line Storage system keeps its data in electronic form (usually magnetic or optical) but on media that is not (necessarily) connected to the computer system while the program that needs the data is running. Examples of Off-Line Storage include magnetic tapes, disk cartridges, optical disks, and floppy diskettes. When a program needs data from an off-line medium, the program must stop and wait for a someone or something to mount the appropriate media on the computer system. This delay can be quite long (perhaps the computer operator decided to take a coffee break?). Near-Line Storage uses the same media as Off-Line Storage, the difference is that the system holds the media in a special robotic jukebox device that can automatically mount the desired media when some program requests it. Tapes and removable media are among the most inexpensive electronic data storage formats available. Hence, these media are great for storing large amounts of data for long time periods.

Hard Copy storage is simply a print-out (in one form or another) of some data. If a program requests some data and that data is present only in hard copy form, someone will have to manually enter the data into the computer. Paper (or other hard copy media) is probably the least expensive form of memory, at least for certain data types.

6.3 How the Memory Hierarchy Operates

The whole point of the memory hierarchy is to allow reasonably fast access to a large amount of memory. If only a little memory was necessary, we'd use fast static RAM (i.e., the stuff they make cache memory out of) for everything. If speed wasn't necessary, we'd just use low-cost dynamic RAM for everything. The whole idea of the memory hierarchy is that we can take advantage of the principle of locality of reference (see "Cache Memory" on page 153) to move often-referenced data into fast memory and leave less-used data in slower memory. Unfortunately, the selection of often-used versus lesser-used data varies over the execution of any given program. Therefore, we cannot simply place our data at various levels in the memory hierarchy and leave the data alone throughout the execution of the program. Instead, the memory subsystems need to be able to move data between themselves dynamically to adjust for changes in locality of reference during the program's execution.

Moving data between the registers and the rest of the memory hierarchy is strictly a program function. The program, of course, loads data into registers and stores register data into memory using instructions like MOV. It is strictly the programmer's or compiler's responsibility to select an instruction sequence that keeps heavily referenced data in the registers as long as possible.

The program is largely unaware of the memory hierarchy. In fact, the program only explicitly controls access to main memory and those components of the memory hierarchy at the file storage level and below (since manipulating files is a program-specific operation). In particular, cache access and virtual memory operation are generally transparent to the program. That is, access to these levels of the memory hierarchy usually take place without any intervention on the program's part. The program just accesses main memory and the hardware (and operating system) take care of the rest.

Of course, if the program really accessed main memory on each access, the program would run quite slowly since modern DRAM main memory subsystems are much slower than the CPU. The job of the cache memory subsystems (and the cache controller) is to move data between main memory and the cache so that the CPU can quickly access data in the cache. Likewise, if data is not available in main memory, but is available in slower virtual memory, the virtual memory subsystem is responsible for moving the data from hard disk to main memory (and then the caching subsystem may move the data from main memory to cache for even faster access by the CPU).

With few exceptions, most transparent memory subsystem accesses always take place between one level of the memory hierarchy and the level immediately below or above it. For example, the CPU rarely accesses main memory directly. Instead, when the CPU requests data from memory, the Level One Cache subsystem takes over. If the requested data is in the cache, then the Level One Cache subsystem returns the data and that's the end of the memory access. On the other hand if the data is not present in the level one cache, then

it passes the request on down to the Level Two Cache subsystem. If the Level Two Cache subsystem has the data, it returns this data to the Level One Cache, which then returns the data to the CPU. Note that requests for this same data in the near future will come from the Level One Cache rather than the Level Two Cache since the Level One Cache now has a copy of the data.

If neither the Level One nor Level Two Cache subsystems have a copy of the data, then the memory subsystem goes to main memory to get the data. If found in main memory, then the memory subsystems copy this data to the Level Two Cache which passes it to the Level One Cache which gives it to the CPU. Once again, the data is now in the Level One Cache, so any references to this data in the near future will come from the Level One Cache.

If the data is not present in main memory, but is present in Virtual Memory on some storage device, the operating system takes over, reads the data from disk (or other devices, such as a network storage server) and places this data in main memory. Main memory then passes this data through the caches to the CPU.

Because of locality of reference, the largest percentage of memory accesses take place in the Level One Cache system. The next largest percentage of accesses occur in the Level Two Cache subsystems. The most infrequent accesses take place in Virtual Memory.

6.4 Relative Performance of Memory Subsystems

If you take another look at Figure 6.1 you'll notice that the speed of the various levels increases at the higher levels of the memory hierarchy. A good question to ask, and one we'll hope to answer in this section, is "how much faster is each successive level in the memory hierarchy?" It actually ranges from "almost no difference" to "four orders of magnitude" as you'll seem momentarily.

Registers are, unquestionably, the best place to store data you need to access quickly. Accessing a register never requires any extra time³. Further, instructions that access data can almost always access that data in a register. Such instructions already encode the register "address" as part of the MOD-REG-R/M byte (see "Encoding Instruction Operands" on page 290). Therefore, it never takes any extra bits in an instruction to use a register. Instructions that access memory often require extra bytes (i.e., displacement bytes) as part of the instruction encoding. This makes the instruction longer which means fewer of them can sit in the cache or in a prefetch queue. Hence, the program may run slower if it uses memory operands more often than register operands simply due to the instruction size difference.

If you read Intel's instruction timing tables, you'll see that they claim that an instruction like "mov(someVar, ecx);" is supposed to run as fast as an instruction of the form "mov(ebx, ecx);" However, if you read the fine print, you'll find that they make several assumptions about the former instruction. First, they assume that *someVar*'s value is present in the level one cache memory. If it is not, then the cache controller needs to look in the level two cache, in main memory, or worse, on disk in the virtual memory subsystem. All of a sudden, this instruction that should execute in one cycle (e.g., one nanosecond on a one gigahertz processor) requires several milliseconds to execution. That's over six orders of magnitude difference, if you're counting. Now granted, locality of reference suggests that future accesses to this variable will take place in one cycle. However, if you access *someVar*'s value one million times immediately thereafter, the average access time of each instruction will be two cycles because of the large amount of time needed to access *someVar* the very first time (when it was on a disk in the virtual memory system). Now granted, the likelihood that some variable will be on disk in the virtual memory subsystem is quite low. But there is a three orders of magnitude difference in performance between the level one cache subsystem and the main memory subsystem. So if the program has to bring in the data from main memory, 999 accesses later you're still paying an average cost of two cycles for the instruction that Intel's documentation claims should execute in one cycle. Note that register accesses never suffer from this problem. Hence, register accesses are much faster.

3. Okay, strictly speaking this is not true. However, we'll ignore data hazards in this discussion and assume that the programmer or compiler has scheduled their instructions properly to avoid pipeline stalls due to data hazards with register data.

The difference between the level one and level two cache systems is not so dramatic. Usually, a level two caching subsystem introduces between one and eight wait states (see “Wait States” on page 151). The difference is usually much greater, though, if the secondary cache is not packaged together with the CPU.

On a one gigahertz processor the level one cache must respond within one nanosecond if the cache operates with zero wait states (note that some processors actually introduce wait states in accesses to the level one cache, but system designers try not to do this). Accessing data in the level two cache is always slower than in the level one cache and there is always the equivalent of at least one wait state, perhaps more, when accessing data in the level two cache. The reason is quite simple – it takes the CPU time to determine that the data it is seeking is not in the L1 (level one) cache; by the time it determines that the data is not present, the memory access cycle is nearly complete and there is no time to access the data in the L2 (level two) cache.

It may also be that the L2 cache is slower than the L1 cache. This is usually done in order to make the L2 cache less expensive. Also, larger memory subsystems tend to be slower than smaller ones, and L2 caches are usually 16 to 64 times larger than the L1 cache, hence they are usually slower as well. Finally, because L2 caches are not usually on the same silicon chip as the CPU, there are some delays associated with getting data in and out of the cache. All this adds up to additional wait states when accessing data in the L2 cache. As noted above, the L2 cache can be as much as an order of magnitude slower than the L1 cache.

Another difference between the L1 and L2 caches is the amount of data the system fetches when there is an L1 cache miss. When the CPU fetches data from the L1 cache, it generally fetches (or writes) only the data requested. If you execute a "mov(al, memory);" instruction, the CPU writes only a single byte to the cache. Likewise, if you execute "mov(mem32, eax);" then the CPU reads 32 bits from the L1 cache. Access to memory subsystems below the L1 cache, however, do not work in small chunks like this. Usually, memory subsystems read blocks (or *cache lines*) of data whenever accessing lower levels of the memory hierarchy. For example, if you execute the "mov(mem32, eax);" instruction and *mem32*'s value is not in the L1 cache, the cache controller doesn't simply read *mem32*'s value from the L2 cache (assuming it's present there). Instead, the cache controller will actually read a block of bytes (generally 16, 32, or 64 bytes, this depends on the particular processor) from the lower memory levels. The hope is that spatial locality exists and reading a block of bytes will speed up accesses to adjacent objects in memory⁴. The bad news, however, is that the "mov(mem32, eax);" instruction doesn't complete until the L1 cache reads the entire cache line (of 16, 32, 64, etc., bytes) from the L2 cache. Although the program may amortize the cost of reading this block of bytes over future accesses to adjacent memory locations, there is a large passage of time between the request for *mem32* and the actual completion of the "mov(mem32, eax);" instruction. This excess time is known as latency. As noted, the hope is that extra time will be worth the cost when future accesses to adjacent memory locations occur; however, if the program does not access memory objects adjacent to *mem32*, this latency is lost time.

A similar performance gulf separates the L2 cache and main memory. Main memory is typically an order of magnitude slower than the L2 cache. Again the L2 cache reads data from main memory in blocks (cache lines) to speed up access to adjacent memory elements.

There is a three to four order of magnitude difference in performance between standard DRAM and disk storage. To overcome this difference, there is usually a two to three orders of magnitude difference in size between the L2 cache and the main memory. In other words, the idea is "if the access time difference between main memory and virtual memory is two orders of magnitude greater than the difference between the L2 cache and main memory, then we'd better make sure we have two orders of magnitude more main memory than we have L2 cache." This keeps the performance loss to a reasonable level since we access virtual memory on disk two orders of magnitude less often.

We will not consider the performance of the other memory hierarchy subsystems since they are more or less under programmer control (their access is not automatic by the CPU or operating system). Hence, very little can be said about how frequently a program will access them.

4. Note that reading a block of *n* bytes is much faster than *n* reads of one byte. So this scheme is many times faster if spatial locality does occur in the program. For information about spatial locality, see “Cache Memory” on page 153.

6.5 Cache Architecture

Up to this point, cache has been this magical place that automatically stores data when we need it, perhaps fetching new data as the CPU requires it. However, a good question is "how exactly does the cache do this?" Another might be "what happens when the cache is full and the CPU is requesting additional data not in the cache?" In this section, we'll take a look at the internal cache organization and try to answer these questions along with a few others.

The basic idea behind a cache is that a program only access a small amount of data at a given time. If the cache is the same size as the typical amount of data the program access at any one given time, then we can put that data into the cache and access most of the data at a very high speed. Unfortunately, the data rarely sits in contiguous memory locations; usually, there's a few bytes here, a few bytes there, and some bytes somewhere else. In general, the data is spread out all over the address space. Therefore, the cache design has got to accommodate the fact that it must map data objects at widely varying addresses in memory.

As noted in the previous section, cache memory is not organized as a group of bytes. Instead, cache organization is usually in blocks of cache lines with each line containing some number of bytes (typically a small number that is a power of two like 16, 32, or 64), see Figure 6.2.

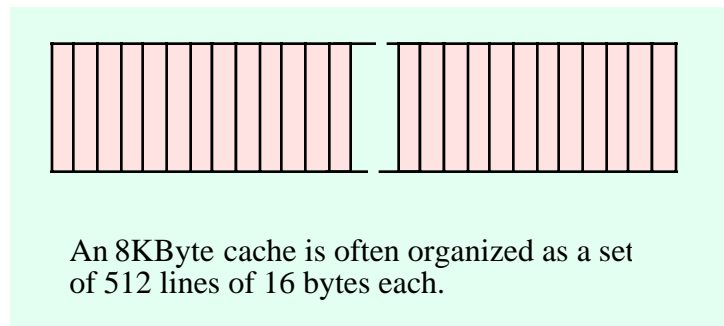


Figure 6.2 Possible Organization of an 8 Kilobyte Cache

The idea of a cache system is that we can attach a different (non-contiguous) address to each of the cache lines. So cache line #0 might correspond to addresses \$10000..\$1000F and cache line #1 might correspond to addresses \$21400..\$2140F. Generally, if a cache line is n bytes long (n is usually some power of two) then that cache line will hold n bytes from main memory that fall on an n -byte boundary. In this example, the cache lines are 16 bytes long, so a cache line holds blocks of 16 bytes whose addresses fall on 16-byte boundaries in main memory (i.e., the L.O. four bits of the address of the first byte in the cache line are always zero).

When the cache controller reads a cache line from a lower level in the memory hierarchy, a good question is "where does the data go in the cache?" The most flexible cache system is the *fully associative cache*. In a fully associative cache subsystem, the caching controller can place a block of bytes in any one of the cache lines present in the cache memory. While this is a very flexible system, the flexibility is not without cost. The extra circuitry to achieve full associativity is expensive and, worse, can slow down the memory subsystem. Most L1 and L2 caches are not fully associative for this reason.

At the other extreme is the *direct mapped cache* (also known as the *one-way set associative cache*). In a direct mapped cache, a block of main memory is always loaded into the same cache line in the cache. Generally, some number of bits in the main memory address select the cache line. For example, Figure 6.3 shows how the cache controller could select a cache line for an 8 Kilobyte cache with 16-byte cache lines and a 32-bit main memory address. Since there are 512 cache lines, this example uses bits four through twelve to select one of the cache lines (bits zero through three select a particular byte within the 16-byte cache line). The direct-mapped cache scheme is very easy to implement. Extracting nine (or some other

number of) bits from the address and using this as an index into the array of cache lines is trivial and fast. However, direct-mapped caches to suffer from some other problems.

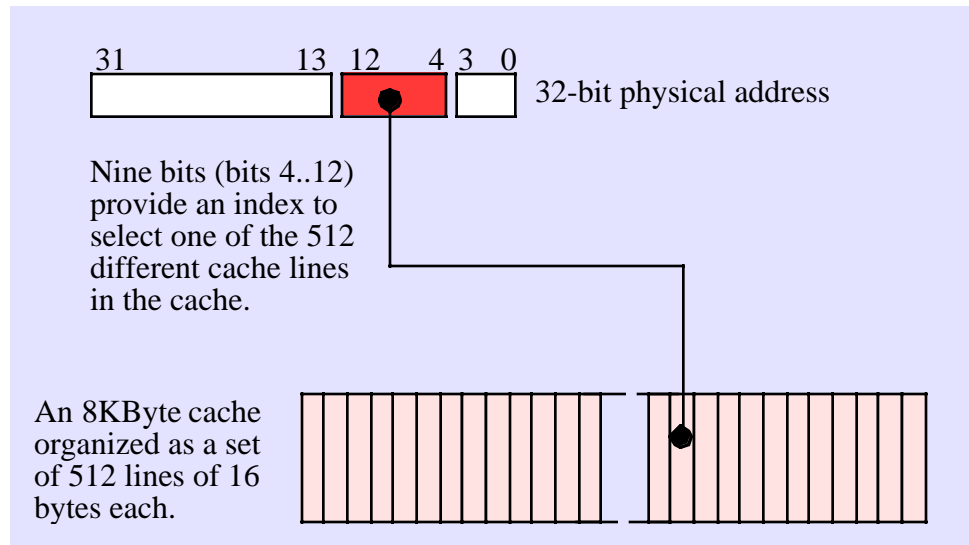


Figure 6.3 Selecting a Cache Line in a Direct-mapped Cache

Perhaps the biggest problem with a direct-mapped cache is that it may not make effective use of all the cache memory. For example, the cache scheme in Figure 6.3 maps address zero to cache line #0. It also maps address \$2000 (8K), \$4000 (16K), \$6000 (24K), \$8000 (32K), and, in fact, it maps every address that is an even multiple of eight kilobytes to cache line #0. This means that if a program is constantly accessing data at addresses that are even multiples of 8K and not accessing any other locations, the system will only use cache line #0, leaving all the other cache lines unused. Each time the CPU requests data at an address that is not at an address within cache line #0, the CPU will have to go down to a lower level in the memory hierarchy to access the data. In this pathological case, the cache is effectively limited to the size of one cache line. Had we used a fully associative cache organization, each access (up to 512 cache lines' worth) could have their own cache line, thus improving performance.

If a fully associative cache organization is too complex, expensive, and slow to implement, but a direct-mapped cache organization isn't as good as we'd like, one might ask if there is a compromise that gives us more capability than a direct-mapped approach without all the complexity of a fully associative cache. The answer is yes, we can create an *n-way set associative cache* which is a compromise between these two extremes. The idea here is to break up the cache into sets of cache lines. The CPU selects a particular set using some subset of the address bits, just as for direct-mapping. Within each set there are *n* cache lines. The caching controller uses a fully associative mapping algorithm to select one of the *n* cache lines within the set.

As an example, an 8 kilobyte two-way set associative cache subsystem with 16-byte cache lines organizes the cache as a set of 256 sets with each set containing two cache lines ("two-way" means each set contains two cache lines). Eight bits from the memory address select one of these 256 different sets. Then the cache controller can map the block of bytes to either cache line within the set (see Figure 6.4). The advantage of a two-way set associative cache over a direct mapped cache is that you can have two accesses on 8 Kilobyte boundaries (using the current example) and still get different cache lines for both accesses. However, once you attempt to access a third memory location at an address that is an even multiple of eight kilobytes you will have a conflict.

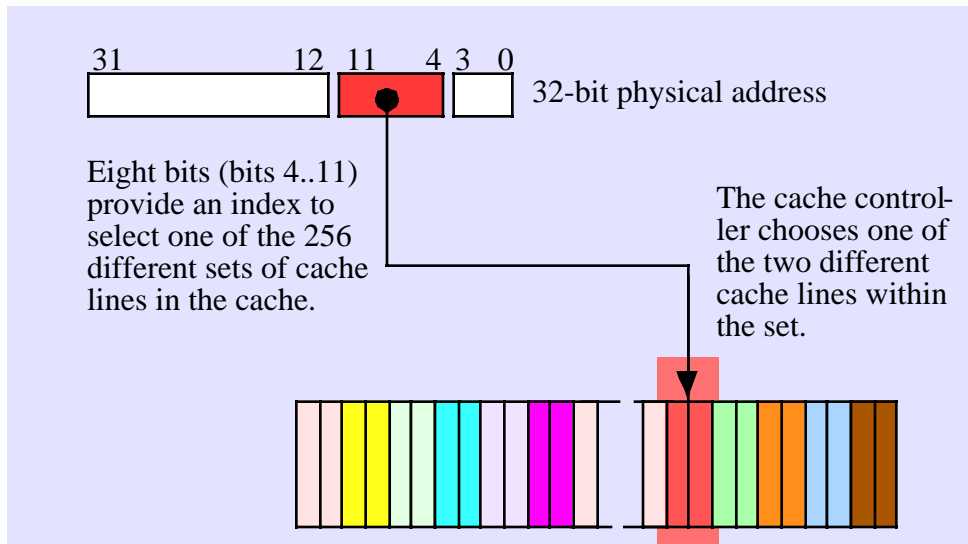


Figure 6.4 A Two-Way Set Associative Cache

A two-way set associative cache is much better than a direct-mapped cache and considerably less complex than a fully associative cache. However, if you're still getting too many conflicts, you might consider using a four-way set associative cache. A four-way set associative cache puts four associative cache lines in each block. In the current 8K cache example, a four-way set associative example would have 128 sets with each set containing four cache lines. This would allow up to four accesses to an address that is an even multiple of eight kilobytes before a conflict would occur.

Obviously, we can create an arbitrary m -way set associative cache (well, m does have to be a power of two). However, if m is equal to n , where n is the number of cache lines, then you've got a fully associative cache with all the attendant problems (complexity and speed). Most cache designs are direct-mapped, two-way set associative, or four-way set associative. The 80x86 family CPUs use all three (depending on the CPU and cache).

Although this section has made direct-mapped cache look bad, they are, in fact, very effective for many types of data. In particular, they are very good for data that you access in a sequential rather than random fashion. Since the CPU typically executes instructions in a sequential fashion, instructions are a good thing to put into a direct-mapped cache. Data access is probably a bit more random access, so a two-way or four-way set associative cache probably makes a better choice.

Because access to data and instructions is different, many CPU designers will use separate caches for instructions and data. For example, the CPU designer could choose to implement an 8K instruction cache and an 8K data cache rather than a 16K unified cache. The advantage is that the CPU designer could choose a more appropriate caching scheme for instructions versus data. The drawback is that the two caches are now each half the size of a unified cache and you may get fewer cache misses from a unified cache. The choice of an appropriate cache organization is a difficult one and can only be made after analyzing lots of running programs on the target processor. How to choose an appropriate cache format is beyond the scope of this text, just be aware that it's not an easy choice you can make by reading some textbook.

Thus far, we've answered the question "where do we put a block of data when we read it into the cache?" An equally important question we ignored until now is "what happens if a cache line isn't available when we need to read data from memory?" Clearly, if all the lines in a set of cache lines contain data, we're going to have to replace one of these lines with the new data. The question is, "how do we choose the cache line to replace?"

For a direct-mapped (one-way set associative) cache architecture, the answer is trivial. We replace exactly the block that the memory data maps to in the cache. The cache controller replaces whatever data

was formerly in the cache line with the new data. Any reference to the old data will result in a cache miss and the cache controller will have to bring that data into the cache replacing whatever data is in that block at that time.

For a two-way set associative cache, the replacement algorithm is a bit more complex. Whenever the CPU references a memory location, the cache controller uses some number of the address bits to select the set that should contain the cache line. Using some fancy circuitry, the caching controller determines if the data is already present in one of the two cache lines in the set. If not, then the CPU has to bring the data in from memory. Since the main memory data can go into either cache line, somehow the controller has to pick one or the other. If either (or both) cache lines are currently unused, the selection is trivial: pick an unused cache line. If both cache lines are currently in use, then the cache controller must pick one of the cache lines and replace its data with the new data. Ideally, we'd like to keep the cache line that will be referenced first (that is, we want to replace the one whose next reference is later in time). Unfortunately, neither the cache controller nor the CPU is omniscient, they cannot predict which is the best one to replace. However, remember the principle of temporal locality (see "Cache Memory" on page 153): if a memory location has been referenced recently, it is likely to be referenced again in the very near future. A corollary to this is "if a memory location has not been accessed in a while, it is likely to be a long time before the CPU accesses it again." Therefore, a good replacement policy that many caching controllers use is the "least recently used" or LRU algorithm. The idea is to pick the cache line that was not most frequently accessed and replace that cache line with the new data. An LRU policy is fairly easy to implement in a two-way set associative cache system. All you need is a bit that is set to zero whenever the CPU accessing one cache line and set it to one when you access the other cache line. This bit will indicate which cache line to replace when a replacement is necessary. For four-way (and greater) set associative caches, maintaining the LRU information is a bit more difficult, which is one of the reasons the circuitry for such caches is more complex. Other possible replacement policies include First-in, First-out⁵ (FIFO) and random. These are easier to implement than LRU, but they have their own problems.

The replacement policies for four-way and n-way set associative caches are roughly the same as for two-way set associative caches. The major difference is in the complexity of the circuit needed to implement the replacement policy (see the comments on LRU in the previous paragraph).

Another problem we've overlooked in this discussion on caches is "what happens when the CPU writes data to memory?" The simple answer is trivial, the CPU writes the data to the cache. However, what happens when the cache line containing this data is replaced by incoming data? If the contents of the cache line is not written back to main memory, then the data that was written will be lost. The next time the CPU reads that data, it will fetch the original data values from main memory and the value written is lost.

Clearly any data written to the cache must ultimately be written to main memory as well. There are two common write policies that caches use: write-back and write-through. Interestingly enough, it is sometimes possible to set the write policy under software control; these aren't hardwired into the cache controller like most of the rest of the cache design. However, don't get your hopes up. Generally the CPU only allows the BIOS or operating system to set the cache write policy, your applications don't get to mess with this. However, if you're the one writing the operating system...

The write-through policy states that any time data is written to the cache, the cache immediately turns around and writes a copy of that cache line to main memory. Note that the CPU does not have to halt while the cache controller writes the data to memory. So unless the CPU needs to access main memory shortly after the write occurs, this writing takes place in parallel with the execution of the program. Still, writing a cache line to memory takes some time and it is likely that the CPU (or some CPU in a multiprocessor system) will want to access main memory during this time, so the write-through policy may not be a high performance solution to the problem. Worse, suppose the CPU reads and writes the value in a memory location several times in succession. With a write-through policy in place the CPU will saturate the bus with cache line writes and this will have a very negative impact on the program's performance. On the positive side, the write-through policy does update main memory with the new value as rapidly as possible. So if two different CPUs are communicating through the use of shared memory, the write-through policy is probably better because the second CPU will see the change to memory as rapidly as possible when using this policy.

5. This policy does exhibit some anomalies. These problems are beyond the scope of this chapter, but a good text on architecture or operating systems will discuss the problems with the FIFO replacement policy.

The second common cache write policy is the write-back policy. In this mode, writes to the cache are not immediately written to main memory; instead, the cache controller updates memory at a later time. This scheme tends to be higher performance because several writes to the same variable (or cache line) only update the cache line, they do not generate multiple writes to main memory.

Of course, at some point the cache controller must write the data in cache to memory. To determine which cache lines must be written back to main memory, the cache controller usually maintains a *dirty bit* with each cache line. The cache system sets this bit whenever it writes data to the cache. At some later time the cache controller checks this dirty bit to determine if it must write the cache line to memory. Of course, whenever the cache controller replaces a cache line with other data from memory, it must first write that cache line to memory if the dirty bit is set. Note that this increases the latency time when replacing a cache line. If the cache controller were able to write dirty cache lines to main memory while no other bus access was occurring, the system could reduce this latency during cache line replacement.

A cache subsystem is not a panacea for slow memory access. In order for a cache system to be effective the software must exhibit locality of reference. If a program accesses memory in a random fashion (or in a fashion guaranteed to exploit the caching controller's weaknesses) then the caching subsystem will actually cause a big performance drop. Fortunately, real-world programs do exhibit locality of reference, so most programs will benefit from the presence of a cache in the memory subsystem.

Another feature to the cache subsystem on modern 80x86 CPUs is that the cache automatically handles many misaligned data references. As you may recall from an earlier chapter, there is a penalty for accesses larger data objects (words or dwords) at an address that is not an even multiple of that object's size. As it turns out, by providing some fancy logic, Intel's designers have eliminated this penalty as long as the data access is completely within a cache line. Therefore, accessing a word or double word at an odd address does not incur a performance penalty as long as the entire object lies within the same cache line. However, if the object crosses a cache line, then there will be a performance penalty for the memory access.

6.6 Virtual Memory, Protection, and Paging

In a modern operating system such as Linux or Windows, it is very common to have several different programs running concurrently in memory. This presents several problems. First, how do you keep the programs from interfering with one another? Second, if one program expects to load into memory at address \$1000 and a second program also expects to load into memory at address \$1000, how can you load and execute both programs at the same time? One last question we might ask is what happens if our computer has 64 megabytes of memory and we decide to load and execute three different applications, two of which require 32 megabytes and one that requires 16 megabytes (not to mention the memory the operating system requires for its own purposes)? The answer to all these questions lies in the virtual memory subsystem the 80x86 processors support⁶.

Virtual memory on the 80x86 gives each process its own 32-bit address space⁷. This means that address \$1000 in one program is physically different than address \$1000 in a separate program. The 80x86 achieves this sleight of hand by using *paging* to remap *virtual addresses* within one program to different *physical addresses* in memory. A virtual address is the memory address that the program uses. A physical address is the bit pattern that actually appears on the CPU's address bus. The two don't have to be the same (and usually, they aren't). For example, program #1's virtual address \$1000 might actually correspond to physical address \$215000 while program #2's virtual address \$1000 might correspond to physical memory address \$300000. How can the CPU do this? Easy, by using *paging*.

6. Actually, virtual memory is really only supported by the 80386 and later processors. We'll ignore this issue here since most people have an 80386 or later processor.

7. Strictly speaking, you actually get a 36-bit address space on Pentium Pro and later processors, but Windows and Linux limits you to 32-bits so we'll use that limitation here.

The concept behind paging is quite simple. First, you break up memory into blocks of bytes called pages. A page in main memory is comparable to a cache line in a cache subsystem, although pages are usually much larger than cache lines. For example, the 80x86 CPUs use a page size of 4,096 bytes.

After breaking up memory into pages, you use a lookup table to translate the H.O. bits of a virtual address to select a page; you use the L.O. bits of the virtual address as an index into the page. For example, with a 4,096-byte page, you'd use the L.O. 12 bits of the virtual address as the offset within the page in physical memory. The upper 20 bits of the address you would use as an index into a lookup table that returns the actual upper 20 bits of the physical address (see Figure 6.5).

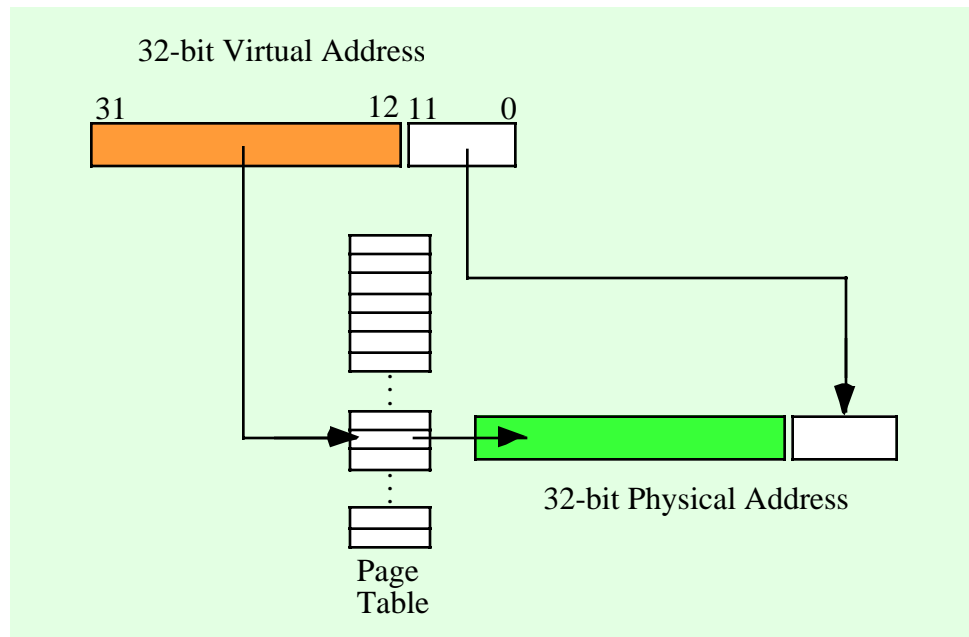


Figure 6.5 Translating a Virtual Address to a Physical Address

Of course, a 20-bit index into the page table would require over one million entries in the page table. If each entry is 32 bits (20 bits for the offset plus 12 bits for other purposes), then the page table would be four megabytes long. This would be larger than most of the programs that would run in memory! However, using what is known as a multi-level page table, it is very easy to create a page table that is only 8 kilobytes long for most small programs. The details are unimportant here, just rest assured that you don't need a four megabyte page table unless your program consumes the entire four gigabyte address space.

If you study Figure 6.5 for a few moments, you'll probably discover one problem with using a page table – it requires two memory accesses in order to access an address in memory: one access to fetch a value from the page table and one access to read or write the desired memory location. To prevent cluttering the data (or instruction) cache with page table entries (thus increasing the number of cache misses), the page table uses its own cache known as the *Translation Lookaside Buffer*, or TLB. This cache typically has 32 entries on a Pentium family processor. This provides a sufficient lookup capability to handle 128 kilobytes of memory (32 pages) without a miss. Since a program typically works with less data than this at any given time, most page table accesses come from the cache rather than main memory.

As noted, each entry in the page table is 32 bits even though the system really only needs 20 bits to remap the addresses. Intel uses some of the remaining 12 bits to provide some memory protection information. For example, one bit marks whether a page is read/write or read-only. Another bit determines if you can execute code on that page. Some bits determine if the application can access that page or if only the operating system can do so. Some bits determine if the page is "dirty" (that is, if the CPU has written to the page) and whether the CPU has accessed the page recently (these bits have the same meaning as for cache

lines). Another bit determines whether the page is actually present in physical memory or if it's stored on secondary storage somewhere. Note that your applications do not have access to the page table, and therefore they cannot modify these bits. However, Windows does provide some functions you can call if you want to change certain bits in the page table (e.g., Windows will allow you to set a page to read-only if you want to do so). Linux users also have some memory mapping functions they can call to play around with the access bits.

Beyond remapping memory so multiple programs can coexist in memory even though they access the same virtual addresses, paging also provides a mechanism whereby the operating system can move infrequently used pages to secondary storage (i.e., a disk drive). Just as locality of reference applies to cache lines, it applies to pages in memory as well. At any one given time a program will only access a small percentage of the pages in memory that contain data and code (this set of pages is known as the *working set*). While this working set of pages varies (slowly) over time, for a reasonable time period the working set remains constant. Therefore, there is little need to have the remainder of the program in memory consuming valuable physical memory that some other process could be using. If the operating system can save those (currently unused) pages to disk, the physical memory they consume would be available for other programs that need it.

Of course, the problem with moving data out of physical memory is that sooner or later the program might actually need it. If you attempt to access a page of memory and the page table bit tells the MMU (memory management unit) that this page is not present in physical memory, then the CPU interrupts the program and passes control to the operating system. The operating system analyzes the memory access request and reads the corresponding page of data from the disk drive to some available page in memory. The process is nearly identical to that used by a fully associative cache subsystem except, of course, accessing the disk is much slower than main memory. In fact, you can think of main memory as a fully associative write-back cache with 4,096 byte cache lines that caches the data on the disk drive. Placement and replacement policies and other issues are very similar to those we've discussed for caches. Discussing how the virtual memory subsystem works beyond equating it to a cache is well beyond the scope of this text. If you're interested, any decent text on operating system design will explain how a virtual memory subsystem swaps pages between main memory and the disk. Our main goal here is to realize that this process takes place in operating systems like Linux or Windows and that accessing the disk is very slow.

One important issue resulting from the fact that each program has a separate page table and the programs themselves don't have access to the page table is that programs cannot interfere with the operation of other programs by overwriting those other program's data (assuming, of course, that the operating system is properly written). Further, if your program crashes by overwriting itself, it cannot crash other programs at the same time. This is a big benefit of a paging memory system.

Note that if two programs want to cooperate and share data, they can do so. All they've got to do is to tell the operating system that they want to share some blocks of memory. The operating system will map their corresponding virtual addresses (of the shared memory area) to the same physical addresses in memory. Under Windows, you can achieve this use *memory mapped files*; see the operating system documentation for more details. Linux also supports memory mapped files as well as some special shared memory operations; again, see the OS documentation for more details.

6.7 Thrashing

Thrashing is a degenerate case that occurs when there is insufficient memory at one level in the memory hierarchy to properly contain the working set required by the upper levels of the memory hierarchy. This can result in the overall performance of the system dropping to the speed of a lower level in the memory hierarchy. Therefore, thrashing can quickly reduce the performance of the system to the speed of main memory or, worse yet, the speed of the disk drive.

There are two primary causes of thrashing: (1) insufficient memory at a given level in the memory hierarchy, and (2) the program does not exhibit locality of reference. If there is insufficient memory to hold a working set of pages or cache lines, then the memory system is constantly replacing one block (cache line or page) with another. As a result, the system winds up operating at the speed of the slower memory in the hierarchy. A common example occurs with virtual memory. A user may have several applications running at the

same time and the sum total of these programs' working sets is greater than all of physical memory available to the program. As a result, as the operating system switches between the applications it has to copy each application's data to and from disk and it may also have to copy the code from disk to memory. Since a context switch between programs is often much faster than retrieving data from the disk, this slows the programs down by a tremendous factor since thrashing slows the context switch down to the speed of swapping the applications to and from disk.

If the program does not exhibit locality of reference and the lower memory subsystems are not fully associative, then thrashing can occur even if there is free memory at the current level in the memory hierarchy. For example, suppose an eight kilobyte L1 caching system uses a direct-mapped cache with 16-byte cache lines (i.e., 512 cache lines). If a program references data objects 8K apart on each access then the system will have to replace the same line in the cache over and over again with each access. This occurs even though the other 511 cache lines are currently unused.

If insufficient memory is the cause of thrashing, an easy solution is to add more memory (if possible, it is rather hard to add more L1 cache when the cache is on the same chip as the processor). Another alternative is to run fewer processes concurrently or modify the program so that it references less memory over a given time period. If lack of locality of reference is causing the problem, then you should restructure your program and its data structures to make references local to one another.

6.8 NUMA and Peripheral Devices

Although most of the RAM memory in a system is based on high-speed DRAM interfaced directly to the processor's bus, not all memory is connected to the CPU in this manner. Sometimes a large block of RAM is part of a peripheral device and you communicate with that device by writing data to the RAM on the peripheral. Video display cards are probably the most common example, but some network interface cards and USB controllers also work this way (as well as other peripherals). Unfortunately, the access time to the RAM on these peripheral devices is often much slower than access to normal memory. We'll call such access NUMA⁸ access to indicate that access to such memory isn't uniform (that is, not all memory locations have the same access times). In this section we'll use the video card as an example, although NUMA performance applies to other devices and memory technologies as well.

A typical video card interfaces to the CPU via the AGP or PCI (or much worse, ISA) bus inside the computer system. The PCI bus nominally runs at 33 MHz and is capable of transferring four bytes per bus cycle. In burst mode, a video controller card, therefore, is capable of transferring 132 megabytes per second (though few would ever come close to achieving this for technical reasons). Now compare this with main memory access. Main memory usually connects directly to the CPU's bus and modern CPUs have a 400 MHz 64-bit wide bus. Technically (if memory were fast enough), the CPU's bus could transfer 800 MBytes/sec. between memory and the CPU. This is six times faster than transferring data across the PCI bus. Game programmers long ago discovered that it's much faster to manipulate a copy of the screen data in main memory and only copy that data to the video display memory when a vertical retrace occurs (about 60 times/sec.). This mechanism is much faster than writing directly to the video memory every time you want to make a change.

Unlike caches and the virtual memory subsystem that operate in a transparent fashion, programs that write to NUMA devices must be aware of this and minimize the accesses whenever possible (e.g., by using an off-screen bitmap to hold temporary results). If you're actually storing and retrieving data on a NUMA device, like a Flash memory card, then you must explicitly cache the data yourself. Later in this text you'll learn about hash tables and searching. Those techniques will help you create your own caching system for NUMA devices.

8. Remember, NUMA stands for NonUniform Memory Access.

6.9 Segmentation

Segmentation is another memory management scheme, like paging, that provides memory protection and virtual memory capabilities. Linux and Windows do not support the use of segments, nor does HLA provide any instructions that let you manipulate segment registers or use segment override prefixes on an instruction⁹. These 32-bit operating systems employ the flat memory model that, essentially, ignores segments on the 80x86. Furthermore, the remainder of this text also ignores segmentation. What this means is that you don't really need to know anything about segmentation in order to write assembly language programs that run under modern OSes. However, it's unthinkable to write a book on 80x86 assembly language programming that doesn't at least mention segmentation. Hence this section.

The basic idea behind the segmentation model is that memory is managed using a set of segments. Each segment is, essentially, its own address space. A segment consists of two components: a base address that contains the address of some physical memory location and a length value that specifies the length of the segment. A segmented address also consists of two components: a segment selector and an offset into the segment. The segment selector specifies the segment to use (that is, the base address and length values) while the offset component specifies the offset from the base address for the actual memory access. The physical address of the actual memory location is the sum of the offset and the base address values. If the offset exceeds the length of the segment, the system generates a protection violation.

Segmentation on the 80x86 got a (deservedly) bad name back in the days of the 8086, 8088, and 80286 processors. The problem back then is that the offset into the segment was only a 16-bit value, effectively limiting segments to 64 kilobytes in length. By creating multiple segments in memory it was possible to address more than 64K within a single program; however, it was a major pain to do so, especially if a single data object exceeded 64 kilobytes in length. With the advent of the 80386, Intel solved this problem (and others) with their segmentation model. By then, however, the damage had been done; segmentation had developed a really bad name that it still bears to this day.

Segments are an especially powerful memory management system when a program needs to manipulate different variable sized objects and the program cannot determine the size of the objects before run time. For example, suppose you want to manipulate several different files using the memory mapped file scheme. Under Windows or Linux, which don't support segmentation, you have to specify the maximum size of the file before you map it into memory. If you don't do this, then the operating system can't leave sufficient space at the end of the first file in memory before the second file starts. On the other hand, if the operating system supported segmentation, it could easily return segmented pointers to these two memory mapped files, each in their own logical address space. This would allow the files to grow to the size of the maximum offset within a segment (or the maximum file size, whichever is smaller). Likewise, if two programs wanted to share some common data, a segmented system could allow the two programs to put the shared data in a segment. This would allow both programs to reference objects in the shared area using like-valued pointer (offset) values. This makes it easier to pass pointer data (within the shared segment) between the two programs, a very difficult thing to do when using a flat memory model without segmentation as Linux and Windows currently do.

One of the more interesting features of the 80386 and later processors is the fact that Intel combined both segmentation and paging in the same memory management unit. Prior to the 80386 most real-world CPUs used paging or segmentation but not both. The 80386 processor merged both of these memory management mechanisms into the same chip, offering the advantages of both systems on a single chip. Unfortunately, most 32-bit operating systems (e.g., Linux and Windows) fail to take advantage of segmentation so this feature goes wasted on the chip.

6.10 .text.text Putting it All Together

CPU architects divide memory into several different types depending on cost, capacity, and speed. They call this the memory hierarchy. Many of the levels in the memory hierarchy are transparent to the program-

9. Though you could easily create macros to do this.

mer. That is, the system automatically moves data between levels in the memory hierarchy without intervention on the programmer's part. However, if you are aware of the effects of the memory hierarchy on program performance, you can write faster programs by organizing your data and code so that it conforms to the expectations of the caching and virtual memory subsystems in the memory hierarchy.

7.1 Chapter Overview

A typical program does three basic activities: input, computation, and output. In this section we will discuss the other two activities beyond computation: input and output or I/O. This chapter concentrates on low-level CPU I/O rather than high level file or character I/O. This chapter discusses how the CPU transfers bytes of data to and from the outside world. This chapter discusses the mechanisms and performance issues behind the I/O.

7.2 Connecting a CPU to the Outside World

Most I/O devices interface to the CPU in a fashion quite similar to memory. Indeed, many devices appear to the CPU as though they were memory devices. To output data to the outside world the CPU simply stores data into a "memory" location and the data magically appears on some connectors external to the computer. Similarly, to input data from some external device, the CPU simply transfers data from a "memory" location into the CPU; this "memory" location holds the value found on the pins of some external connector.

An output port is a device that looks like a memory cell to the computer but contains connections to the outside world. An I/O port typically uses a latch rather than a flip-flop to implement the memory cell. When the CPU writes to the address associated with the latch, the latch device captures the data and makes it available on a set of wires external to the CPU and memory system (see Figure 7.1). Note that output ports can be write-only, or read/write. The port in Figure 7.1, for example, is a write-only port. Since the outputs on the latch do not loop back to the CPU's data bus, the CPU cannot read the data the latch contains. Both the address decode and write control lines must be active for the latch to operate; when reading from the latch's address the decode line is active, but the write control line is not.

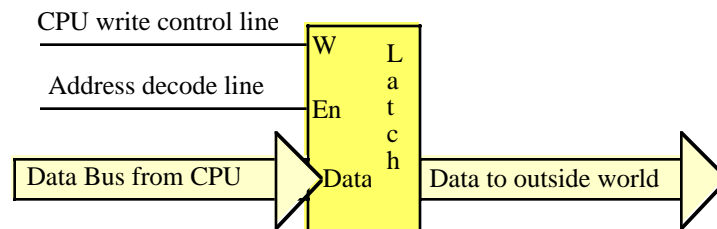


Figure 7.1 A Typical Output Port

Figure 7.2 shows how to create a read/write input/output port. The data written to the output port loops back to a transparent latch. Whenever the CPU reads the decoded address the read and decode lines are active and this activates the lower latch. This places the data previously written to the output port on the CPU's data bus, allowing the CPU to read that data. A read-only (input) port is simply the lower half of Figure 7.2; the system ignores any data written to an input port.

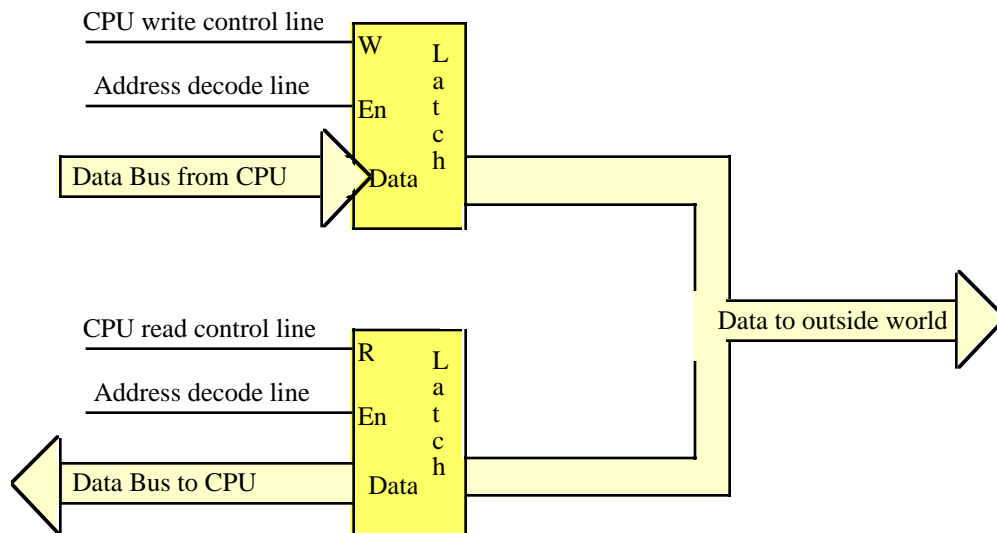


Figure 7.2 An Output Port that Supports Read/Write Access

Note that the port in Figure 7.2 is not an input port. Although the CPU can read this data, this port organization simply lets the CPU read the data it previously wrote to the port. The data appearing on an external connector is an output port (only). One could create a (read-only) input port by using the lower half of the circuit in Figure 7.2. The input to the latch would appear on the CPU's data bus whenever the CPU reads the latch data.

A perfect example of an output port is a parallel printer port. The CPU typically writes an ASCII character to a byte-wide output port that connects to the DB-25F connector on the back of the computer's case. A cable transmits this data to the printer where an input port (to the printer) receives the data. A processor inside the printer typically converts this ASCII character to a sequence of dots it prints on the paper.

Generally, a given peripheral device will use more than a single I/O port. A typical PC parallel printer interface, for example, uses three ports: a read/write port, an input port, and an output port. The read/write port is the data port (it is read/write to allow the CPU to read the last ASCII character it wrote to the printer port). The input port returns control signals from the printer; these signals indicate whether the printer is ready to accept another character, is off-line, is out of paper, etc. The output port transmits control information to the printer such as whether data is available to print.

The first thing to learn about the input/output subsystem is that I/O in a typical computer system is radically different than I/O in a typical high level programming language. In a real computer system you will rarely find machine instructions that behave like *writeln*, *cout*, *printf*, or even the HLA *stdin* and *stdout* statements. In fact, most input/output instructions behave exactly like the 80x86's MOV instruction. To send data to an output device, the CPU simply moves that data to a special memory location. To read data from an input device, the CPU simply moves data from the address of that device into the CPU. Other than there are usually more wait states associated with a typical peripheral device than actual memory, the input or output operation looks very similar to a memory read or write operation.

7.3 Read-Only, Write-Only, Read/Write, and Dual I/O Ports

We can classify input/output ports into four categories based on the CPU's ability to read and write data at a given port address. These four categories are read-only ports, write-only ports, read/write ports, and dual I/O ports.

A read-only port is (obviously) an input port. If the CPU can only read the data from the port, then that port is providing data appearing on lines external to the CPU. The system typically ignores any attempt to write data to a read-only port¹. A good example of a read-only port is the status port on a PC's parallel printer interface. Reading data from this port lets you test the current condition of the printer. The system ignores any data written to this port.

A write-only port is always an output port. Writing data to such a port presents the data for use by an external device. Attempting to read data from a write-only port generally returns garbage (i.e., whatever values that just happen to be on the data bus at that time). You generally cannot depend on the meaning of any value read from a write-only port.

A read/write port is an output port as far as the outside world is concerned. However, the CPU can read as well as write data to such a port. Whenever the CPU reads data from a read/write port, it reads the data that was last written to the port. Reading the port does not affect the data the external peripheral device sees, reading the port is a simple convenience for the programmer so that s/he doesn't have to save the value last written to the port should they want to retrieve the value.

A dual I/O port is also a read/write port, but reading the port reads data from some external device while writing data to the port transmits data to a different external device. Figure 7.3 shows how you could interface such a device to the system. Note that the input and output ports are actually a read-only and a write-only port that share the same address. Reading the address accesses one port while writing to the address accesses the other port. Essentially, this port arrangement uses the R/W control line(s) as an extra address bit when selecting these ports.

1. Note, however, that some devices may fail if you attempt to write to their corresponding input ports, so it's never a good idea to write data to a read-only port.

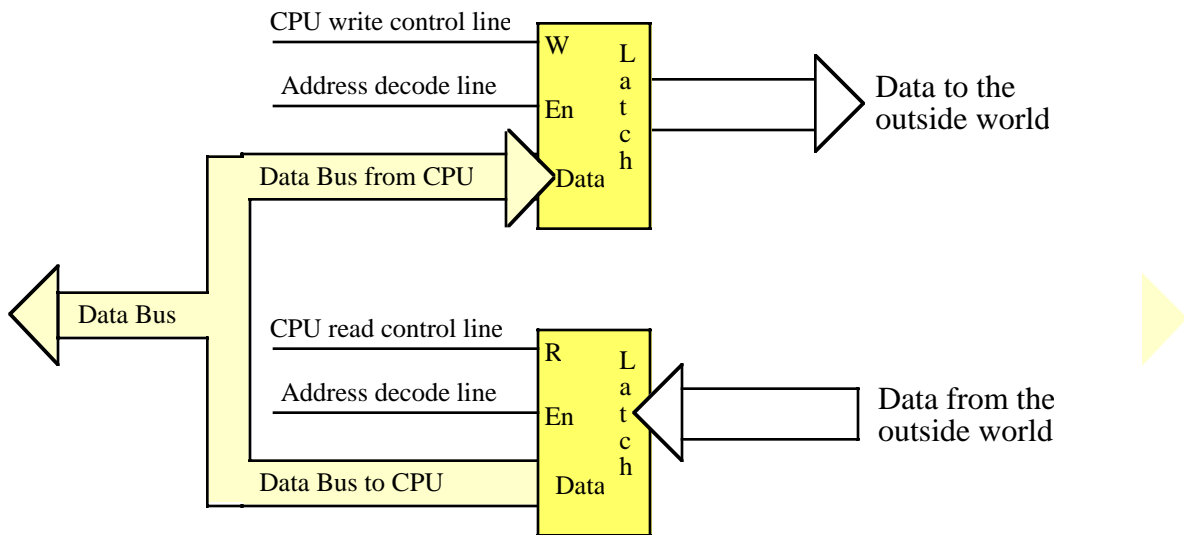


Figure 7.3 An Input and an Output Device That Share the Same Address (a Dual I/O Port)

These examples may leave you with the impression that the CPU always reads and writes data to peripheral devices using data on the data bus (that is, whatever data the CPU places on the data bus when it writes to an output port is the data actually written to that output port). While this is generally true for input ports (that is, the CPU transfers input data across the data bus when reading data from the input port), this isn't necessarily true for output ports. In fact, a very common output mechanism is simply accessing a port. Figure 7.4 provides a very simple example. In this circuit, an address decoder decodes two separate addresses. Any access (read or write) to the first address sets the output line high; any read or write of the second address clears the output line. Note that this circuit ignores the data on the CPU's data lines. It is not important whether the CPU reads or writes data to these addresses, nor is the data written of any consequence. The only thing that matters is that the CPU access one of these two addresses.

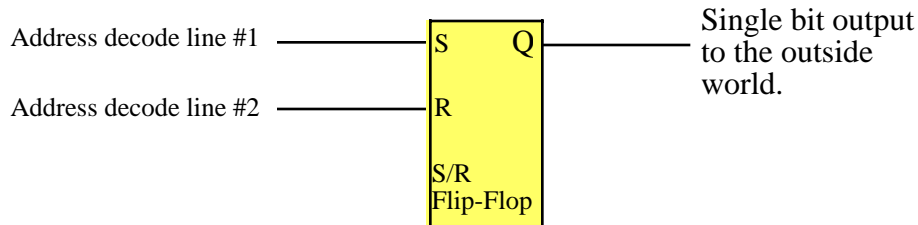


Figure 7.4 Outputting Data to a Port by Simply Accessing That Port

Another possible way to connect an output port to the CPU is to use a D flip-flop and connect the read/write status lines to the D input on the flip-flop. Figure 7.5 shows how you could design such a device. In this diagram any read of the selected port sets the output bit to zero while a write to this output port sets the output bit to one.

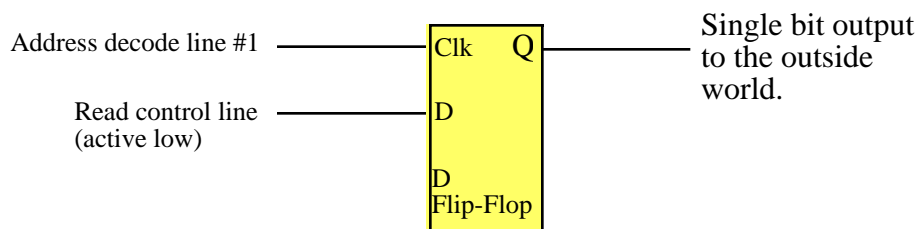


Figure 7.5 Outputting Data Using the Read/Write Control as the Data to Output

There are a wide variety of ways you can connect external devices to the CPU. This section only provides a few examples as a sampling of what is possible. In the real world, there are an amazing number of different ways that engineers connect external devices to the CPU. Unless otherwise noted, the rest of this chapter will assume that the CPU reads and writes data to an external device using the data bus. This is not to imply that this is the only type of I/O that one could use in a given example.

7.4 I/O (Input/Output) Mechanisms

There are three basic forms of input and output that a typical computer system will use: I/O-mapped I/O, memory-mapped I/O, and direct memory access (DMA). I/O-mapped input/output uses special instructions to transfer data between the computer system and the outside world; memory-mapped I/O uses special memory locations in the normal address space of the CPU to communicate with real-world devices; DMA is a special form of memory-mapped I/O where the peripheral device reads and writes data in memory without going through the CPU. Each I/O mechanism has its own set of advantages and disadvantages, we will discuss these in this section.

7.4.1 Memory Mapped Input/Output

A memory mapped peripheral device is connected to the CPU's address and data lines exactly like memory, so whenever the CPU reads or writes the address associated with the peripheral device, the CPU transfers data to or from the device. This mechanism has several benefits and only a few disadvantages.

The principle advantage of a memory-mapped I/O subsystem is that the CPU can use any instruction that accesses memory to transfer data between the CPU and a memory-mapped I/O device. The MOV instruction is the one most commonly used to send and receive data from a memory-mapped I/O device, but any instruction that reads or writes data in memory is also legal. For example, if you have an I/O port that is read/write, you can use the ADD instruction to read the port, add data to the value read, and then write data back to the port.

Of course, this feature is only usable if the port is a read/write port (or the port is readable and you've specified the port address as the source operand of your ADD instruction). If the port is read-only or write-only, an instruction that reads memory, modifies the value, and then writes the modified value back to memory will be of little use. You should use such read/modify/write instructions only with read/write ports (or dual I/O ports if such an operation makes sense).

Nevertheless, the fact that you can use any instruction that accesses memory to manipulate port data is often a big advantage since you can operate on the data with a single instruction rather than first moving the data into the CPU, manipulating the data, and then writing the data back to the I/O port.

The big disadvantage of memory-mapped I/O devices is that they consume addresses in the memory map. Generally, the minimum amount of space you can allocate to a peripheral (or block of related peripherals) is a four kilobyte page. Therefore, a few independent peripherals can wind up consuming a fair amount of the physical address space. Fortunately, a typical PC has only a couple dozen such devices, so this isn't much of a problem. However, some devices, like video cards, consume a large chunk of the address space (e.g., some video cards have 32 megabytes of on-board memory that they map into the memory address space).

7.4.2 I/O Mapped Input/Output

I/O-mapped input/output uses special instructions to access I/O ports. Many CPUs do not provide this type of I/O, though the 80x86 does. The Intel 80x86 family uses the IN and OUT instructions to provide I/O-mapped input/output capabilities. The 80x86 IN and OUT instructions behave somewhat like the MOV instruction except they transmit their data to and from a special I/O address space that is distinct from the memory address space. The IN and OUT instructions use the following syntax:

```
in( port, al ); // ... or AX or EAX, port is a constant in the range
out( al, port ); // 0..255.
```

```
in( dx, al ); // Or AX or EAX.
out( al, dx );
```

The 80x86 family uses a separate address bus for I/O transfers². This bus is only 16-bits wide, so the 80x86 can access a maximum of 65,536 different bytes in the I/O space. The first two instructions encode the port address as an eight-bit constant, so they're actually limited to accessing only the first 256 I/O addresses in this address space. This makes the instruction shorter (two bytes instead of three). Unfortunately, most of the interesting peripheral devices are at addresses above 255, so the first pair of instructions above are only useful for accessing certain on-board peripherals in a PC system.

To access I/O ports at addresses beyond 255 you must use the latter two forms of the IN and OUT instructions above. These forms require that you load the 16-bit I/O address into the DX register and use DX as a pointer to the specified I/O address. For example, to write a byte to the I/O address \$378³ you would use an instruction sequence like the following:

```
mov( $378, dx );
out( al, dx );
```

The advantage of an I/O address space is that peripheral devices mapped to this area do not consume space in the memory address space. This allows you to fully expand the memory address space with RAM or other memory. On the other hand, you cannot use arbitrary memory instructions to access peripherals in the I/O address space, you can only use the IN and OUT instructions.

Another disadvantage to the 80x86's I/O address space is that it is quite small. Although most peripheral devices only use a couple of I/O addresses (and most use fewer than 16 I/O addresses), a few devices, like video display cards, can occupy millions of different I/O locations (e.g., three bytes for each pixel on the screen). As

-
2. Physically, the I/O address bus is the same as the memory address bus, but additional control lines determine whether the address on the bus is accessing memory or an I/O device.
 3. This is typically the address of the data port on the parallel printer port.

noted earlier, some video display cards have 32 megabytes of dual-ported RAM on board. Clearly we cannot easily map this many locations into the 64K I/O address space.

7.4.3 Direct Memory Access

Memory-mapped I/O subsystems and I/O-mapped subsystems both require the CPU to move data between the peripheral device and main memory. For this reason, we often call these two forms of input/output programmed I/O. For example, to input a sequence of ten bytes from an input port and store these bytes into memory the CPU must read each value and store it into memory. For very high-speed I/O devices the CPU may be too slow when processing this data a byte (or word or double word) at a time. Such devices generally have an interface to the CPU's bus so they can directly read and write memory. This is known as direct memory access since the peripheral device accesses memory directly, without using the CPU as an intermediary. This often allows the I/O operation to proceed in parallel with other CPU operations, thereby increasing the overall speed of the system. Note, however, that the CPU and DMA device cannot both use the address and data buses at the same time. Therefore, concurrent processing only occurs if the CPU has a cache and is executing code and accessing data found in the cache (so the bus is free). Nevertheless, even if the CPU must halt and wait for the DMA operation to complete, the I/O is still much faster since many of the bus operations during I/O or memory-mapped input/output consist of instruction fetches or I/O port accesses which are not present during DMA operations.

A typical DMA controller consists of a pair of counters and other circuitry that interfaces with memory and the peripheral device. One of the counters serves as an address register. This counter supplies an address on the address bus for each transfer. The second counter specifies the number of transfers to complete. Each time the peripheral device wants to transfer data to or from memory, it sends a signal to the DMA controller. The DMA controller places the value of the address counter on the address bus. At the same time, the peripheral device places data on the data bus (if this is an input operation) or reads data from the data bus (if this is an output operation). After a successful data transfer, the DMA controller increments its address register and decrements the transfer counter. This process repeats until the transfer counter decrements to zero.

7.5 I/O Speed Hierarchy

Different devices have different data transfer rates. Some devices, like keyboards, are extremely slow (comparing their speed to CPU speeds). Other devices like disk drives can actually transfer data faster than the CPU can read it. The mechanisms for data transfer differ greatly based on the transfer speed of the device. Therefore, it makes sense to create some terminology to describe the different transfer rates of peripheral devices.

Low-speed devices are those that produce or consume data at a rate much slower than the CPU is capable of processing. For the purposes of discussion, we'll claim that low-speed devices operate at speeds that are two to three orders of magnitude (or more) slower than the CPU. *Medium-speed devices* are those that transfer data at approximately the same rate (within an order of magnitude slower, but never faster) than the CPU. *High-speed devices* are those that transfer data faster than the CPU is capable of moving data between the device and the CPU. Clearly, high-speed devices must use DMA since the CPU is incapable of transferring the data between the CPU and memory.

With typical bus architectures, modern day PCs are capable of one transfer per microsecond or better. Therefore, high-speed devices are those that transfer data more rapidly than once per microsecond. Medium-speed transfers are those that involve a data transfer every one to 100 microseconds. Low-speed devices usually trans-

fer data less often than once every 100 microseconds. The difference between these speeds will decide the mechanism we use for the I/O operation (e.g., high-speed transfers require the use of DMA or other techniques).

Note that one transfer per microsecond is not the same thing as a one megabyte per second data transfer rate. A peripheral device can actually transfer more than one byte per data transfer operation. For example, when using the "in(dx, eax);" instruction, the peripheral device can transfer four bytes in one transfer. Therefore, if the device is reaching one transfer per microsecond, then the device can transfer four megabytes per second. Likewise, a DMA device on a Pentium processor can transfer 64 bits at a time, so if the device completes one transfer per microsecond it will achieve an eight megabyte per second data transfer rate.

7.6 System Busses and Data Transfer Rates

Earlier in this text (see *The System Bus* on page 138) you saw that the CPU communicates to memory and I/O devices using the system bus. In that chapter you saw that a typical Von Neumann Architecture machine has three different busses: the address bus, the data bus, and the control bus. If you've ever opened up a computer and looked inside or read the specifications for a system, you've probably heard terms like *PCI*, *ISA*, *EISA*, or even *NuBus* mentioned when discussing the computer's bus. If you're familiar with these terms, you may wonder what their relationship is with the CPU's bus. In this section we'll discuss this relationship and describe how these different busses affect the performance of a system.

Computer system busses like PCI (Peripheral Connection Interface) and ISA (Industry Standard Architecture) are definitions for physical connectors inside a computer system. These definitions describe a set of signals, physical dimensions (i.e., connector layouts and distances from one another), and a data transfer protocol for connecting different electronic devices. These busses are related to the CPU's bus only insofar as many of the signals on one of the peripheral busses also appear on the CPU's bus. For example, all of the aforementioned busses provide lines for address, data, and control functions.

Peripheral interconnection busses do not necessarily mirror the CPU's bus. All of these busses contain several additional lines that are not present on the CPU's bus. These additional lines let peripheral devices communicate with one other directly (without having to go through the CPU or memory). For example, most busses provide a common set of interrupt control signals that let various I/O devices communicate directly with the system's interrupt controller (which is also a peripheral device). Nor does the peripheral bus always include all the signals found on the CPU's bus. For example, the ISA bus only supports 24 address lines whereas the Pentium IV supports 36 address lines. Therefore, peripherals on the ISA bus only have access to 16 megabytes of the Pentium IV's 64 gigabyte address range.

A typical modern-day PC supports the PCI bus (although some older systems also provide ISA connectors). The organization of the PCI and ISA busses in a typical computer system appears in Figure 7.6.

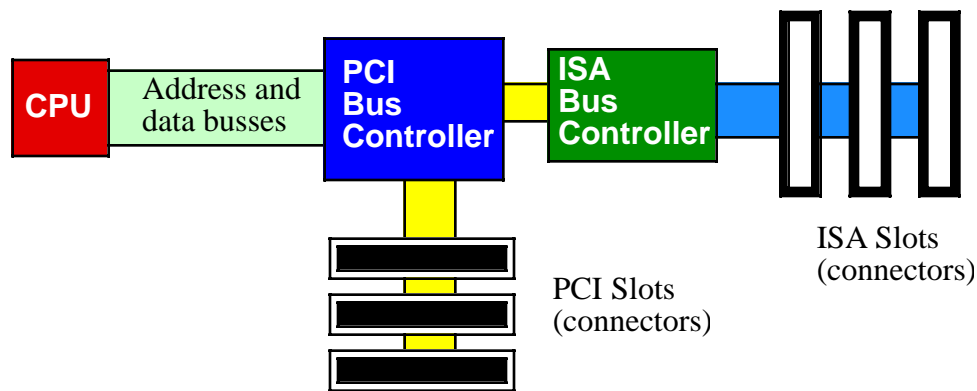


Figure 7.6 Connection of the PCI and ISA Busses in a Typical PC

Notice how the CPU's address and data busses connect to a PCI Bus Controller device (which is, itself, a peripheral of sorts). The actual PCI bus is connected to this chip. Note that the CPU does not connect directly to the PCI bus. Instead, the PCI Bus Controller acts as an intermediary, rerouting all data transfer requests between the CPU and the PCI bus.

Another interesting thing to note is that the ISA Bus Controller is not directly connected to the CPU. Instead, it is connected to the PCI Bus Controller. There is no logical reason why the ISA Controller couldn't be connected directly to the CPU's bus, however, in most modern PCs the ISA and PCI controllers appear on the same chip and the manufacturer of this chip has chosen to interface the ISA bus through the PCI controller for cost or performance reasons.

The CPU's bus (often called the local bus) usually runs at some submultiple of the CPU's frequency. Typical local bus frequencies include 66 MHz, 100 MHz, 133 MHz, 400 MHz, and, possibly, beyond⁴. Usually, only memory and a few selected peripherals (e.g., the PCI Bus Controller) sit on the CPU's bus and operate at this high frequency. Since the CPU's bus is typically 64 bits wide (for Pentium and later processors) and it is theoretically possible to achieve one data transfer per cycle, the CPU's bus has a maximum possible data transfer rate (or maximum bandwidth) of eight times the clock frequency (e.g., 800 megabytes/second for a 100 MHz bus). In practice, CPUs rarely achieve the maximum data transfer rate, but they do achieve some percentage of this, so the faster the bus, the more data can move in and out of the CPU (and caches) in a given amount of time.

The PCI bus comes in several configurations. The base configuration has a 32-bit wide data bus operating at 33 MHz. Like the CPU's local bus, the PCI is theoretically capable of transferring data on each clock cycle. This provides a theoretical maximum of 132 MBytes/second data transfer rate (33 MHz times four bytes). In practice, the PCI bus doesn't come anywhere near this level of performance except in short bursts. Whenever the CPU wishes to access a peripheral on the PCI bus, it must negotiate with other peripheral devices for the right to use the bus. This negotiation can take several clock cycles before the PCI controller grants the CPU the bus. If a CPU writes a sequence of values to a peripheral a double word per bus request, then the negotiation takes the majority of the time and the data transfer rate drops dramatically. The only way to achieve anywhere near the maximum theoretical bandwidth on the bus is to use a DMA controller and move blocks of data. In this block mode the DMA controller can negotiate just once for the bus and transfer a fair sized block of data without giving up the bus between each transfer. This "burst mode" allows the device to move lots of data quickly.

There are a couple of enhancements to the PCI bus that improve performance. Some PCI busses support a 64-bit wide data path. This, obviously, doubles the maximum theoretical data transfer rate. Another enhance-

4. 400 MHz was the maximum CPU bus frequency as this was being written.

ment is to run the bus at 66 MHz, which also doubles the throughput. In theory, you could have a 64-bit wide 66 MHz bus that quadruples the data transfer rate (over the performance of the baseline configuration). Few systems or peripherals currently support anything other than the base configuration, but these optional enhancements to the PCI bus allow it to grow with the CPU as CPUs increase their performance.

The ISA bus is a carry over from the original PC/AT computer system. This bus is 16 bits wide and operates at 8 MHz. It requires four clock cycles for each bus cycle. For this and other reasons, the ISA bus is capable of about only one data transmission per microsecond. With a 16-bit wide bus, data transfer is limited to about two megabytes per second. This is much slower than the CPU's local bus and the PCI bus. Generally, you would only attach low-speed devices like an RS-232 communications device, a modem, or a parallel printer to the ISA bus. Most other devices (disks, scanners, network cards, etc.) are too fast for the ISA bus. The ISA bus is really only capable of supporting low-speed and medium speed devices.

Note that accessing the ISA bus on most systems involves first negotiating for the PCI bus. The PCI bus is so much faster than the ISA bus that this has very little impact on the performance of peripherals on the ISA bus. Therefore, there is very little difference to be gained by connecting the ISA controller directly to the CPU's local bus.

7.7 The AGP Bus

Video display cards are a very special peripheral that need the maximum possible amount of bus bandwidth to ensure quick screen updates and fast graphic operations. Unfortunately, if the CPU has to constantly negotiate with other peripherals for the use of the PCI bus, graphics performance can suffer. To overcome this problem, video card designers created the AGP (Advanced Graphics Port) interface between the CPU and the video display card.

The AGP is a secondary bus interface that a video card uses in addition to the PCI bus. The AGP connection lets the CPU quickly move data to and from the video display RAM. The PCI bus provides a connection to the other I/O ports on the video display card (see Figure 7.7). Since there is only one AGP port per system, only one card can use the AGP and the system never has to negotiate for access to the AGP bus.

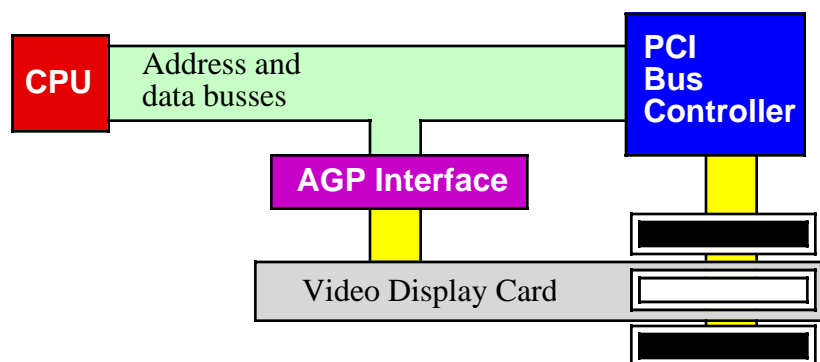


Figure 7.7 AGP Bus Interface

Buffering

If a particular I/O device produces or consumes data faster than the system is capable of transferring data to that device, the system designer has two choices: provide a faster connection between the CPU and the device or slow down the rate of transfer between the two.

Creating a faster connection is possible if the peripheral device is already connected to a slow bus like ISA. Another possibility is going to a wider bus (e.g., to the 64-bit PCI bus) to increase bandwidth, or to use a bus with a higher frequency (e.g., a 66 MHz bus rather than a 33 MHz bus). Systems designers can sometimes create a faster interface to the bus; the AGP connection is a good example. However, once you're using the fastest bus available on the system, improving system performance by selecting a faster connection to the computer can be very expensive.

The other alternative is to slow down the transfer rate between the peripheral and the computer system. This isn't always as bad as it seems. Most high-speed devices don't transfer data at a constant rate to the system. Instead, devices typically transfer a block of data rapidly and then sit idle for some period of time. Although the burst rate is high (and faster than the CPU or system can handle), the average data transfer rate is usually lower than what the CPU/system can handle. If you could average out the peaks and transfer some of the data when the peripheral is inactive, you could easily move data between the peripheral and the computer system without resorting to an expensive, high-bandwidth, solution.

The trick is to use memory to buffer the data on the peripheral side. The peripheral can rapidly fill this buffer with data (or extract data from the buffer). Once the buffer is empty (or full) and the peripheral device is inactive, the system can refill (or empty) the buffer at a sustainable rate. As long as the average data rate of the peripheral device is below the maximum bandwidth the system will support, and the buffer is large enough to hold bursts of data to/from the peripheral, this scheme lets the peripheral communicate with the system at a lower data transfer rate than the device requires during burst operation.

7.8 Handshaking

Many I/O devices cannot accept data at an arbitrary rate. For example, a Pentium based PC is capable of sending several hundred million characters a second to a printer, but that printer is (probably) unable to print that many characters each second. Likewise, an input device like a keyboard is unable to provide several million keystrokes per second (since it operates at human speeds, not computer speeds). The CPU needs some mechanism to coordinate data transfer between the computer system and its peripheral devices.

One common way to coordinate data transfer is to provide some status bits in a secondary input port. For example, a one in a single bit in an I/O port can tell the CPU that a printer is ready to accept more data, a zero would indicate that the printer is busy and the CPU should not send new data to the printer. Likewise, a one bit in a different port could tell the CPU that a keystroke from the keyboard is available at the keyboard data port, a zero in that same bit could indicate that no keystroke is available. The CPU can test these bits prior to reading a key from the keyboard or writing a character to the printer.

Using status bits to indicate that a device is ready to accept or transmit data is known as handshaking. It gets this name because the protocol is similar to two people agreeing on some method of transfer by a hand shake.

Figure 7.8 shows the layout of the parallel printer port's status register. For the LPT1: printer interface, this port appears at I/O address \$379. As you can see from this diagram, bit seven determines if the printer is capable of receiving data from the system; this bit will contain a one when the printer is capable of receiving data.

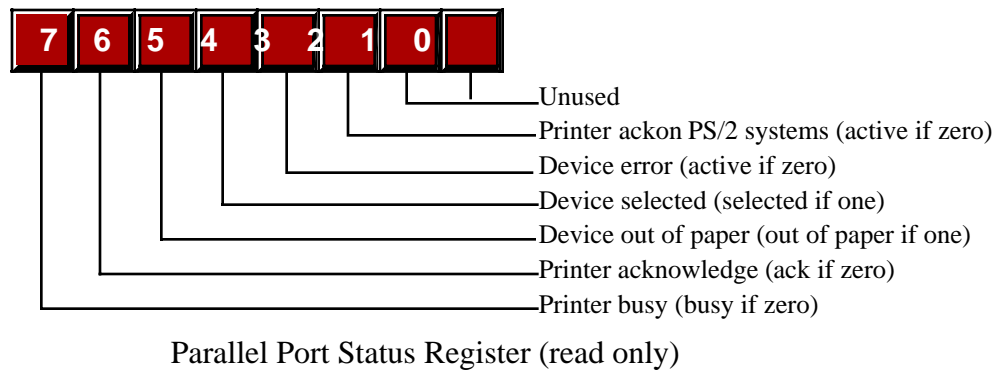


Figure 7.8 The Parallel Port Status Port

The following short program segment will continuously loop while the H.O. bit of the printer status register contains zero and will exit once the printer is ready to accept data:

```

mov( $379, dx );
repeat

    in( dx, al );
    and( $80, al ); // Clears Z flag if bit seven is set.

until( @nz );

// Okay to write another byte to the printer data port here.

```

The code above begins by setting DX to \$379 since this is the I/O address of the printer status port. Within the loop the code reads a byte from the status port (the IN instruction) and then tests the H.O. bit of the port using the AND instruction. Note that logically ANDing the AL register with \$80 will produce zero if the H.O. bit of AL was zero (that is, if the byte read from the input port was zero). Similarly, logically anding AL with \$80 will produce \$80 (a non-zero result) if the H.O. bit of the printer status port was set. The 80x86 zero flag reflects the result of the AND instruction; therefore, the zero flag will be set if AND produces a zero result, it will be reset otherwise. The REPEAT..UNTIL loop repeats this test until the AND instruction produces a non-zero result (meaning the H.O. bit of the status port is set).

One problem with using the AND instruction to test bits as the code above is that you might want to test other bits in AL once the code leaves the loop. Unfortunately, the "and(\$80, al);" instruction destroys the values of the other bits in AL while testing the H.O. bit. To overcome this problem, the 80x86 supports another form of the AND instruction —TEST. The TEST instruction works just like AND except it only updates the flags; it does not store the result of the logical AND operation back into the destination register (AL in this case). One other advantage to TEST is that it only reads its operands, so there are less problems with data hazards when using this instruction (versus AND). Also, you can safely use the TEST instruction directly on read-only memory-mapped I/O ports since it does not write data back to the port. As an example, let's recode the previous loop using the TEST instruction:

```

mov( $379, dx );

```


repeat

```
in( dx, al );  
test( $80, al ); // Clears Z flag if bit seven is set.
```

```
until( @nz );
```

```
// Okay to write another byte to the printer data port here.
```

Once the H.O. bit of the printer status port is set, it is okay to transmit another byte to the printer. The computer can make a byte available by storing the byte data into I/O address \$378 (for LPT1:). However, simply storing data to this port does not inform the printer that it can take the byte. The system must complete the other half of the handshake operation and send the printer a signal to indicate that a byte is available.

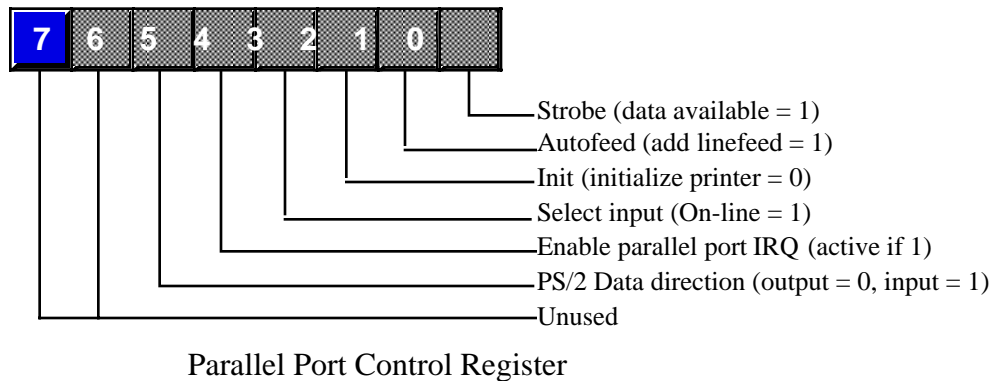


Figure 7.9 The Parallel Port Command Register

Bit zero (the strobe line) must be set to one and then back to zero when the CPU makes data available for the printer (the term "strobe" suggests that the system pulses this line in the command port). In order to pulse this bit without affecting the other control lines, the CPU must first read this port, OR a one into the L.O. bit, write the data to the port, then mask out the L.O. bit using an AND instruction, and write the final result back to the control port again. Therefore, it takes three accesses (a read and two writes) to send the strobe to the printer. The following code handles this transmission:

```
mov( $378, dx ); // Data port address  
mov( Data2Xmit, al ); // Send the data to the printer.  
out( al, dx );
```

```
mov( $37a, dx ); // Point DX at the control port.  
in( dx, al ); // Get the current port setting.  
or( 1, al ); // Set the L.O. bit.  
out( al, dx ); // Set the strobe line high.  
and( $fe, al ); // Clear the L.O. bit.
```

```
out( al, dx );    // Set the strobe line low.
```

The code above would normally follow the REPEAT..UNTIL loop in the previous example. To transmit a second byte to the printer you would jump back to the REPEAT..UNTIL loop and wait for the printer to consume the current byte.

Note that it takes a minimum of five I/O port accesses to transmit a byte to the printer use the code above (minimum one IN instruction in the REPEAT..UNTIL loop plus four instructions to send the byte and strobe). If the parallel port is connected to the ISA bus, this means it takes a minimum of five microseconds to transmit a single byte; that works out to less than 200,000 bytes per second. If you are sending ASCII characters to the printer, this is far faster than the printer can print the characters. However, if you are sending a bitmap or a Post-script file to the printer, the printer port bandwidth limitation will become the bottleneck since it takes considerable data to print a page of graphics. For this reason, most graphic printers use a different technique than the above to transmit data to the printer (some parallel ports support DMA in order to get the data transfer rate up to a reasonable level).

7.9 Time-outs on an I/O Port

One problem with the REPEAT..UNTIL loop in the previous section is that it could spin indefinitely waiting for the printer to become ready to accept additional input. If someone turns the printer off or the printer cable becomes disconnected, the program could freeze up, forever waiting for the printer to become available. Usually, it's a good idea to indicate to the user that something has gone wrong rather than simply freezing up the system. A typical way to handle this problem is using a time-out period to determine that something is wrong with the peripheral device.

With most peripheral devices you can expect some sort of response within a reasonable amount of time. For example, most printers will be ready to accept additional character data within a few seconds of the last transmission (worst case). Therefore, if 30 seconds or more have passed since the printer was last willing to accept a character, this is probably an indication that something is wrong. If the program could detect this, then it could ask the user to check the printer and tell the program to resume printing once the problem is resolved.

Choosing a good time-out period is not an easy task. You must carefully balance the irritation of having the program constantly ask you what's wrong when there is nothing wrong with the printer (or other device) with the program locking up for long periods of time when there is something wrong. Both situations are equally annoying to the end user.

Any easy way to create a time-out period is to count the number of times the program loops while waiting for a handshake signal from a peripheral. Consider the following modification to the REPEAT..UNTIL loop of the previous section:

```
mov( $379, dx );
mov( 30_000_000, ecx );
repeat

    dec( ecx );    // Count down to see if the time-out has expired.
    breakif( @z ); // Leave this loop if ecx counted down to zero.

in( dx, al );
```

```

    test( $80, al ); // Clears Z flag if bit seven is set.

until( @nz );

    if( ecx = 0 ) then

        // We had a time-out error.

    else

        // Okay to write another byte to the printer data port here.

    endif;

```

The code above will exit once the printer is ready to accept data or when approximately 30 seconds have expired. You may question the 30 second figure. After all, a software based loop (counting down ECX to zero) should run at different speeds on different processors. However, don't miss the fact that there is an IN instruction inside this loop. The IN instruction reads a port on the ISA bus and that means this instruction will take approximately one microsecond to execute (about the fastest operation on the ISA bus). Hence, every one million times through the loop will take about a second (–50%, but close enough for our purposes). This is true regardless of the CPU frequency.

The 80x86 provides a couple of instructions that are quite useful for implementing time-outs in a polling loop: LOOPZ and LOOPNZ. We'll consider the LOOPZ instruction here since it's perfect for the loop above. The LOOPZ instruction decrements the ECX register by one and falls through to the next instruction if ECX contains zero. If ECX does not contain zero, then this instruction checks the zero flag setting prior to decrementing ECX; if the zero flag was set, then LOOPZ transfers control to a label specified as LOOPZ's operand. Consider the implementation of the previous REPEAT..UNTIL loop using LOOPZ:

```

    mov( $379, dx );
    mov( 30_000_000, ecx );
PollingLoop:

    in( dx, al );
    test( $80, al ); // Clears Z flag if bit seven is set.

loopz PollingLoop; // Repeat while zero and ECX > 0.

    if( ecx = 0 ) then

        // We had a time-out error.

```

else

```
// Okay to write another byte to the printer data port here.
```

endif;

Notice how this code doesn't need to explicitly decrement ECX and check to see if it became zero.

Warning: the LOOPZ instruction can only transfer control to a label with –127 bytes of the LOOPZ instruction. Due to a design problem, HLA cannot detect this problem. If the branch range exceeds 127 bytes HLA will not report an error. Instead, the underlying assembler (e.g., MASM or Gas) will report the error when it assembles HLA's output. Since it's somewhat difficult to track down these problems in the MASM or Gas listing, the best solution is to never use the LOOPZ instruction to jump more than a few instructions in your code. It's perfect for short polling loops like the one above, it's not suitable for branching large distances.

7.10 Interrupts and Polled I/O

Polling is constantly testing a port to see if data is available. That is, the CPU polls (asks) the port if it has data available or if it is capable of accepting data. The REPEAT..UNTIL loop in the previous section is a good example of polling. The CPU continually polls the port to see if the printer is ready to accept data. Polled I/O is inherently inefficient. Consider what happens in the previous section if the printer takes ten seconds to accept another byte of data — the CPU spins in a loop doing nothing (other than testing the printer status port) for those ten seconds.

In early personal computer systems, this is exactly how a program would behave; when it wanted to read a key from the keyboard it would poll the keyboard status port until a key was available. Such computers could not do other operations while waiting for the keyboard.

The solution to this problem is to provide an *interrupt mechanism*. An interrupt is an external hardware event (such as the printer becoming ready to accept another byte) that causes the CPU to interrupt the current instruction sequence and call a special interrupt service routine. (ISR). An interrupt service routine typically saves all the registers and flags (so that it doesn't disturb the computation it interrupts), does whatever operation is necessary to handle the source of the interrupt, it restores the registers and flags, and then it resumes execution of the code it interrupted. In many computer systems (e.g., the PC), many I/O devices generate an interrupt whenever they have data available or are able to accept data from the CPU. The ISR quickly processes the request in the background, allowing some other computation to proceed normally in the foreground.

An interrupt is essentially a procedure call that the hardware makes (rather than explicit call to some procedure, like a call to the *stdout.put* routine). The most important thing to remember about an interrupt is that it can pause the execution of some program at any point between two instructions when an interrupt occurs. Therefore, you typically have no guarantee that one instruction always executes immediately after another in the program because an interrupt could occur between the two instructions. If an interrupt occurs in the middle of the execution of some instruction, then the CPU finishes that instruction before transferring control to the appropriate interrupt service routine. However, the interrupt generally interrupts execution before the start of the next instruction⁵. Suppose, for example, that an interrupt occurs between the execution of the following two instructions:

```
add( i, eax );
```

<---- Interrupt occurs here.

```
mov( eax, j );
```

When the interrupt occurs, control transfers to the appropriate ISR that handles the hardware event. When that ISR completes and executes the IRET (interrupt return) instruction, control returns back to the point of interruption and execution of the original code continues with the instruction immediately after the point of interrupt (e.g., the MOV instruction above). Imagine an interrupt service routine that executes the following code:

```
mov( 0, eax );  
iret;
```

If this ISR executes in response to the interrupt above, then the main program will not produce a correct result. Specifically, the main program should compute "j := eax +i;" Instead, it computes "j := 0;" (in this particular case) because the interrupt service routine sets EAX to zero, wiping out the sum of *i* and the previous value of EAX. This highlights a very important fact about ISRs: **ISRs must preserve all registers and flags whose values they modify**. If an ISR does not preserve some register or flag value, this will definitely affect the correctness of the programs running when an interrupt occurs. Usually, the ISR mechanism itself preserves the flags (e.g., the interrupt pushes the flags onto the stack and the IRET instruction restores those flags). However, the ISR itself is responsible for preserving any registers that it modifies.

Although the preceding discussion makes it clear that ISRs must preserve registers and the flags, your ISRs must exercise similar care when manipulating any other resources the ISR shares with other processes. This includes variables, I/O ports, etc. Note that preserving the values of such objects isn't always the correct solution. Many ISRs communicate their results to the foreground program using shared variables. However, as you will see, the ISR and the foreground program must coordinate access to shared resources or they may produce incorrect results. Writing code that correctly works with shared resources is a difficult challenge; the possibility of subtle bugs creeping into the program is very great. We'll consider some of these issues a little later in this chapter; the messy details will have to wait for a later volume of this text.

CPUs that support interrupts must provide some mechanism that allows the programmer to specify the address of the ISR to execute when an interrupt occurs. Typically, an interrupt vector is a special memory location that contains the address of the ISR to execute when an interrupt occurs. PCs typically support up to 16 different interrupts.

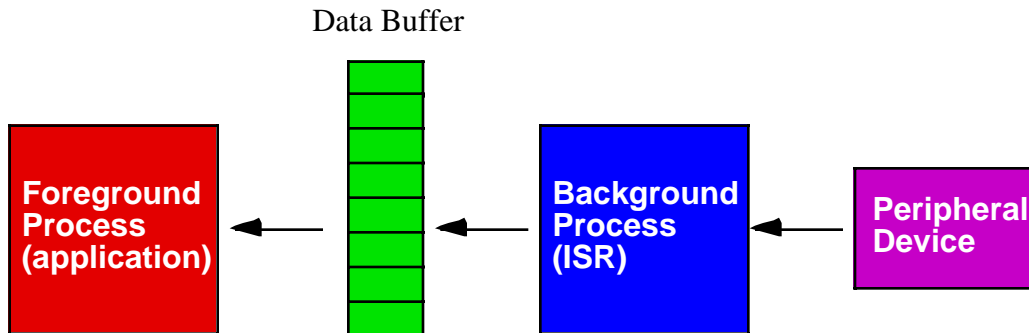
After an ISR completes its operation, it generally returns control to the foreground task with a special return from interrupt instruction. On the Y86 hypothetical processor, for example, the IRET (interrupt return) instruction handles this task. This same instruction does a similar task on the 80x86. An ISR should always end with this instruction so the ISR can return control to the program it interrupted.

7.11 Using a Circular Queue to Buffer Input Data from an ISR

A typical interrupt-driven input system uses the ISR to read data from an input port and buffer it up whenever data becomes available. The foreground program can read that data from the buffer at its leisure without losing any data from the port. A typical foreground/ISR arrangement appears in Figure 7.10. In this diagram the ISR

5. The situation is somewhat fuzzy if you have pipelines and superscalar operation. Exactly what instruction does an interrupt precede if there are multiple instructions executing simultaneously? The answer is somewhat irrelevant, however, since the interrupt does take place between the execution of some pair of instructions; in reality, the interrupt may occur immediately after the last instruction to enter the pipeline when the interrupt occurs. Nevertheless, the system does interrupt the execution of the foreground process after the execution of some instruction.

reads a value from the peripheral device and then stores the data into a common buffer that the ISR shares with the foreground application. Sometime later, the foreground process removes the data from the buffer. If (during a burst of input) the device and ISR produce data faster than the foreground application reads data from the buffer, the ISR will store up multiple unread data values in the buffer. As long as the average consumption rate of the foreground process matches the average production rate of the ISR, and the buffer is large enough to hold bursts of data, there will be no lost data.



The background process produces data (by reading it from the device) and places it in the buffer. The foreground process consumes data by removing it from the buffer.

Figure 7.10 Interrupt Service Routine as a Data Produce/Application as a Data Consumer

If the foreground process in Figure 7.10 consumes data faster than the ISR produces it, sooner or later the buffer will become empty. When this happens the foreground process will have to wait for the background process to produce more data. Typically the foreground process would poll the data buffer (or, in a more advanced system, block execution) until additional data arrives. Then the foreground process can easily extract the new data from the buffer and continue execution.

There is nothing special about the data buffer. It is just a block of contiguous bytes in memory and a few additional pieces of information to maintain the list of data in the buffer. While there are lots of ways to maintain data in a buffer such as this one, probably the most popular technique is to use a *circular buffer*. A typical circular buffer implementation contains three objects: an array that holds the actual data, a pointer to the next available data object in the buffer, and a length value that specifies how many objects are currently in the buffer.

Later in this text you will see how to declare and use arrays. However, in the chapter on Memory Access you saw how to allocate a block of data in the STATIC section (see The Static Sections on page 167) or how to use *malloc* to allocate a block of bytes (see Dynamic Memory Allocation and the Heap Segment on page 187). For our purposes, declaring a block of bytes in the STATIC section is just fine; the following code shows one way to set aside 16 bytes for a buffer:

```
static
    buffer:    byte := 0;                // Reserves one byte.
             byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 // 15 additional bytes.
```

Of course, this technique would not be useful if you wanted to set aside storage for a really large buffer, but it works fine for small buffers (like our example above). See the chapter on arrays (appearing later in this text) if you need to allocate storage for a larger buffer.

In addition to the buffer data itself, a circular buffer also needs at least two other values: an index into the buffer that specifies where the next available data object appears and a count of valid items in the buffer. Given that the 80x86's addressing

modes all use 32-bit registers, we'll find it most convenient to use a 32-bit unsigned integer for this purpose even though the index and count values never exceed 16. The declaration for these values might be the following:

```
static
    index: uns32 := 0; // Start with first element of the array.
    count: uns32 := 0; // Initially, there is no data in the array.
```

The data producer (the ISR in our example) inserts data into the buffer by following these steps:

- ¥ Check the count. If the count is equal to the buffer size, then the buffer is full and some corrective action is necessary.
- ¥ Store the new data object at location $((\text{index} + \text{count}) \bmod \text{buffer_size})$.
- ¥ Increment the count variable.

Suppose that the producer wishes to add a character to the initially empty buffer. The *count* is zero so we don't have to deal with a buffer overflow. The *index* value is also zero, so $((\text{index} + \text{count}) \bmod 16)$ is zero and we store our first data byte at index zero in the array. Finally, we increment *count* by one so that the producer will put the next byte at offset one in the array of bytes.

If the consumer never removes any bytes and the producer keeps producing bytes, sooner or later the buffer will fill up and *count* will hit 16. Any attempt to insert additional data into the buffer is an error condition. The producer needs to decide what to do at that point. Some simple routines may simply ignore any additional data (that is, any additional incoming data from the device will be lost). Some routines may signal an exception and leave it up to the main application to deal with the error. Some other routines may attempt to expand the buffer size to allow additional data in the buffer. The corrective action is application-specific. In our examples we'll assume the program either ignores the extra data or immediately stops the program if a buffer overflow occurs.

You'll notice that the producer stores the data at location $((\text{index} + \text{count}) \bmod \text{buffer_size})$ in the array. This calculation, as you'll soon see, is how the circular buffer obtains its name. HLA does provide a MOD instruction that will compute the remainder after the division of two values, however, most buffer routines don't compute remainder using the MOD instruction. Instead, most buffer routines rely on a cute little trick to compute this value much more efficiently than with the MOD instruction. The trick is this: if a buffer's size is a power of two (16 in our case), you can compute $(x \bmod \text{buffer_size})$ by logically ANDing x with $\text{buffer_size} - 1$. In our case, this means that the following instruction sequence computes $((\text{index} + \text{count}) \bmod 16)$ in the EBX register:

```
mov( index, ebx );
add( count, ebx );
and( 15, ebx );
```

Remember, this trick only works if the buffer size is an integral power of two. If you look at most programs that use a circular buffer for their data, you'll discover that they commonly use a buffer size that is an integral power of two. The value is not arbitrary; they do this so they can use the AND trick to efficiently compute the remainder.

To remove data from the buffer, the consumer half of the program follows these steps:

- ¥ The consumer first checks to see if there is any data in the buffer. If not, the consumer waits until data is available.
- ¥ If (or when) data is available, the consumer fetches the value at the location *index* specifies within the buffer.
- ¥ The consumer then decrements the *count* and computes $\text{index} := (\text{index} + 1) \bmod \text{buffer_size}$.

To remove a byte from the circular buffer in our current example, you'd use code like the following:

```
// wait for data to appear in the buffer.

repeat
until( count <> 0 );

// Remove the character from the buffer.

mov( index, ebx );
mov( buffer[ ebx ], al ); // Fetch the byte from the buffer.
dec( count );           // Note that we've removed a character.
inc( ebx );             // Index := Index + 1;
and( 15, ebx );        // Index := (index + 1) mod 16;
mov( ebx, index );     // Save away the new index value.
```

As the consumer removes data from the circular queue, it advances the index into the array. If you're wondering what happens at the end of the array, well that's the purpose of the MOD calculation. If *index* starts at zero and increments with each character, you'd expect the sequence 0, 1, 2, ... At some point or another the *index* will exceed the bounds of the buffer (i.e., when *index* increments to 16). However, the MOD operation resets this value back to zero (since 16 MOD 16 is zero). Therefore, the consumer, after that point, will begin removing data from the beginning of the buffer.

Take a close look at the REPEAT..UNTIL loop in the previous code. At first blush you may be tempted to think that this is an infinite loop if *count* initially contains zero. After all, there is no code in the body of the loop that modifies *count*'s value. So if *count* contains zero upon initial entry, how does it ever change? Well, that's the job of the ISR. When an interrupt comes along the ISR suspends the execution of this loop at some arbitrary point. Then the ISR reads a byte from the device, puts the byte into the buffer, and updates the *count* variable (from zero to one). Then the ISR returns and the consumer code above resumes where it left off. On the next loop iteration, however, *count*'s value is no longer zero, so the loop falls through to the following code. This is a classic example of how an ISR communicates with a foreground process — by writing a value to some shared variable.

There is a subtle problem with the producer/consumer code in this section. It will fail if the producer is attempting to insert data into the buffer at exactly the same time the consumer is removing data. Consider the following sequence of instructions:

```
// wait for data to appear in the buffer.

repeat
until( count <> 0 );

// Remove the character from the buffer.

mov( index, ebx );
mov( buffer[ ebx ], al ); // Fetch the byte from the buffer.
dec( count );           // Note that we've removed a character.

*** Assume the interrupt occurs here, so we begin executing
*** the data insertion sequence:

mov( index, ebx );
add( count, ebx );
and( 15, ebx );
mov( al, buffer[ ebx] );
inc( count );
```



```
*** now the ISR returns to the consumer code (assume we've preserved EBX):
```

```
inc( ebx );           // Index := Index + 1;
and( 15, ebx );      // Index := (index + 1) mod 16;
mov( ebx, index );   // Save away the new index value.
```

The problem with this code, which is very subtle, is that the consumer has decremented the count variable and an interrupt occurs before the consumer can update the index variable as well. Therefore, upon arrival into the ISR, the *count* value and the *index* value are inconsistent. That is, *index+count* now points at the last value placed in the buffer rather than the next available location. Therefore, the ISR will overwrite the last byte in the buffer rather than properly placing this byte after the (current) last byte. Worse, once the ISR returns to the consumer code, the consumer will update the *index* value and effectively add a byte of garbage to the end of the circular buffer. The end result is that we wipe out the next to last value in the buffer and add a garbage byte to the end of the buffer.

Note that this problem doesn't occur all the time, or even frequently for that matter. In fact, it only occurs in the very special case where the interrupt occurs between the "dec(count);" and "mov(ebx, index);" instructions in this code. If this code executes a very tiny percentage of the time, the likelihood of encountering this error is quite small. This may seem good, but this is actually worse than having the problem occur all the time; the fact that the problem rarely occurs just means that it's going to be really hard to find and correct this problem when you finally do detect that something has gone wrong. ISRs and concurrent programs are among the most difficult programs in the world to test and debug. The best solution is to carefully consider the interaction between foreground and background tasks when writing ISRs and other concurrent programs. In a later volume, this text will consider the issues in concurrent programming, for now, be very careful about using shared objects in an ISR.

There are two ways to correct the problem that occurs in this example. One way is to use a pair of (somewhat) independent variables to manipulate the queue. The original PC's type-ahead keyboard buffer, for example, used two index variables rather than an index and a count to maintain the queue. The ISR would use one index to insert data and the foreground process would use the second index to remove data from the buffer. The only sharing of the two pointers was a comparison for equality, which worked okay even in an interrupt environment. Here's how the code worked:

```
// Declarations

static
  buffer: byte := 0; byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
  Ins: uns32 := 0; // Insert bytes starting here.
  Rmv: uns32 := 0; // Remove bytes starting here.

// Insert a byte into the queue (the ISR ):

mov( Ins, ebx );
inc( ebx );
and( 15, ebx );
if( ebx <> Rmv ) then

  mov( al, buffer[ ebx ] );
  mov( ebx, Ins );

else

  // Buffer overflow error.
  // Note that we don't update INS in this case.
```

```

endif;

// Remove a byte from the queue (the consumer process).

mov( Rmv, ebx );
repeat

    // Wait for data to arrive

until( ebx <> Ins );
mov( buffer[ ebx ], al );
inc( ebx );
and( 15, ebx );
mov( ebx, Rmv );

```

If you study this code (very) carefully, you'll discover that the two code sequences don't interfere with one another. The difference between this code and the previous code is that the foreground and background processes don't write to a (control) variable that the other routine uses. The ISR only writes to *Ins* while the foreground process only writes to *Rmv*. In general, this is not a sufficient guarantee that the two code sequences won't interfere with one another, but it does work in this instance.

One drawback to this code is that it doesn't fully utilize the buffer. Specifically, this code sequence can only hold 15 characters in the buffer; one byte must go unused because this code determines that the buffer is full when the value of *Ins* is one less than *Rmv* (MOD 16). When the two indices are equal, the buffer is empty. Since we need to test for both these conditions, we can't use one of the bytes in the buffer.

A second solution, that many people prefer, is to protect that section of code in the foreground process that could fail if an interrupt comes along. There are lots of ways to protect this *critical section*⁶ of code. Alas, most of the mechanisms are beyond the scope of this chapter and will have to wait for a later volume in this text. However, one simple way to protect a critical section is to simply disable interrupts during the execution of that code. The 80x86 family provides two instructions, CLI and STI that let you enable and disable interrupts. The CLI instruction (clear interrupt enable flag) disables interrupts by clearing the "I" bit in the flags register (this is the interrupt enable flag). Similarly, the STI instruction enables interrupts by setting this flag. These two instructions use the following syntax:

```

cli();    // Disables interrupts from this point forward...
.
.
.
sti();    // Enables interrupts from this point forward...

```

You can surround a critical section in your program with these two instructions to protect that section from interrupts. The original consumer code could be safely written as follows:

```

// wait for data to appear in the buffer.

repeat
until( count <> 0 );

// Remove the character from the buffer.

```

6. A critical section is a region of code during which certain resources have to be protected from other processes. For example, the consumer code that fetches data from the buffer needs to be protected from the ISR.

```

cli(); // Protect the following critical section.
mov( index, ebx );
mov( buffer[ ebx ], al ); // Fetch the byte from the buffer.
dec( count ); // Note that we've removed a character.
inc( ebx ); // Index := Index + 1;
and( 15, ebx ); // Index := (index + 1) mod 16;
mov( ebx, index ); // Save away the new index value.
sti(); // Critical section is done, restore interrupts.

```

Perhaps a better sequence to use is to push the EFLAGS register (that contains the I flag) and turn off the interrupts. Then, rather than blindly turning interrupts back on, you can restore the original I flag setting using a POPFD instruction:

```

// Remove the character from the buffer.

pushfd(); // Preserve current I flag value.
cli(); // Protect the following critical section.
mov( index, ebx );
mov( buffer[ ebx ], al ); // Fetch the byte from the buffer.
dec( count ); // Note that we've removed a character.
inc( ebx ); // Index := Index + 1;
and( 15, ebx ); // Index := (index + 1) mod 16;
mov( ebx, index ); // Save away the new index value.
popfd(); // Restore original I flag value

```

This mechanism is arguably safer since it doesn't turn the interrupts on even if they were already off before executing this sequence.

In our simple example (with a single producer and a single consumer) there is no need to protect the code in the ISR. However, if it were possible for two different ISRs to insert data into the buffer, and one ISR could interrupt another, then you would have to protect the code inside the ISR as well.

You must be very careful about turning the interrupts on and off. If you turn the interrupts off and forget to turn them back on, the next time you enter a loop like one of the REPEAT..UNTIL loops in this section the program will lock up because the loop control variable (*count*) will never change if an ISR cannot execute and update its value. This situation is called *deadlock* and you must take special care to avoid it.

Note that applications under Windows or Linux cannot change the state of the interrupt disable flag. This technique is useful mainly in embedded system or under simpler operating systems like DOS. Fortunately, advanced 32-bit operating systems like Linux and Windows provide other mechanisms for protecting critical sections.

7.12 Using a Circular Queue to Buffer Output Data for an ISR

You can also use a circular buffer to hold data waiting for transmission. For example, a program can buffer up data in bursts while an output device is busy and then the output device can empty the buffer at a steady state. The queuing and dequeuing routines are very similar to those found in the previous section with one major difference: output devices don't automatically initiate the transfer like input devices. This problem is a source of many bugs in output buffering routines and one that we'll pay careful attention to in this section.

As noted above, one advantage of an input device is that it automatically interrupts the system whenever data is available. This activates the corresponding ISR that reads the data from the device and places the data in the buffer. No special processing is necessary to prepare the interrupt system for the next input value.

There is a subtle difference between the interrupts an input device generates and the interrupts an output device generates. An input device generates an interrupt when data is available, output devices generate an interrupt when they are ready to accept more data. For example, a keyboard device generates an interrupt when the user presses a key and the system has to read the character from the keyboard. A printer device, on the other hand, generates an interrupt once it is done with the last character transmitted to it and it's ready to accept another character. Whenever the user presses a keyboard for the very first time, the system will generate an interrupt in response to that event. However, the printer does not generate an interrupt when the system first powers up to tell the system that it's ready to accept a character. Even if it did, the system would ignore the interrupt since it (probably) doesn't have any data to transmit to the printer at that point. Later, when the system puts data in the printer's buffer for transmission, there is no interrupt that activates the ISR to remove a character from the buffer and send it to the printer. The printer device only sends interrupts when it is done processing a character; if it isn't processing any characters, it won't generate any interrupts.

This creates a bit of a problem. If the foreground process places characters in the queue and the background process (the ISR, which is the consumer in this case) only removes those characters when an interrupt occurs, the system will never activate the ISR since the device isn't currently processing anything. To correct this problem, the producer code (the foreground process) must maintain a flag that indicates whether the output device is currently processing a character; if so, then the producer can simply queue up the character in the buffer. If the device is not currently processing any data, then the producer should send the data directly to the device rather than queue up the data in the buffer⁷. Old time programmers refer to this as "priming the pump" since we have to put data in the transmission pipeline in order to get the process working properly.

Once the producer "primes the pump" the process continues automatically as long as there is data in the buffer. After the output device processes the current byte it generates an interrupt. The ISR removes a byte from the buffer and transmits this data to the device. When that byte completes transmission the device generates another interrupt and the process repeats. This process repeats automatically as long as there is data in the buffer to transmit to the output device.

When the ISR transmits the last character from the buffer, the output device still generates an interrupt at the end of the transmission. The ISR, upon noting that the buffer is empty, returns without sending any new data to the output device. Since there is no pending data transmission to the output device, there will be no new interrupts to activate the ISR when new data appears in the buffer. Once again the foreground process (producer) will have to prime the pump to get the process going when it attempts to put data in the buffer.

Perhaps the easiest way to handle this process is to use a boolean variable to indicate whether the output device is currently transmitting data (and will generate an interrupt to process the next byte). If the flag is set, the foreground process can simply enqueue the data; if the flag is clear, the foreground process must transmit the data directly to the device (or call the code that does this). In this latter case, the foreground process must also set the flag to denote a transmission in progress.

Here is some code that can implement this functionality:

```
static
OutBuf: byte := 0; byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;
Index: uns32 := 0;
Count: uns32 := 0;
Xmitting: boolean := false; // Flag to denote transmission in progress.
.
.
.
// Code to enqueue a byte (foreground process executes this)
```

7. Another possibility is to go ahead and queue up the data and then manually activate the code that dequeues the data and sends it to the output device.

```

if( Count = 16 ) then

    // Error, buffer is full. Do whatever processing is necessary to
    // deal with this problem.
    .
    .
    .

elseif( Xmitting ) then

    // If we're currently transmitting data, just add the byte to the queue.

    pushfd();          // Critical region! Turn off the interrupts.
    cli();
    mov( Index, ebx ); // Store the new byte at address (Index+Count) mod 16
    add( Count, ebx );
    and( 15, ebx );
    mov( al, OutBuf[ ebx ] );
    inc( Count );
    popfd();           // Restore the interrupt flag's value.

else

    // The buffer is empty and there is no character in transmission.
    // Do whatever is necessary to transmit the character to the output
    // device.
    .
    .
    .

    // Be sure to set the Xmitting flag since a character is now being
    // transmitted to the output device.

    mov( true, Xmitting );

endif;
.
.
.
// Here's the code that would appear inside the ISR to remove a character
// from the buffer and send it to the output device. The system calls this
// ISR whenever the device finishes processing some character.
// (Presumably, the ISR preserves all registers this code sequence modifies)

if( Count > 0 ) then

    // Okay, there are characters in the buffer. Remove the first one
    // and transmit it to the device:

    mov( Index, ebx );
    mov( OutBuf[ ebx ], al ); // Get the next character to output
    inc( ebx );              // Point Index at the next byte in the
    and( 15, ebx );         // circular buffer.
    mov( ebx, Index );
    dec( Count );          // Decrement count since we removed a char.

```

```

    << At this point, do whatever needs to be done in order to
        transmit a character to the output device >>
    .
    .
    .

else

    // At this point, the ISR was called but the buffer is empty.
    // Simply clear the Xmitting flag and return. (this will force the
    // next buffer insertion operation to transmit the data directly to the
    // device.)

    mov( true, Xmitting );

endif;

```

7.13 I/O and the Cache

It goes without saying that the CPU cannot cache values for memory-mapped I/O ports. If a port is an input port, caching the data from that port would always return the first value read; subsequent reads would read the value in the cache rather than the possible (volatile) data at the input port. Similarly, with a write-back cache mechanism, writes to an output port may never reach that port (i.e., the CPU may save up several writes in the cache and send the last such write to the actual I/O port). Therefore, there must be some mechanism to tell the CPU not to cache up accesses to certain memory locations.

The solution is in the virtual memory subsystem of the 80x86. The 80x86's page table entries contain information that the CPU can use to determine whether it is okay to map data from a page in memory to cache. If this flag is set one way, then the cache operates normally; if the flag is set the other way, then the CPU does not cache up accesses to that page.

Unfortunately, the granularity (that is, the minimum size) of this access is the 4K page. So if you need to map 16 device registers into memory somewhere and cannot cache them, you must actually consume 4K of the address space to hold these 16 locations. Fortunately, there is a lot of room in the 4 GByte virtual address space and there aren't that many peripheral devices that need to be mapped into the memory address space. So assigning these device addresses sparsely in the memory map will not present too many problems.

7.14 Protected Mode Operation

Windows and Linux employ the 80x86's protected mode of operation. In this mode of operation, direct access to devices is restricted to the operating system and certain privileged programs. Standard applications, even those written in assembly language, are not so privileged. If you write a simple program that attempts to send data to an I/O port via an IN or an OUT instruction, the system will generate an illegal access exception and halt your program. Unless you're willing to write a device driver for your operating system, you'll probably not be able to access the I/O devices directly.

Like Windows, Linux does not allow an arbitrary application program to access I/O ports as it pleases. Only programs with "super-user" (root) privileges may do so. For limited I/O access, it is possible to use the Linux IOPERM system call to make certain I/O ports accessible from user applications (note that only a process with super-user privileges may call IOPERM, but that program may then invoke a standard user application and the application it runs will have access to the specified ports). For more details, Linux users should read the "man" page on "ioperm".

This chapter has provided an introduction to I/O in a very general, architectural sense. It hasn't spent too much time discussing the particular peripheral devices present in a typical PC. This is an intended omission; there is no need to confuse readers with information they can't use. Furthermore, as manufacturers introduce new PCs they are removing many of the common peripherals like parallel and serial ports that are relatively easy to program in assembly language. They are replacing these devices with complex peripherals like USB and Firewire. Unfortunately, programming these newer peripheral devices is well beyond the scope of this text (Microsoft's USB code, for example, is well over 100 pages of C++ code).

Those who are interested in additional information about programming standard PC peripherals may want to consult one of the many excellent hardware references available for the PC or take a look at the DOS/16-bit version of this text.

IN and OUT aren't the only instructions that you cannot execute in an application running under protected mode. The system considers many instructions to be "privileged" and will abort your program if you attempt to use these instructions. The CLI and STI instructions are good examples. If you attempt to execute either of these instructions, the system will stop your program.

Some instructions will execute in an application, but behave differently than they do when the operating system executes them. The PUSHFD and POPFD instructions are good examples. These instruction push and pop the interrupt enable flag (among others). Therefore, you could use PUSHFD to push the flags on the stack, pop this double word off the stack and clear the bit associated with the interrupt flag, push the value back onto the stack and then use POPFD to restore the flags (and, in the process, clear the interrupt flag). This would seem like a sneaky way around clearing the interrupt flag. The CPU must allow applications to push and pop the flags for other reasons. However, for various security reasons the CPU cannot allow applications to manipulate the interrupt disable flag. Therefore, the POPFD instruction behaves a little differently in an application than it does when the operating system executes it. In an application, the CPU ignores the interrupt flag bit it pops off the stack. In operating system ("kernel") mode, popping the flags register does restore the interrupt flag.

7.15 Device Drivers

If Linux and Windows don't allow direct access to peripheral devices, how does a program communicate with these devices? Clearly this can be done since applications interact with real-world devices all the time. If you reread the previous section carefully, you'll note that it doesn't claim that programs can't access the devices, it only states that user application programs are denied such access. Specially written modules, known as device drivers, are able to access I/O ports by special permission from the operating system. Writing device drivers is well beyond the scope of this chapter (though it will make an excellent subject for a later volume in this text). Nevertheless, an understanding of how device drivers work may help you understand the possibilities and limitations of I/O under a "protected mode" operating system.

A device driver is a special type of program that connects to the operating system. The device driver must follow some special protocols and it must make some special calls to the operating system that are not available to standard applications. Further, in order to install a device driver in your system you must have administrator privileges (device drivers create all kinds of security and resource allocation problems; you can't have every hacker in the world taking advantage of rogue device drivers running on your system). Therefore, "whipping out a device driver" is not a trivial process and application programs cannot load and unload arbitrary drivers at will.

Fortunately, there are only a limited number of devices you'd typically find on a PC, therefore you only need a limited number of device drivers. You would typically install a device driver in the operating system the same time you install the device (or when you install the operating system if the device is built into the PC). About the only time you'd really need to write your own device driver is when you build your own device or in some special instance when you need to take advantage of some devices capabilities that the standard device drivers don't allow for.

One big advantage to the device driver mechanism is that the operating system (or device vendors) must provide a reasonable set of device drivers or the system will never become popular (one of the reasons Microsoft and IBM's OS/2 operating system was never successful was the dearth of device drivers). This means that applications can easily manipulate lots of devices without the application programmer having to know much about the device itself; the real work has been taken care of by the operating system.

The device driver model does have a few drawbacks, however. The device driver model is great for low-speed devices, where the OS and device driver can respond to the device much more quickly than the device requires. The device driver model is also great for medium and high-speed devices where the system transmits large blocks of data in one direction at a time; in such a situation the application can pass a large block of data to the operating system and the OS can transmit this data to the device (or conversely, read a large block of data from the device and place it in an application-supplied buffer). One problem with the device driver model is that it does not support medium and high-speed data transfers that require a high degree of interaction between the device and the application.

The problem is that calling the operating system is an expensive process. Whenever an application makes a call to the OS to transmit data to the device it could actually take hundreds of microseconds, if not milliseconds, before the device driver actually sees the data. If the interaction between the device and the application requires a constant flurry of bytes moving back and forth, there will be a big delay if each transfer has to go through the operating system. For such applications you will need to write a special device driver to handle the transactions directly in the driver rather than continually returning to the application.

7.16 Putting It All Together

Although the CPU is where all the computation takes place in a computer system, that computation would be for naught if there was no way to get information into and out of the computer system. This is the responsibility of the I/O subsystem. I/O at the machine level is considerably different than the interface high level languages and I/O subroutine libraries (like *std-out.put*) provide. At the machine level, I/O transfers consist of moving bytes (or other data units) between the CPU and device registers or memory.

The 80x86 family supports two types of *programmed I/O*: memory-mapped input/output and I/O-mapped I/O. PCs also provide a third form of I/O that is mostly independent of the CPU: direct memory access or DMA. Memory-mapped input/output uses standard instructions that access memory to move data between the system and the peripheral devices. I/O-mapped input/output uses special instructions, IN and OUT, to move data between the CPU and peripheral devices. I/O-mapped devices have the advantage that they do not consume memory addresses normally intended for system memory. However, the only access to devices using this scheme is through the IN and OUT instructions; you cannot use arbitrary instructions that manipulate memory to control such peripherals. Devices that use DMA have special hardware that let them transmit data to and from system memory without going through the CPU. Devices that use DMA tend to be very high performance, but this I/O mechanism is really only useful for devices that transmit large blocks of data at high speeds.

I/O devices have many different operating speeds. Some devices are far slower than the CPU while other devices can actually produce or consume data faster than the CPU. For devices that are slower than the CPU, some sort of handshaking mechanism is necessary in order to coordinate the data transfer between the CPU and the device. High-speed devices require a DMA controller or buffering since the CPU cannot handle the data rates of these devices. In all cases, some mechanism is necessary to tell the CPU that the I/O operation is complete so the CPU can go about other business.

In modern 32-bit operating systems like Windows and Linux, applications programs do not have direct access to the peripheral devices. The operating system coordinates all I/O via the use of device drivers. The good thing about device drivers is that you (usually) don't have to write them – the operating system provides them for you. The bad thing about writing device drivers is if you have to write one, they are very complex. A later volume in this text may discuss how to do this.

Because HLA programs usually run as applications under the OS, you will not be able to use most of the coding techniques this chapter discusses within your HLA applications. Nevertheless, understanding how device I/O works can help you write better applications. Of course, if you ever have to write a device driver for some device, then the basic knowledge this chapter presents is a good foundation for learning how to write such code.

Volume Three: Basic Assembly Language

- Chapter One: Constants, Variables, and Data Types
How to declare objects in HLA.
- Chapter Two: Character Strings
A discussion of HLA's character string representation and an introduction to string routines in the HLA Standard Library.
- Chapter Three: Characters and Character Sets
A discussion of characters and the operations on them plus character sets and HLA's representation of character sets.
- Chapter Four: Arrays
How to declare and access elements of arrays.
- Chapter Five: Records, Unions, and Namespaces
How to declare records (structures) and how to access the fields within those records.
- Chapter Six: Dates and Times
Dates and Times are two important data types whose importance was underscored by the Y2K problem. This chapter discusses how to properly implement these data types.
- Chapter Seven: File I/O
Maintaining persistent information (across executions) within your programs.
- Chapter Eight: Introduction to Procedures
The ability to create your own procedures is of great importance in any program. This chapter discusses HLA's high level procedure declaration syntax and how to call procedures you've written.
- Chapter Nine: Managing Large Programs
This chapter discusses how to break up a program into modules and separately compile them.

Volume Three:

Basic Assembly Language

Chapter Ten: Integer Arithmetic

This chapter discusses how to compute the values of integer expressions. In particular, it describes how to convert arithmetic expressions into assembly language.

Chapter Eleven: Real Arithmetic

This chapter discusses how to compute the values of floating point expressions. In particular, it describes how to convert arithmetic expressions into assembly language.

Chapter Twelve: Calculation Via Table Lookups

This chapter discusses how to quickly compute some value using a table lookup.

Chapter Thirteen: Questions, Projects, and Laboratory Exercises

Test your knowledge.

This Volume provides a basic introduction to assembly language programming. By the end of this volume you should be able to write meaningful programs using HLA. This Volume plus Volume Four present all the basic skills a typical assembly language programmer needs to write real-world applications in assembly language.

Chapters One through Seven provide information about important data types and data structures found in typical assembly language programs. For courses that have a limited amount of time available, Chapters One, Four, and Five from this set are the most important, closely followed by Chapters Two and Seven. Chapters Three and Six are optional though students should read these on their own.

Chapters Eight and Ten are also essential. Chapters Nine and Eleven are important and the course should cover them if time permits. Chapter Twelve discusses an optimization that is becoming less and less important as CPU speeds vastly outstrip memory access times. Those interested in programming embedded systems should read this chapter, other instructors may elect to skip this material.

Constants, Variables, and Data Types Chapter One

Volume One discussed the basic format for data in memory. Volume Two covered how a computer system physically organizes that data. This chapter finishes this discussion by connecting the concept of *data representation* to its actual physical representation. As the title implies, this chapter concerns itself with three main topics: constants, variables and data structures. This chapter does not assume that you've had a formal course in data structures, though such experience would be useful.

1.1 Chapter Overview

This chapter discusses how to declare and use constants, scalar variables, integers, reals, data types, pointers, arrays, and structures. You must master these subjects before going on to the next chapter. Declaring and accessing arrays, in particular, seems to present a multitude of problems to beginning assembly language programmers. However, the rest of this text depends on your understanding of these data structures and their memory representation. Do not try to skim over this material with the expectation that you will pick it up as you need it later. You will need it right away and trying to learn this material along with later material will only confuse you more.

1.2 Some Additional Instructions: INTMUL, BOUND, INTO

This chapter introduces arrays and other concepts that will require the expansion of your 80x86 instruction set knowledge. In particular, you will need to learn how to multiply two values; hence the first instruction we will look at is the `intmul` (integer multiply) instruction. Another common task when accessing arrays is to check to see if an array index is within bounds. The 80x86 `bound` instruction provides a convenient way to check a register's value to see if it is within some range. Finally, the `into` (interrupt on overflow) instruction provides a quick check for signed arithmetic overflow. Although `into` isn't really necessary for array (or other data type access), its function is very similar to `bound`, hence the presentation at this point.

The `intmul` instruction takes one of the following forms:

```
// The following compute destreg = destreg * constant

intmul( constant, destreg16 );
intmul( constant, destreg32 );

// The following compute dest = src * constant

intmul( constant, srcreg16, destreg16 );
intmul( constant, srcmem16, destreg16 );

intmul( constant, srcreg32, destreg32 );
intmul( constant, srcmem32, destreg32 );

// The following compute dest = dest * src

intmul( srcreg16, destreg16 );
intmul( srcmem16, destreg16 );
intmul( srcreg32, destreg32 );
intmul( srcmem32, destreg32 );
```

Note that the syntax of the `intmul` instruction is different than the `add` and `sub` instructions. In particular, note that the destination operand must be a register (`add` and `sub` both allow a memory operand as a destination). Also note that `intmul` allows three operands when the first operand is a constant. Another important

difference is that the `intmul` instruction only allows 16-bit and 32-bit operands; it does not allow eight-bit operands.

`intmul` computes the product of its specified operands and stores the result into the destination register. If an overflow occurs (which is always a signed overflow, since `intmul` only multiplies signed integer values), then this instruction sets both the carry and overflow flags. `intmul` leaves the other condition code flags undefined (so, for example, you cannot check the sign flag or the zero flag after `intmul` and expect them to tell you anything about the `intmul` operation).

The `bound` instruction checks a 16-bit or 32-bit register to see if it is between one of two values. If the value is outside this range, the program raises an exception and aborts. This instruction is particularly useful for checking to see if an array index is within a given range. The `bound` instruction takes one of the following forms:

```
bound( reg16, LBconstant, UBconstant );
bound( reg32, LBconstant, UBconstant );

bound( reg16, Mem16[2] );1
bound( reg32, Mem32[2] );2
```

The `bound` instruction compares its register operand against an unsigned lower bound value and an unsigned upper bound value to ensure that the register is in the range:

$$\text{lower_bound} \leq \text{register} \leq \text{upper_bound}$$

The form of the `bound` instruction with three operands compares the register against the second and third parameters (the lower bound and upper bound, respectively)³. The `bound` instruction with two operands checks the register against one of the following ranges:

```
Mem16[0] <= register16 <= Mem16[2]
Mem32[0] <= register32 <= Mem32[4]
```

If the specified register is not within the given range, then the 80x86 raises an exception. You can trap this exception using the HLA `try..endtry` exception handling statement. The `excepts.hhf` header file defines an exception, `ex.BoundInstr`, specifically for this purpose. The following code fragment demonstrates how to use the `bound` instruction to check some user input:

```
program BoundDemo;
#include( "stdlib.hhf" );

static
    InputValue:int32;
    GoodInput:boolean;

begin BoundDemo;

    // Repeat until the user enters a good value:

    repeat

        // Assume the user enters a bad value.

        mov( false, GoodInput );
```

-
1. The “[2]” suggests that this variable must be an array of two consecutive word values in memory.
 2. Likewise, this memory operand must be two consecutive dwords in memory.
 3. This form isn’t a true 80x86 instruction. HLA converts this form of the `bound` instruction to the two operand form by creating two readonly memory variables initialized with the specified constant.

```

// Catch bad numeric input via the try..endtry statement.

try

    stdout.put( "Enter an integer between 1 and 10: " );
    stdin.flushInput();
    stdin.geti32();

    mov( eax, InputValue );

    // Use the BOUND instruction to verify that the
    // value is in the range 1..10.

    bound( eax, 1, 10 );

    // If we get to this point, the value was in the
    // range 1..10, so set the boolean "GoodInput"
    // flag to true so we can exit the loop.

    mov( true, GoodInput );

    // Handle inputs that are not legal integers.
exception( ex.ConversionError )

    stdout.put( "Illegal numeric format, reenter", nl );

    // Handle integer inputs that don't fit into an int32.
exception( ex.ValueOutOfRange )

    stdout.put( "Value is *way* too big, reenter", nl );

    // Handle values outside the range 1..10 (BOUND instruction)
/*
exception( ex.BoundInstr )

    stdout.put
    (
        "Value was ",
        InputValue,
        ", it must be between 1 and 10, reenter",
        nl
    );
*/

endtry;

until( GoodInput );
stdout.put( "The value you entered, ", InputValue, " is valid.", nl );

end BoundDemo;

```

Program 1.1 Demonstration of the BOUND Instruction

The `into` instruction, like `bound`, also generates an exception under certain conditions. Specifically, `into` generates an exception if the overflow flag is set. Normally, you would use `into` immediately after a signed arithmetic operation (e.g., `intmul`) to see if an overflow occurs. If the overflow flag is not set, the system ignores the `into` instruction; however, if the overflow flag is set, then the `into` instruction raises the HLA *ex.IntoInstr* exception. The following code sample demonstrates the use of the `into` instruction:

```

program INTOdemo;
#include( "stdlib.hhf" );

static
    LOperand:int8;
    ResultOp:int8;

begin INTOdemo;

    // The following try..endtry checks for bad numeric
    // input and handles the integer overflow check:

    try

        // Get the first of two operands:

        stdout.put( "Enter a small integer value (-128..+127):" );
        stdin.geti8();
        mov( al, LOperand );

        // Get the second operand:

        stdout.put( "Enter a second small integer value (-128..+127):" );
        stdin.geti8();

        // Produce their sum and check for overflow:

        add( LOperand, al );
        into();

        // Display the sum:

        stdout.put( "The eight-bit sum is ", (type int8 al), nl );

        // Handle bad input here:

    exception( ex.ConversionError )

        stdout.put( "You entered illegal characters in the number", nl );

        // Handle values that don't fit in a byte here:

    exception( ex.ValueOutOfRange )

        stdout.put( "The value must be in the range -128..+127", nl );

        // Handle integer overflow here:

    /*
    exception( ex.IntoInstr )

```

```

        stdout.put
        (
            "The sum of the two values is outside the range -128..+127",
            nl
        );
    */

endtry;

end INTOdemo;

```

Program 1.2 Demonstration of the INTO Instruction

1.3 The QWORD and TBYTE Data Types

HLA lets you declare eight-byte and ten-byte variables using the *qword*, and *tbyte* data types, respectively. Since HLA does not allow the use of 64-bit or 80-bit non-floating point constants, you may not associate an initializer with these two data types. However, if you wish to reserve storage for a 64-bit or 80-bit variable, you may use these two data types to do so.

The *qword* type lets you declare *quadword* (eight byte) variables. Generally, *qword* variables will hold 64-bit integer or unsigned integer values, although HLA and the 80x86 certainly don't enforce this. The HLA Standard Library contains several routines to let you input and display 64-bit signed and unsigned integer values. The chapter on advanced arithmetic will discuss how to calculate 64-bit results on the 80x86 if you need integers of this size.

The *tbyte* directive allocates ten bytes of storage. There are two data types indigenous to the 80x87 (math coprocessor) family that use a ten byte data type: ten byte BCD values and extended precision (80 bit) floating point values. Since you would normally use the *real80* data type for floating point values, about the only purpose of *tbyte* in HLA is to reserve storage for a 10-byte BCD value (or other data type that needs 80 bits). Once again, the chapter on advanced arithmetic may provide some insight into the use of this data type. However, except for very advanced applications, you could probably ignore this data type and not suffer.

1.4 HLA Constant and Value Declarations

HLA's CONST and VAL sections let you declare symbolic constants. The CONST section lets you declare identifiers whose value is constant throughout compilation and run-time; the VAL section lets you declare symbolic constants whose value can change at compile time, but whose values are constant at run-time (that is, the same name can have a different value at several points in the source code, but the value of a VAL symbol at a given point in the program cannot change while the program is running).

The CONST section appears in the same declaration section of your program that contains the STATIC, READONLY, STORAGE, and VAR, sections. It begins with the CONST reserved word and has a syntax that is nearly identical to the READONLY section, that is, the CONST section contains a list of identifiers followed by a type and a constant expression. The following example will give you an idea of what the CONST section looks like:

```

const
    pi:          real32 := 3.14159;
    MaxIndex:    uns32  := 15;
    Delimiter:   char   := '/';
    BitMask:     byte   := $F0;

```

```
DebugActive:  boolean:= true;
```

Once you declare these constants in this manner, you may use the symbolic identifiers anywhere the corresponding literal constant is legal. These constants are known as *manifest constants*. A manifest constant is a symbolic representation of a constant that allows you to substitute the literal value for the symbol anywhere in the program. Contrast this with READONLY variables; a READONLY variable is certainly a constant value since you cannot change such a variable at run time. However, there is a memory location associated with READONLY variables and the operating system, not the HLA compiler, enforces the read-only attribute at run-time. Although it will certainly crash your program when it runs, it is perfectly legal to write an instruction like “MOV(EAX, ReadOnlyVar);” On the other hand, it is no more legal to write “MOV(EAX, MaxIndex);” (using the declaration above) than it is to write “MOV(EAX, 15);” In fact, both of these statements are equivalent since the compiler substitutes “15” for *MaxIndex* whenever it encounters this manifest constant.

If there is absolutely no ambiguity about a constant’s type, then you may declare a constant by specifying only the name and the constant’s value, omitting the type specification. In the example earlier, the *pi*, *Delimiter*, *MaxIndex*, and *DebugActive* constants could use the following declarations:

```
const
  pi           := 3.14159;           // Default type is real80.
  MaxIndex     := 15;                // Default type is uns32.
  Delimiter:   := '/';              // Default type is char.
  DebugActive: := true;             // Default type is boolean.
```

Symbol constants that have an integer literal constant are always given the type *uns32* if the constant is zero or positive, or *int32* if the value is negative. This is why *MaxIndex* was okay in this CONST declaration but *BitMask* was not. Had we included the statement “BitMask := \$F0;” in this latter CONST section, the declaration would have been legal but *BitMask* would be of type *uns32* rather than *byte*.

Constant declarations are great for defining “magic” numbers that might possibly change during program modification. The following provides an example of using constants to parameterize “magic” values in the program.

```
program ConstDemo;
#include( "stdlib.hhf" );

const
  MemToAllocate := 4_000_000;
  NumDWords     := MemToAllocate div 4;
  MisalignBy    := 62;

  MainRepetitions := 1000;
  DataRepetitions := 999_900;

  CacheLineSize := 16;

begin ConstDemo;

  //console.cls();
  stdout.put
  (
    "Memory Alignment Exercise",nl,
    nl,
    "Using a watch (preferably a stopwatch), time the execution of", nl
    "the following code to determine how many seconds it takes to", nl
    "execute.", nl
    nl
    "Press Enter to begin timing the code:"
  );
```



```

// Allocate enough dynamic memory to ensure that it does not
// all fit inside the cache. Note: the machine had better have
// at least four megabytes free or virtual memory will kick in
// and invalidate the timing.

malloc( MemToAllocate );

// Zero out the memory (this loop really exists just to
// ensure that all memory is mapped in by the OS).

mov( NumDWords, ecx );
repeat

    dec( ecx );
    mov( 0, (type dword [eax+ecx*4]));

until( !ecx ); // Repeat until ECX = 0.

// Okay, wait for the user to press the Enter key.

stdin.readLine();

// Note: as processors get faster and faster, you may
// want to increase the size of the following constant.
// Execution time for this loop should be approximately
// 10-30 seconds.

mov( MainRepetitions, edx );
add( MisalignBy, eax ); // Force misalignment of data.

repeat

    mov( DataRepetitions, ecx );
    align( CacheLineSize );
    repeat

        sub( 4, ecx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );

    until( !ecx );
    dec( edx );

until( !edx ); // Repeat until EAX is zero.

stdout.put( stdio.bell, "Stop timing and record time spent", nl, nl );

// Okay, time the aligned access.

stdout.put
(
    "Press Enter again to begin timing access to aligned variable:"
);
stdin.readLine();

```

```

// Note: if you change the constant above, be sure to change
// this one, too!

mov( MainRepetitions, edx );
sub( MisalignBy, eax ); // Realign the data.
repeat

    mov( DataRepetitions, ecx );
    align( CacheLineSize );
    repeat

        sub( 4, ecx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );
        mov( [eax+ecx*4], ebx );

    until( !ecx );
    dec( edx );

until( !edx ); // Repeat until EAX is zero.

stdout.put( stdio.bell, "Stop timing and record time spent", nl, nl );
free( eax );

end ConstDemo;

```

Program 1.3 Data Alignment Program Rewritten Using CONST Definitions

1.4.1 Constant Types

Manifest constants can be any of the HLA primitive types plus a few of the composite types this chapter discusses. Volumes One and Two discussed most of the primitive types; these primitive types include the following:

- Boolean constants (true or false)
- Uns8 constants (0..255)
- Uns16 constants (0..65535)
- Uns32 constants (0..4,294,967,295)
- Int8 constants (-128..+127)
- Int16 constants (-32768..+32767)
- Int32 constants (-2,147,483,648..+2,147,483,647)
- Char constants (any ASCII character with a character code in the range 0..255)
- Byte constants (any eight-bit value including integers, booleans, and characters)
- Word constants (any 16-bit value)
- DWord constants (any 32-bit value)
- Real32 constants (floating point values)
- Real64 constants (floating point values)
- Real80 constants (floating point values)

In addition to the constant types appearing above, the CONST section supports six additional constant types:

- String constants
- Text constants
- Enumerated constant values

- Array constants
- Record/Union constants
- Character set constants

These data types are the subject of this Volume and the discussion of most of them appears in later chapters. However, the string and text constants are sufficiently important to warrant an early discussion of these constant types.

1.4.2 String and Character Literal Constants

HLA, like most programming languages, draws a distinction between a sequence of characters, a *string*, and a single character. This distinction is present both in the type declarations and in the syntax for literal character and string constants. Until now, this text has not drawn a fine distinction between character and string literal constants; now it is time to do so.

String literal constants consist of a sequence of zero or more characters surrounded by the ASCII quote characters. The following are all examples of legal literal string constants:

```
"This is a string"      // String with 16 characters.
""                     // Zero length string.
"a"                    // String with a single character.
"123"                  // String of length three.
```

A string of length one is not the same thing as a character constant. HLA uses two completely different internal representations for character and string values. Hence, "a" is not a character value, it is a string value that just happens to contain a single character.

Character literal constants take a couple forms, but the most common consist of a single character surrounded by ASCII apostrophe characters:

```
`2'                    // Character constant equivalent to ASCII code $32.
`a'                    // Character constant for lower case 'A'.
```

As noted above, "a" and 'a' are not equivalent.

Those who are familiar with C/C++/Java probably recognize these literal constant forms, since they are similar to the character and string constants in C/C++/Java. In fact, this text has made a tacit assumption to this point that you are somewhat familiar with C/C++ insofar as examples appearing up to this point use character and string constants without an explicit definition of them⁴.

Another similarity between C/C++ strings and HLA's is the automatic concatenation of adjacent literal string constants within your program. For example, HLA concatenates the two string constants

```
"First part of string, " "second part of string"
```

to form the single string constant

```
"First part of string, second part of string"
```

Beyond these few similarities, however, HLA strings and C/C++ strings are different. For example, C/C++ strings let you specify special character values using the escape character sequence consisting of a backslash character followed by one or more special characters; HLA does not use this escape character mechanism. HLA does provide, however, several other ways to achieve this same goal.

Since HLA does not allow escape character sequences in literal string and character constants, the first question you might ask is "How does one embed quote characters in string constants and apostrophe characters in character constants?" To solve this problem, HLA uses the same technique as Pascal and many other

4. Apologies are due to those of you who do not know C/C++/Java or a language that shares these string and constant definitions.

languages: you insert two quotes in a string constant to represent a single quote or you place two apostrophes in a character constant to represent a single apostrophe character, e.g.,

```
"He wrote a ""Hello World"" program as an example."
```

The above is equivalent to:

```
He wrote a "Hello World" program as an example.
```

```
''''
```

The above is equivalent to a single apostrophe character.

HLA provides a couple of other features that eliminate the need for escape characters. In addition to concatenating two adjacent string constants to form a longer string constant, HLA will also concatenate any combination of adjacent character and string constants to form a single string constant:

```
`1' `2' `3' // Equivalent to "123"
"He wrote a " `"' "Hello World" `"' " program as an example."
```

Note that the two “He wrote...” strings in the above examples are identical to HLA.

HLA provides a second way to specify character constants that handles all the other C/C++ escape character sequences: the ASCII code literal character constant. This literal character constant form uses the syntax:

```
#integer_constant
```

This form creates a character constant whose value is the ASCII code specified by *integer_constant*. The numeric constant can be a decimal, hexadecimal, or binary value, e.g.,

```
#13    #$d    %#1101    // All three are the same character, a
                          // carriage return.
```

Since you may concatenate character literals with strings, and the *#constant* form is a character literal, the following are all legal strings:

```
"Hello World" #13 #10 // #13 #10 is the Windows newline sequence
                          // (carriage return followed by line feed).

"Error: Bad Value" #7 // #7 is the bell character.
"He wrote a " # $22 "Hello World" # $22 " program as an example."
```

Since \$22 is the ASCII code for the quote character, this last example is yet a third form of the “He wrote...” string literal.

1.4.3 String and Text Constants in the CONST Section

String and text constants in the CONST section use the following declaration syntax:

```
const
AStringConst:    string := "123";
ATextConst:      text   := "123";
```

Other than the data type of these two constants, their declarations are identical. However, their behavior in an HLA program is quite different.

Whenever HLA encounters a symbolic string constant within your program, it substitutes the string literal constant in place of the string name. So a statement like “`stdout.put(AStringConst);`” prints the string “123” (without quotes, of course) to the display. No real surprise here.

Whenever HLA encounters a symbolic text constant within your program, it substitutes the text of that string (rather than the string literal constant) for the identifier. That is, HLA substitutes the characters

between the delimiting quotes in place of the symbolic text constant. Therefore, the following statement is perfectly legal given the declarations above:

```
mov( ATextConst, al );      // equivalent to mov( 123, al );
```

Note that substituting *AStringConst* for *ATextConst* in this example is illegal:

```
mov( AStringConst, al );   // equivalent to mov( "123", al );
```

This latter example is illegal because you cannot move a string literal constant into the AL register.

Whenever HLA encounters a symbolic text constant in your program, it immediately substitutes the value of the text constant's string for that text constant and continues the compilation as though you had written the text constant's value rather than the symbolic identifier in your program. This can save some typing and help make your programs a little more readable if you often enter some sequence of text in your program. For example, consider the *nl* (newline) text constant declaration found in the HLA `stdio.hhf` library header file:

```
const
  nl: text := "#$d #$a"; // Windows version. Linux is just a line feed.
```

Whenever HLA encounters the symbol *nl*, it immediately substitutes the value of the string “#\$d #\$a” for the *nl* identifier. When HLA sees the *#\$d* (carriage return) character constant followed by the *#\$a* (line feed) character constants, it concatenates the two to form the string containing the Windows newline sequence (a carriage return followed by a line feed). Consider the following two statements:

```
stdout.put( "Hello World", nl );
stdout.put( "Hello World" nl );
```

(Notice that the second statement above does not separate the string literal and the *nl* symbol with a comma.) In the first example, HLA emits code that prints the string “Hello World” and then emits some additional code that prints a newline sequence. In the second example, HLA expands the *nl* symbol as follows:

```
stdout.put( "Hello World" #$d #$a );
```

Now HLA sees a string literal constant (“Hello World”) followed by two character constants. It concatenates the three of them together to form a single string and then prints this string with a single call. Therefore, leaving off the comma between the string literal and the *nl* symbol produces slightly more efficient code. Keep in mind that this only works with string literal constants. You cannot concatenate string variables, or a string variable with a string literal, by using this technique.

Linux users should note that the Linux end of line sequence is just a single linefeed character. Therefore, the declaration for *nl* is slightly different in Linux.

In the constant section, if you specify only a constant identifier and a string constant (i.e., you do not supply a type), HLA defaults to type *string*. If you want to declare a *text* constant you must explicitly supply the type.

```
const
  AStrConst := "String Constant";
  ATextConst: text := "mov( 0, eax );";
```

1.4.4 Constant Expressions

Thus far, this chapter has given the impression that a symbolic constant definition consists of an identifier, an optional type, and a literal constant. Actually, HLA constant declarations can be a lot more sophisticated than this because HLA allows the assignment of a constant expression, not just a literal constant, to a symbolic constant. The generic constant declaration takes one of the following two forms:

```
Identifier : typeName := constant_expression ;
Identifier := constant_expression ;
```

Constant expressions take the familiar form you're used to in high level languages like C/C++ and Pascal. They may contain literal constant values, previously declared symbolic constants, and various arithmetic operators. The following lists some of the operations possible in a constant expression:

Arithmetic Operators

- (unary negation) Negates the expression immediately following the "-".
- * Multiplies the integer or real values around the asterisk.
- div Divides the left integer operand by the right integer operand producing an integer (truncated) result.
- mod Divides the left integer operand by the right integer operand producing an integer remainder.
- / Divides the left numeric operand by the second numeric operand producing a floating point result.
- + Adds the left and right numeric operands.
- Subtracts the right numeric operand from the left numeric operand.

Comparison Operators

- =, == Compares left operand with right operand. Returns TRUE if equal.
- <>, != Compares left operand with right operand. Returns TRUE if not equal.
- < Returns true if left operand is less than right operand.
- <= Returns true if left operand is <= right operand.
- > Returns true if left operand is greater than right operand.
- >= Returns true if left operand is >= right operand.

Logical Operators⁵:

- & For boolean operands, returns the logical AND of the two operands.
- | For boolean operands, returns the logical OR of the two operands.
- ^ For boolean operands, returns the logical exclusive-OR.
- ! Returns the logical NOT of the single operand following "!".

Bitwise Logical Operators:

- & For integer numeric operands, returns bitwise AND of the operands.
- | For integer numeric operands, returns bitwise OR of the operands.
- ^ For integer numeric operands, returns bitwise XOR of the operands.
- ! For an integer numeric operand, returns bitwise NOT of the operand.

String Operators:

- '+' Returns the concatenation of the left and right string operands.

The constant expression operators follow standard precedence rules; you may use the parentheses to override the precedence if necessary. See the HLA reference in the appendix for the exact precedence relationships between the operators. In general, if the precedence isn't obvious, use parentheses to exactly state the order of evaluation. HLA actually provides a few more operators than these, though the ones above are the ones you will most commonly use. Please see the HLA documentation for a complete list of constant expression operators.

If an identifier appears in a constant expression, that identifier must be a constant identifier that you have previously defined in your program. You may not use variable identifiers in a constant expression; their values are not defined at compile-time when HLA evaluates the constant expression. Also, don't confuse compile-time and run-time operations:

```
// Constant expression, computed while HLA is compiling your program:
```

5. Note to C/C++ and Java users. HLA's constant expressions use complete boolean evaluation rather than short-circuit boolean evaluation. Hence, HLA constant expressions do not behave identically to C/C++/Java expressions.

```

const
    x      := 5;
    y      := 6;
    Sum    := x + y;

// Run-time calculation, computed while your program is running, long after
// HLA has compiled it:

    mov( x, al );
    add( y, al );

```

HLA directly interprets the value of a constant expression during compilation. It does not emit any machine instructions to compute “x+y” in the constant expression above. Instead, it directly computes the sum of these two constant values. From that point forward in the program, HLA associates the value 11 with the constant *Sum* just as if the program had contained the statement “Sum := 11;” rather than “Sum := x+y;” On the other hand, HLA does not precompute the value 11 in AL for the MOV and ADD instructions above⁶, it faithfully emits the object code for these two instructions and the 80x86 computes their sum when the program is run (sometime after the compilation is complete).

In general, constant expressions don’t get very sophisticated. Usually, you’re adding, subtracting, or multiplying two integer values. For example, the following CONST section defines a set of constants that have consecutive values:

```

const
    TapeDAT    := 1;
    Tape8mm    := TapeDAT + 1;
    TapeQIC80  := Tape8mm + 1;
    TapeTravan := TapeQIC80 + 1;
    TapeDLT    := TapeTravan + 1;

```

The constants above have the following values: TapeDAT = 1, Tape8mm = 2, TapeQIC80 = 3, TapeTravan = 4, and TapeDLT = 5.

1.4.5 Multiple CONST Sections and Their Order in an HLA Program

Although CONST sections must appear in the declaration section of an HLA program (e.g., between the “PROGRAM *pgmname*;” header and the corresponding “BEGIN *pgmname*;” statement), they do not have to appear before or after any other items in the declaration section. In fact, like the variable declaration sections, you can place multiple CONST sections in the declaration section. The only restriction on HLA constant declarations is that you must declare any constant symbol before you use it in your program.

Some C/C++ programmers, for example, are more comfortable writing their constant declarations as follows (since this is closer to C/C++’s syntax for declaring constants):

```

const  TapeDAT    := 1;
const  Tape8mm    := TapeDAT + 1;
const  TapeQIC80  := Tape8mm + 1;
const  TapeTravan := TapeQIC80 + 1;
const  TapeDLT    := TapeTravan + 1;

```

The placement of the CONST section in a program seems to be a personal issue among programmers. Other than the requirements of defining all constants before you use them, you may feel free to insert the constant declaration section anywhere in the declaration section. Some programmers prefer to put all their

6. Technically, if HLA had an optimizer it could replace these two instructions with a single “MOV(11, al);” instruction. HLA v1.x, however, does not do this.

CONST declarations at the beginning of their declaration section, some programmers prefer to spread them throughout declaration section, defining the constants just before they need them for some other purpose. Putting all your constants at the beginning of an HLA declaration section is probably the wisest choice right now. Later in this text you'll see reasons why you might want to define your constants later in a declaration section.

1.4.6 The HLA VAL Section

You cannot change the value of a constant you define in the CONST section. While this seems perfectly reasonable (constants after all, are supposed to be, well, constant), there are different ways we can define the term *constant* and CONST objects only follow the rules of one specific definition. HLA's VAL section lets you define constant objects that follow slightly different rules. This section will discuss the VAL section and the difference between VAL constants and CONST constants.

The concept of “*const-ness*” can exist at two different times: while HLA is compiling your program and later when your program executes (and HLA is no longer running). All reasonable definitions of a constant require that a value not change while the program is running. Whether or not the value of a “constant” can change during compilation is a separate issue. The difference between HLA CONST objects and HLA VAL objects is whether the value of the constant can change during compilation.

Once you define a constant in the CONST section, the value of that constant is immutable from that point forward *both at run-time and while HLA is compiling your program*. Therefore, an instruction like “`mov(SymbolicCONST, EAX);`” always moves the same value into EAX, regardless of where this instruction appears in the HLA main program. Once you define the symbol *SymbolicCONST* in the CONST section, this symbol has the same value from that point forward.

The HLA VAL section lets you declare symbolic constants, just like the CONST section. However, HLA VAL constants can change their value throughout the source code in your program. The following HLA declarations are perfectly legal:

```
val    InitialValue    := 0;
const  SomeVal         := InitialValue + 1;    // = 1
const  AnotherVal      := InitialValue + 2;    // = 2

val    InitialValue    := 100;
const  ALargerVal      := InitialValue;        // = 100
const  LargeValTwo     := InitialValue*2;      // = 200
```

All of the symbols appearing in the CONST sections use the symbolic value *InitialValue* as part of the definition. Note, however, that *InitialValue* has different values at different points in this code sequence; at the beginning of the code sequence *InitialValue* has the value zero, while later it has the value 100.

Remember, at run-time a VAL object is not a variable; it is still a manifest constant and HLA will substitute the current value of a VAL identifier for that identifier⁷. Statements like “`MOV(25, InitialValue);`” are no more legal than “`MOV(25, 0);`” or “`MOV(25, 100);`”

1.4.7 Modifying VAL Objects at Arbitrary Points in Your Programs

If you declare all your VAL objects in the declaration section, it would seem that you would not be able to change the value of a VAL object between the BEGIN and END statements of your program. After all, the VAL section must appear in the declaration section of the program and the declaration section ends before the BEGIN statement. Later, you will learn that most VAL object modifications occur between the BEGIN and END statements; hence, HLA must provide some way to change the value of a VAL object outside the declaration section. The mechanism to do this is the “?” operator.

7. In this context, *current* means the value last assigned to a VAL object looking backward in the source code.

Not only does HLA allow you to change the value of a VAL object outside the declaration section, it allows you to change the value of a VAL object almost *anywhere* in the program. Anywhere a space is allowed inside an HLA program, you can insert a statement of the form:

```
? ValIdentifier := constant_expression ;
```

This means that you could write a short program like the following:

```
program VALdemo;
#include( "stdlib.hhf" );

val
    NotSoConstant := 0;

begin VALdemo;

    mov( NotSoConstant, eax );
    stdout.put( "EAX = ", (type uns32 eax ), nl );

    ?NotSoConstant := 10;
    mov( NotSoConstant, eax );
    stdout.put( "EAX = ", (type uns32 eax ), nl );

    ?NotSoConstant := 20;
    mov( NotSoConstant, eax );
    stdout.put( "EAX = ", (type uns32 eax ), nl );

    ?NotSoConstant := 30;
    mov( NotSoConstant, eax );
    stdout.put( "EAX = ", (type uns32 eax ), nl );

end VALdemo;
```

Program 1.4 Demonstration of VAL Redefinition Using "?" Operator

You probably won't have much use for VAL objects at this time. However, later on you'll see (in the chapter on the HLA compile-time language) how useful VAL objects can be to you.

1.5 The HLA TYPE Section

Let's say that you simply do not like the names that HLA uses for declaring byte, word, double word, real, and other variables. Let's say that you prefer Pascal's naming convention or, perhaps, C's naming convention. You want to use terms like *integer*, *float*, *double*, or whatever. If this were Pascal you could redefine the names in the **type** section of the program. With C you could use a **#define** or a **typedef** statement to accomplish the task. Well, HLA, like Pascal, has it's own TYPE statement that also lets you create aliases of these names. The following example demonstrates how to set up some C/C++/Pascal compatible names in your HLA programs:

```
type
    integer:    int32;
```

```
float:    real32;
double:  real64;
colors:  byte;
```

Now you can declare your variables with more meaningful statements like:

```
static
  i:          integer;
  x:          float;
  HouseColor: colors;
```

If you are an Ada, C/C++, or FORTRAN programmer (or any other language, for that matter), you can pick type names you're more comfortable with. Of course, this doesn't change how the 80x86 or HLA reacts to these variables one iota, but it does let you create programs that are easier to read and understand since the type names are more indicative of the actual underlying types. One warning for C/C++ programmers: don't get too excited and go off and define an *int* data type. Unfortunately, INT is an 80x86 machine instruction (interrupt) and therefore, this is a reserved word in HLA.

The TYPE section is useful for much more than creating type isomorphism (that is, giving a new name to an existing type). The following sections will demonstrate many of the possible things you can do in the TYPE section.

1.6 ENUM and HLA Enumerated Data Types

In a previous section discussing constants and constant expressions, you saw the following example:

```
const  TapeDAT      := 1;
const  Tape8mm     := TapeDAT + 1;
const  TapeQIC80   := Tape8mm + 1;
const  TapeTravan  := TapeQIC80 + 1;
const  TapeDLT     := TapeTravan + 1;
```

This example demonstrates how to use constant expressions to develop a set of constants that contain unique, consecutive, values. There are, however, a couple of problems with this approach. First, it involves a lot of typing (and extra reading when reviewing this program). Second, it's very easy to make a mistake when creating long lists of unique constants and reuse or skip some values. The HLA ENUM type provides a better way to create a list of constants with unique values.

ENUM is an HLA type declaration that lets you associate a list of names with a new type. HLA associates a unique value with each name (that is, it *enumerates* the list). The ENUM keyword typically appears in the TYPE section and you use it as follows:

```
type
  enumTypeID:  enum { comma_separated_list_of_names };
```

The symbol *enumTypeID* becomes a new type whose values are specified by the specified list of names. As a concrete example, consider the data type *TapeDrives* and a corresponding variable declaration of type *TypeDrives*:

```
type
  TapeDrives: enum{ TapeDAT, Tape8mm, TapeQIC80, TapeTravan, TapeDLT};

static
  BackupUnit:  TapeDrives := TapeDAT;

  .
  .
  .

  mov( BackupUnit, al );
```

```

if( al = Tape8mm ) then
    ...
endif;

// etc.

```

By default, HLA reserves one byte of storage for enumerated data types. So the *BackupUnit* variable will consume one byte of memory and you would typically use an eight-bit register to access it⁸. As for the constants, HLA associates consecutive *uns8* constant values starting at zero with each of the enumerated identifiers. In the *TapeDrives* example, the tape drive identifiers would have the values *TapeDAT*=0, *Tape8mm*=1, *TapeQIC80*=2, *TapeTravan*=3, and *TapeDLT*=4. You may use these constants exactly as though you had defined them with these values in a *CONST* section.

1.7 Pointer Data Types

Some people refer to pointers as scalar data types, others refer to them as composite data types. This text will treat them as scalar data types even though they exhibit some tendencies of both scalar and composite data types.

Of course, the place to start is with the question “What is a pointer?” Now you’ve probably experienced pointers first hand in the Pascal, C, or Ada programming languages and you’re probably getting worried right now. Almost everyone has a real bad experience when they first encounter pointers in a high level language. Well, fear not! Pointers are actually *easier* to deal with in assembly language. Besides, most of the problems you had with pointers probably had nothing to do with pointers, but rather with the linked list and tree data structures you were trying to implement with them. Pointers, on the other hand, have lots of uses in assembly language that have nothing to do with linked lists, trees, and other scary data structures. Indeed, simple data structures like arrays and records often involve the use of pointers. So if you’ve got some deep-rooted fear about pointers, well forget everything you know about them. You’re going to learn how *great* pointers really are.

Probably the best place to start is with the definition of a pointer. Just exactly what is a pointer, anyway? Unfortunately, high level languages like Pascal tend to hide the simplicity of pointers behind a wall of abstraction. This added complexity (which exists for good reason, by the way) tends to frighten programmers because *they don’t understand what’s going on*.

Now if you’re afraid of pointers, well, let’s just ignore them for the time being and work with an array. Consider the following array declaration in Pascal:

```
M: array [0..1023] of integer;
```

Even if you don’t know Pascal, the concept here is pretty easy to understand. *M* is an array with 1024 integers in it, indexed from *M[0]* to *M[1023]*. Each one of these array elements can hold an integer value that is independent of all the others. In other words, this array gives you 1024 different integer variables each of which you refer to by number (the array index) rather than by name.

If you encountered a program that had the statement “*M[0]:=100;*” you probably wouldn’t have to think at all about what is happening with this statement. It is storing the value 100 into the first element of the array *M*. Now consider the following two statements:

```

i := 0; (* Assume "i" is an integer variable *)
M [i] := 100;

```

You should agree, without too much hesitation, that these two statements perform the same exact operation as “*M[0]:=100;*”. Indeed, you’re probably willing to agree that you can use any integer expression in the

8. HLA provides a mechanism by which you can specify that enumerated data types consume two or four bytes of memory. See the HLA documentation for more details.

range 0...1023 as an index into this array. The following statements *still* perform the same operation as our single assignment to index zero:

```
i := 5;    (* assume all variables are integers*)
j := 10;
k := 50;
m [i*j-k] := 100;
```

“Okay, so what’s the point?” you’re probably thinking. “Anything that produces an integer in the range 0...1023 is legal. So what?” Okay, how about the following:

```
M [1] := 0;
M [ M [1] ] := 100;
```

Whoa! Now that takes a few moments to digest. However, if you take it slowly, it makes sense and you’ll discover that these two instructions perform the exact same operation you’ve been doing all along. The first statement stores zero into array element $M[1]$. The second statement fetches the value of $M[1]$, which is an integer so you can use it as an array index into M , and uses that value (zero) to control where it stores the value 100.

If you’re willing to accept the above as reasonable, perhaps bizarre, but usable nonetheless, then you’ll have no problems with pointers. *Because $m[1]$ is a pointer!* Well, not really, but if you were to change “M” to “memory” and treat this array as all of memory, this is the exact definition of a pointer.

1.7.1 Using Pointers in Assembly Language

A pointer is simply a memory location whose value is the address (or index, if you prefer) of some other memory location. Pointers are very easy to declare and use in an assembly language program. You don’t even have to worry about array indices or anything like that.

An HLA pointer is a 32 bit value that may contain the address of some other variable. If you have a dword variable p that contains \$1000_0000, then p “points” at memory location \$1000_0000. To access the dword that p points at, you could use code like the following:

```
mov( p, ebx );      // Load EBX with the value of pointer p.
mov( [ebx], eax );  // Fetch the data that p points at.
```

By loading the value of p into EBX this code loads the value \$1000_0000 into EBX (assuming p contains \$1000_0000 and, therefore, points at memory location \$1000_0000). The second instruction above loads the EAX register with the word starting at the location whose offset appears in EBX. Since EBX now contains \$1000_0000, this will load EAX from locations \$1000_0000 through \$1000_0003.

Why not just load EAX directly from location \$1000_0000 using an instruction like “MOV(mem, EAX);” (assuming mem is at address \$1000_0000)? Well, there are lots of reasons. But the primary reason is that this single instruction always loads EAX from location mem . You cannot change the location from which it loads EAX. The former instructions, however, always load EAX from the location where p is pointing. This is very easy to change under program control. In fact, the simple instruction “MOV(&mem2, p);” will cause those same two instructions above to load EAX from $mem2$ the next time they execute. Consider the following instructions:

```
mov( &i, p );      // Assume all variables are STATIC variables.
.
.
.
if( some_expression ) then

    mov( &j, p );  // Assume the code above skips this instruction and
.                // you get to the next instruction by jumping
.                // to this point from somewhere else.
.
.
```

```

endif;
mov( p, ebx );      // Assume both of the above code paths wind up
mov( [ebx], eax ); // down here.

```

This short example demonstrates two execution paths through the program. The first path loads the variable *p* with the address of the variable *i*. The second path through the code loads *p* with the address of the variable *j*. Both execution paths converge on the last two MOV instructions that load EAX with *i* or *j* depending upon which execution path was taken. In many respects, this is like a *parameter* to a procedure in a high level language like Pascal. Executing the same instructions accesses different variables depending on whose address (*i* or *j*) winds up in *p*.

1.7.2 Declaring Pointers in HLA

Since pointers are 32 bits long, you could simply use the dword directive to allocate storage for your pointers. However, there is a much better way to do this: HLA provides the POINTER TO phrase specifically for declaring pointer variables. Consider the following example:

```

static
    b:          byte;
    d:          dword;
    pByteVar:   pointer to byte := &b;
    pDWordVar:  pointer to dword := &d;

```

This example demonstrates that it is possible to initialize as well as declare pointer variables in HLA. Note that you may only take addresses of static variables (STATIC, READONLY, and STORAGE objects) with the address-of operator, so you can only initialize pointer variables with the addresses of static objects.

You can also define your own pointer types in the TYPE section of an HLA program. For example, if you often use pointers to characters, you'll probably want to use a TYPE declaration like the one in the following example:

```

type
    ptrChar:   pointer to char;

static
    cString:  ptrChar;

```

1.7.3 Pointer Constants and Pointer Constant Expressions

HLA allows two literal pointer constant forms: the address-of operator followed by the name of a static variable or the constant zero. In addition to these two literal pointer constants, HLA also supports simple pointer constant expressions.

The constant zero represents the NULL or NIL pointer, that is, an illegal address that does not exist⁹. Programs typically initialize pointers with NULL to indicate that a pointer has explicitly *not* been initialized. The HLA Standard Library predefines both the “NULL” and “nil” constants in the memory.hhf header file¹⁰.

In addition to simple address literals and the value zero, HLA allows very simple constant expressions wherever a pointer constant is legal. Pointer constant expressions take one of the two following forms:

```

&StaticVarName + PureConstantExpression
&StaticVarName - PureConstantExpression

```

9. Actually, address zero does exist, but if you try to access it under Windows or Linux you will get a general protection fault.

10. NULL is for C/C++ programmers and nil is familiar to Pascal/Delphi programmers.

The *PureConstantExpression* term is a numeric constant expression that does not involve any pointer constants. This type of expression produces a memory address that is the specified number of bytes before or after (“-” or “+”, respectively) the *StaticVarName* variable in memory.

Since you can create pointer constant expressions, it should come as no surprise to discover that HLA lets you define manifest pointer constants in the CONST section. The following program demonstrates how you can do this.

```

program PtrConstDemo;
#include( "stdlib.hhf" );

static
    b: byte := 0;
    byte 1, 2, 3, 4, 5, 6, 7;

const
    pb:= &b + 1;

begin PtrConstDemo;

    mov( pb, ebx );
    mov( [ebx], al );
    stdout.put( "Value at address pb = $", al, nl );

end PtrConstDemo;

```

Program 1.5 Pointer Constant Expressions in an HLA Program

Upon execution, this program prints the value of the byte just beyond *b* in memory (which contains the value \$01).

1.7.4 Pointer Variables and Dynamic Memory Allocation

Pointer variables are the perfect place to store the return result from the HLA Standard Library *malloc* function. The *malloc* function returns the address of the storage it allocates in the EAX register; therefore, you can store the address directly into a pointer variable with a single MOV instruction immediately after a call to *malloc*:

```

type
    bytePtr: pointer to byte;

var
    bPtr: bytePtr;
    .
    .
    .
    malloc( 1024 );           // Allocate a block of 1,024 bytes.
    mov( eax, bPtr );        // Store address of block in bPtr.
    .
    .
    .
    free( bPtr );           // Free the allocated block when done using it.

```

·
·
·

In addition to *malloc* and *free*, the HLA Standard Library provides a *realloc* procedure. The *realloc* routine takes two parameters, a pointer to a block of storage that *malloc* (or *realloc*) previously created, and a new size. If the new size is less than the old size, *realloc* releases the storage at the end of the allocated block back to the system. If the new size is larger than the current block, then *realloc* will allocate a new block and move the old data to the start of the new block, then free the old block.

Typically, you would use *realloc* to correct a bad guess about a memory size you'd made earlier. For example, suppose you want to read a set of values from the user but you won't know how many memory locations you'll need to hold the values until after the user has entered the last value. You could make a wild guess and then allocate some storage using *malloc* based on your estimate. If, during the input, you discover that your estimate was too low, simply call *realloc* with a larger value. Repeat this as often as required until all the input is read. Once input is complete, you can make a call to *realloc* to release any unused storage at the end of the memory block.

The *realloc* procedure uses the following calling sequence:

```
realloc( ExistingPointer, NewSize );
```

Realloc returns a pointer to the newly allocated block in the EAX register.

One danger exists when using *realloc*. If you've made multiple copies of pointers into a block of storage on the heap and then call *realloc* to resize that block, all the existing pointers are now invalid. Effectively *realloc* frees the existing storage and then allocates a new block. That new block may not be in the same memory location as the old block, so any existing pointers (into the block) that you have will be invalid after the *realloc* call.

1.7.5 Common Pointer Problems

There are five common problems programmers encounter when using pointers. Some of these errors will cause your programs to immediately stop with a diagnostic message; other problems are more subtle, yielding incorrect results without otherwise reporting an error or simply affecting the performance of your program without displaying an error. These five problems are

- Using an uninitialized pointer
- Using a pointer that contains an illegal value (e.g., NULL)
- Continuing to use malloc'd storage after that storage has been free'd
- Failing to free storage once the program is done using it
- Accessing indirect data using the wrong data type.

The first problem above is using a pointer variable before you have assigned a valid memory address to the pointer. Beginning programmers often don't realize that declaring a pointer variable only reserves storage for the pointer itself, it does not reserve storage for the data that the pointer references. The following short program demonstrates this problem:

```
// Program to demonstrate use of
// an uninitialized pointer. Note
// that this program should terminate
// with a Memory Access Violation exception.

program UinitPtrDemo;
#include( "stdlib.hhf" );

static
```

```

// Note: by default, variables in the
// static section are initialized with
// zero (NULL) hence the following
// is actually initialized with NULL,
// but that will still cause our program
// to fail because we haven't initialized
// the pointer with a valid memory address.

Uninitialized: pointer to byte;

begin UninitPtrDemo;

    mov( Uninitialized, ebx );
    mov( [ebx], al );
    stdout.put( "Value at address Uninitialized: = $", al, nl );

end UninitPtrDemo;

```

Program 1.6 Uninitialized Pointer Demonstration

Although variables you declare in the `STATIC` section are, technically, initialized; static initialization still doesn't initialize the pointer in this program with a valid address.

Of course, there is no such thing as a truly uninitialized variable on the 80x86. What you really have are variables that you've explicitly given an initial value and variables that just happen to inherit whatever bit pattern was in memory when storage for the variable was allocated. Much of the time, these garbage bit patterns laying around in memory don't correspond to a valid memory address. Attempting to *dereference* such a pointer (that is, access the data in memory at which it points) raises a Memory Access Violation exception.

Sometimes, however, those random bits in memory just happen to correspond to a valid memory location you can access. In this situation, the CPU will access the specified memory location without aborting the program. Although to a naive programmer this situation may seem preferable to aborting the program, in reality this is far worse because your defective program continues to run with a defect without alerting you to the problem. If you store data through an uninitialized pointer, you may very well overwrite the values of other important variables in memory. This defect can produce some very difficult to locate problems in your program.

The second problem programmers have with pointers is storing invalid address values into a pointer. The first problem, above, is actually a special case of this second problem (with garbage bits in memory supplying the invalid address rather than you producing via a miscalculation). The effects are the same; if you attempt to dereference a pointer containing an invalid address you will either get a Memory Access Violation exception or you will access an unexpected memory location.

The third problem listed above is also known as the dangling pointer problem. To understand this problem, consider the following code fragment:

```

malloc( 256 ); // Allocate some storage.
mov( eax, ptr ); // Save address away in a pointer variable.
.
. // Code that use the pointer variable "ptr".
.
free( ptr ); // Free the storage associated with "ptr".
.
. // Code that does not change the value in "ptr".
.
mov( ptr, ebx );
mov( al, [ebx] );

```


In this example you will note that the program allocates 256 bytes of storage and saves the address of that storage away in the *ptr* variable. Then the code uses this block of 256 bytes for a while and frees the storage, returning it to the system for other uses. Note that calling *free* does not change the value of *ptr* in any way; *ptr* still points at the block of memory allocated by *malloc* earlier. Indeed, *free* does not change any data in this block, so upon return from *free*, *ptr* still points at the data stored into the block by this code. However, note that the call to *free* tells the system that this 256-byte block of memory is no longer needed by the program and the system can use this region of memory for other purposes. The *free* function cannot enforce that fact that you will never access this data again, you are simply promising that you won't. Of course, the code fragment above breaks this promise; as you can see in the last two instructions above the program fetches the value in *ptr* and accesses the data it points at in memory.

The biggest problem with dangling pointers is that you can get away with using them a good part of the time. As long as the system doesn't reuse the storage you've free'd, using a dangling pointer produces no ill effects in your program. However, with each new call to *malloc*, the system may decide to reuse the memory released by that previous call to *free*. When this happens, any attempt to dereference the dangling pointer may produce some unintended consequences. The problems range from reading data that has been overwritten (by the new, legal, use of the data storage), to overwriting the new data, to (the worst case) overwriting system heap management pointers (doing so will probably cause your program to crash). The solution is clear: *never use a pointer value once you free the storage associated with that pointer.*

Of all the problems, the fourth (failing to free allocated storage) will probably have the least impact on the proper operation of your program. The following code fragment demonstrates this problem:

```

malloc( 256 );
mov( eax, ptr );
.          // Code that uses the data where ptr is pointing.
.          // This code does not free up the storage
.          // associated with ptr.
malloc( 512 );
mov( eax, ptr );

// At this point, there is no way to reference the original
// block of 256 bytes pointed at by ptr.

```

In this example the program allocates 256 bytes of storage and references this storage using the *ptr* variable. At some later time, the program allocates another block of bytes and overwrites the value in *ptr* with the address of this new block. Note that the former value in *ptr* is lost. Since this address no longer exists in the program, there is no way to call *free* to return the storage for later use. As a result, this memory is no longer available to your program. While making 256 bytes of memory inaccessible to your program may not seem like a big deal, imagine now that this code is in a loop that repeats over and over again. With each execution of the loop the program loses another 256 bytes of memory. After a sufficient number of loop iterations, the program will exhaust the memory available on the heap. This problem is often called a *memory leak* because the effect is the same as though the memory bits were leaking out of your computer (yielding less and less available storage) during program execution¹¹.

Memory leaks are far less damaging than using dangling pointers. Indeed, there are only two problems with memory leaks: the danger of running out of heap space (which, ultimately, may cause the program to abort, though this is rare) and performance problems due to virtual memory page swapping. Nevertheless, you should get in the habit of always free all storage once you are done using it. Note that when your program quits, the operating system reclaims all storage including the data lost via memory leaks. Therefore, memory lost via a leak is only lost to your program, not the whole system.

The last problem with pointers is the lack of type-safe access. HLA cannot and does not enforce pointer type checking. For example, consider the following program:

11. Note that the storage isn't lost from you computer; once your program quits it returns all memory (including unfree'd storage) to the O/S. The next time the program runs it will start with a clean slate.

```

// Program to demonstrate use of
// lack of type checking in pointer
// accesses.

program BadTypePtrDemo;
#include( "stdlib.hhf" );

static
    ptr:    pointer to char;
    cnt:    uns32;

begin BadTypePtrDemo;

    // Allocate sufficient characters
    // to hold a line of text input
    // by the user:

    malloc( 256 );
    mov( eax, ptr );

    // Okay, read the text a character
    // at a time by the user:

    stdout.put( "Enter a line of text: " );
    stdin.flushInput();
    mov( 0, cnt );
    mov( ptr, ebx );
    repeat

        stdin.getc();           // Read a character from the user.
        mov( al, [ebx] );       // Store the character away.
        inc( cnt );             // Bump up count of characters.
        inc( ebx );             // Point at next position in memory.

    until( stdin.eoln());

    // Okay, we've read a line of text from the user,
    // now display the data:

    mov( ptr, ebx );
    for( mov( cnt, ecx ); ecx > 0; dec( ecx ) ) do

        mov( [ebx], eax );
        stdout.put( "Current value is $", eax, nl );
        inc( ebx );

    endfor;
    free( ptr );

end BadTypePtrDemo;

```

Program 1.7 Type-Unsafe Pointer Access Example

This program reads in data from the user as character values and then displays the data as double word hexadecimal values. While a powerful feature of assembly language is that it lets you ignore data types at

will and automatically coerce the data without any effort, this power is a two-edged sword. If you make a mistake and access indirect data using the wrong data type, HLA and the 80x86 may not catch the mistake and your program may produce inaccurate results. Therefore, you need to take care when using pointers and indirection in your programs that you use the data consistently with respect to data type.

1.8 Putting It All Together

This chapter contains an eclectic combination of subjects. It begins with a discussion of the INTMUL, BOUND, and INTO instructions that will prove useful throughout this text. Then this chapter discusses how to declare constants and data types, including enumerated data types. This chapter also introduces constant expressions and pointers. The following chapters in this text will make extensive use of these concepts.

Introduction to Character Strings

Chapter Two

2.1 Chapter Overview

This chapter discusses how to declare and use character strings in your programs. While not a complete treatment of this subject (additional material appears later in this text), this chapter will provide sufficient information to allow basic string manipulation within your HLA programs.

2.2 Composite Data Types

Composite data types are those that are built up from other (generally scalar) data types. This chapter will cover one of the more important composite data types – the character string. A string is a good example of a composite data type – it is a data structure built up from a sequence of individual characters and some other data.

2.3 Character Strings

After integer values, character strings are probably the most popular data type that modern programs use. The 80x86 does support a handful of string instructions, but these instructions are really intended for block memory operations, not a specific implementation of a character string. Therefore, this section will concentrate mainly on the HLA definition of character strings and also discuss the string handling routines available in the HLA Standard Library.

In general, a *character string* is a sequence of ASCII characters that possesses two main attributes: a *length* and the *character data*. Different languages use different data structures to represent strings. To better understand the reasoning behind HLA strings, it is probably instructive to look at two different string representations popularized by various high level languages.

Without question, *zero-terminated strings* are probably the most common string representation in use today because this is the native string format for C/C++ and programs written in C/C++. A zero terminated string consists of a sequence of zero or more ASCII characters ending with a byte containing zero. For example, in C/C++, the string “abc” requires four characters: the three characters ‘a’, ‘b’, and ‘c’ followed by a byte containing zero. As you’ll soon see, HLA character strings are upwards compatible with zero terminated strings, but in the meantime you should note that it is very easy to create zero terminated strings in HLA. The easiest place to do this is in the STATIC section using code like the following:

```
static
    zeroTerminatedString: char; @nostorage;
                        byte "This is the zero terminated string", 0;
```

Remember, when using the @NOSTORAGE option, no space is actually reserved for a variable declaration, so the *zeroTerminatedString* variable’s address in memory corresponds to the first character in the following BYTE directive. Whenever a character string appears in the BYTE directive as it does here, HLA emits each character in the string to successive memory locations. The zero value at the end of the string properly terminates this string.

Zero terminated strings have two principle attributes: they are very simple to implement and the strings can be any length. On the other hand, zero terminated string has a few drawbacks. First, though not usually important, zero terminated strings cannot contain the NUL character (whose ASCII code is zero). Generally, this isn’t a problem, but it does create havoc once in a great while. The second problem with zero terminated strings is that many operations on them are somewhat inefficient. For example, to compute the length of a zero terminated string you must scan the entire string looking for that zero byte (counting each

character as you encounter it). The following program fragment demonstrates how to compute the length of the string above:

```

mov( &zeroTerminatedString, ebx );
mov( 0, eax );
while( (type byte [ebx]) <> 0 ) do

    inc( ebx );
    inc( eax );

endwhile;

// String length is now in EAX.

```

As you can see from this code, the time it takes to compute the length of the string is proportional to the length of the string; as the string gets longer it will take longer to compute its length.

A second string format, *length-prefixed strings*, overcomes some of the problems with zero terminated strings. Length-prefixed strings are common in languages like Pascal; they generally consist of a length byte followed by zero or more character values. The first byte specifies the length of the string, the remaining bytes (up to the specified length) are the character data itself. In a length-prefixed scheme, the string “abc” would consist of the four bytes \$03 (the string length) followed by ‘a’, ‘b’, and ‘c’. You can create length prefixed strings in HLA using code like the following:

```

data
    lengthPrefixedString:char;
        byte 3, "abc";

```

Counting the characters ahead of time and inserting them into the byte statement, as was done here, may seem like a major pain. Fortunately, there are ways to have HLA automatically compute the string length for you.

Length-prefixed strings solve the two major problems associated with zero-terminated strings. It is possible to include the NUL character in length-prefixed strings and those operations on zero terminated strings that are relatively inefficient (e.g., string length) are more efficient when using length prefixed strings. However, length prefixed strings suffer from their own drawbacks. The principal drawback to length-prefixed strings, as described, is that they are limited to a maximum of 255 characters in length (assuming a one-byte length prefix).

HLA uses an expanded scheme for strings that is upwards compatible with both zero-terminated and length-prefixed strings. HLA strings enjoy the advantages of both zero-terminated and length-prefixed strings without the disadvantages. In fact, the only drawback to HLA strings over these other formats is that HLA strings consume a few additional bytes (the overhead for an HLA string is nine bytes compared to one byte for zero-terminated or length-prefixed strings; the overhead being the number of bytes needed above and beyond the actual characters in the string).

An HLA string value consists of four components. The first element is a double word value that specifies the maximum number of characters that the string can hold. The second element is a double word value specifying the current length of the string. The third component is the sequence of characters in the string. The final component is a zero terminating byte. You could create an HLA-compatible string in the STATIC section using the following code¹:

```

static
    dword 11;
    dword 11;
    TheString: char; @nostorage;
        byte "Hello there";
        byte 0;

```

1. Actually, there are some restrictions on the placement of HLA strings in memory. This text will not cover those issues. See the HLA documentation for more details.

Note that the address associated with the HLA string is the address of the first character, not the maximum or current length values.

“So what is the difference between the current and maximum string lengths?” you’re probably wondering. Well, in a fixed string like the above they are usually the same. However, when you allocate storage for a string variable at run-time, you will normally specify the maximum number of characters that can go into the string. When you store actual string data into the string, the number of characters you store must be less than or equal to this maximum value. The HLA Standard Library string routines will raise an exception if you attempt to exceed this maximum length (something the C/C++ and Pascal formats can’t do).

The terminating zero byte at the end of the HLA string lets you treat an HLA string as a zero-terminated string if it is more efficient or more convenient to do so. For example, most calls to Windows and Linux require zero-terminated strings for their string parameters. Placing a zero at the end of an HLA string ensures compatibility with Windows, Linux, and other library modules that use zero-terminated strings.

2.4 HLA Strings

As noted in the previous section, HLA strings consist of four components: a maximum length, a current string length, character data, and a zero terminating byte. However, HLA never requires you to create string data by manually emitting these components yourself. HLA is smart enough to automatically construct this data for you whenever it sees a string literal constant. So if you use a string constant like the following, understand that somewhere HLA is creating the four-component string in memory for you:

```
stdout.put( "This gets converted to a four-component string by HLA" );
```

HLA doesn’t actually work directly with the string data described in the previous section. Instead, when HLA sees a string object it always works with a *pointer* to that object rather than the object directly. Without question, this is the most important fact to know about HLA strings, and is the biggest source of problems beginning HLA programmers have with strings in HLA: *strings are pointers!* A string variable consumes exactly four bytes, the same as a pointer (because it *is* a pointer!). Having said all that, let’s take a look at a simple string variable declaration in HLA:

```
static
    StrVariable: string;
```

Since a string variable is a pointer, you must initialize it before you can use it. There are three general ways you may initialize a string variable with a legal string address: using static initializers, using the *stralloc* routine, or calling some other HLA Standard Library that initializes a string or returns a pointer to a string.

In one of the static declaration sections that allow initialized variables (STATIC, and READONLY) you can initialize a string variable using the standard initialization syntax, e.g.,

```
static
    InitializedString: string := "This is my string";
```

Note that this does not initialize the string variable with the string data. Instead, HLA creates the string data structure (see the previous section) in a special, hidden, memory segment and initializes the *InitializedString* variable with the address of the first character in this string (the “T” in “This”). *Remember, strings are pointers!* The HLA compiler places the actual string data in a read-only memory segment. Therefore, you cannot modify the characters of this string literal at run-time. However, since the string variable (a pointer, remember) is in the static section, you can change the string variable so that it points at different string data.

Since string variables are pointers, you can load the value of a string variable into a 32-bit register. The pointer itself points at the first character position of the string. You can find the current string length in the double word four bytes prior to this address, you can find the maximum string length in the double word eight bytes prior to this address. The following program demonstrates one way to access this data².

```

// Program to demonstrate accessing Length and Maxlength fields of a string.

program StrDemo;
#include( "stdlib.hhf" );

static
  theString:string := "String of length 19";

begin StrDemo;

  mov( theString, ebx ); // Get pointer to the string.

  mov( [ebx-4], eax );   // Get current length
  mov( [ebx-8], ecx );   // Get maximum length

  stdout.put
  (
    "theString = '", theString, "'", nl,
    "length( theString )= ", (type uns32 eax ), nl,
    "maxLength( theString )= ", (type uns32 ecx ), nl
  );

end StrDemo;

```

Program 2.1 Accessing the Length and Maximum Length Fields of a String

When accessing the various fields of a string variable it is not wise to access them using fixed numeric offsets as done in this example. In the future, the definition of an HLA string may change slightly. In particular, the offsets to the maximum length and length fields are subject to change. A safer way to access string data is to coerce your string pointer using the *str.strRec* data type. The *str.strRec* data type is a record data type (see “Records, Unions, and Name Spaces” on page 483) that defines symbolic names for the offsets of the length and maximum length fields in the string data type. Were the offsets to the length and maximum length fields to change in a future version of HLA, then the definitions in *str.strRec* would also change, so if you use *str.strRec* then recompiling your program would automatically make any necessary changes to your program.

To use the *str.strRec* data type properly, you must first load the string pointer into a 32-bit register, e.g., “MOV(SomeString, EBX);” Once the pointer to the string data is in a register, you can coerce that register to the *str.strRec* data type using the HLA construct “(type str.strRec [EBX])”. Finally, to access the length or maximum length fields, you would use either “(type str.strRec [EBX]).length” or “(type str.strRec [EBX]).MaxStrLen” (respectively). Although there is a little more typing involved (versus using simple offsets like “-4” or “-8”), these forms are far more descriptive and much safer than straight numeric offsets. The following program corrects the previous example by using the *str.strRec* data type.

```

// Program to demonstrate accessing Length and Maxlength fields of a string.

program LenMaxlenDemo;
#include( "stdlib.hhf" );

static

```

2. Note that this scheme is not recommended. If you need to extract the length information from a string, use the routines provided in the HLA string library for this purpose.


```

theString:string := "String of length 19";

begin LenMaxlenDemo;

    mov( theString, ebx ); // Get pointer to the string.

    mov( (type str.strRec [ebx]).length, eax ); // Get current length
    mov( (type str.strRec [ebx]).MaxStrLen, ecx ); // Get maximum length

    stdout.put
    (
        "theString = '", theString, "' ", nl,
        "length( theString )= ", (type uns32 eax ), nl,
        "maxLength( theString )= ", (type uns32 ecx ), nl
    );

end LenMaxlenDemo;

```

Program 2.2 Correct Way to Access Length and MaxStrLen Fields of a String

A second way to manipulate strings in HLA is to allocate storage on the heap to hold string data. Because strings can't directly use pointers returned by *malloc* (since strings need to access eight bytes prior to the pointer address), you shouldn't use *malloc* to allocate storage for string data. Fortunately, the HLA Standard Library memory module provides a memory allocation routine specifically designed to allocate storage for strings: *stralloc*. Like *malloc*, *stralloc* expects a single dword parameter. This value specifies the (maximum) number of characters needed in the string. The *stralloc* routine will allocate the specified number of bytes of memory, plus between nine and thirteen additional bytes to hold the extra string information³.

The *stralloc* routine will allocate storage for a string, initialize the maximum length to the value passed as the *stralloc* parameter, initialize the current length to zero, and store a zero (terminating byte) in the first character position of the string. After all this, *stralloc* returns the address of the zero terminating byte (that is, the address of the first character element) in the EAX register.

Once you've allocated storage for a string, you can call various string manipulation routines in the HLA Standard Library to operate on the string. The next section will discuss the HLA string routines in detail; this section will introduce a couple of string related routines for the sake of example. The first such routine is the "stdin.gets(strvar)". This routine reads a string from the user and stores the string data into the string storage pointed at by the string parameter (*strvar* in this case). If the user attempts to enter more characters than you've allocated for the string, then *stdin.gets* raises the *ex.StringOverflow* exception. The following program demonstrates the use of *stralloc*.

```

// Program to demonstrate stralloc and stdin.gets.

program strallocDemo;
#include( "stdlib.hhf" );

static
    theString:string;

begin strallocDemo;

    stralloc( 16 ); // Allocate storage for the string and store

```

3. *Stralloc* may allocate more than nine bytes for the overhead data because the memory allocated to an HLA string must always be double word aligned and the total length of the data structure must be an even multiple of four.

```

mov( eax, theString ); // the pointer into the string variable.

// Prompt the user and read the string from the user:

stdout.put( "Enter a line of text (16 chars, max): " );
stdin.flushInput();
stdin.gets( theString );

// Echo the string back to the user:

stdout.put( "The string you entered was: ", theString, nl );

end strallocDemo;

```

Program 2.3 Reading a String from the User

If you look closely, you see a slight defect in the program above. It allocates storage for the string by calling *stralloc* but it never frees the storage allocated. Even though the program immediately exits after the last use of the string variable, and the operating system will deallocate the storage anyway, it's always a good idea to explicitly free up any storage you allocate. Doing so keeps you in the habit of freeing allocated storage (so you don't forget to do it when it's important) and, also, programs have a way of growing such that an innocent defect that doesn't affect anything in today's program becomes a show-stopping defect in tomorrow's version.

To free storage allocated via *stralloc*, you must call the corresponding *strfree* routine, passing the string pointer as the single parameter. The following program is a correction of the previous program with this minor defect corrected:

```

// Program to demonstrate stralloc, strfree, and stdin.gets.

program strfreeDemo;
#include( "stdlib.hhf" );

static
    theString:string;

begin strfreeDemo;

    stralloc( 16 ); // Allocate storage for the string and store
    mov( eax, theString ); // the pointer into the string variable.

    // Prompt the user and read the string from the user:

    stdout.put( "Enter a line of text (16 chars, max): " );
    stdin.flushInput();
    stdin.gets( theString );

    // Echo the string back to the user:

    stdout.put( "The string you entered was: ", theString, nl );

    // Free up the storage allocated by stralloc:

    strfree( theString );

end strfreeDemo;

```

Program 2.4 **Corrected Program that Reads a String from the User**

When looking at this corrected program, please take note that the *stdin.gets* routine expects you to pass it a string parameter that points at an allocated string object. Without question, one of the most common mistakes beginning HLA programmers make is to call *stdin.gets* and pass it a string variable that has not been initialized. This may be getting old now, but keep in mind that *strings are pointers!* Like pointers, if you do not initialize a string with a valid address, your program will probably crash when you attempt to manipulate that string object. The call to *stralloc* plus moving the returned result into *theString* is how the programs above initialize the string pointer. If you are going to use string variables in your programs, you must ensure that you allocate storage for the string data prior to writing data to the string object.

Allocating storage for a string option is such a common operation that many HLA Standard Library routines will automatically do the allocation to save you the effort. Generally, such routines have an “a_” prefix as part of their name. For example, the *stdin.a_gets* combines a call to *stralloc* and *stdin.gets* into the same routine. This routine, which doesn’t have any parameters, reads a line of text from the user, allocates a string object to hold the input data, and then returns a pointer to the string in the EAX register. The following program is an adaptation of the previous two programs that uses *stdin.a_gets*:

```
// Program to demonstrate  strfree and stdin.a_gets.

program strfreeDemo2;
#include( "stdlib.hhf" );

static
    theString:string;

begin strfreeDemo2;

    // Prompt the user and read the string from the user:

    stdout.put( "Enter a line of text: " );
    stdin.flushInput();
    stdin.a_gets();
    mov( eax, theString );

    // Echo the string back to the user:

    stdout.put( "The string you entered was: ", theString, nl );

    // Free up the storage allocated by stralloc:

    strfree( theString );

end strfreeDemo2;
```

Program 2.5 **Reading a String from the User with *stdin.a_gets***

Note that, as before, you must still free up the storage *stdin.a_gets* allocates by calling the *strfree* routine. One big difference between this routine and the previous two is the fact that HLA will automatically allocate exactly enough space for the string read from the user. In the previous programs, the call to *stralloc*

only allocates 16 bytes. If the user types more than this then the program raises an exception and quits. If the user types less than 16 characters, then some space at the end of the string is wasted. The `stdin.a_gets` routine, on the other hand, always allocates the minimum necessary space for the string read from the user. Since it allocates the storage, there is little chance of overflow⁴.

2.5 Accessing the Characters Within a String

Extracting individual characters from a string is a very common and easy task. In fact, it is so easy that HLA doesn't provide any specific procedure or language syntax to accomplish this - it's easy enough just to use machine instructions to accomplish this. Once you have a pointer to the string data, a simple indexed addressing mode will do the rest of the work for you.

Of course, the most important thing to keep in mind is that *strings are pointers*. Therefore, you cannot apply an indexed addressing mode directly to a string variable and expect to extract characters from the string. I.e., if `s` is a string variable, then “`MOV(s[ebx], al);`” does not fetch the character at position `EBX` in string `s` and place it in the `AL` register. Remember, `s` is just a pointer variable, an addressing mode like `s[ebx]` will simply fetch the byte at offset `EBX` in memory starting at the address of `s` (see Figure 2.1).

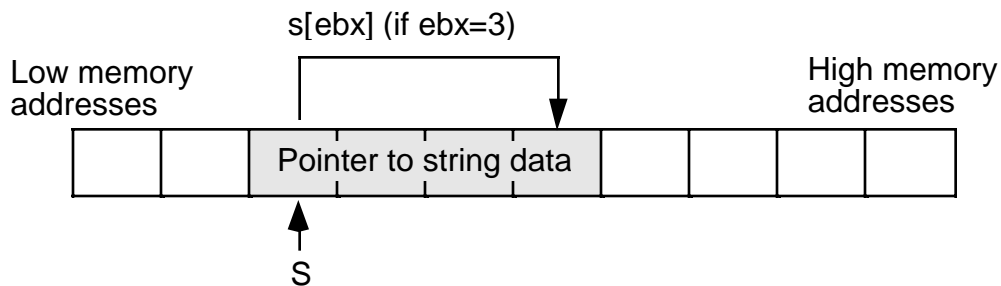


Figure 2.1 Incorrectly Indexing Off a String Variable

In Figure 2.1, assuming `EBX` contains three, “`s[ebx]`” does not access the fourth character in the string `s`, instead it fetches the fourth byte of the pointer to the string data. It is very unlikely that this is the desired effect you would want. Figure 2.2 shows the operation that is necessary to fetch a character from the string, assuming `EBX` contains the value of `s`:

4. Actually, there are limits on the maximum number of characters that `stdin.a_gets` will allocate. This is typically between 1,024 bytes and 4,096 bytes; See the HLA Standard Library source listings for the exact value.

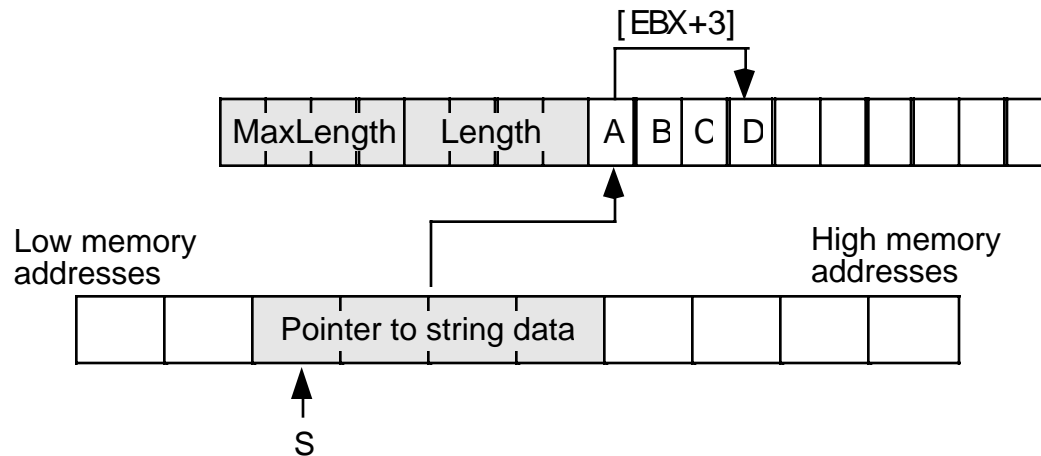


Figure 2.2 Correctly Indexing Off the Value of a String Variable

In Figure 2.2 EBX contains the value of string *s*. The value of *s* is a pointer to the actual string data in memory. Therefore, EBX will point at the first character of the string when you load the value of *s* into EBX. The following code demonstrates how to access the fourth character of string *s* in this fashion:

```
mov( s, ebx );           // Get pointer to string data into EBX.
mov( [ebx+3], al );     // Fetch the fourth character of the string.
```

If you want to load the character at a variable, rather than fixed, offset into the string, then you can use one of the 80x86's scaled indexed addressing modes to fetch the character. For example, if an *uns32* variable contains the desired offset into the string, you could use the following code to access the character at *s[index]*:

```
mov( s, ebx );           // Get address of string data into EBX.
mov( index, ecx );      // Get desired offset into string.
mov( [ebx+ecx], al );   // Get the desired character into AL.
```

There is only one problem with the code above- it does not check to ensure that the character at offset *index* actually exists. If *index* is greater than the current length of the string, then this code will fetch a garbage byte from memory. Unless you can a priori determine that *index* is always less than the length of the string, code like this is dangerous to use. A better solution is to check the index against the string's current length before attempting to access the character. the following code provides one way to do this.

```
mov( s, ebx );
mov( index, ecx );
if( ecx < (type str.strRec [ebx]).Length ) then

    mov( [ebx+ecx], al );

else

    << error, string index is of bounds >>

endif;
```

In the ELSE portion of this IF statement you could take corrective action, print an error message, or raise an exception. If you want to explicitly raise an exception, you can use the HLA RAISE statement to accomplish this. The syntax for the RAISE statement is

```
raise( integer_constant );
```

```
raise( reg32 );
```

The value of the *integer_constant* or 32-bit register must be an exception number. Usually, this is one of the predefined constants in the `excepts.hhf` header file. An appropriate exception to raise when a string index is greater than the length of the string is `ex.StringIndexError`. The following code demonstrates raising this exception if the string index is out of bounds:

```
mov( s, ebx );
mov( index, ecx );
if( ecx < (type str.strRec [ebx]).Length ) then

    mov( [ebx+ecx], al );

else

    raise( ex.StringIndexError );

endif;
```

2.6 The HLA String Module and Other String-Related Routines

Although HLA provides a powerful definition for string data, the real power behind HLA's string capabilities lies in the HLA Standard Library, not in the definition of HLA string data. HLA provides several dozen string manipulation routines that far exceed the capabilities found in standard HLLs like C/C++, Java, or Pascal; indeed, HLA's string handling capabilities rival those in string processing languages like Icon or SNOBOL4. While it is premature to introduce all of HLA's character string handling routines, this chapter will discuss many of the string facilities that HLA provides.

Perhaps the most basic string operation you will need is to assign one string to another. There are three different ways to assign strings in HLA: by reference, by copying a string, and by duplicating a string. Of these, assignment by reference is the fastest and easiest. If you have two strings and you wish to assign one string to the other, a simple and fast way to do this is to copy the string pointer. The following code fragment demonstrates this:

```
static
string1:  string    := "Some String Data";
string2:  string;
.
.
.
mov( string1, eax );
mov( eax, string2 );
.
.
.
```

String assignment by reference is very efficient because it only involves two simple MOV instructions, regardless of the actual length of the string. Assignment by reference works great if you never modify the string data after the assignment operation. Do keep in mind, though, that both string variables (`string1` and `string2` in the example above) *wind up pointing at the same data*. So if you make a change to the data pointed at by one string variable, you will change the string data pointed at by the second string object since both objects point at the same data. The following program demonstrates this problem:

```
// Program to demonstrate the problem
```

```

// with string assignment by reference.

program strRefAssignDemo;
#include( "stdlib.hhf" );

static
    string1:    string;
    string2:    string;

begin strRefAssignDemo;

    // Get a value into string1

    forever

        stdout.put( "Enter a string with at least three characters: " );
        stdin.a_gets();
        mov( eax, string1 );

        breakif( (type str.strRec [eax]).length >= 3 );

        stdout.put( "Please enter a string with at least three chars." nl );

    endfor;

    stdout.put( "You entered: '", string1, "'" nl );

    // Do the string assignment by copying the pointer

    mov( string1, ebx );
    mov( ebx, string2 );

    stdout.put( "String1= '", string1, "'" nl );
    stdout.put( "String2= '", string2, "'" nl );

    // Okay, modify the data in string1 by overwriting
    // the first three characters of the string (note that
    // a string pointer always points at the first character
    // position in the string and we know we've got at least
    // three characters here).

    mov( 'a', (type char [ebx]) );
    mov( 'b', (type char [ebx+1]) );
    mov( 'c', (type char [ebx+2]) );

    // Okay, demonstrate the problem with assignment via
    // pointer copy.

    stdout.put
    (
        "After assigning 'abc' to the first three characters in string1:"
        nl
        nl
    );
    stdout.put( "String1= '", string1, "'" nl );
    stdout.put( "String2= '", string2, "'" nl );

    strfree( string1 );    // Don't free string2 as well!

end strRefAssignDemo;

```

Program 2.6 Problem with String Assignment by Copying Pointers

Since both *string1* and *string2* point at the same string data in this example, any change you make to one string is reflected in the other. While this is sometimes acceptable, most programmers expect assignment to produce a different copy of a string; they expect the semantics of string assignment to produce two unique copies of the string data.

An important point to remember when using *copy by reference* (this term means copying a pointer rather than copying the actual data) is that you have created an *alias* to the string data. The term “alias” means that you have two names for the same object in memory (e.g., in the program above, *string1* and *string2* are two different names for the same string data). When you read a program it is reasonable to expect that different variables refer to different memory objects. Aliases violate this rule, thus making your program harder to read and understand because you’ve got to remember that aliases do not refer to different objects in memory. Failing to keep this in mind can lead to subtle bugs in your program. For instance, in the example above you have to remember that *string1* and *string2* are aliases so as not to free both objects at the end of the program. Worse still, you to remember that *string1* and *string2* are aliases so that you don’t continue to use *string2* after freeing *string1* in this code since *string2* would be a dangling reference at that point.

Since using copy by reference makes your programs harder to read and increases the possibility that you might introduce subtle defects in your programs, you might wonder why someone would use copy by reference at all. There are two reasons for this: first, copy by reference is very efficient; it only involves the execution of two MOV instructions. Second, some algorithms actually depend on copy by reference semantics. Nevertheless, you should carefully consider whether copying string pointers is the appropriate way to do a string assignment in your program before using this technique.

The second way to assign one string to another is to actually copy the string data. The HLA Standard Library *str.cpy* routine provides this capability. A call to the *str.cpy* procedure using the following form:

```
str.cpy( source_string, destination_string );
```

The source and destination strings must be string variables (pointers) or 32-bit registers containing the addresses of the string data in memory.

The *str.cpy* routine first checks the maximum length field of the destination string to ensure that it is at least as big as the current length of the source string. If it is not, then *str.cpy* raises the *ex.StringOverflow* exception. If the maximum string length field of the destination string is at least as big as the current string length of the source string, then *str.cpy* copies the string length, the characters, and the zero terminating byte from the source string to the data area at which the destination string points. When this process is complete, the two strings point at identical data, but they do not point at the same data in memory⁵. The following program is a rework of the previous example using *str.cpy* rather than copy by reference.

```
// Program to demonstrate string assignment using str.cpy.

program strcpyDemo;
#include( "stdlib.hhf" );

static
    string1:    string;
    string2:    string;

begin strcpyDemo;
```

5. Unless, of course, both string pointers contained the same address to begin with, in which case *str.cpy* copies the string data over the top of itself.


```

// Allocate storage for string2:

stralloc( 64 );
mov( eax, string2 );

// Get a value into string1

forever

    stdout.put( "Enter a string with at least three characters: " );
    stdin.a_gets();
    mov( eax, string1 );

    breakif( (type str.strRec [eax]).length >= 3 );

    stdout.put( "Please enter a string with at least three chars." nl );

endfor;

// Do the string assignment via str.cpy

str.cpy( string1, string2 );

stdout.put( "String1= ", string1, "" nl );
stdout.put( "String2= ", string2, "" nl );

// Okay, modify the data in string1 by overwriting
// the first three characters of the string (note that
// a string pointer always points at the first character
// position in the string and we know we've got at least
// three characters here).

mov( string1, ebx );
mov( 'a', (type char [ebx]) );
mov( 'b', (type char [ebx+1]) );
mov( 'c', (type char [ebx+2]) );

// Okay, demonstrate that we have two different strings
// since we used str.cpy to copy the data:

stdout.put
(
    "After assigning 'abc' to the first three characters in string1:"
    nl
    nl
);
stdout.put( "String1= ", string1, "" nl );
stdout.put( "String2= ", string2, "" nl );

// Note that we have to free the data associated with both
// strings since they are not aliases of one another.

strfree( string1 );
strfree( string2 );

end strcpyDemo;

```

Program 2.7 Copying Strings using str.cpy

There are two really important things to note about this program. First, note that this program begins by allocating storage for *string2*. Remember, the *str.cpy* routine does not allocate storage for the destination string, it assumes that the destination string already has storage allocated to it. Keep in mind that *str.cpy* does not initialize *string2*, it only copies data to the location where *string2* is pointing. It is the program's responsibility to initialize the string by allocating sufficient memory before calling *str.cpy*. The second thing to notice here is that the program calls *strfree* to free up the storage for both *string1* and *string2* before the program quits.

Allocating storage for a string variable prior to calling *str.cpy* is so common that the HLA Standard Library provides a routine that allocates and copies the string: *str.a_cpy*. This routine uses the following call syntax:

```
str.a_cpy( source_string );
```

Note that there is no destination string. This routine looks at the length of the source string, allocates sufficient storage, makes a copy of the string, and then returns a pointer to the new string in the EAX register. The following program demonstrates the current example using the *str.a_cpy* procedure.

```
// Program to demonstrate string assignment using str.a_cpy.

program stra_cpyDemo;
#include( "stdlib.hhf" );

static
    string1:    string;
    string2:    string;

begin stra_cpyDemo;

    // Get a value into string1

    forever

        stdout.put( "Enter a string with at least three characters: " );
        stdin.a_gets();
        mov( eax, string1 );

        breakif( (type str.strRec [eax]).length >= 3 );

        stdout.put( "Please enter a string with at least three chars." nl );

    endfor;

    // Do the string assignment via str.a_cpy

    str.a_cpy( string1 );
    mov( eax, string2 );

    stdout.put( "String1= `", string1, "`" nl );
    stdout.put( "String2= `", string2, "`" nl );

    // Okay, modify the data in string1 by overwriting
    // the first three characters of the string (note that
```

```

// a string pointer always points at the first character
// position in the string and we know we've got at least
// three characters here).

mov( string1, ebx );
mov( 'a', (type char [ebx]) );
mov( 'b', (type char [ebx+1]) );
mov( 'c', (type char [ebx+2]) );

// Okay, demonstrate that we have two different strings
// since we used str.cpy to copy the data:

stdout.put
(
    "After assigning 'abc' to the first three characters in string1:"
    nl
    nl
);
stdout.put( "String1= ", string1, "" nl );
stdout.put( "String2= ", string2, "" nl );

// Note that we have to free the data associated with both
// strings since they are not aliases of one another.

strfree( string1 );
strfree( string2 );

end stra_cpyDemo;

```

Program 2.8 Copying Strings using `str.a_cpy`

Warning: Whenever using copy by reference or `str.a_cpy` to assign a string, don't forget to free the storage associated with the string when you are (completely) done with that string's data. Failure to do so may produce a memory leak if you do not have another pointer to the previous string data laying around.

Obtaining the length of a character string is such a common need that the HLA Standard Library provides a `str.length` routine specifically for this purpose. Of course, you can fetch the length by using the `str.strRec` data type to access the length field directly, but constant use of this mechanism can be tiring since it involves a lot of typing. The `str.length` routine provides a more compact and convenient way to fetch the length information. You call `str.length` using one of the following two formats:

```

str.length( Reg32 );
str.length( string_variable );

```

This routine returns the current string length in the EAX register.

Another pair of useful string routines are the `str.cat` and `str.a_cat` procedures. They use the following calling sequence:

```

str.cat( srcStr, destStr );
str.a_cat( src1Str, src2Str );

```

These two routines concatenate two strings (that is, they create a new string by joining the two strings together). The `str.cat` procedure concatenates the source string to the end of the destination string. Before

the concatenation actually takes place, *str.cat* checks to make sure that the destination string is large enough to hold the concatenated result, it raises the *ex.StringOverflow* exception if the destination string is too small.

The *str.a_cat*, as its name suggests, allocates storage for the resulting string before doing the concatenation. This routine will allocate sufficient storage to hold the concatenated result, then it will copy the *src1Str* to the allocated storage, finally it will append the string data pointed at by *src2Str* to the end of this new string and return a pointer to the new string in the EAX register.

Warning: note a potential source of confusion. The *str.cat* procedure concatenates its first operand to the end of the second operand. Therefore, *str.cat* follows the standard (src, dest) operand format present in many HLA statements. The *str.a_cat* routine, on the other hand, has two source operands rather than a source and destination operand. The *str.a_cat* routine concatenates its two operands in an intuitive left-to-right fashion. This is the opposite of *str.cat*. Keep this in mind when using these two routines.

The following program demonstrates the use of the *str.cat* and *str.a_cat* routines:

```
// Program to demonstrate str.cat and str.a_cat.

program strcatDemo;
#include( "stdlib.hhf" );

static
    UserName:    string;
    Hello:       string;
    a_Hello:     string;

begin strcatDemo;

    // Allocate storage for the concatenated result:

    stralloc( 1024 );
    mov( eax, Hello );

    // Get some user input to use in this example:

    stdout.put( "Enter your name: " );
    stdin.flushInput();
    stdin.a_gets();
    mov( eax, UserName );

    // Use str.cat to combine the two strings:

    str.cpy( "Hello ", Hello );
    str.cat( UserName, Hello );

    // Use str.a_cat to combine the string strings:

    str.a_cat( "Hello ", UserName );
    mov( eax, a_Hello );

    stdout.put( "Concatenated string #1 is '", Hello, "' " nl );
    stdout.put( "Concatenated string #2 is '", a_Hello, "' " nl );

    strfree( UserName );
    strfree( a_Hello );
    strfree( Hello );

end strcatDemo;
```

Program 2.9 Demonstration of `str.cat` and `str.a_cat` Routines

The `str.insert` and `str.a_insert` routines are closely related to the string concatenation procedures. However, the `str.insert` and `str.a_insert` routines let you insert one string anywhere into another string, not just at the end of the string. The calling sequences for these two routines are

```
str.insert( src, dest, index );
str.a_insert( StrToInsert, StrToInsertInto, index );
```

These two routines insert the source string (`src` or `StrToInsert`) into the destination string (`dest` or `StrToInsertInto`) starting at character position `index`. The `str.insert` routine inserts the source string directly into the destination string; if the destination string is not large enough to hold both strings, `str.insert` raises an `ex.StringOverflow` exception. The `str.a_insert` routine first allocates a new string on the heap, copies the destination string (`StrToInsertInto`) to the new string, and then inserts the source string (`StrToInsert`) into this new string at the specified offset; `str.a_insert` returns a pointer to the new string in the EAX register.

Indexes into a string are zero-based. This means that if you supply the value zero as the index in `str.insert` or `str.a_insert`, then these routines will insert the source string before the first character of the destination string. Likewise, if the index is equal to the length of the string, then these routines will simply concatenate the source string to the end of the destination string. Note: if the index is greater than the length of the string, the `str.insert` and `str.a_insert` procedures will not raise an exception; instead, they will simply append the source string to the end of the destination string.

The `str.delete` and `str.a_delete` routines let you remove characters from a string. They use the following calling sequence:

```
str.delete( str, StartIndex, Length );
str.a_delete( str, StartIndex, Length );
```

Both routines delete `Length` characters starting at character position `StartIndex` in string `str`. The difference between the two is that `str.delete` deletes the characters directly from `str` whereas `str.a_delete` first allocates storage and copies `str`, then deletes the characters from the new string (leaving `str` untouched). The `str.a_delete` routine returns a pointer to the new string in the EAX register.

The `str.delete` and `str.a_delete` routines are very forgiving with respect to the values you pass in `StartIndex` and `Length`. If `StartIndex` is greater than the current length of the string, these routines do not delete any characters from the string. If `StartIndex` is less than the current length of the string, but `StartIndex+Length` is greater than the length of the string, then these routines will delete all characters from `StartIndex` to the end of the string.

Another very common string operation is the need to copy a portion of a string to a different string without otherwise affecting the source string. The `str.substr` and `str.a_substr` routines provide this capability. These routines use the following calling sequence:

```
str.substr( src, dest, StartIndex, Length );
str.a_substr( src, StartIndex, Length );
```

The `str.substr` routine copies `length` characters, starting at position `StartIndex`, from the `src` string to the `dest` string. The `dest` string must have sufficient storage allocated to hold the new string or `str.substr` will raise an `ex.StringOverflow` exception. If the `StartIndex` value is greater than the length of the string, then `str.substr` will raise an `ex.StringIndexError` exception. If `StartIndex+Length` is greater than the length of the source string, but `StartIndex` is less than the length of the string, then `str.substr` will extract only those characters from `StartIndex` to the end of the string.

The `str.a_substr` procedure behaves in a fashion nearly identical to `str.substr` except it allocates storage on the heap for the destination string. Other than overflow never occurs, `str.a_substr` handles exceptions the identically to `str.substr`⁶. As you can probably guess by now, `str.a_substr` returns a pointer to the newly allocated string in the EAX register.

After you begin working with string data for a little while, the need will invariably arise to compare two strings. A first attempt at string comparison, using the standard HLA relational operators, will compile but not necessarily produce the desired results:

```

mov( s1, eax );
if( eax = s2 ) then

    << code to execute if the strings are equal >>

else

    << code to execute if the strings are not equal >>

endif;
```

As stated above, this code will compile and execute just fine. However, it's probably not doing what you expect it to do. Remember *strings are pointers*. This code compares the two pointers to see if they are equal. If they are equal, clearly the two strings are equal (since both *s1* and *s2* point at the exact same string data). However, the fact that the two pointers are different doesn't necessarily mean that the strings are not equivalent. Both *s1* and *s2* could contain different values (that is, they point at different addresses in memory) yet the string data at those two different addresses could be identical. Most programmers expect a string comparison for equality to be true if the data for the two strings is the same. Clearly a pointer comparison does not provide this type of comparison. To overcome this problem, the HLA Standard Library provides a set of string comparison routines that will compare the string data, not just their pointers. These routines use the following calling sequences:

```

str.eq( src1, src2 );
str.ne( src1, src2 );
str.lt( src1, src2 );
str.le( src1, src2 );
str.gt( src1, src2 );
str.ge( src1, src2 );
```

Each of these routines compares the *src1* string to the *src2* string and return true (1) or false (0) in the EAX register depending on the comparison. For example, "str.eq(s1, s2);" returns true in EAX if *s1* is equal to *s2*. HLA provides a small extension that allows you to use the string comparison routines within an IF statement⁷. The following code demonstrates the use of some of these comparison routines within an IF statement:

```

stdout.put( "Enter a single word: " );
stdin.a_gets();
if( str.eq( eax, "Hello" ) ) then

    stdout.put( "You entered 'Hello'", nl );

endif;
strfree( eax );
```

Note that the string the user enters in this example must exactly match "Hello", including the use of an upper case "H" at the beginning of the string. When processing user input, it is best to ignore alphabetic case in string comparisons because different users have different ideas about when they should be pressing the shift key on the keyboard. An easy solution is to use the HLA case insensitive string comparison functions. These routines compare two strings ignoring any differences in alphabetic case. These routines use the following calling sequences:

```

str.ieq( src1, src2 );
```

6. Technically, *str.a_substr*, like all routines that call *malloc* to allocate storage, can raise an *ex.MemoryAllocationFailure* exception, but this is very unlikely to occur.

7. This extension is actually a little more general than this section describes. A later chapter will explain it fully.

```

str.ine( src1, src2 );
str.ilt( src1, src2 );
str.ile( src1, src2 );
str.igt( src1, src2 );
str.ige( src1, src2 );

```

Other than they treat upper case characters the same as their lower case equivalents, these routines behave exactly like the former routines, returning true or false in EAX depending on the result of the comparison.

Like most high level languages, HLA compares strings using *lexicographical ordering*. This means that two strings are equal if and only if their lengths are the same and the corresponding characters in the two strings are exactly the same. For less than or greater than comparisons, lexicographical ordering corresponds to the way words appear in a dictionary. That is, “a” is less than “b” is less than “c” etc. Actually, HLA compares the strings using the ASCII numeric codes for the characters, so if you are unsure whether “a” is less than a period, simply consult the ASCII character chart (incidentally, “a” is greater than a period in the ASCII character set, just in case you were wondering).

If two strings have different lengths, lexicographical ordering only worries about the length if the two strings exactly match up through the length of the shorter string. If this is the case, then the longer string is greater than the shorter string (and, conversely, the shorter string is less than the longer string). Note, however, that if the characters in the two strings do not match at all, then HLA’s string comparison routines ignore the length of the string; e.g., “z” is always greater than “aaaaa” even though it has a shorter length.

The *str.eq* routine checks to see if two strings are equal. Sometimes, however, you might want to know whether one string *contains* another string. For example, you may want to know if some string contains the substring “north” or “south” to determine some action to take in a game. The HLA *str.index* routine lets you check to see if one string is contained as a substring of another. The *str.index* routine uses the following calling sequence:

```
str.index( StrToSearch, SubstrToSearchFor );
```

This function returns, in EAX, the offset into *StrToSearch* where *SubstrToSearchFor* appears. This routine returns -1 in EAX if *SubstrToSearchFor* is not present in *StrToSearch*. Note that *str.index* will do a case sensitive search. Therefore the strings must exactly match. There is no case insensitive variant of *str.index* you can use⁸.

The HLA strings module contains many additional routines besides those this section presents. Space limitations and prerequisite knowledge prevent the presentation of all the string functions here; however, this does not mean that the remaining string functions are unimportant. You should definitely take a look at the HLA Standard Library documentation to learn everything you can about the powerful HLA string library routines. The chapters on advanced string handling contain more information on HLA string and pattern matching routines.

2.7 In-Memory Conversions

The HLA Standard Library’s string module contains dozens of routines for converting between strings and other data formats. Although it’s a little premature in this text to present a complete description of those functions, it would be rather criminal not to discuss at least one of the available functions: the *str.put* routine. This one routine (which is actually a macro) encapsulates the capabilities of all the other string conversion functions, so if you learn how to use this one, you’ll have most of the capabilities of those other routines at your disposal. For more information on the other string conversions, see the chapters in the volume on Advanced String Handling.

8. However, HLA does provide routines that will convert all the characters in a string to one case or another. So you can make copies of the strings, convert all the characters in both copies to lower case, and then search using these converted strings. This will achieve the same result.

You use the *str.put* routine in a manner very similar to the *stdout.put* routine. The only difference is that the *str.put* routine “writes” its data to a string instead of the standard output device. A call to *str.put* has the following syntax:

```
str.put( destString, values_to_convert );
```

Example of a call to *str.put*:

```
str.put( destString, "I =", i:4, " J= ", j, " s=", s );
```

Note: generally you would not put a newline character sequence at the end of the string as you would if you were printing the string to the standard output device.

The *destString* parameter at the beginning of the *str.put* parameter list must be a string variable and it must already have storage associated with it. If *str.put* attempts to store more characters than allowed into the *destString* parameter, then this function raises the *ex.StringOverflow* exception.

Most of the time you won’t know the length of the string that *str.put* will produce. In those instances, you should simply allocate sufficient storage for a really large string, one that is way larger than you expect, and use this string data as the first parameter of the *str.put* call. This will prevent an exception from crashing your program. Generally, if you expect to produce about one screen line of text, you should probably allocate at least 256 characters for the destination string. If you’re creating longer strings, you should probably use a default of 1024 characters (or more, if you’re going to produce *really* large strings).

Example:

```
static
  s: string;
  .
  .
  .
  mov( stralloc( 256 ), s );
  .
  .
  .
  str.put( s, "R: ", r:16:4, " strval: `", strval:-10, "`" );
```

You can use the *str.put* routine to convert any data to a string that you can print using *stdout.put*. You will probably find this routine invaluable for common value-to-string conversions.

At the time this is being written, there is no corresponding *str.get* routine that will read values from an input string (this routine will probably appear in a future version of the HLA Standard Library, so watch out for it). In the meantime, the HLA strings and conversions modules in the Standard Library do provide lots of stand-alone conversion functions you can use to convert string data to some other format. See the volume on “Advanced String Handling” for more details about these routines.

2.8 Putting It All Together

There are many different ways to represent character strings. This chapter began by discussing how the C/C++ and Pascal languages represent strings using zero-terminated and length prefixed strings. HLA uses a hybrid representation for its string. HLA strings consist of a pointer to a zero terminated sequence of character with a pair of prefix length values. HLA’s format offers all the advantages of the other two forms with the slight disadvantage of a few extra bytes of overhead.

After discussing string formats, this chapter discussed how to operate on string data. In addition to accessing the characters in a string directly (which is easy, you just index off the pointer to the string data), this chapter described how to manipulate strings using several routines from the HLA Standard Library. This chapter provides a very basic introduction to string handling in HLA. To learn more about string manipulation in assembly language (and the use of the routines in the HLA Standard Library), see the separate volume on “Advanced String Handling” in this text.

Characters and Character Sets

Chapter Three

3.1 Chapter Overview

This chapter completes the discussion of the character data type by describing several character translation and classification functions found in the HLA Standard Library. These functions provide most of the character operations that aren't trivial to realize using a few 80x86 machine instructions.

This chapter also introduces another composite data type based on the character – the character set data type. Character sets and their associated operations, let you quickly test characters to see if they belong to some set. These operations also let you manipulate sets of characters using familiar operations like set union, intersection, and difference.

3.2 The HLA Standard Library CHARS.HHF Module

The HLA Standard Library `chars.hhf` module provides a couple of routines that convert characters from one form to another and several routines that classify characters according to their graphic representation. These functions are especially useful for processing user input to verify that it is correct.

The first two routines we will consider are the translation/conversion functions. These functions are *chars.toUpper* and *chars.toLower*. These functions use the following syntax:

```
chars.toLower( characterValue ); // Returns converted character in AL.
chars.toUpper( characterValue ); // Returns converted character in AL.
```

These two functions require a byte-sized parameter (typically a register or a *char* variable). They check the character to see if it is an alphabetic character; if it is not, then these functions return the unmodified parameter value in the AL register. If the character is an alphabetic character, then these functions may translate the value depending on the particular function. The *chars.toUpper* function translates lower case alphabetic characters to upper case; it returns upper case character unmodified. The *chars.toLower* function does the converse – it translates upper case characters to lower case characters and leaves lower case characters alone.

These two functions are especially useful when processing user input containing alphabetic characters. For example, suppose you expect a “Y” or “N” answer from the user at some point in your program. Your code might look like the following:

```
forever

    stdout.put( "Answer 'Y' or 'N':" );
    stdin.FlushInput(); // Force input of new line of text.
    stdin.getc(); // Read user input in AL.
    breakif( al = 'Y' );
    breakif( al = 'N' );
    stdout.put( "Illegal input, please reenter", nl );

endfor;
```

The problem with this program is that the user must answer exactly “Y” or “N” (using upper case) or the program will reject the user's input. This means that the program will reject “y” and “n” since the ASCII codes for these characters are different than “Y” and “N”.

One way to solve this problem is to include two additional `BREAKIF` statements in the code above that test for “y” and “n” as well as “Y” and “N”. The problem with this approach is that AL will still contain one of four different characters, complicating tests of AL once the program exits the loop. A better solution is to use either *chars.toUpper* or *chars.toLower* to translate all alphabetic characters to a single case. Then you

can test AL for a single pair of characters, both in the loop and outside the loop. The resulting code would look like the following:

```

forever

    stdout.put( "Answer 'Y' or 'N':" );
    stdin.FlushInput(); // Force input of new line of text.
    stdin.getc();      // Read user input in AL.
    chars.toUpper( al ); // Convert "y" and "n" to "Y" and "N".
    breakif( al = 'Y' );
    breakif( al = 'N' );
    stdout.put( "Illegal input, please reenter", nl );

endfor;
<< test for "Y" or "N" down here to determine user input >>

```

As you can see from this example, the case conversion functions can be quite useful when processing user input. As a final example, consider a program that presents a menu of options to the user and the user selects an option using an alphabetic character. Once again, you can use *chars.toUpper* or *chars.toLower* to map the input character to a single case so that it is easier to process the user's input:

```

stdout.put( "Enter selection (A-G):" );
stdin.FlushInput();
stdin.getc();
chars.toLower( al );
if( al = 'a' ) then

    << Handle Menu Option A >>

elseif( al = 'b' ) then

    << Handle Menu Option B >>

elseif( al = 'c' ) then

    << Handle Menu Option C >>

elseif( al = 'd' ) then

    << Handle Menu Option D >>

elseif( al = 'e' ) then

    << Handle Menu Option E >>

elseif( al = 'f' ) then

    << Handle Menu Option F >>

elseif( al = 'g' ) then

    << Handle Menu Option G >>

else

    stdout.put( "Illegal input!" nl );

endif;

```

The remaining functions in the `chars.hhf` module all return a boolean result depending on the type of the character you pass them as a parameter. These classification functions let you quickly and easily test a character to determine if its type is valid for the some intended use. These functions expect a single byte (`char`) parameter and they return true (1) or false (0) in the EAX register. These functions use the following calling syntax:

```
chars.isAlpha( c ); // Returns true if c is alphabetic
chars.isUpper( c ); // Returns true if c is upper case alphabetic.
chars.isLower( c ); // Returns true if c is lower case alphabetic.
chars.isAlphaNum( c ); // Returns true if c is alphabetic or numeric.
chars.isDigit( c ); // Returns true if c is a decimal digit.
chars.isXDigit( c ); // Returns true if c is a hexadecimal digit.
chars.isGraphic( c ); // See notes below.
chars.isSpace( c ); // Returns true if c is a whitespace character.
chars.isASCII( c ); // Returns true if c is in the range #00..#7f.
chars.isCtrl( c ); // Returns true if c is a control character.
```

Notes: Graphic characters are the printable characters whose ASCII codes fall in the range \$21..\$7E. Note that a space is not considered a graphic character (nor are the control characters). Whitespace characters are the space, the tab, the carriage return, and the linefeed. Control characters are those characters whose ASCII code is in the range \$00..\$1F and \$7F.

These classification functions are great for validating user input. For example, if you want to check to ensure that a user has entered nothing but numeric characters in a string you read from the standard input, you could use code like the following:

```
stdin.a_gets(); // Read line of text from the user.
mov( eax, ebx ); // save ptr to string in EBX.
mov( ebx, ecx ); // Another copy of string pointer to test each char.
while( (type char [ecx]) <> #0 ) do // Repeat while not at end of string.

    breakif( !chars.isDigit( (type char [ecx] ) ) );
    inc( ecx ); // Move on to the next character;

endwhile;
if( (type char [ecx] ) = #0 ) then

    << Valid string, process it >>

else

    << invalid string >>

endif;
```

Although the `chars.hhf` module's classification functions handle many common situations, you may find that you need to test a character to see if it belongs in a class that the `chars.hhf` module does not handle. Fear not, checking for such characters is very easy. The next section will explain how to do this.

3.3 Character Sets

Character sets are another composite data type, like strings, built upon the character data type. A character set is a mathematical set of characters with the most important attribute being membership. That is, a character is either a member of a set or it is not a member of a set. The concept of sequence (e.g., whether one character comes before another, as in a string) is completely foreign to a character set. If two characters are members of a set, their order in the set is irrelevant. Also, membership is a binary relation; a character is either in the set or it is not in the set; you cannot have multiple copies of the same character in a character set. Finally, there are various operations that are possible on character sets including the mathematical set operations of union, intersection, difference, and membership test.

HLA implements a restricted form of character sets that allows set members to be any of the 128 standard ASCII characters (i.e., HLA's character set facilities do not support extended character codes in the range #128..#255). Despite this restriction, however, HLA's character set facilities are very powerful and are very handy when writing programs that work with string data. The following sections describe the implementation and use of HLA's character set facilities so you may take advantage of character sets in your own programs.

3.4 Character Set Implementation in HLA

There are many different ways to represent character sets in an assembly language program. HLA implements character sets by using an array of 128 boolean values. Each boolean value determines whether the corresponding character is or is not a member of the character set; i.e., a true boolean value indicates that the specified character is a member of the set, a false value indicates that the corresponding character is not a member of the set. To conserve memory, HLA allocates only a single bit for each character in the set; therefore, HLA character sets consume 16 bytes of memory since there are 128 bits in 16 bytes. This array of 128 bits is organized in memory as shown in Figure 3.1.

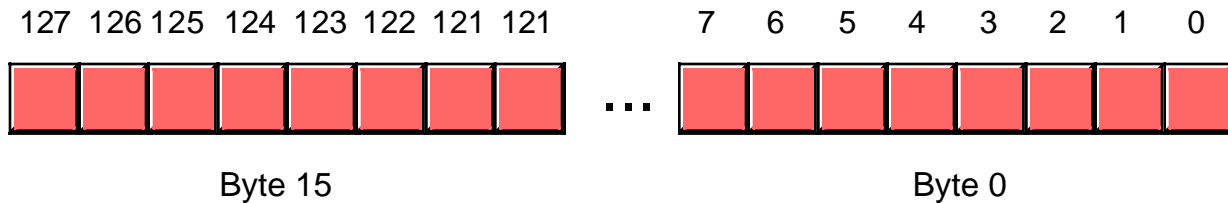


Figure 3.1 Bit Layout of a Character Set Object

Bit zero of byte zero corresponds to ASCII code zero (the NUL character). If this bit is one, then the character set contains the NUL character; if this bit contains false, then the character set does not contain the NUL character. Likewise, bit zero of byte one (the eighth bit in the 128-bit array) corresponds to the back-space character (ASCII code is eight). Bit one of byte eight corresponds to ASCII code 65, an upper case 'A'. Bit 65 will contain a one if 'A' is a current member of the character set, it will contain zero if 'A' is not a member of the set.

While there are other possible ways to implement character sets, this bit vector implementation has the advantage that it is very easy to implement set operations like union, intersection, difference comparison, and membership tests.

HLA supports character set variables using the *cset* data type. To declare a character set variable, you would use a declaration like the following:

```
static
  CharSetVar: cset;
```

This declaration will reserve 16 bytes of storage to hold the 128 bits needed to represent an ASCII character set.

Although it is possible to manipulate the bits in a character set using instructions like AND, OR, XOR, etc., the 80x86 instruction set includes several bit test, set, reset, and complement instructions that are nearly perfect for manipulating character sets. The BT (bit test) instruction, for example will copy a single bit in memory to the carry flag. The BT instruction allows the following syntactical forms:

```
bt( BitNumber, BitsToTest );

bt( reg16, reg16 );
```

```

bt( reg32, reg32 );
bt( constant, reg16 );
bt( constant, reg32 );

bt( reg16, mem16 );
bt( reg32, mem32 ); //HLA treats cset objects as dwords within bt.
bt( constant, mem16 );
bt( constant, mem32 ); //HLA treats cset objects as dwords within bt.

```

The first operand holds a bit number, the second operand specifies a register or memory location whose bit should be copied into the carry flag. If the second operand is a register, the first operand must contain a value in the range $0..n-1$, where n is the number of bits in the second operand. If the first operand is a constant and the second operand is a memory location, the constant must be in the range $0..255$. Here are some examples of these instructions:

```

bt( 7, ax );           // Copies bit #7 of AX into the carry flag (CF).
mov( 20, eax );
bt( eax, ebx );       // Copies bit #20 of EBX into CF.

// Copies bit #0 of the byte at CharSetVar+3 into CF.

bt( 24, CharSetVar );

// Copies bit #4 of the byte at DWmem+2 into CF.

bt( eax, CharSetVar );

```

The BT instruction turns out to be quite useful for testing set membership. For example, to see if the character 'A' is a member of a character set, you could use a code sequence like the following:

```

bt( 'A', CharSetVar );
if( @c ) then

    << Do something if 'A' is a member of the set >>

endif;

```

The BTS (bit test and set), BTR (bit test and reset), and BTC (bit test and complement) instructions are also quite useful for manipulating character set variables. Like the BT instruction, these instructions copy the specified bit into the carry flag; after copying the specified bit, these instructions will set, clear, or invert (respectively) the specified bit. Therefore, you can use the BTS instruction to add a character to a character set via set union (that is, it adds a character to the set if the character was not already a member of the set, otherwise the set is unaffected). You can use the BTR instruction to remove a character from a character set via set intersection (That is, it removes a character from the set if and only if it was previously in the set; otherwise it has no effect on the set). The BTC instruction lets you add a character to the set if it wasn't previously in the set, it removes the character from the set if it was previously a member (that is, it toggles the membership of that character in the set).

The HLA Standard Library provides lots of character set handling routines. See "Character Set Support in the HLA Standard Library" on page 445. for more details about HLA's character set facilities.

3.5 HLA Character Set Constants and Character Set Expressions

HLA supports literal character set constants. These *cset* constants make it easy to initialize *cset* variables at compile time and they make it very easy to pass character set constants as procedure parameters. An HLA character set constant takes the following form:

```
{ Comma_separated_list_of_characters_and_character_ranges }
```

The following is an example of a simple character set holding the numeric digit characters:

```
{ '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' }
```

When specifying a character set literal that has several contiguous values, HLA lets you concisely specify the values using only the starting and ending values of the range thusly:

```
{ '0'..'9' }
```

You may combine characters and various ranges within the same character set constant. For example, the following character set constant is all the alphanumeric characters:

```
{ '0'..'9', 'a'..'z', 'A'..'Z' }
```

You can use these *cset* literal constants in the CONST and VAL sections. The following example demonstrates how to create the symbolic constant *AlphaNumeric* using the character set above:

```
const
  AlphaNumeric: cset := { '0'..'9', 'a'..'z', 'A'..'Z' };
```

After the above declaration, you can use the identifier *AlphaNumeric* anywhere the character set literal is legal.

You can also use character set literals (and, of course, character set symbolic constants) as the initializer field for a *STATIC* or *READONLY* variable. The following code fragment demonstrates this:

```
static
  Alphabetic: cset := { 'a'..'z', 'A'..'Z' };
```

Anywhere you can use a character set literal constant, a character set constant expression is also legal. HLA supports the following operators in character set constant expressions:

<i>CSetConst</i> + <i>CSetConst</i>	Computes the union of the two sets ¹ .
<i>CSetConst</i> * <i>CSetConst</i>	Computes the intersection of the two sets ² .
<i>CSetConst</i> - <i>CSetConst</i>	Computes the set difference of the two sets ³ .
- <i>CSetConst</i>	Computes the set complement ⁴ .

Note that these operators only produce compile-time results. That is, the expressions above are computed by the compiler during compilation, they do not emit any machine code. If you want to perform these operations on two different sets while your program is running, the HLA Standard Library provides routines you can call to achieve the results you desire. HLA also provides other compile-time character set operators. See the chapter on the compile-time language and macros for more details.

3.6 The IN Operator in HLA HLL Boolean Expressions

The HLA *IN* operator can dramatically reduce the logic in your HLA programs. This text has waited until now to discuss this operator because certain forms require a knowledge of character sets and character set constants. Now that you've seen character set constants, there is no need to delay the introduction of this important language feature.

In addition to the standard boolean expressions in *IF*, *WHILE*, *REPEAT..UNTIL*, and other statements, HLA also supports boolean expressions that take the following forms:

reg_s in *CSetConstant*

-
1. The set union is the set of all characters that are in either set.
 2. The set intersection is the set of all characters that appear in both operand sets.
 3. The set difference is the set of characters that appear in the first set but do not appear in the second set.
 4. The set complement is the set of all characters not in the set.

```

reg8 not in CSetConstant
reg8 in CSetVariable
reg8 not in CSetVariable

```

These four forms of the IN and NOT IN operators check to see if a character in an eight-bit register is a member of a character set (either a character set constant or a character set variable). The following code fragment demonstrates these operators:

```

const
  Alphabetic: cset := {'a'..'z', 'A'..'Z'};
  .
  .
  .
  stdin.getc();
  if( al in Alphabetic ) then

      stdout.put( "You entered an alphabetic character" nl );

  elseif( al in {'0'..'9'} ) then

      stdout.put( "You entered a numeric character" nl );

  endif;

```

3.7 Character Set Support in the HLA Standard Library

As noted in the previous sections, the HLA Standard Library provides several routines that provide character set support. The character set support routines fall into four categories: standard character set functions, character set tests, character set conversions, and character set I/O. This section describes these routines in the HLA Standard Library.

To begin with, let's consider the Standard Library routines that help you construct character sets. These routines include: *cs.empty*, *cs.cpy*, *cs.charToCset*, *cs.unionChar*, *cs.removeChar*, *cs.rangeChar*, *cs.strToCset*, and *cs.unionStr*. These procedures let you build up character sets at run-time using character and string objects.

The *cs.empty* procedure initializes a character set variable to the empty set by setting all the bits in the character set to zero. This procedure call uses the following syntax (*CSvar* is a character set variable):

```
cs.empty( CSvar );
```

The *cs.cpy* procedure copies one character set to another, replacing any data previously held by the destination character set. The syntax for *cs.cpy* is

```
cs.cpy( srcCsetValue, destCsetVar );
```

The *cs.cpy* source character set can be either a character set constant or a character set variable. The destination character set must be a character set variable.

The *cs.unionChar* procedure adds a character to a character set. It uses the following calling sequence:

```
cs.unionChar( CharVar, CSvar );
```

This call will add the first parameter, a character, to the set via set union. Note that you could use the *BTS* instruction to achieve this same result although the *cs.unionChar* call is often more convenient (though slower).

The *cs.charToCset* function creates a singleton set (a set containing a single character). The calling format for this function is

```
cs.charToCset( CharValue, CSvar );
```

The first operand, the character value *CharValue*, can be an eight-bit register, a constant, or a character variable. The second operand (*CSvar*) must be a character set variable. This function clears the destination character set to all zeros and then adds the specified character to the character set.

The *cs.removeChar* procedure lets you remove a single character from a character set without affecting the other characters in the set. This function uses the same syntax as *cs.charToCset* and the parameters have the same attributes. The calling sequence is

```
cs.removeChar( CharValue, CSVar );
```

The *cs.rangeChar* constructs a character set containing all the characters between two characters you pass as parameters. This function sets all bits outside the range of these two characters to zero. The calling sequence is

```
cs.rangeChar( LowerBoundChar, UpperBoundChar, CSVar );
```

The *LowerBoundChar* and *UpperBoundChar* parameters can be constants, registers, or character variables. *CSVar*, the destination character set, must be a *cset* variable.

The *cs.strToCset* procedure creates a new character set containing the union of all the characters in a character string. This procedure begins by setting the destination character set to the empty set and then unions in the characters in the string one by one until it exhausts all characters in the string. The calling sequence is

```
cs.strToCset( StringValue, CSVar );
```

Technically, the *StringValue* parameter can be a string constant as well as a string variable, however, it doesn't make any sense to call *cs.strToCset* in this fashion since *cs.cpy* is a much more efficient way to initialize a character set with a constant set of characters. As usual, the destination character set must be a *cset* variable. Typically, you'd use this function to create a character set based on a string input by the user.

The *cs.unionStr* procedure will add the characters in a string to an existing character set. Like *cs.strToCset*, you'd normally use this function to union characters into a set based on a string input by the user. The calling sequence for this is

```
cs.unionStr( StringValue, CSVar );
```

Standard set operations include union, intersection, and set difference. The HLA Standard Library routines *cs.setunion*, *cs.intersection*, and *cs.difference* provide these operations, respectively⁵. These routines all use the same calling sequence:

```
cs.setunion( srcCset, destCset );
cs.intersection( srcCset, destCset );
cs.difference( srcCset, destCset );
```

The first parameter can be a character set constant or a character set variable. The second parameter must be a character set variable. These procedures compute “*destCset := destCset op srcCset*” where *op* represents set union, intersection, or difference, depending on the function call.

The third category of character set routines test character sets in various ways. They typically return a boolean value indicating the result of the test. The HLA character set routines in this category include *cs.IsEmpty*, *cs.member*, *cs.subset*, *cs.psubset*, *cs.superset*, *cs.psuperset*, *cs.eq*, and *cs.ne*.

The *cs.IsEmpty* function tests a character set to see if it is the empty set. The function returns true or false in the EAX register. This function uses the following calling sequence:

```
cs.IsEmpty( CSetValue );
```

5. “*cs.setunion*” was used rather than “*cs.union*” because “union” is an HLA reserved word.

The single parameter may be a constant or a character set variable, although it doesn't make much sense to pass a character set constant to this procedure (since you would know at compile-time whether this set is empty or not empty).

The *cs.member* function tests to see if a character value is a member of a set. This function returns true in the EAX register if the supplied character is a member of the specified set. Note that you can use the BT instruction to (more efficiently) test this same condition. However, the *cs.member* function is probably a little more convenient to use. The calling sequence for *cs.member* is

```
cs.member( CharValue, CsetValue );
```

The first parameter is a register, character variable, or a constant. The second parameter is either a character set constant or a character set variable. It would be unusual for both parameters to be constants.

The *cs.subset*, *cs.psubset* (proper subset), *cs.superset*, and *cs.psuperset* (proper superset) functions let you check to see if one character set is a subset or superset of another. The calling sequence for these four routines is nearly identical, it is one of the following:

```
cs.subset( CsetValue1, CsetValue2 );
cs.psubset( CsetValue1, CsetValue2 );
cs.superset( CsetValue1, CsetValue2 );
cs.psuperset( CsetValue1, CsetValue2 );
```

These routines compare the first parameter against the second parameter and return true or false in the EAX register depending upon the result of the comparison. One set is a subset of another if all the members of the first character set can be found in the second character set. It is a proper subset if the second character set also contains characters not found in the first (left) character set. Likewise, one character set is a superset of another if it contains all the characters in the second (right) set (and, possibly, more). A proper superset contains additional characters above and beyond those found in the second set. The parameters can be either character set variables or character set constants; however, it would be unusual for both parameters to be character set constants (since you can determine this at compile time, there would be no need to call a run-time function to compute this).

The *cs.eq* and *cs.ne* check to see if two sets are equal or not equal. These functions return true or false in EAX depending upon the set comparison. The calling sequence is identical to the sub/superset functions above:

```
cs.eq( CsetValue1, CsetValue2 );
cs.ne( CsetValue1, CsetValue2 );
```

The *cs.extract* routine removes an arbitrary character from a character set and returns that character in the EAX register⁶. The calling sequence is the following:

```
cs.extract( CsetVar );
```

The single parameter must be a character set variable. Note that this function will modify the character set variable by removing some character from the character set. This function returns \$FFFF_FFFF (-1) in EAX if the character set was empty prior to the call.

In addition to the routines found in the *cs* (character set) library module, the string and standard output modules also provide functions that allow or expect character set parameters. For example, if you supply a character set value as a parameter to *stdout.put*, the *stdout.put* routine will print the characters currently in the set. See the HLA Standard Library documentation for more details on character set handling procedures.

3.8 Using Character Sets in Your HLA Programs

Character sets are valuable for many different applications in your programs. For example, in the volume on Advanced String Handling you'll discover how to use character sets to match complex patterns.

6. This routine returns the character in AL and zeros out the H.O. three bytes of EAX.

However, such use of character sets is a little beyond the scope of this chapter, so at this point we'll concentrate on another common use of character sets: validating user input. This section will also present a couple of other applications for character sets to help you start thinking about how you could use them in your program.

Consider the following short code segment that gets a yes/no type answer from the user:

```
static
  answer: char;
  .
  .
  .
  repeat
    .
    .
    .
    stdout.put( "Would you like to play again? " );
    stdin.FlushInput();
    stdin.get( answer );

  until( answer = 'n' );
```

A major problem with this code sequence is that it will only stop if the user presses a lower case 'n' character. If they type anything other than 'n' (including upper case 'N') the program will treat this as an affirmative answer and transfer back to the beginning of the repeat..until loop. A better solution would be to validate the user input before the UNTIL clause above to ensure that the user has only typed "n", "N", "y", or "Y". The following code sequence will accomplish this:

```
repeat
  .
  .
  .
  repeat

    stdout.put( "Would you like to play again? " );
    stdin.FlushInput();
    stdin.get( answer );

    until( cs.member( answer, { 'n', 'N', 'Y', 'y' } ) );
    if( answer = 'N' ) then

      mov( 'n', answer );

    endif;

  until( answer = 'n' );
```

While an excellent use for character sets is to validate user input, especially when you must restrict the user to a small set of non-contiguous input characters, you should not use the *cs.member* function to test to see if a character value is within literal set. For example, you should never do something like the following:

```
repeat

  stdout.put( "Enter a character between 0..9: " );
  stdin.getc();

  until( cs.member( a1, {'0'..'9' } ) );
```

While there is nothing logically wrong with this code, keep in mind that HLA run-time boolean expressions allow simple membership tests using the IN operator. You could write the code above far more efficiently using the following sequence:

```

repeat

    stdout.put( "Enter a character between 0..9: " );
    stdin.getc();

until( al in '0'..'9' );

```

The place where the *cs.member* function becomes useful is when you need to see if an input character is within a set of characters that you build at run time.

3.9 Low-level Implementation of Set Operations

Although the HLA Standard Library character set module simplifies the use of character sets within your assembly language programs, it is instructive to look at how all these functions operate so you know the cost associated with each function. Also, since many of these functions are quite trivial, you might want to implement them in-line for performance reasons. The following subsections describe how each of the functions operate.

3.9.1 Character Set Functions That Build Sets

The first group of functions we will look at in the Character Set module are those that construct or copy character sets. These functions are *cs.empty*, *cs.cpy*, *cs.charToCset*, *cs.unionChar*, *cs.removeChar*, *cs.rangeChar*, *cs.strToCset*, and *cs.unionStr*.

Creating an empty set is, perhaps, the easiest of all the operations. To create an empty set all we need to is zero out all 128 bits in the *cset* object. Program 3.1 provides the implementation of this function.

```

// Program that demonstrates the implmentation of
// the cs.empty function.

program csEmpty;
#include( "stdlib.hhf" )

static
    csetDest: cset;
    csetSrc: cset := {'a'..'z', 'A'..'Z'};

begin csEmpty;

    // How to create an empty set (cs.empty):
    // (Zero out all bits in the cset)

    mov( 0, eax );
    mov( eax, (type dword csetDest) );
    mov( eax, (type dword csetDest[4]) );
    mov( eax, (type dword csetDest[8]) );
    mov( eax, (type dword csetDest[12]) );

    stdout.put( "Empty set = {", csetDest, "}" nl );

end csEmpty;

```

Program 3.1 cs.empty Implementation

Note that *cset* objects are 16 bytes long. Therefore, this code zeros out those 16 bytes by storing EAX into the four consecutive double words that comprise the object. Note the use of type coercion in the MOV statements; this is necessary since *cset* objects are not the same size as *dword* objects.

To copy one character set to another is only a little more difficult than creating an empty set. All we have to do is copy the 16 bytes from the source character set to the destination character set. We can accomplish this with four pairs of double word MOV statements. Program 3.2 provides the sample implementation.

```
// Program that demonstrates the implementation of
// the cs.empty function.

program csCpy;
#include( "stdlib.hhf" )

static
    csetDest: cset;
    csetSrc: cset := {'a'..'z', 'A'..'Z'};

begin csCpy;

    // How to create an empty set (cs.empty):
    // (Zero out all bits in the cset)

    mov( (type dword csetSrc), eax );
    mov( eax, (type dword csetDest) );

    mov( (type dword csetSrc[4]), eax );
    mov( eax, (type dword csetDest[4]) );

    mov( (type dword csetSrc[8]), eax );
    mov( eax, (type dword csetDest[8]) );

    mov( (type dword csetSrc[12]), eax );
    mov( eax, (type dword csetDest[12]) );

    stdout.put( "Copied set = {" , csetDest, "}" nl );

end csCpy;
```

Program 3.2 cs.cpy Implementation

The *cs.charToCset* function creates a singleton set containing the specified character. To implement this function we first begin by creating an empty set (using the same code as *cs.empty*) and then we set the bit corresponding to the single character in the character set. We can use the BTS (bit test and set) instruction to easily set the specified bit in the *cset* object. Program 3.3 provides the implementation of this function.

```
// Program that demonstrates the implementation of
// the cs.charToCset function.

program cscharToCset;
#include( "stdlib.hhf" )

static
```

```

csetDest: cset;
chrValue: char := 'a';

begin cscharToCset;

    // Begin by creating an empty set:

    mov( 0, eax );
    mov( eax, (type dword csetDest) );
    mov( eax, (type dword csetDest[4]) );
    mov( eax, (type dword csetDest[8]) );
    mov( eax, (type dword csetDest[12]) );

    // Okay, use the BTS instruction to set the specified bit in
    // the character set.

    movzx( chrValue, eax );
    bts( eax, csetDest );

    stdout.put( "Singleton set = {" , csetDest, "}" nl );

end cscharToCset;

```

Program 3.3 cs.charToCset Implementation

If you study this code carefully, you will note an interesting fact: the BTS instruction's operands are not the same size (*dword* and *cset*). Since programmers often use the BTx instructions to manipulate items in a character set, HLA allows you to specify a *cset* object as the destination operand of a BTx(reg₃₂, mem) instruction. Technically, the memory operand should be a double word object; HLA automatically coerces *cset* objects to *dword* for these instructions. Note that BTS requires a 16 or 32-bit register. Therefore, this code zero extends the character's value into EAX prior to executing the BTS instruction. Note that the value in EAX must not exceed 127 or this code will manipulate data beyond the end of the character set in memory. The use of a BOUND instruction might be warranted here if you can't ensure that the *chrValue* variable contains a value in the range 0..127.

The *cs.unionChar* adds a single character to the character set (if that character was not already present in the character set). This code is actually a bit simpler than the *cs.charToCset* function; the only difference is that the code does not clear the set to begin with – it simply sets the bit corresponding to the given character. Program 3.4 provides the implementation.

```

// Program that demonstrates the implementation of
// the cs.unionChar function.

program csUnionChar;
#include( "stdlib.hhf" )

static
    csetDest: cset := {'0'..'9'};
    chrValue: char := 'a';

begin csUnionChar;

    // Okay, use the BTS instruction to add the specified bit to
    // the character set.

```

```

movzx( chrValue, eax );
bts( eax, csetDest );

stdout.put( "New set = {", csetDest, "}" nl );

end csUnionChar;

```

Program 3.4 cs.unionChar Implementation

Once again, note that this code assumes that the character value is in the range 0..127. If it is possible for the character to fall outside this range, you should check the value before attempting to union the character into the character set. You can use an IF statement or the BOUND instruction for this check.

The *cs.removeChar* function removes a character from a character set, provided that character was a member of the set. If the character was not originally in the character set, then *cs.removeChar* does not affect the original character set. To accomplish this, the code must clear the bit associated with the character to remove from the set. Program 3.5 uses the BTR (bit test and reset) instruction to achieve this.

```

// Program that demonstrates the implmentation of
// the cs.removeChar function.

program csRemoveChar;
#include( "stdlib.hhf" )

static
  csetDest: cset := {'0'..'9'};
  chrVal1: char := '0';
  chrVal2: char := 'a';

begin csRemoveChar;

  // Okay, use the BTC instruction to remove the specified bit from
  // the character set.

  movzx( chrVal1, eax );
  btr( eax, csetDest );

  stdout.put( "Set w/o '0' = {", csetDest, "}" nl );

  // Now remove a character not in the set to demonstrate
  // that removal of a non-existant character doesn't affect
  // the set:

  movzx( chrVal2, eax );
  btr( eax, csetDest );
  stdout.put( "Final set = {", csetDest, "}" nl );

end csRemoveChar;

```

Program 3.5 cs.removeChar Implementation

Don't forget to use a BOUND instruction or an IF statement in Program 3.5 if it is possible for the character's value to fall outside the range 0..127. This will prevent the code from manipulating memory beyond the end of the character set.

The *cs.rangeChar* function creates a set containing all the characters between two specified boundaries. This function begins by creating an empty set; then it loops over the range of character to insert, inserting each character into the set as appropriate. Program 3.6 provides an example implementation of this function.

```
// Program that demonstrates the implementation of
// the cs.rangeChar function.

program csRangeChar;
#include( "stdlib.hhf" )

static
    csetDest: cset;
    startRange: char := 'a';
    endRange: char := 'z';

begin csRangeChar;

    // Begin by creating the empty set:

    mov( 0, eax );
    mov( eax, (type dword csetDest) );
    mov( eax, (type dword csetDest[4]) );
    mov( eax, (type dword csetDest[8]) );
    mov( eax, (type dword csetDest[12]) );

    // Run the following loop for each character between
    // 'startRange' and 'endRange' and set the corresponding
    // bit in the cset for each character in the range.

    movzx( startRange, eax );
    while( al <= endRange ) do

        bts( eax, csetDest );
        inc( al );

    endwhile;
    stdout.put( "Final set = {", csetDest, "}" nl );

end csRangeChar;
```

Program 3.6 cs.rangeChar Implementation

One interesting thing to note about the code in Program 3.6 is how it takes advantage of the fact that AL contains the actual character index even though it has to use EAX with the BTS instruction. As usual, you should check the range of the two values if there is any possibility that they could be outside the range 0..127.

One problem with this particular implementation of the *cs.rangeChar* function is that it is not particularly efficient if you ask it to create a set with a lot of characters in it. As you can see by studying the code, the execution time of this function is proportional to the number of characters in the range. In particular, the loop in this function iterates once for each character in the range. So if the range is large the loop executes

many more times than if the range is small. There is a more efficient solution to this problem using table lookups whose execution time is independent of the size of the character set it creates. For more details on using table lookups, see “Calculation Via Table Lookups” on page 647.

The *cs.strToCset* function scans through an HLA string character by character and creates a new character set by adding each character in the string to an empty character set.

```

// Program that demonstrates the implementation of
// the cs.strToCset function.

program csStrToCset;
#include( "stdlib.hhf" )

static
  StrToAdd: string := "Hello_World";
  csetDest: cset;

begin csStrToCset;

  // Begin by creating the empty set:

  mov( 0, eax );
  mov( eax, (type dword csetDest) );
  mov( eax, (type dword csetDest[4]) );
  mov( eax, (type dword csetDest[8]) );
  mov( eax, (type dword csetDest[12]) );

  // For each character in the source string, add that character
  // to the set.

  mov( StrToAdd, eax );
  while( (type char [eax]) <> #0 ) do // While not at end of string.

    movzx( (type char [eax]), ebx );
    bts( ebx, csetDest );
    inc( eax );

  endwhile;
  stdout.put( "Final set = {", csetDest, "}" nl );

end csStrToCset;

```

Program 3.7 cs.strToCset Implementation

This code begins by fetching the pointer to the first character in the string. The loop repeats for each character in the string up to the zero terminating byte of the string. For each character, this code uses the BTS instruction to set the corresponding bit in the destination character set. As usual, don't forget to use an IF statement or BOUND instruction if it is possible for the characters in the string to have values outside the range 0..127.

The *cs.unionStr* function is very similar to the *cs.strToCset* function; in fact, the only difference is that it doesn't create an empty character set prior to adding the characters in a string to the destination character set.

```

// Program that demonstrates the implementation of
// the cs.unionStr function.

program csUnionStr;
#include( "stdlib.hhf" )

static
  StrToAdd: string := "Hello_World";
  csetDest: cset := {'0'..'9'};

begin csUnionStr;

  // For each character in the source string, add that character
  // to the set.

  mov( StrToAdd, eax );
  while( (type char [eax]) <> #0 ) do // While not at end of string.

    movzx( (type char [eax]), ebx );
    bts( ebx, csetDest );
    inc( eax );

  endwhile;
  stdout.put( "Final set = {", csetDest, "}" nl );

end csUnionStr;

```

Program 3.8 cs.unionStr Implementation

3.9.2 Traditional Set Operations

The previous section describes how to construct character sets from characters and strings. In this section we'll take a look at how you can manipulate character sets using the traditional set operations of intersection, union, and difference.

The union of two sets A and B is the collection of all items that are in set A, set B, or both. In the bit array representation of a set, this means that a bit in the destination character set will be one if either or both of the corresponding bits in sets A or B are set. This of course, corresponds to the logical OR operation. Therefore, we can easily create the set union of two sets by logically ORing their bytes together. Program 3.9 provides the complete implementation of this function.

```

// Program that demonstrates the implementation of
// the cs.setunion function.

program cssetUnion;
#include( "stdlib.hhf" )

static
  csetSrc1: cset := {'a'..'z'};
  csetSrc2: cset := {'A'..'Z'};
  csetDest: cset;

begin cssetUnion;

```

```

// To compute the union of csetSrc1 and csetSrc2 all we have
// to do is logically OR the two sets together.

mov( (type dword csetSrc1), eax );
or( (type dword csetSrc2), eax );
mov( eax, (type dword csetDest));

mov( (type dword csetSrc1[4]), eax );
or( (type dword csetSrc2[4]), eax );
mov( eax, (type dword csetDest[4]));

mov( (type dword csetSrc1[8]), eax );
or( (type dword csetSrc2[8]), eax );
mov( eax, (type dword csetDest[8]));

mov( (type dword csetSrc1[12]), eax );
or( (type dword csetSrc2[12]), eax );
mov( eax, (type dword csetDest[12]));

stdout.put( "Final set = {", csetDest, "}" nl );

end cssetUnion;

```

Program 3.9 cs.setunion Implementation

The intersection of two sets is those elements that are members of both sets. In the bit array representation of character sets that HLA uses, this means that a bit is set in the destination character set if the corresponding bit is set in both the source sets; this corresponds to the logical AND operation; therefore, to compute the set intersection of two character sets, all you need do is logically AND the 16 bytes of the two source sets together. Program 3.10 provides a sample implementation.

```

// Program that demonstrates the implementation of
// the cs.intersection function.

program csIntersection;
#include( "stdlib.hhf" )

static
  csetSrc1: cset := {'a'..'z'};
  csetSrc2: cset := {'A'..'z'};
  csetDest: cset;

begin csIntersection;

  // To compute the intersection of csetSrc1 and csetSrc2 all we have
  // to do is logically AND the two sets together.

  mov( (type dword csetSrc1), eax );
  and( (type dword csetSrc2), eax );
  mov( eax, (type dword csetDest));

  mov( (type dword csetSrc1[4]), eax );
  and( (type dword csetSrc2[4]), eax );
  mov( eax, (type dword csetDest[4]));

```

```

mov( (type dword csetSrc1[8]), eax );
and( (type dword csetSrc2[8]), eax );
mov( eax, (type dword csetDest[8]));

mov( (type dword csetSrc1[12]), eax );
and( (type dword csetSrc2[12]), eax );
mov( eax, (type dword csetDest[12]));

stdout.put( " Set A = {" , csetSrc1, "}" nl );
stdout.put( " Set B = {" , csetSrc2, "}" nl );
stdout.put( "Intersection of A and B = {" , csetDest, "}" nl );

end csIntersection;

```

Program 3.10 cs.intersection Implementation

The difference of two sets is all the elements in the first set that are not also present in the second set. To compute this result we must logically AND the values from the first set with the inverted values of the second set; i.e., to compute $C := A - B$ we use the following expression:

$$C := A \text{ and } (\text{not } B);$$

Program 3.11 provides the code to implement this operation.

```

// Program that demonstrates the implementation of
// the cs.difference function.

program csDifference;
#include( "stdlib.hhf" )

static
  csetSrc1: cset := {'0'..'9', 'a'..'z'};
  csetSrc2: cset := {'A'..'z'};
  csetDest: cset;

begin csDifference;

  // To compute the difference of csetSrc1 and csetSrc2 all we have
  // to do is logically AND A and NOT B together.

  mov( (type dword csetSrc2), eax );
  not( eax );
  and( (type dword csetSrc1), eax );
  mov( eax, (type dword csetDest));

  mov( (type dword csetSrc2[4]), eax );
  not( eax );
  and( (type dword csetSrc1[4]), eax );
  mov( eax, (type dword csetDest[4]));

  mov( (type dword csetSrc2[8]), eax );
  not( eax );
  and( (type dword csetSrc1[8]), eax );
  mov( eax, (type dword csetDest[8]));

  mov( (type dword csetSrc2[12]), eax );

```

```

not( eax );
and( (type dword csetSrc1[12]), eax );
mov( eax, (type dword csetDest[12]));

stdout.put( " Set A = {" , csetSrc1, "}" nl );
stdout.put( " Set B = {" , csetSrc2, "}" nl );
stdout.put( "Difference of A and B = {" , csetDest, "}" nl );

end csDifference;

```

Program 3.11 cs.difference Implementation

3.9.3 Testing Character Sets

In addition to manipulating the members of a character set, the need often arises to compare character sets, check to see if a character is a member of a set, and check to see if a set is empty. In this section we'll discuss how HLA implements the relational operations on character sets.

Occasionally you'll want to check a character set to see if it contains any members. Although you could achieve this by creating a static *cset* variable with no elements and comparing the set in question against this empty set, there is a more efficient way to do this – just check to see if all the bits in the set in question are zero. An easy way to do this, that uses, is to logically OR the four double words in a *cset* object together. If the result is zero, then all the bits in the *cset* variable are zero and, hence, the character set is empty.

```

// Program that demonstrates the implmentation of
// the cs.IsEmpty function.

program csIsEmpty;
#include( "stdlib.hhf" )

static
  csetSrc1: cset := {};
  csetSrc2: cset := {'A'..'Z'};

begin csIsEmpty;

  // To see if a set is empty, simply OR all the dwords
  // together and see if the result is zero:

  mov( (type dword csetSrc1[0]), eax );
  or( (type dword csetSrc1[4]), eax );
  or( (type dword csetSrc1[8]), eax );
  or( (type dword csetSrc1[12]), eax );

  if( @z ) then

    stdout.put( "csetSrc1 is empty ({" , csetSrc1, "}" nl );

  else

    stdout.put( "csetSrc1 is not empty ({" , csetSrc1, "}" nl );

  endif;

  // Repeat the test for csetSrc2:

```

```

mov( (type dword csetSrc2[0]), eax );
or( (type dword csetSrc2[4]), eax );
or( (type dword csetSrc2[8]), eax );
or( (type dword csetSrc2[12]), eax );

if( @z ) then

    stdout.put( "csetSrc2 is empty ({", csetSrc2, "}" nl );

else

    stdout.put( "csetSrc2 is not empty ({", csetSrc2, "}" nl );

endif;

end csIsEmpty;

```

Program 3.12 Implementation of cs.IsEmpty

Perhaps the most common check on a character set is set membership; that is, checking to see if some character is a member of a given character set. As you've seen already (see "Character Set Implementation in HLA" on page 442), the BT instruction is perfect for this. Since we've already discussed how to use the BT instruction (along with, perhaps, a MOVZX instruction), there is no need to repeat the implementation of this operation here.

Two sets are equal if and only if all the bits are equal in the two set objects. Therefore, we can implement the cs.ne and cs.eq (set inequality and set equality) functions by comparing the four double words in a *cset* object and noting if there are any differences. Program 3.13 demonstrates how you can do this.

```

// Program that demonstrates the implementation of
// the cs.eq and cs.ne functions.

program cseqne;
#include( "stdlib.hhf" )

static
    csetSrc1: cset := {'a'..'z'};
    csetSrc2: cset := {'a'..'z'};
    csetSrc3: cset := {'A'..'Z'};

begin cseqne;

    // To see if a set equal to another, check to make sure
    // all four dwords are equal. One sneaky way to do this
    // is to use the XOR operator (XOR is "not equals" as you
    // may recall).

    mov( (type dword csetSrc1[0]), eax ); // Set EAX to zero if these
    xor( (type dword csetSrc2[0]), eax ); // two dwords are equal
    mov( eax, ebx ); // Accumulate result here.

    mov( (type dword csetSrc1[4]), eax );
    xor( (type dword csetSrc2[4]), eax );
    or( eax, ebx );

```

```

mov( (type dword csetSrc1[8]), eax );
xor( (type dword csetSrc2[8]), eax );
or( eax, ebx );

mov( (type dword csetSrc1[12]), eax );
xor( (type dword csetSrc2[12]), eax );
or( eax, ebx );

// At this point, EBX is zero if the two csets are equal
// (also, the zero flag is set if they are equal).

if( @z ) then

    stdout.put( "csetSrc1 is equal to csetSrc2" nl );

else

    stdout.put( "csetSrc1 is not equal to csetSrc2" nl );

endif;

// Implementation of cs.ne:

mov( (type dword csetSrc1[0]), eax ); // Set EAX to zero if these
xor( (type dword csetSrc3[0]), eax ); // two dwords are equal
mov( eax, ebx ); // Accumulate result here.

mov( (type dword csetSrc1[4]), eax );
xor( (type dword csetSrc3[4]), eax );
or( eax, ebx );

mov( (type dword csetSrc1[8]), eax );
xor( (type dword csetSrc3[8]), eax );
or( eax, ebx );

mov( (type dword csetSrc1[12]), eax );
xor( (type dword csetSrc3[12]), eax );
or( eax, ebx );

// At this point, EBX is non-zero if the two csets are not equal
// (also, the zero flag is clear if they are not equal).

if( @nz ) then

    stdout.put( "csetSrc1 is not equal to csetSrc3" nl );

else

    stdout.put( "csetSrc1 is equal to csetSrc3" nl );

endif;

end cseqne;

```

Program 3.13 Implementation of cs.ne and cs.eq

The remaining tests on character sets roughly correspond to tests for less than or greater than; though in set theory we refer to these as superset and subset. One set is a subset of another if the second set contains all the elements of the first set; the second set may contain additional elements. The subset relationship is roughly equivalent to “less than or equal.” The proper subset relation of two sets states that the elements of one set are all present in a second set and the two sets are not equal (i.e., the second set contains additional elements). This is roughly equivalent to the “less than” relationship.

Testing for a subset is an easy task. All you have to do is take the set intersection of the two sets and verify that the intersection of the two is equal to the first set. That is, $A \leq B$ if and only if:

$$A == (A * B) \quad "*" \text{ denotes set intersection}$$

Testing for a proper subset is a little more work. The same relationship above must hold but the resulting inspection must not be equal to B. That is, $A < B$ if and only if,

$$(A == (A * B)) \text{ and } (B <> (A * B))$$

The algorithms for superset and proper superset are nearly identical. They are:

$$\begin{aligned} B == (A * B) & \qquad \qquad \qquad A \geq B \\ (B == (A * B)) \text{ and } (A <> (A * B)) & \qquad A > B \end{aligned}$$

The implementation of these four relational operations is left as an exercise.

3.10 Putting It All Together

This chapter describes HLA’s implementation of character sets. Character sets are a very useful tool for validating user input and for other character scanning and manipulation operations. HLA uses a bit array implementation for character set objects. HLA’s implementation allows for 128 different character values in a character set.

The HLA Standard Library provides a wide set of functions that let you build, manipulate, and compare character sets. Although these functions are convenient to use, most of the character set operations are so simple that you can implement them directly using in-line code. This chapter provided the implementation of many of the HLA Standard Library character set functions.

Note that this chapter does not cover all the uses of character sets in an assembly language program. In the volume on “Advanced String Handling” you will see many more uses for character sets in your programs.

Arrays

Chapter Four

4.1 Chapter Overview

This chapter discusses how to declare and use arrays in your assembly language programs. This is probably the most important chapter on composite data structures in this text. Even if you elect to skip the chapters on Strings, Character Sets, Records, and Dates and Times, be sure you read and understand the material in this chapter. Much of the rest of the text depends on your understanding of this material.

4.2 Arrays

Along with strings, arrays are probably the most commonly used composite data type. Yet most beginning programmers have a very weak understanding of how arrays operate and their associated efficiency trade-offs. It's surprising how many novice (and even advanced!) programmers view arrays from a completely different perspective once they learn how to deal with arrays at the machine level.

Abstractly, an array is an aggregate data type whose members (elements) are all the same type. Selection of a member from the array is by an integer index¹. Different indices select unique elements of the array. This text assumes that the integer indices are contiguous (though this is by no means required). That is, if the number x is a valid index into the array and y is also a valid index, with $x < y$, then all i such that $x < i < y$ are valid indices into the array.

Whenever you apply the indexing operator to an array, the result is the specific array element chosen by that index. For example, $A[i]$ chooses the i^{th} element from array A . Note that there is no formal requirement that element i be anywhere near element $i+1$ in memory. As long as $A[i]$ always refers to the same memory location and $A[i+1]$ always refers to its corresponding location (and the two are different), the definition of an array is satisfied.

In this text, we will assume that array elements occupy contiguous locations in memory. An array with five elements will appear in memory as shown in Figure 4.1

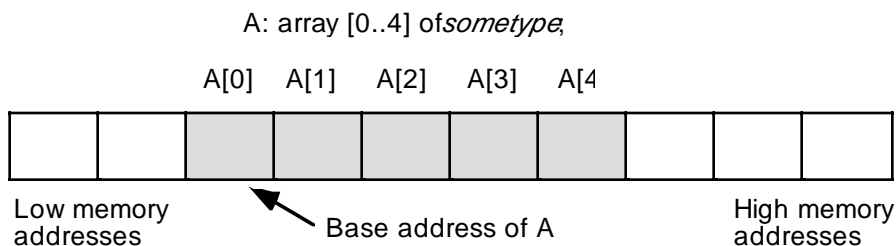


Figure 4.1 Array Layout in Memory

The *base address* of an array is the address of the first element on the array and always appears in the lowest memory location. The second array element directly follows the first in memory, the third element follows the second, etc. Note that there is no requirement that the indices start at zero. They may start with any number as long as they are contiguous. However, for the purposes of discussion, it's easier to discuss accessing array elements if the first index is zero. This text generally begins most arrays at index zero unless

1. Or some value whose underlying representation is integer, such as character, enumerated, and boolean types.

there is a good reason to do otherwise. However, this is for consistency only. There is no efficiency benefit one way or another to starting the array index at zero.

To access an element of an array, you need a function that translates an array index to the address of the indexed element. For a single dimension array, this function is very simple. It is

$$\text{Element_Address} = \text{Base_Address} + ((\text{Index} - \text{Initial_Index}) * \text{Element_Size})$$

where *Initial_Index* is the value of the first index in the array (which you can ignore if zero) and the value *Element_Size* is the size, in bytes, of an individual element of the array.

4.3 Declaring Arrays in Your HLA Programs

Before you access elements of an array, you need to set aside storage for that array. Fortunately, array declarations build on the declarations you've seen thus far. To allocate *n* elements in an array, you would use a declaration like the following in one of the variable declaration sections:

```
ArrayName: basetype[n];
```

ArrayName is the name of the array variable and *basetype* is the type of an element of that array. This sets aside storage for the array. To obtain the base address of the array, just use *ArrayName*.

The “[n]” suffix tells HLA to duplicate the object *n* times. Now let's look at some specific examples:

```
static
```

```
CharArray: char[128];           // Character array with elements 0..127.
IntArray: integer[ 8 ];        // "integer" array with elements 0..7.
ByteArray: byte[10];           // Array of bytes with elements 0..9.
PtrArray: dword[4];            // Array of double words with elements 0..3.
```

The second example, of course, assumes that you have defined the *integer* data type in the TYPE section of the program.

These examples all allocate storage for uninitialized arrays. You may also specify that the elements of the arrays be initialized to a single value using declarations like the following in the STATIC and READ-ONLY sections:

```
RealArray: real32[8] := [ 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0 ];
IntegerAry: integer[8] := [ 1, 1, 1, 1, 1, 1, 1, 1 ];
```

These definitions both create arrays with eight elements. The first definition initializes each four-byte real value to 1.0, the second declaration initializes each integer element to one. Note that the number of constants within the square brackets must match the size you declare for the array.

This initialization mechanism is fine if you want each element of the array to have the same value. What if you want to initialize each element of the array with a (possibly) different value? No sweat, just specify a different set of values in the list surrounded by the square brackets in the example above:

```
RealArray: real32[8] := [ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 ];
IntegerAry: integer[8] := [ 1, 2, 3, 4, 5, 6, 7, 8 ];
```

4.4 HLA Array Constants

The last few examples in the last section demonstrate the use of HLA array constants. An HLA array constant is nothing more than a list of values (all the same time) surrounded by a pair of brackets. The following are all legal array constants:

```
[ 1, 2, 3, 4 ]
[ 2.0, 3.14159, 1.0, 0.5 ]
```

```
[ 'a', 'b', 'c', 'd' ]
[ "Hello", "world", "of", "assembly" ]
```

(note that this last array constant contains four double word pointers to the four HLA strings appearing elsewhere in memory.)

As you saw in the previous section you can use array constants in the `STATIC` and `READONLY` sections to provide initial values for array variables. Of course, the number of comma separated items in an array constant must exactly match the number of array elements in the variable declaration. Likewise, the type of the array constant's elements must match the type of the elements in the array variable.

Using array constants to initialize small arrays is very convenient. Of course, if your array has several thousand elements in it, typing them all in will not be very much fun. Most arrays initialized this way have no more than a couple hundred entries, and generally far less than 100. It is reasonable to use an array constant to initialize such variables. However, at some point it will become far too tedious and error-prone to initialize arrays in this fashion. It is doubtful, for example, that you would want to manually initialize an array with 1,000 different elements using an array constant². However, if you want to initialize all the elements of an array with the same value, HLA does provide a special array constant syntax for doing so. Consider the following declaration:

```
BigArray: uns32[ 1000 ] := 1000 dup [ 1 ] ;
```

This declaration creates a 1,000 element integer array initializing each element of the array with the value one. The “1000 dup [1]” expression tells HLA to create an array constant by duplicating the single value “[1]” one thousand times. You can even use the `DUP` operator to duplicate a series of values (rather than a single value) as the following example indicates:

```
SixteenInts: int32[16] := 4 dup [1,2,3,4];
```

This example initializes *SixteenInts* with four copies of the sequence “1, 2, 3, 4” yielding a total of sixteen different integers (i.e., 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4).

You will see some more possibilities with the `DUP` operator when looking at multidimensional arrays a little later.

4.5 Accessing Elements of a Single Dimension Array

To access an element of a zero-based array, you can use the simplified formula:

$$\text{Element_Address} = \text{Base_Address} + \text{index} * \text{Element_Size}$$

For the *Base_Address* entry you can use the name of the array (since HLA associates the address of the first element of an array with the name of that array). The *Element_Size* entry is the number of bytes for each array element. If the object is an array of bytes, the *Element_Size* field is one (resulting in a very simple computation). If each element of the array is a word (or other two-byte type) then *Element_Size* is two. And so on. To access an element of the *SixteenInts* array in the previous section, you'd use the formula:

$$\text{Element_Address} = \text{SixteenInts} + \text{index} * 4$$

The 80x86 code equivalent to the statement “`EAX:=SixteenInts[index]`” is

```
mov( index, ebx );
shl( 2, ebx );           //Sneaky way to compute 4*ebx
mov( SixteenInts[ ebx ], eax );
```

There are two important things to notice here. First of all, this code uses the `SHL` instruction rather than the `INTMUL` instruction to compute $4 * \text{index}$. The main reason for choosing `SHL` is that it was more efficient. It turns out that `SHL` is a *lot* faster than `INTMUL` on many processors.

2. In the chapter on Macros and the HLA Run-Time Language you will learn how to automate the initialization of large array objects. So initializing large objects is not completely out of the question.

The second thing to note about this instruction sequence is that it does not explicitly compute the sum of the base address plus the index times two. Instead, it relies on the indexed addressing mode to implicitly compute this sum. The instruction “`mov(SixteenInts[ebx], eax);`” loads EAX from location *SixteenInts*+EBX which is the base address plus *index**4 (since EBX contains *index**4). Sure, you could have used

```
lea( eax, SixteenInts );
mov( index, ebx );
shl( 2, ebx );           //Sneaky way to compute 4*ebx
add( eax, ebx );        //Compute base address plus index*4
mov( SixteenInts[ ebx ], eax );
```

in place of the previous sequence, but why use five instructions where three will do the same job? This is a good example of why you should know your addressing modes inside and out. Choosing the proper addressing mode can reduce the size of your program, thereby speeding it up.

Of course, as long as we’re discussing efficiency improvements, it’s worth pointing out that the 80x86 scaled indexed addressing modes let you automatically multiply an index by one, two, four, or eight. Since this current example multiplies the index by four, we can simplify the code even farther by using the scaled indexed addressing mode:

```
mov( index, ebx );
mov( SixteenInts[ ebx*4 ], eax );
```

Note, however, that if you need to multiply by some constant other than one, two, four, or eight, then you cannot use the scaled indexed addressing modes. Similarly, if you need to multiply by some element size that is not a power of two, you will not be able to use the SHL instruction to multiply the index by the element size; instead, you will have to use INTMUL or some other instruction sequence to do the multiplication.

The indexed addressing mode on the 80x86 is a natural for accessing elements of a single dimension array. Indeed, it’s syntax even suggests an array access. The only thing to keep in mind is that you must remember to multiply the index by the size of an element. Failure to do so will produce incorrect results.

Before moving on to multidimensional arrays, a couple of additional points about addressing modes and arrays are in order. The above sequences work great if you only access a single element from the *SixteenInts* array. However, if you access several different elements from the array within a short section of code, and you can afford to dedicate another register to the operation, you can certainly shorten your code and, perhaps, speed it up as well. Consider the following code sequence:

```
lea( ebx, SixteenInts );
mov( index, esi );
mov( [ebx+esi*4], eax );
```

Now EBX contains the base address and ESI contains the *index* value. Of course, this hardly appears to be a good trade-off. However, when accessing additional elements of *SixteenInts* you do not have to reload EBX with the base address of *SixteenInts* for the next access. The following sequence is a little shorter than the comparable sequence that doesn’t load the base address into EBX:

```
lea( ebx, SixteenInts );
mov( index, esi );
mov( [ebx+esi*4], eax );
.
.           //Assumption: EBX is left alone
.           //           through this code.
mov( index2, esi );
mov( [ebx+esi*4], eax );
```

This code is slightly shorter because the “`mov([ebx+esi*4], eax);`” instruction is slightly shorter than the “`mov(SixteenInts[ebx*4], eax);`” instruction. Of course the more accesses to *SixteenInts* you make without reloading EBX, the greater your savings will be. Tricky little code sequences such as this one sometimes pay off handsomely. However, the savings depend entirely on which processor you’re using. Code

sequences that run faster on one 80x86 CPU might actually run *slower* on a different CPU. Unfortunately, if speed is what you're after there are no hard and fast rules. In fact, it is very difficult to predict the speed of most instructions on the 80x86 CPUs.

4.5.1 Sorting an Array of Values

Almost every textbook on this planet gives an example of a sort when introducing arrays. Since you've probably seen how to do a sort in high level languages already, it's probably instructive to take a quick look at a sort in HLA. The example code in this section will use a variant of the Bubble Sort which is great for short lists of data and lists that are nearly sorted, but horrible for just about everything else³.

```

const
  NumElements:= 16;

static
  DataToSort: uns32[ NumElements ] :=
    [
      1, 2, 16, 14,
      3, 9, 4, 10,
      5, 7, 15, 12,
      8, 6, 11, 13
    ];

  NoSwap: boolean;

  .
  .
  .

  // Bubble sort for the DataToSort array:

  repeat

    mov( true, NoSwap );
    for( mov( 0, ebx ); ebx <= NumElements-2; inc( ebx ) ) do

      mov( DataToSort[ ebx*4], eax );
      if( eax > DataToSort[ ebx*4 + 4] ) then

        mov( DataToSort[ ebx*4 + 4 ], ecx );
        mov( ecx, DataToSort[ ebx*4 ] );
        mov( eax, DataToSort[ ebx*4 + 4 ] ); // Note: EAX contains
        mov( false, NoSwap );              // DataToSort[ ebx*4 ]

      endif;

    endfor;

  until( NoSwap );

```

The bubble sort works by comparing adjacent elements in an array. The interesting thing to note in this code fragment is how it compares adjacent elements. You will note that the IF statement compares EAX (which contains DataToSort[ebx*4]) against DataToSort[EBX*4 + 4]. Since each element of this array is four bytes (*uns32*), the index [EBX*4 + 4] references the next element beyond [EBX*4].

3. Fear not, you'll see some better sorting algorithms in the chapter on procedures and recursion.

As is typical for a bubble sort, this algorithm terminates if the innermost loop completes without swapping any data. If the data is already presorted, then the bubble sort is very efficient, making only one pass over the data. Unfortunately, if the data is not sorted (worst case, if the data is sorted in reverse order), then this algorithm is extremely inefficient. Indeed, although it is possible to modify the code above so that, on the average, it runs about twice as fast, such optimizations are wasted on such a poor algorithm. However, the Bubble Sort is very easy to implement and understand (which is why introductory texts continue to use it in examples). Fortunately, you will learn about more advanced sorts later in this text, so you won't be stuck with it for very long.

4.6 Multidimensional Arrays

The 80x86 hardware can easily handle single dimension arrays. Unfortunately, there is no magic addressing mode that lets you easily access elements of multidimensional arrays. That's going to take some work and lots of instructions.

Before discussing how to declare or access multidimensional arrays, it would be a good idea to figure out how to implement them in memory. The first problem is to figure out how to store a multi-dimensional object into a one-dimensional memory space.

Consider for a moment a Pascal array of the form "A:array[0..3,0..3] of char;". This array contains 16 bytes organized as four rows of four characters. Somehow you've got to draw a correspondence with each of the 16 bytes in this array and 16 contiguous bytes in main memory. Figure 4.2 shows one way to do this:

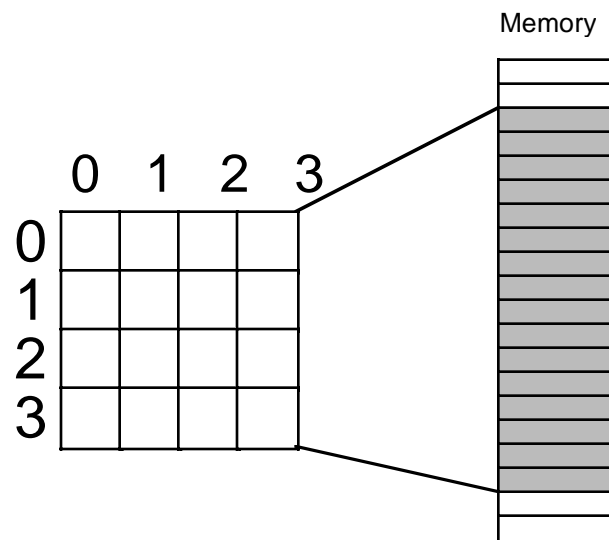


Figure 4.2 Mapping a 4x4 Array to Sequential Memory Locations

The actual mapping is not important as long as two things occur: (1) each element maps to a unique memory location (that is, no two entries in the array occupy the same memory locations) and (2) the mapping is consistent. That is, a given element in the array always maps to the same memory location. So what you really need is a function with two input parameters (row and column) that produces an offset into a linear array of sixteen memory locations.

Now any function that satisfies the above constraints will work fine. Indeed, you could randomly choose a mapping as long as it was unique. However, what you really want is a mapping that is efficient to compute at run time and works for any size array (not just 4x4 or even limited to two dimensions). While there are a

large number of possible functions that fit this bill, there are two functions in particular that most programmers and most high level languages use: *row major ordering* and *column major ordering*.

4.6.1 Row Major Ordering

Row major ordering assigns successive elements, moving across the rows and then down the columns, to successive memory locations. This mapping is demonstrated in Figure 4.3:

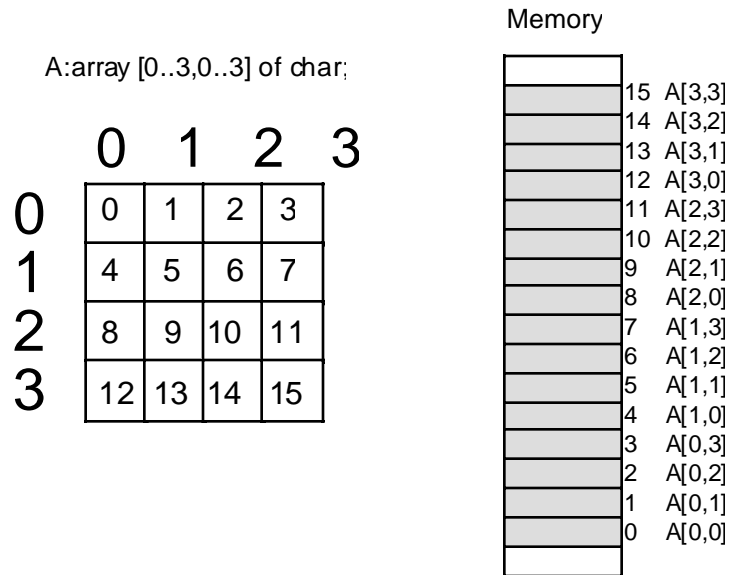


Figure 4.3 Row Major Array Element Ordering

Row major ordering is the method employed by most high level programming languages including Pascal, C/C++, Java, Ada, Modula-2, etc. It is very easy to implement and easy to use in machine language. The conversion from a two-dimensional structure to a linear array is very intuitive. You start with the first row (row number zero) and then concatenate the second row to its end. You then concatenate the third row to the end of the list, then the fourth row, etc. (see Figure 4.4).

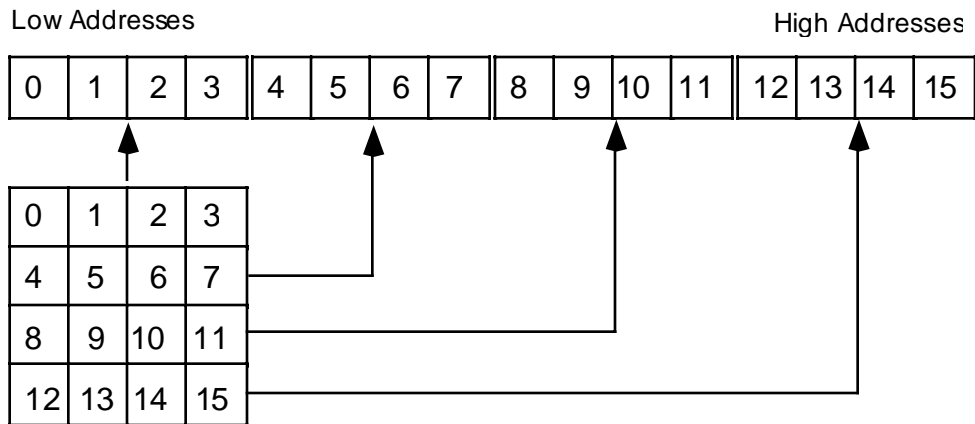


Figure 4.4 Another View of Row-Major Ordering for a 4x4 Array

For those who like to think in terms of program code, the following nested Pascal loop also demonstrates how row major ordering works:

```
index := 0;
for colindex := 0 to 3 do
  for rowindex := 0 to 3 do
    begin
      memory [index] := rowmajor [colindex][rowindex];
      index := index + 1;
    end;
```

The important thing to note from this code, that applies regardless of the number of dimensions, is that the rightmost index increases the fastest. That is, as you allocate successive memory locations you increment the rightmost index until you reach the end of the current row. Upon reaching the end, you reset the index back to the beginning of the row and increment the next successive index by one (that is, move down to the next row.). This works equally well for any number of dimensions⁴. The following Pascal segment demonstrates row major organization for a 4x4x4 array:

```
index := 0;
for depthindex := 0 to 3 do
  for colindex := 0 to 3 do
    for rowindex := 0 to 3 do begin
      memory [index] := rowmajor [depthindex][colindex][rowindex];
      index := index + 1;
    end;
```

The actual function that converts a list of index values into an offset doesn't involve loops or much in the way of fancy computations. Indeed, it's a slight modification of the formula for computing the address of an element of a single dimension array. The formula to compute the offset for a two-dimension row major ordered array declared in Pascal as "A:array [0..3,0..3] of integer" is

$$\text{Element_Address} = \text{Base_Address} + (\text{colindex} * \text{row_size} + \text{rowindex}) * \text{Element_Size}$$

As usual, *Base_Address* is the address of the first element of the array (*A[0][0]* in this case) and *Element_Size* is the size of an individual element of the array, in bytes. *Colindex* is the leftmost index, *rowindex* is the rightmost index into the array. *Row_size* is the number of elements in one row of the array (four, in

4. By the way, the number of dimensions of an array is its *arity*.

this case, since each row has four elements). Assuming *Element_Size* is one, this formula computes the following offsets from the base address:

Column index	Row Index	Offset into Array
0	0	0
0	1	1
0	2	2
0	3	3
1	0	4
1	1	5
1	2	6
1	3	7
2	0	8
2	1	9
2	2	10
2	3	11
3	0	12
3	1	13
3	2	14
3	3	15

For a three-dimensional array, the formula to compute the offset into memory is the following:

$$\text{Address} = \text{Base} + ((\text{depthindex} * \text{col_size} + \text{colindex}) * \text{row_size} + \text{rowindex}) * \text{Element_Size}$$

Col_size is the number of items in a column, *row_size* is the number of items in a row. In C/C++, if you've declared the array as "type A[i] [j] [k];" then *row_size* is equal to *k* and *col_size* is equal to *j*.

For a four dimensional array, declared in C/C++ as "type A[i] [j] [k] [m];" the formula for computing the address of an array element is

$$\text{Address} = \text{Base} + (((\text{LeftIndex} * \text{depth_size} + \text{depthindex}) * \text{col_size} + \text{colindex}) * \text{row_size} + \text{rowindex}) * \text{Element_Size}$$

Depth_size is equal to *j*, *col_size* is equal to *k*, and *row_size* is equal to *m*. *LeftIndex* represents the value of the leftmost index.

By now you're probably beginning to see a pattern. There is a generic formula that will compute the offset into memory for an array with *any* number of dimensions, however, you'll rarely use more than four.

Another convenient way to think of row major arrays is as arrays of arrays. Consider the following single dimension Pascal array definition:

```
A: array [0..3] of sometype;
```

Assume that *sometype* is the type "sometype = array [0..3] of char;".

A is a single dimension array. Its individual elements happen to be arrays, but you can safely ignore that for the time being. The formula to compute the address of an element of a single dimension array is

$$\text{Element_Address} = \text{Base} + \text{Index} * \text{Element_Size}$$

In this case *Element_Size* happens to be four since each element of *A* is an array of four characters. So what does this formula compute? It computes the base address of each row in this 4x4 array of characters (see Figure 4.5):

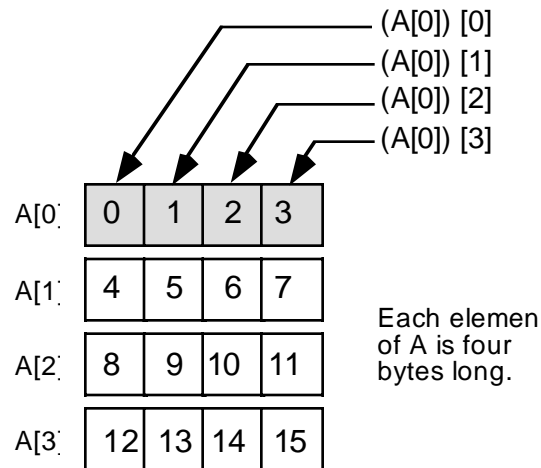


Figure 4.5 Viewing a 4x4 Array as an Array of Arrays

Of course, once you compute the base address of a row, you can reapply the single dimension formula to get the address of a particular element. While this doesn't affect the computation at all, conceptually it's probably a little easier to deal with several single dimension computations rather than a complex multidimensional array element address computation.

Consider a Pascal array defined as "A:array [0..3] [0..3] [0..3] [0..3] [0..3] of char;" You can view this five-dimension array as a single dimension array of arrays. The following Pascal code demonstrates such a definition:

```
type
  OneD = array [0..3] of char;
  TwoD = array [0..3] of OneD;
  ThreeD = array [0..3] of TwoD;
  FourD = array [0..3] of ThreeD;
var
  A : array [0..3] of FourD;
```

The size of *OneD* is four bytes. Since *TwoD* contains four *OneD* arrays, its size is 16 bytes. Likewise, *ThreeD* is four *TwoDs*, so it is 64 bytes long. Finally, *FourD* is four *ThreeDs*, so it is 256 bytes long. To compute the address of "A [b, c, d, e, f]" you could use the following steps:

- Compute the address of *A [b]* as "Base + b * size". Here size is 256 bytes. Use this result as the new base address in the next computation.
- Compute the address of *A [b, c]* by the formula "Base + c*size", where *Base* is the value obtained immediately above and size is 64. Use the result as the new base in the next computation.
- Compute the address of *A [b, c, d]* by "Base + d*size" with *Base* coming from the above computation and size being 16.
- Compute the address of *A [b, c, d, e]* with the formula "Base + e*size" with *Base* from above and size being four. Use this value as the base for the next computation.
- Finally, compute the address of *A [b, c, d, e, f]* using the formula "Base + f*size" where base comes from the above computation and size is one (obviously you can simply ignore this final multiplication). The result you obtain at this point is the address of the desired element.

Not only is this scheme easier to deal with than the fancy formulae given earlier, but it is easier to compute (using a single loop) as well. Suppose you have two arrays initialized as follows

$$A1 = [256, 64, 16, 4, 1] \quad \text{and} \quad A2 = [b, c, d, e, f]$$

then the Pascal code to perform the element address computation becomes:

```

for i := 0 to 4 do
  base := base + A1[i] * A2[i];

```

Presumably *base* contains the base address of the array before executing this loop. Note that you can easily extend this code to any number of dimensions by simply initializing *A1* and *A2* appropriately and changing the ending value of the for loop.

As it turns out, the computational overhead for a loop like this is too great to consider in practice. You would only use an algorithm like this if you needed to be able to specify the number of dimensions at run time. Indeed, one of the main reasons you won't find higher dimension arrays in assembly language is that assembly language displays the inefficiencies associated with such access. It's easy to enter something like "A [b,c,d,e,f]" into a Pascal program, not realizing what the compiler is doing with the code. Assembly language programmers are not so cavalier – they see the mess you wind up with when you use higher dimension arrays. Indeed, good assembly language programmers try to avoid two dimension arrays and often resort to tricks in order to access data in such an array when its use becomes absolutely mandatory. But more on that a little later.

4.6.2 Column Major Ordering

Column major ordering is the other function frequently used to compute the address of an array element. FORTRAN and various dialects of BASIC (e.g., older versions of Microsoft BASIC) use this method to index arrays.

In row major ordering the rightmost index increased the fastest as you moved through consecutive memory locations. In column major ordering the leftmost index increases the fastest. Pictorially, a column major ordered array is organized as shown in Figure 4.6:

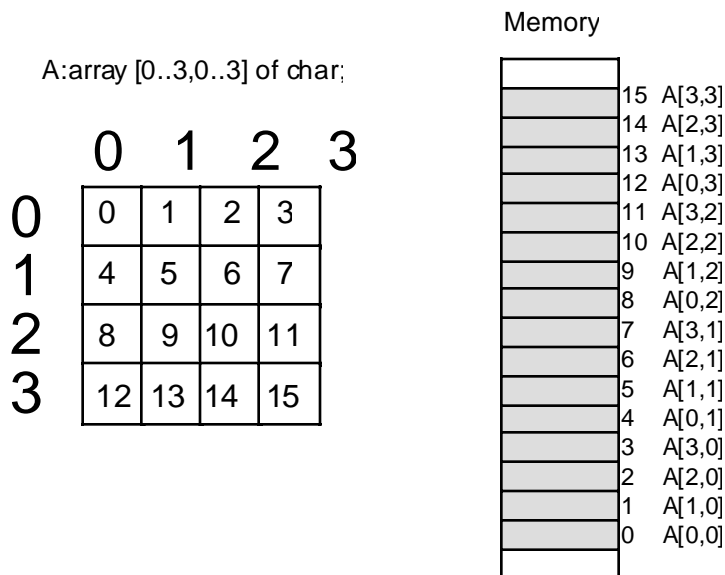


Figure 4.6 Column Major Array Element Ordering

The formulae for computing the address of an array element when using column major ordering is very similar to that for row major ordering. You simply reverse the indexes and sizes in the computation:

For a two-dimension column major array:

$$\text{Element_Address} = \text{Base_Address} + (\text{rowindex} * \text{col_size} + \text{colindex}) * \text{Element_Size}$$

For a three-dimension column major array:

```
Address = Base + ((rowindex*col_size+colindex) * depth_size + depthindex) *
Element_Size
```

For a four-dimension column major array:

```
Address =
  Base + (((rowindex * col_size + colindex)*depth_size + depthindex) *
  Left_size + Leftindex) * Element_Size
```

The single Pascal loop provided for row major access remains unchanged (to access $A[b][c][d][e][f]$):

```
for i := 0 to 4 do
  base := base + A1[i] * A2[i];
```

Likewise, the initial values of the $A1$ array remain unchanged:

```
A1 = {256, 64, 16, 4, 1}
```

The only thing that needs to change is the initial values for the $A2$ array, and all you have to do here is reverse the order of the indices:

```
A2 = {f, e, d, c, b}
```

4.7 Allocating Storage for Multidimensional Arrays

If you have an $m \times n$ array, it will have $m * n$ elements and require $m*n*Element_Size$ bytes of storage. To allocate storage for an array you must reserve this amount of memory. As usual, there are several different ways of accomplishing this task. Fortunately, HLA's array declaration syntax is very similar to high level language array declaration syntax, so C/C++, BASIC, and Pascal programmers will feel right at home. To declare a multidimensional array in HLA, you use a declaration like the following:

```
ArrayName: elementType [ comma_separated_list_of_dimension_bounds ];
```

For example, here is a declaration for a 4x4 array of characters:

```
GameGrid: char[ 4, 4 ];
```

Here is another example that shows how to declare a three dimensional array of strings:

```
NameItems: string[ 2, 3, 3 ];
```

Remember, string objects are really pointers, so this array declaration reserves storage for 18 double word pointers ($2*3*3=18$).

As was the case with single dimension arrays, you may initialize every element of the array to a specific value by following the declaration with the assignment operator and an array constant. Array constants ignore dimension information; all that matters is that the number of elements in the array constant correspond to the number of elements in the actual array. The following example shows the *GameGrid* declaration with an initializer:

```
GameGrid: char[ 4, 4 ] :=
  [
    'a', 'b', 'c', 'd',
    'e', 'f', 'g', 'h',
    'i', 'j', 'k', 'l',
    'm', 'n', 'o', 'p'
  ];
```

Note that HLA ignores the indentation and extra whitespace characters (e.g., newlines) appearing in this declaration. It was laid out to enhance readability (which is always a good idea). HLA does not interpret the four separate lines as representing rows of data in the array. Humans do, which is why it's good to lay out the initial data in this manner, but HLA completely ignores the physical layout of the declaration. All that

matters is that there are 16 (4*4) characters in the array constant. You'll probably agree that this is much easier to read than

```
GameGrid: char[ 4,4 ] :=
    [ 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
      'n', 'o', 'p' ];
```

Of course, if you have a large array, an array with really large rows, or an array with many dimensions, there is little hope for winding up with something reasonable. That's when comments that carefully explain everything come in handy.

As with single dimension arrays, you can use the DUP operator to initialize each element of a really large array with the same value. The following example initializes a 256x64 array of bytes so that each byte contains the value \$FF:

```
StateValue: byte[ 256, 64 ] := 256*64 dup [$ff];
```

Note the use of a constant expression to compute the number of array elements rather than simply using the constant 16,384 (256*64). The use of the constant expression more clearly suggests that this code is initializing each element of a 256x64 element array than does the simple literal constant 16,384.

Another HLA trick you can use to improve the readability of your programs is to use *nested array constants*. The following is an example of an HLA nested array constant:

```
[ [0, 1, 2], [3, 4], [10, 11, 12, 13] ]
```

Whenever HLA encounters an array constant nested inside another array constant, it simply removes the brackets surrounding the nested array constant and treats the whole constant as a single array constant. For example, HLA converts the nested array constant above to the following:

```
[ 0, 1, 2, 3, 4, 10, 11, 12, 13 ]
```

You can take advantage of this fact to help make your programs a little more readable. For multidimensional array constants you can enclose each row of the constant in square brackets to denote that the data in each row is grouped and separate from the other rows. As an example, consider the following declaration for the *GameGrid* array that is identical (as far as HLA is concerned) to the previous declaration:

```
GameGrid: char[ 4, 4 ] :=
    [
        [ 'a', 'b', 'c', 'd' ],
        [ 'e', 'f', 'g', 'h' ],
        [ 'i', 'j', 'k', 'l' ],
        [ 'm', 'n', 'o', 'p' ]
    ];
```

This declaration makes it clearer that the array constant is a 4x4 array rather than just a 16-element one-dimensional array whose elements wouldn't fit all on one line of source code. Little aesthetic improvements like this are what separate mediocre programmers from good programmers.

4.8 Accessing Multidimensional Array Elements in Assembly Language

Well, you've seen the formulae for computing the address of an array element. You've even looked at some Pascal code you could use to access elements of a multidimensional array. Now it's time to see how to access elements of those arrays using assembly language.

The MOV, SHL, and INTMUL instructions make short work of the various equations that compute offsets into multidimensional arrays. Let's consider a two dimension array first:

```
static
```

```

i:      int32;
j:      int32;
TwoD:   int32[ 4, 8 ];
.
.
.

// To perform the operation TwoD[i,j] := 5; you'd use code like the following.
// Note that the array index computation is (i*8 + j)*4.

mov( i, ebx );
shl( 3, ebx );      // Multiply by eight (shl by 3 is a multiply by 8).
add( j, ebx );
mov( 5, TwoD[ ebx*4 ] );

```

Note that this code does *not* require the use of a two register addressing mode on the 80x86. Although an addressing mode like `TwoD[ebx][esi]` looks like it should be a natural for accessing two dimensional arrays, that isn't the purpose of this addressing mode.

Now consider a second example that uses a three dimension array:

```

static
i:      int32;
j:      int32;
k:      int32;
ThreeD: int32[ 3, 4, 5 ];
.
.
.

// To perform the operation ThreeD[i,j,k] := ESI; you'd use the following code
// that computes ((i*4 + j)*5 + k)*4 as the address of ThreeD[i,j,k].

mov( i, ebx );
shl( 2, ebx );      // Four elements per column.
add( j, ebx );
intmul( 5, ebx );   // Five elements per row.
add( k, ebx );
mov( esi, ThreeD[ ebx*4 ] );

```

Note that this code uses the INTMUL instruction to multiply the value in EBX by five. Remember, the SHL instruction can only multiply a register by a power of two. While there are ways to multiply the value in a register by a constant other than a power of two, the INTMUL instruction is more convenient⁵.

4.9 Large Arrays and MASM

There is a defect in later versions of MASM v6.x that create some problems when you declare large static arrays in your programs. Now you may be wondering what this has to do with you since we're using HLA, but don't forget that HLA v1.x compiles to MASM assembly code and then runs MASM to assemble this output. Therefore, any defect in MASM is going to be a problem for HLA users.

The problem occurs when the total number of array elements you declare in a static section (STATIC, READONLY, or STORAGE) starts to get large. Large in this case is CPU dependent, but it falls somewhere between 128,000 and one million elements for most systems. MASM, for whatever reason, uses a very slow algorithm to emit array code to the object file; by the time you declare 64K array elements, MASM starts to produce a noticeable delay while compiling your code. After that point, the delay grows linearly with the

5. A full discussion of multiplication by constants other than a power of two appears in the chapter on arithmetic.

number of array elements (i.e., as you double the number of array elements you double the assembly time) until the data saturates MASM's internal buffers and the cache. Then there is a big jump in execution time. For example, on a 300 MHz Pentium II processor, compiling a program with an array with 256,000 elements takes about 30 seconds, compiling a program with an array having 512,000 element takes several minutes. Compiling a program with a one-megabyte array seems to take forever.

There are a couple of ways to solve this problem. First, of course, you can limit the size of your arrays in your program. Unfortunately, this isn't always an option available to you. The second possibility is to use MASM v6.11; the defect was introduced in MASM after this version. The problem with MASM v6.11 is that it doesn't support the MMX instruction set, so if you're going to compile MMX instructions (or other instructions that MASM v6.11 doesn't support) with HLA you will not be able to use this option. A third option is to put your arrays in a VAR section rather than a static declaration section; HLA processes arrays you declare in the VAR section so MASM never sees them. Hence, arrays you declare in the VAR section don't suffer from this problem.

4.10 Dynamic Arrays in Assembly Language

One problem with arrays up to this point is that their size is static. That is, the number of elements in all of the examples is chosen when writing the program, it is not set while the program is running (i.e., dynamically). Alas, sometimes you simply don't know how big an array needs to be when you're writing the program; you can only determine the size of the array while the program is running. This section describes how to allocate storage for arrays dynamically so you can set their size at run time.

Allocating storage for a single dimension array, and accessing elements of that array, is a nearly trivial task at run time. All you need to do is call the HLA Standard Library *malloc* routine specifying the size of the array, in bytes. *Malloc* will return a pointer to the base address of the new array in the EAX register. Typically, you would save this address in a pointer variable and use that value as the base address of the array in all future array accesses.

To access an element of a single dimensional dynamic array, you would generally load the base address into a register and compute the index in a second register. Then you could use the based indexed addressing mode to access elements of that array. This is not a whole lot more work than accessing elements of a statically allocated array. The following code fragment demonstrates how to allocate and access elements of a single dimension dynamic array:

```
static
    ArySize:      uns32;
    BaseAdrs:    pointer to uns32;
    .
    .
    .
    stdout.put( "How many elements do you want in your array? " );
    stdin.getu32();
    mov( eax, ArySize;           // Save away the upper bounds on this array.
    shl( 2, eax );              // Multiply eax by four to compute the number of bytes.
    malloc( eax );              // Allocate storage for the array.
    mov( eax, BaseAdrs );       // Save away the base address of the new array.
    .
    .
    .

    // Zero out each element of the array:

    mov( BaseAdrs, ebx );
    mov( 0, eax );
    for( mov(0, esi); esi < ArySize; inc( esi ) ) do

        mov( eax, [ebx + esi*4 ]);
```

```
endfor;
```

Dynamically allocating storage for a multidimensional array is fairly straight-forward. The number of elements in a multidimensional array is the product of all the dimension values; e.g., a 4x5 array has 20 elements. So if you get the bounds for each dimension from the user, all you need do is compute the product of all of these bound values and multiply the final result by the size of a single element. This computes the total number of bytes in the array, the value that *malloc* expects.

Accessing elements of multidimensional arrays is a little more problematic. The problem is that you need to keep the dimension information (that is, the bounds on each dimension) around because these values are needed when computing the row major (or column major) index into the array⁶. The conventional solution is to store these bounds into a static array (generally you know the *arity*, or number of dimensions, at compile-time, so it is possible to statically allocate storage for this array of dimension bounds). This array of dynamic array bounds is known as a *dope vector*. The following code fragment shows how to allocate storage for a two-dimensional dynamic array using a simple *dope vector*.

```
var
  ArrayPtr:  pointer to uns32;
  ArrayDims: uns32[2];
  .
  .
  .
// Get the array bounds from the user:

stdout.put( "Enter the bounds for dimension #1: " );
stdin.get( ArrayDims[0] );

stdout.put( "Enter the bounds for dimension #2: " );
stdin.get( ArrayDims[1*4] );

// Allocate storage for the array:

mov( ArrayDims[0], eax );
intmul( ArrayDims[1*4], eax );
shl( 2, eax );          // Multiply by four since each element is 4 bytes.
malloc( eax );          // Allocate storage for the array and
mov( eax, ArrayPtr );  // save away the pointer to the array.

// Initialize the array:

mov( 0, edx );
mov( ArrayPtr, edi );
for( mov( 0, ebx ); ebx < ArrayDims[0]; inc( ebx ) ) do

    for( mov( 0, ecx ); ecx < ArrayDims[1*4]; inc( ecx ) ) do

        // Compute the index into the array
        // as esi := ( ebx * ArrayDims[1*4] + ecx ) * 4
        // (Note that the final multiplication by four is
        // handled by the scaled indexed addressing mode below.)

        mov( ebx, esi );
        intmul( ArrayDims[1*4], esi );
        add( ecx, esi );

        // Initialize the current array element with edx.
```

6. Technically, you don't need the value of the left-most dimension bound to compute an index into the array, however, if you want to check the index bounds using the BOUND instruction (or some other technique), you will need this value around at run-time as well.


```

        mov( edx, [edi+esi*4] );
        inc( edx );

    endfor;

endfor;

```

4.11 HLA Standard Library Array Support

The HLA Standard Library provides an array module that helps reduce the effort needed to support static and dynamic arrays in your program. The “arrays.hhf” library module provides code to declare and allocate dynamic arrays, compute the index into an array, copy arrays, perform row reductions, transpose arrays, and more. This section will explore some of the more useful features the arrays module provides.

One of the more interesting features of the HLA Standard Library arrays module is that most of the array manipulation procedures support both statically allocated and dynamically allocated arrays. In fact, the HLA array procedures can automatically figure out if an array is static or dynamic and generate the appropriate code for that array. There is one catch, however. In order for HLA to be able to differentiate statically and dynamically allocated arrays, you must use the dynamic array declarations found in the arrays package. This won’t be a problem because HLA’s dynamic array facilities are powerful and very easy to use.

To declare a dynamic array with the HLA arrays package, you use a variable declaration like the following:

```
variableName: array.dArray( elementType, Arity );
```

The *elementType* parameter is a regular HLA data type identifier (e.g., *int32* or some type identifier you’ve defined in the TYPE section). The *Arity* parameter is a constant that specifies the number of dimensions for the array (*arity* is the formal name for “number of dimensions”). Note that you do not specify the bounds of each dimension in this declaration. Storage allocation occurs later, at run time. The following is an example of a declaration for a dynamically allocated two-dimensional matrix:

```
ScreenBuffer: array.dArray( char, 2 );
```

The *array.dArray* data type is actually an HLA macro⁷ that expands the above to the following:

```
ScreenBuffer: record
    dataPtr:      dword;
    dopeVector:   uns32[ 2 ];
    elementType: char;
endrecord;
```

The *dataPtr* field will hold the base address of the array once the program allocates storage for it. The *dopeVector* array has one element for each array dimension (the macro uses the second parameter of the *array.dArray* type as the number of dimensions for the *dopeVector* array). The *elementType* field is a single object that has the same type as an element of the dynamic array. HLA provides a couple of built-in functions that you can use on these fields to extract important information. The *@Elements* function returns the number of elements in an array. Therefore, “*@Elements(ScreenBuffer.dopeVector)*” will return the number of elements (two) in the *ScreenBuffer.dopeVector* array. Since this array contains one element for each dimension in the dynamic array, you can use the *@Elements* function with the *dopeVector* field to determine the arity of the array. You can use the HLA *@Size* function on the *ScreenBuffer.elementType* field to determine the size of an array element in the dynamic array. Most of the time you will know the arity and type of your dynamic arrays (after all, you declared them), so you probably won’t use these functions often until you start writing macros that process dynamic arrays.

7. See the chapter on Macros and the HLA Compile-Time Language for details on macros.

After you declare a dynamic array, you must initialize the dynamic array object before attempting to use the array. The HLA Standard Library *array.daAlloc* routine handles this task for you. This routine uses the following syntax:

```
array.daAlloc( arrayName, comma_separated_list_of_array_bounds );
```

To allocate storage for the *ScreenBuffer* variable in the previous example you could use a call like the following:

```
array.daAlloc( ScreenBuffer, 20, 40 );
```

This call will allocate storage for a 20x40 array of characters. It will store the base address of the array into the *ScreenBuffer.dataPtr* field. It will also initialize *ScreenBuffer.dopeVector[0]* with 20 and *ScreenBuffer.dopeVector[1*4]* with 40. To access elements of the *ScreenBuffer* array you can use the techniques of the previous section, or you could use the *array.index* function.

The *array.index* function automatically computes the address of an array element for you. This function uses the following call syntax:

```
array.index( reg32, arrayName, comma_separated_list_of_index_values );
```

The first parameter must be a 32-bit register. The *array.index* function will store the address of the specified array element in this register. The second *array.index* parameter must be the name of an array; this can be either a statically allocated array or an array you've declared with *array.dArray* and allocated dynamically with *array.daAlloc*. Following the array name parameter is a list of one or more array indices. The number of array indices must match the arity of the array. These array indices can be constants, *dword* memory variables, or registers (however, you must not specify the same register that appears in the first parameter as one of the array indices). Upon return from this function, you may access the specified array element using the register indirect addressing mode and the register appearing as the first parameter.

One last routine you'll want to know about when manipulating HLA dynamic arrays is the *array.daFree* routine. This procedure expects a single parameter that is the name of an HLA dynamic array. Calling *array.daFree* will free the storage associated with the dynamic array. The following code fragment is a rewrite of the example from the previous section that uses HLA dynamic arrays:

```
var
  da:    array.dArray( uns32, 2 );
  Bnd1:  uns32;
  Bnd2:  uns32;
  .
  .
  .
  // Get the array bounds from the user:

  stdout.put( "Enter the bounds for dimension #1: " );
  stdin.get( Bnd1 );

  stdout.put( "Enter the bounds for dimension #2: " );
  stdin.get( Bnd2 );

  // Allocate storage for the array:

  array.daAlloc( da, Bnd1, Bnd2 );

  // Initialize the array:

  mov( 0, edx );
  for( mov( 0, ebx ); ebx < Bnd1; inc( ebx ) ) do

    for( mov( 0, ecx ); ecx < Bnd2; inc( ecx ) ) do
```

```

// Initialize the current array element with edx.
// Use array.index to compute the address of the array element.

array.index( edi, da, ebx, ecx );
mov( edx, [edi] );
inc( edx );

endfor;

endifor;

```

Another extremely useful library module is the *array.cpy* routine. This procedure will copy the data from one array to another. The calling syntax is:

```
array.cpy( sourceArrayName, destArrayName );
```

The source and destination arrays can be static or dynamic arrays. The *array.cpy* automatically adjusts and emits the proper code for each combination of parameters. With most of the array manipulation procedures in the HLA Standard Library, you pay a small performance penalty for the convenience of these library modules. Not so with *array.cpy*. This procedure is very, very fast; much faster than writing a loop to copy the data element by element.

4.12 Putting It All Together

Accessing elements of an array is a very common operation in assembly language programs. This chapter provides the basic information you need to efficiently access array elements. After mastering the material in this chapter you should know how to declare arrays in HLA and access elements of those arrays in your programs.

Records, Unions, and Name Spaces

Chapter Five

5.1 Chapter Overview

This chapter discusses how to declare and use record (structures), unions, and name spaces in your programs. After strings and arrays, records are among the most commonly used composite data types; indeed, records are the mechanism you use to create user-defined composite data types. Many assembly language programmers never bother to learn how to use records in assembly language, yet would never consider not using them in high level language programs. This is somewhat inconsistent since records (structures) are just as useful in assembly language programs as in high level language programs. Given that you use records in assembly language (and especially HLA) in a manner quite similar to high level languages, there really is no reason for excluding this important tool from your programmer's tool chest. Although you'll use unions and name spaces far less often than records, their presence in the HLA language is crucial for many advanced applications. This brief chapter provides all the information you need to successfully use records, unions, and name spaces within your HLA programs.

5.2 Records

Another major composite data structure is the Pascal *record* or C/C++ *structure*¹. The Pascal terminology is probably better, since it tends to avoid confusion with the more general term *data structure*. Since HLA uses the term "record" we'll adopt that term here.

Whereas an array is homogeneous, whose elements are all the same, the elements in a record can be of any type. Arrays let you select a particular element via an integer index. With records, you must select an element (known as a *field*) by name.

The whole purpose of a record is to let you encapsulate different, but logically related, data into a single package. The Pascal record declaration for a student is probably the most typical example:

```
student =
  record
    Name: string [64];
    Major: integer;
    SSN: string[11];
    Midterm1: integer;
    Midterm2: integer;
    Final: integer;
    Homework: integer;
    Projects: integer;
  end;
```

Most Pascal compilers allocate each field in a record to contiguous memory locations. This means that Pascal will reserve the first 65 bytes for the name², the next two bytes hold the major code, the next 12 the Social Security Number, etc.

In HLA, you can also create structure types using the RECORD/ENDRECORD declaration. You would encode the above record in HLA as follows:

```
type
  student: record
    Name: char[65];
    Major: int16;
    SSN: char[12];
```

1. It also goes by some other names in other languages, but most people recognize at least one of these names.
2. Strings require an extra byte, in addition to all the characters in the string, to encode the length.

```

Midterm1: int16;
Midterm2: int16;
Final: int16;
Homework: int16;
Projects: int16;
endrecord;

```

As you can see, the HLA declaration is very similar to the Pascal declaration. Note that, to be true to the Pascal declaration, this example uses character arrays rather than strings for the *Name* and *SSN* (U.S Social Security Number) fields. In a real HLA record declaration you'd probably use a string type for at least the name (keeping in mind that a string variable is only a four byte pointer).

The field names within the record must be unique. That is, the same name may not appear two or more times in the same record. However, all field names are local to that record. Therefore, you may reuse those field names elsewhere in the program.

The RECORD/ENDRECORD type declaration may appear in a variable declaration section (e.g., STATIC or VAR) or in a TYPE declaration section. In the previous example the *Student* declaration appears in the TYPE section, so this does not actually allocate any storage for a *Student* variable. Instead, you have to explicitly declare a variable of type *Student*. The following example demonstrates how to do this:

```

var
John: Student;

```

This allocates 81 bytes of storage laid out in memory as shown in Figure 5.1.

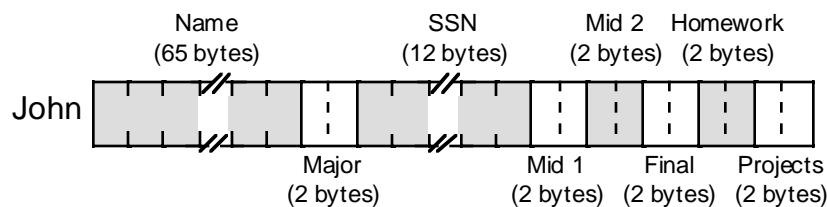


Figure 5.1 Student Data Structure Storage in Memory

If the label *John* corresponds to the *base address* of this record, then the *Name* field is at offset *John+0*, the *Major* field is at offset *John+65*, the *SSN* field is at offset *John+67*, etc.

To access an element of a structure you need to know the offset from the beginning of the structure to the desired field. For example, the *Major* field in the variable *John* is at offset 65 from the base address of *John*. Therefore, you could store the value in AX into this field using the instruction

```

mov( ax, (type word John[65]) );

```

Unfortunately, memorizing all the offsets to fields in a record defeats the whole purpose of using them in the first place. After all, if you've got to deal with these numeric offsets why not just use an array of bytes instead of a record?

Well, as it turns out, HLA lets you refer to field names in a record using the same mechanism C/C++ and Pascal use: the dot operator. To store AX into the *Major* field, you could use “mov(ax, John.Major);” instead of the previous instruction. This is much more readable and certainly easier to use.

Note that the use of the dot operator does *not* introduce a new addressing mode. The instruction “mov(ax, John.Major);” still uses the displacement only addressing mode. HLA simply adds the base address of *John* with the offset to the *Major* field (65) to get the actual displacement to encode into the instruction.

Like any type declaration, HLA requires all record type declarations to appear in the program before you use them. However, you don't have to define all records in the TYPE section to create record variables. You can use the RECORD/ENDRECORD declaration directly in a variable declaration section. This is con-

venient if you have only one instance of a given record object in your program. The following example demonstrates this:

```
storage
```

```
    OriginPoint: record
        x: uns8;
        y: uns8;
        z: uns8;
    endrecord;
```

5.3 Record Constants

HLA lets you define record constants. In fact, HLA supports both symbolic record constants and literal record constants. Record constants are useful as initializers for static record variables. They are also quite useful as compile-time data structures when using the HLA compile-time language (see the chapters on Macros and the HLA Compile-Time Language). This section discusses how to create record constants.

A record literal constant takes the following form:

```
RecordTypeName: [ List_of_comma_separated_constants ]
```

The *RecordTypeName* is the name of a record data type you've defined in an HLA TYPE section prior to this point. To create a record constant you must have previously defined the record type in a TYPE section of your program.

The constant list appearing between the brackets are the data items for each of the fields in the specified record. The first item in the list corresponds to the first field of the record, the second item in the list corresponds to the second field, etc. The data types of each of the constants appearing in this list must match their respective field types. The following example demonstrates how to use a literal record constant to initialize a record variable:

```
type
    point: record
        x:int32;
        y:int32;
        z:int32;
    endrecord;

static
    Vector: point := point:[ 1, -2, 3 ];
```

This declaration initializes *Vector.x* with 1, *Vector.y* with -2, and *Vector.z* with 3.

You can also create symbolic record constants by declaring record objects in the CONST or VAL sections of your program. You access fields of these symbolic record constants just as you would access the field of a record variable, using the dot operator. Since the object is a constant, you can specify the field of a record constant anywhere a constant of that field's type is legal. You can also employ symbolic record constants as record variable initializers. The following example demonstrates this:

```
type
    point: record
        x:int32;
        y:int32;
        z:int32;
    endrecord;

const
    PointInSpace: point := point:[ 1, 2, 3 ];
```

```
static
    Vector: point := PointInSpace;
```

```

XCoord: int32 := PointInSpace.x;
.
.
.
stdout.put( "Y Coordinate is ", PointInSpace.y, nl );
.
.
.

```

5.4 Arrays of Records

It is a perfectly reasonable operation to create an array of records. To do so, you simply create a record type and then use the standard array declaration syntax when declaring an array of that record type. The following example demonstrates how you could do this:

```

type
  recElement:
    record
      << fields for this record >>
    endrecord;
  .
  .
  .
static
  recArray: recElement[4];

```

To access an element of this array you use the standard array indexing techniques found in the chapter on arrays. Since *recArray* is a single dimension array, you'd compute the address of an element of this array using the formula "baseAddress + index*@size(recElement)." For example, to access an element of *recArray* you'd use code like the following:

```

// Access element i of recArray:

intmul( @size( recElement ), i, ebx ); // ebx := i*@size( recElement )
mov( recArray.someField[ebx], eax );

```

Note that the index specification follows the entire variable name; remember, this is assembly not a high level language (in a high level language you'd probably use "recArray[i].someField").

Naturally, you can create multidimensional arrays of records as well. You would use the standard row or column major order functions to compute the address of an element within such records. The only thing that really changes (from the discussion of arrays) is that the size of each element is the size of the record object.

```

static
  rec2D: recElement[ 4, 6 ];
  .
  .
  .
// Access element [i,j] of rec2D and load "someField" into EAX:

intmul( 6, i, ebx );
add( j, ebx );
intmul( @size( recElement ), ebx );
mov( rec2D.someField[ ebx ], eax );

```


5.5 Arrays/Records as Record Fields

Records may contain other records or arrays as fields. Consider the following definition:

```
type
  Pixel:
    record
      Pt:      point;
      color:   dword;
    endrecord;
```

The definition above defines a single point with a 32 bit color component. When initializing an object of type *Pixel*, the first initializer corresponds to the *Pt* field, *not the x-coordinate field*. **The following definition is incorrect:**

```
static
  ThisPt: Pixel := Pixel:[ 5, 10 ];    // Syntactically incorrect!
```

The value of the first field (“5”) is not an object of type *point*. Therefore, the assembler generates an error when encountering this statement. HLA will allow you to initialize the fields of *Pixel* using declarations like the following:

```
static
  ThisPt: Pixel := Pixel:[ point:[ 1, 2, 3 ], 10 ];
  ThatPt: Pixel := Pixel:[ point:[ 0, 0, 0 ], 5 ];
```

Accessing *Pixel* fields is very easy. Like a high level language you use a single period to reference the *Pt* field and a second period to access the *x*, *y*, and *z* fields of *point*:

```
    stdout.put( "ThisPt.Pt.x = ", ThisPt.Pt.x, nl );
    stdout.put( "ThisPt.Pt.y = ", ThisPt.Pt.y, nl );
    stdout.put( "ThisPt.Pt.z = ", ThisPt.Pt.z, nl );
    .
    .
    .
  mov( eax, ThisPt.Color );
```

You can also declare *arrays* as record fields. The following record creates a data type capable of representing an object with eight points (e.g., a cube):

```
type
  Object8:
    record
      Pts:      point[8];
      Color:    dword;
    endrecord;
```

This record allocates storage for eight different points. Accessing an element of the *Pts* array requires that you know the size of an object of type *point* (remember, you must multiply the index into the array by the size of one element, 12 in this particular case). Suppose, for example, that you have a variable *CUBE* of type *Object8*. You could access elements of the *Pts* array as follows:

```
// CUBE.Pts[i].x := 0;

    mov( i, ebx );
    intmul( 12, ebx );
    mov( 0, CUBE.Pts.x[ebx] );
```

The one unfortunate aspect of all this is that you must know the size of each element of the *Pts* array. Fortunately, HLA provides a built-in function that will compute the size of an array element (in bytes) for you: the *@size* function. You can rewrite the code above using *@size* as follows:

```
// CUBE.Pts[i].x := 0;

    mov( i, ebx );
```

```
intmul( @size( point ), ebx );
mov( 0, CUBE.Pts.x[ebx] );
```

This solution is much better than multiplying by the literal constant 12. Not only does HLA figure out the size for you (so you don't have to), it automatically substitutes the correct size if you ever change the definition of the *point* record in your program. For this reason, you should always use the `@size` function to compute the size of array element objects in your programs.

Note in this example that the index specification (“[ebx]”) follows the whole object name even though the array is *Pts*, not *x*. Remember, the “[ebx]” specification is an indexed addressing mode, not an array index. Indexes always follow the entire name, you do not attach them to the array component as you would in a high level language like C/C++ or Pascal. This produces the correct result because addition is commutative, and the dot operator (as well as the index operator) corresponds to addition. In particular, the expression “CUBE.Pts.x[ebx]” tells HLA to compute the sum of *CUBE* (the base address of the object) plus the offset to the *Pts* field, plus the offset to the *x* field plus the value of EBX. Technically, we're really computing `offset(CUBE)+offset(Pts)+EBX+offset(x)` but we can rearrange this since addition is commutative.

You can also define two-dimensional arrays within a record. Accessing elements of such arrays is no different than any other two-dimensional array other than the fact that you must specify the array's field name as the base address for the array. E.g.,

```
type
  RecW2DArray:
    record
      intField: int32;
      aField: int32[4,5];
      .
      .
      .
    endrecord;

static
  recVar: RecW2DArray;
  .
  .
  .
  // Access element [i,j] of the aField field using Row-major ordering:

  mov( i, ebx );
  intmul( 5, ebx );
  add( j, ebx );
  mov( recVar.aField[ ebx*4 ], eax );
  .
  .
  .
```

The code above uses the standard row-major calculation to index into a 4x5 array of double words. The only difference between this example and a stand-alone array access is the fact that the base address is *recVar.aField*.

There are two common ways to nest record definitions. As noted earlier in this section, you can create a record type in a TYPE section and then use that type name as the data type of some field within a record (e.g., the *Pt:point* field in the *Pixel* data type above). It is also possible to declare a record directly within another record without creating a separate data type for that record; the following example demonstrates this:

```
type
  NestedRecs:
    record
      iField: int32;
      sField: string;
      rField:
```

```

        record
            i:int32;
            u:uns32;
        endrecord;
    cField:char;
endrecord;

```

Generally, it's a better idea to create a separate type rather than embed records directly in other records, but nesting them is perfectly legal and a reasonable thing to do on occasion.

If you have an array of records and one of the fields of that record type is an array, you must compute the indexes into the arrays independently of one another and then use the sum of these indexes as the ultimate index. The following example demonstrates how to do this:

```

type
    recType:
        record
            arrayField: dword[4,5];
            << Other Fields >>
        endrecord;

static
    aryOfRecs: recType[3,3];
    .
    .
    .
    // Access aryOfRecs[i,j].arrayField[k,l]:

    intmul( 5, i, ebx );           // Computes index into aryOfRecs
    add( j, ebx );                // as (i*5 +j)*@size( recType ).
    intmul( @size( recType ), ebx );

    intmul( 3, k, eax );          // Computes index into aryOfRecs
    add( l, eax );                // as (k*3 + j) (*4 handled later).

    mov( aryOfRecs.arrayField[ ebx + eax*4 ], eax );

```

Note the use of the base plus scaled indexed addressing mode to simplify this operation.

5.6 Controlling Field Offsets Within a Record

By default, whenever you create a record, HLA automatically assigns the offset zero to the first field of that record. This corresponds to records in a high level language and is the intuitive default condition. In some instances, however, you may want to assign a different starting offset to the first field of the record. HLA provides a mechanism that lets you set the starting offset of the first field in the record.

The syntax to set the first offset is

```

name:
    record := startingOffset;
        << Record Field Declarations >>
    endrecord;

```

Using the syntax above, the first field will have the starting offset specified by the *startingOffset int32* constant expression. Since this is an *int32* value, the starting offset value can be positive, zero, or negative.

One circumstance where this feature is invaluable is when you have a record whose base address is actually somewhere within the data structure. The classic example is an HLA string. An HLA string uses a record declaration similar to the following:

```

record

```

```

    MaxStrLen: dword;
    length: dword;
    charData: char[xxxx];
endrecord;

```

As you're well aware by now, HLA string pointers do not contain the address of the *MaxStrLen* field; they point at the *charData* field. The *str.strRec* record type found in the HLA Standard Library Strings module uses a record declaration similar to the following:

```

type
  strRec:
    record := -8;
      MaxStrLen: dword;
      length: dword;
      charData: char;
    endrecord;

```

The starting offset for the *MaxStrLen* field is -8. Therefore, the offset for the *length* field is -4 (four bytes later) and the offset for the *charData* field is zero. Therefore, if *EBX* points at some string data, then “(type *str.strRec* [*ebx*]).length” is equivalent to “[*ebx*-4]” since the *length* field has an offset of -4.

Generally, you will not use HLA's ability to specify the starting field offset when creating your own record types. Instead, this feature finds most of its use when you are mapping an HLA data type over the top of some other predefined data type in memory (strings are a good example, but there are many other examples as well).

5.7 Aligning Fields Within a Record

To achieve maximum performance in your programs, or to ensure that HLA's records properly map to records or structures in some high level language, you will often need to be able to control the alignment of fields within a record. For example, you might want to ensure that a *dword* field's offset is an even multiple of four. You use the *ALIGN* directive to do this, the same way you would use *ALIGN* in the *STATIC* declaration section of your program. The following example shows how to align some fields on important boundaries:

```

type
  PaddedRecord:
    record
      c:char;
      align(4);
      d:dword;
      b:boolean;
      align(2);
      w:word;
    endrecord;

```

Whenever HLA encounters the *ALIGN* directive within a record declaration, it automatically adjusts the following field's offset so that it is an even multiple of the value the *ALIGN* directive specifies. It accomplishes this by increasing the offset of that field, if necessary. In the example above, the fields would have the following offsets: *c*:0, *d*:4, *b*:8, *w*:10. Note that HLA inserts three bytes of padding between *c* and *d* and it inserts one byte of padding between *b* and *w*. It goes without saying that you should never assume that this padding is present. If you want to use those extra bytes, then declare fields for them.

Note that specifying alignment within a record declaration does not guarantee that the field will be aligned on that boundary in memory; it only ensures that the field's offset is aligned on the specified boundary. If a variable of type *PaddedRecord* starts at an odd address in memory, then the *d* field will also start at an odd address (since any odd address plus four is an odd address). If you want to ensure that the fields are aligned on appropriate boundaries in memory, you must also use the *ALIGN* directive before variable declarations of that record type, e.g.,

```

static
    .
    .
    .
    align(4);
    PRvar: PaddedRecord;

```

The value of the `ALIGN` operand should be an even value that is evenly divisible by the largest `ALIGN` expression within the record type (four is the largest value in this case, and it's already evenly divisible by two).

If you want to ensure that the record's size is a multiple of some value, then simply stick an `ALIGN` directive as the last item in the record declaration. HLA will emit an appropriate number of bytes of padding at the end of the record to fill it in to the appropriate size. The following example demonstrates how to ensure that the record's size is a multiple of four bytes:

```

type
    PaddedRec:
        record
            << some field declarations >>

            align(4);
        endrecord;

```

5.8 Pointers to Records

During execution, your program may refer to structure objects directly or indirectly using a pointer. When you use a pointer to access fields of a structure, you must load one of the 80x86's 32-bit registers with the address of the desired record. Suppose you have the following variable declarations (assuming the *Object8* structure from "Arrays/Records as Record Fields" on page 487):

```

static
    Cube:      Object8;
    CubePtr:   pointer to Object8 := &Cube;

```

CubePtr contains the address of (i.e., it is a pointer to) the *Cube* object. To access the *Color* field of the *Cube* object, you could use an instruction like "mov(Cube.Color, eax);". When accessing a field via a pointer you need to load the address of the object into a 32-bit register such as `EBX`. The instruction "mov(CubePtr EBX);" will do the trick. After doing so, you can access fields of the *Cube* object using the `[EBX+offset]` addressing mode. The only problem is "How do you specify which field to access?" Consider briefly, the following *incorrect* code:

```

    mov( CubePtr, ebx );
    mov( [ebx].Color, eax );           // This does not work!

```

There is one major problem with the code above. Since field names are local to a structure and it's possible to reuse a field name in two or more structures, how does HLA determine which offset *Color* represents? When accessing structure members directly (e.g., "mov(Cube.Color, EAX);") there is no ambiguity since *Cube* has a specific type that the assembler can check. "[EBX]", on the other hand, can point at *anything*. In particular, it can point at any structure that contains a *Color* field. So the assembler cannot, on its own, decide which offset to use for the *Color* symbol.

HLA resolves this ambiguity by requiring that you explicitly supply a type. To do this, you must coerce "[EBX]" to type *Cube*. Once you do this, you can use the normal dot operator notation to access the *Color* field:

```

    mov( CubePtr, ebx );
    mov( (type Cube [ebx]).Color, eax );

```

By specifying the record name, HLA knows which offset value to use for the *Color* symbol.

If you have a pointer to a record and one of that record's fields is an array, the easiest way to access elements of that field is by using the base plus indexed addressing mode. To do so, you just load the pointer to the record into one register and compute the index into the array in a second register. Then you combine these two registers in the address expression. In the example above, the *Pts* field is an array of eight *point* objects. To access field *x* of the *i*th element of the *Cube.Pts* field, you'd use code like the following:

```
mov( CubePtr, ebx );
intmul( @size( point ), i, esi ); // Compute index into point array.
mov( (type Object8 [ebx]).Pts.x[ esi*4 ], eax );
```

As usual, the index appears after all the field names.

If you use a pointer to a particular record type frequently in your program, typing a coercion operator like "(type Object8 [ebx])" can get old pretty quick. One way to reduce the typing needed to coerce EBX is to use a TEXT constant. For example, consider the following statement in a program:

```
const
  O8ptr: text := "(type Object8 [ebx])";
```

With this statement at the beginning of your program you can use *O8ptr* in place of the type coercion operator and HLA will automatically substitute the appropriate text. With a text constant like the above, the former example becomes a little more readable and writable:

```
mov( CubePtr, ebx );
intmul( @size( point ), i, esi ); // Compute index into point array.
mov( O8Ptr.Pts.x[ esi*4 ], eax );
```

5.9 Unions

A record definition assigns different offsets to each field in the record according to the size of those fields. This behavior is quite similar to the allocation of memory offsets in a VAR or STATIC section. HLA provides a second type of structure declaration, the UNION, that does not assign different addresses to each object; instead, each field in a UNION declaration has the same offset – zero. The following example demonstrates the syntax for a UNION declaration:

```
type
  unionType:
    union
      << fields (syntactically identical to record declarations) >>
    endunion;
```

You access the fields of a UNION exactly the same way you access the fields of a record: using dot notation and field names. The following is a concrete example of a UNION type declaration and a variable of the UNION type:

```
type
  numeric:
    union
      i: int32;
      u: uns32;
      r: real64;
    endunion;
  .
  .
  .
static
  number: numeric;
  .
  .
```

```

    .
mov( 55, number.u );
    .
    .
    .
mov( -5, number.i );
    .
    .
    .
stdout.put( "Real value = ", number.r, nl );

```

The important thing to note about UNION objects is that all the fields of a UNION have the same offset in the structure. In the example above, the *number.u*, *number.i*, and *number.r* fields all have the same offset: zero. Therefore, the fields of a UNION overlap one another in memory; this is very similar to the way the 80x86 eight, sixteen, and thirty-two bit registers overlap one another. Usually, access to the fields of a UNION are mutually exclusive; that is, you do not manipulate separate fields of a particular UNION variable concurrently because writing to one field overwrites the other fields. In the example above, any modification of *number.u* would also change *number.i* and *number.r*.

Programmers typically use UNIONS for two different reasons: to conserve memory or to create aliases. Memory conservation is the intended use of this data structure facility. To see how this works, let's compare the *numeric* UNION above with a corresponding record type:

```

type
  numericRec:
    record
      i: int32;
      u: uns32;
      r: real64;
    endrecord;

```

If you declare a variable, say *n*, of type *numericRec*, you access the fields as *n.i*, *n.u*, and *n.r*; exactly as though you had declared the variable to be type *numeric*. The difference between the two is that *numericRec* variables allocate separate storage for each field of the record while numeric objects allocate the same storage for all fields. Therefore, *@size(numericRec)* is 16 since the record contains two double word fields and a quad word (*real64*) field. *@size(numeric)*, however, is eight. This is because all the fields of a UNION occupy the same memory locations and the size of a UNION object is the size of the largest field of that object (see Figure 5.2).

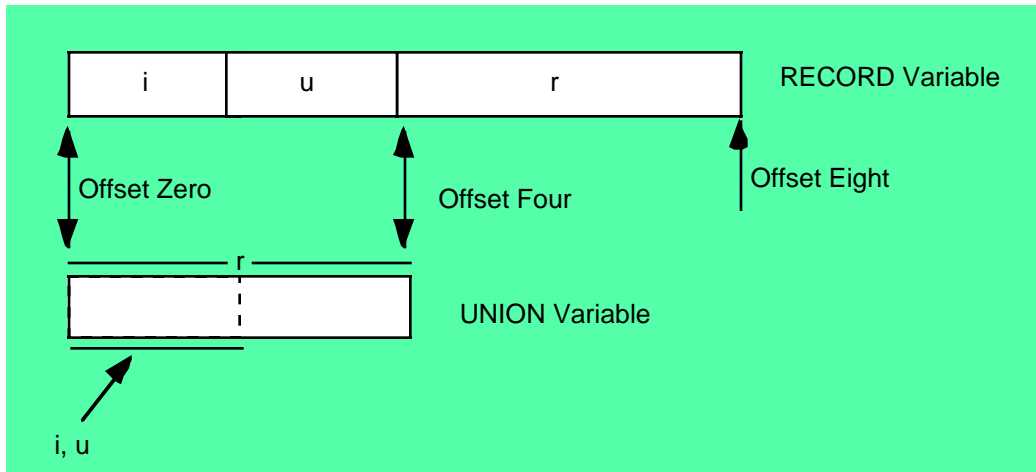


Figure 5.2 Layout of a UNION versus a RECORD Variable

In addition to conserving memory, programmers often use UNIONS to create aliases in their code. As you may recall, an alias is a different name for the same memory object. Aliases are often a source of confusion in a program so you should use them sparingly; sometimes, however, using an alias can be quite convenient. For example, in some section of your program you might need to constantly use type coercion to refer to an object using a different type. Although you can use an HLA TEXT constant to simplify this process, another way to do this is to use a UNION variable with the fields representing the different types you want to use for the object. As an example, consider the following code:

```
type
  CharOrUns:
    union
      c:char;
      u:uns32;
    endrecord;

static
  v:CharOrUns;
```

With a declaration like the above, you can manipulate an *uns32* object by accessing *v.u*. If, at some point, you need to treat the L.O. byte of this *uns32* variable as a character, you can do so by simply accessing the *v.c* variable, e.g.,

```
mov( eax, v.u );
stdout.put( "v, as a character, is '", v.c, "'\n" );
```

You can use UNIONS exactly the same way you use RECORDS in an HLA program. In particular, UNION declarations may appear as fields in RECORDS, RECORD declarations may appear as fields in UNIONS, array declarations may appear within UNIONS, you can create arrays of UNIONS, etc.

5.10 Anonymous Unions

Within a RECORD declaration you can place a UNION declaration without specifying a fieldname for the union object. The following example demonstrates the syntax for this:

```
type
  HasAnonUnion:
    record
```



```

    r:real64;
    union
        u:uns32;
        i:int32;
    endunion;
    s:string;
endrecord;

static
    v: HasAnonUnion;

```

Whenever an anonymous union appears within an RECORD you can access the fields of the UNION as though they were direct fields of the RECORD. In the example above, for example, you would access *v*'s *u* and *i* fields using the syntax “*v.u*” and “*v.i*”, respectively. The *u* and *i* fields have the same offset in the record (eight, since they follow a *real64* object). The fields of *v* have the following offsets from *v*'s base address:

```

v.r      0
v.u      8
v.i      8
v.s     12

```

@size(*v*) is 16 since the *u* and *i* fields only consume four bytes between them.

Warning: HLA gets confused if you attempt to create a record constant when that record has anonymous unions (HLA doesn't allow UNION constants). So don't create record constants of a record if that record contains anonymous unions as fields.

HLA also allows anonymous records within unions. Please see the HLA documentation for more details, though the syntax and usage is identical to anonymous unions within records.

5.11 Variant Types

One big use of UNIONS in programs is to create *variant* types. A variant variable can change its type dynamically while the program is running. A variant object can be an integer at one point in the program, switch to a string at a different part of the program, and then change to a real value at a later time. Many very high level language systems use a dynamic type system (i.e., variant objects) to reduce the overall complexity of the program; indeed, proponents of many very high level languages insist that the use of a dynamic typing system is one of the reasons you can write complex programs in so few lines. Of course, if you can create variant objects in a very high level language, you can certainly do it in assembly language. In this section we'll look at how we can use the UNION structure to create variant types.

At any one given instant during program execution a variant object has a specific type, but under program control the variable can switch to a different type. Therefore, when the program processes a variant object it must use an IF statement or SWITCH statement to execute a different sequence of instructions based on the object's current type. Very high level languages (VHLLs) do this transparently. In assembly language you will have to provide the code to test the type yourself. To achieve this, the variant type needs some additional information beyond the object's value. Specifically, the variant object needs a field that specifies the current type of the object. This field (often known as the *tag* field) is a small enumerated type or integer that specifies the type of the object at any given instant. The following code demonstrates how to create a variant type:

```

type
    VariantType:
        record
            tag:uns32; // 0-uns32, 1-int32, 2-real64
        union
            u:uns32;
            i:int32;
            r:real64;

```

```

        endunion;
    endrecord;

static
    v:VariantType;

```

The program would test the *v.tag* field to determine the current type of the *v* object. Based on this test, the program would manipulate the *v.i*, *v.u*, or *v.r* field.

Of course, when operating on variant objects, the program's code must constantly be testing the tag field and executing a separate sequence of instructions for *uns32*, *int32*, or *real64* values. If you use the variant fields often, it makes a lot of sense to write procedures to handle these operations for you (e.g., *vadd*, *vsub*, *vmul*, and *vdiv*). Better yet, you might want to make a class out of your variant types. For details on this, see the chapter on Classes appearing later in this text.

5.12 Namespaces

One really nice feature of RECORDs and UNIONs is that the field names are local to a given RECORD or UNION declaration. That is, you can reuse field names in different RECORDs or UNIONs. This is an important feature of HLA because it helps avoid *name space pollution*. Name space pollution occurs when you use up all the "good" names within the current scope and you have to start creating non-descriptive names for some object because you've already used the most appropriate name for something else. Because you can reuse names in different RECORD/UNION definitions (and you can even reuse those names outside of the RECORD/UNION definitions) you don't have to dream up new names for the objects that have less meaning. We use the term *namespace* to describe how HLA associates names with a particular object. The field names of a RECORD have a namespace that is limited to objects of that record type. HLA provides a generalization of this namespace mechanism that lets you create arbitrary namespaces. These namespace objects let you shield the names of constants, types, variables, and other objects so their names do not interfere with other declarations in your program.

An HLA NAMESPACE section encapsulates a set of generic declarations in much the same way that a RECORD encapsulates a set of variable declarations. A NAMESPACE declaration takes the following form:

```

namespace name;

    << declarations >>

end name;

```

The *name* identifier provides the name for the NAMESPACE. The identifier after the END clause must exactly match the identifier after NAMESPACE. You may have several NAMESPACE declarations within a program as long as the identifiers for the name spaces are all unique. Note that a NAMESPACE declaration section is a section unto itself. It does not have to appear in a TYPE or VAR section. A NAMESPACE may appear anywhere one of the HLA declaration sections is legal. A program may contain any number of NAMESPACE declarations; in fact, the name space identifiers don't even have to be unique as you will soon see.

The declarations that appear between the NAMESPACE and END clauses are all the standard HLA declaration sections except that you cannot nest name space declarations. You may, however, put CONST, VAL, TYPE, STATIC, READONLY, STORAGE, and VAR sections within a namespace³. The following code provides an example of a typical NAMESPACE declaration in an HLA program:

```

namespace myNames;

```

3. Procedure and macro declarations, the subjects of later chapters, are also legal within a name space declaration section.

```

type
    integer: int32;

static
    i:integer;
    j:uns32;

const
    pi:real64 := 3.14159;

end myNames;

```

To access the fields of a name space you use the same dot notation that records and unions use. For example, to access the fields of *myNames* outside of the name space you'd use the following identifiers:

```

myNames.integer - A type declaration equivalent to int32.
myNames.i - An integer variable (int32).
myNames.j - An uns32 variable.
myNames.pi - A real64 constant.

```

This example also demonstrates an important point about NAMESPACE declarations: within a name space you may reference other identifiers in that same NAMESPACE declaration without using the dot notation. For example, the *i* field above uses type *integer* from the *myNames* name space without the “mynames.” prefix.

What is not obvious from the example above is that NAMESPACE declarations create a clean symbol table whenever you open up a declaration. The only external symbols that HLA recognizes in a NAMESPACE declaration are the predefined type identifiers (e.g., *int32*, *uns32*, and *char*). HLA does not recognize any symbols you've declared outside the NAMESPACE while it is processing your namespace declaration. This creates a problem if you want to use symbols outside the NAMESPACE when declaring other symbols inside the NAMESPACE. For example, suppose the type *integer* had been defined outside *myNames* as follows:

```

type
    integer: int32;

namespace myNames;

static
    i:integer;
    j:uns32;

const
    pi:real64 := 3.14159;

end myNames;

```

If you were to attempt to compile this code, HLA would complain that the symbol *integer* is undefined. Clearly *integer* is defined in this program, but HLA hides all external symbols when creating a name space so that you can reuse (and redefine) those symbols within the name space. Of course, this doesn't help much if you actually want to use a name that you've defined outside *myNames* within that name space. HLA provides a solution to this problem: the *@global:* operator. If, within a name space declaration section, you prefix a name with “@global:” then HLA will use the global definition of that name rather than the local definition (if a local definition even exists). To correct the problem in the previous example, you'd use the following code:

```

type
    integer: int32;

```

```

namespace myNames;

    static
        i:@global:integer;
        j:uns32;

    const
        pi:real64 := 3.14159;

end myNames;

```

With the *@global:* prefix, the *i* variable will be type *int32* even if a different declaration of integer appears within the *myNames* name space.

You cannot nest NAMESPACE declarations⁴. However, you can have multiple NAMESPACE declarations in the same program that use the same name space identifier, e.g.,

```

namespace ns;

    << declaration group #1 >>

end ns;
.
.
.
namespace ns;

    << declaration group #2 >>

end ns;

```

When HLA encounters a second NAMESPACE declaration for a given identifier, it simply appends the declarations in the second group to the end of the symbol list it created for the first group. Therefore, after processing the two NAMESPACE declarations, the *ns* name space would contain the set of all symbols you've declared in both the name space blocks.

Perhaps the most common use of name spaces is in library modules. If you create a set of library routines to use in various projects or distribute to others, you have to be careful about the names you choose for your functions and other objects. If you use common names like *get* and *put*, the users of your module will complain when your names collide with theirs. An easily solution is to put all your code in a NAMESPACE block. Then the only name you have to worry about is the name of the NAMESPACE itself. This is the only name that will collide with other users' code. That can happen, but it's much less likely to happen than if you don't use a name space and your library module introduces dozens, if not hundreds, of new names into the global name space⁵. The HLA Standard Library provides many good examples of name spaces in use. The HLA Standard Library defines several name spaces like *stdout*, *stdin*, *str*, *cs*, and *chars*. You refer to functions in these name spaces using names like *stdout.put*, *stdin.get*, *cs.intersection*, *str.eq*, and *chars.toUpper*. The use of name spaces in the HLA Standard Library prevents conflicts with similar names in your own programs.

5.13 Putting It All Together

One of the more amazing facts about programmer psychology is the fact that a high level language programmer would refuse to use a high level language that doesn't support records or structures; then that same programmer won't bother to learn how to use them in assembly language (all the time, grumbling about their

4. There really doesn't seem to be a need to do this; hence its omission from HLA.

5. The global name space is the global section of your program.

absence). You use records in assembly language for the same reason you use them in high level languages. Given that most programmers consider records and structure essential in high level languages, it is surprising they aren't as concerned about using them in assembly language.

This short chapter demonstrates that it doesn't take much effort to master the concept of records in an assembly language program. Taken together with UNIONS and NAMESPACES, RECORDS can help you write HLA programs that are far more readable and easier to understand. Therefore, you should use these language features as appropriate when writing assembly code.

Dates and Times

Chapter Six

6.1 Chapter Overview

This chapter discusses dates and times as a data type. In particular, this chapter discusses the data/time data structures the HLA Standard Library defines and it also discusses date arithmetic and other operations on dates and times.

6.2 Dates

For the first 50 years, or so, of the computer's existence, programmers did not give much thought to date calculations. They either used a date/time package provided with their programming language, or they kludged together their own date processing libraries. It wasn't until the Y2K¹ problem came along that programmers began to give dates serious consideration in their programs. The purpose of this chapter is two-fold. First, this chapter teaches that date manipulation is not as trivial as most people would like to believe – it takes a lot of work to properly compute various date functions. Second, this chapter presents the HLA date and time formats found in the “datetime.hhf” library module. Hopefully this chapter will convince you that considerable thought has gone into the HLA datetime.hhf module so you'll be inclined to use it rather than trying to create your own date/time formats and routines.

Although date and time calculations may seem like they should be trivial, they are, in fact, quite complex. Just remember the Y2K problem to get a good idea of the kinds of problems your programs may create if they don't calculate date and time values correctly. Fortunately, you don't have to deal with the complexities of date and time calculations, the HLA Standard Library does the hard stuff for you.

The HLA Standard Library date routines produce valid results for dates between January 1, 1583 and December 31, 9999². HLA represents dates using the following record definition (in the *date* namespace):

```
type
  daterec:
    record
      day:uns8;
      month:uns8;
      year:uns16;
    endrecord;
```

This format (*date.daterec*) compactly represents all legal dates using only four bytes. Note that this is the same date format that the chapter on Data Representation presents for the extended data format (see “Bit Fields and Packed Data” on page 81). You should use the *date.daterec* data type when declaring date objects in your HLA programs, e.g.,

```
static
  TodaysDate: date.daterec;
  Century21: date.daterec := date.daterec:[ 1, 1, 2001 ]; // note: d, m ,y
```

As the second example above demonstrates, the first field is the day field and the second field is the month field if you use a *date.daterec* constant to initialize a static *date.daterec* object. Don't fall into the trap of using the mm/dd/yy or yy/mm/dd organization common in most countries.

1. For those who missed it, the Y2K (or Year 2000) problem occurred when programmers used two digits for the date and assumed that the H.O. two digits were “19”. Clearly this code malfunctioned when the year 2000 came along.

2. The Gregorian Calendar came into existence in Oct, 1582, so any dates earlier than this are meaningless as far as date calculations are concerned. The last legal date, 9999, was chosen arbitrarily as a trap for wild dates entering the calculation. This means, of course, that code calling the HLA Standard Library Date/Time package will suffer from the Y10K problem. However, you'll probably not consider this a severe limitation!

The HLA *date.daterec* format has a couple of advantages. First, it is a trivial matter to convert between the internal and external representations of a date. All you have to do is extract the *d*, *m*, and *y* fields and manipulate them as integers of the appropriate sizes. Second, this format makes it very easy to compare two dates to see if one date follows another in time; all you've got to do is compare the *date.daterec* object as though it were a 32-bit unsigned integer and you'll get the correct result. The Standard Library *date.daterec* format does have a few disadvantages. Specifically, certain calculations like computing the number of days between two dates is a bit difficult. Fortunately, the HLA Standard Library Date module provides most of the functions you'll ever need for date calculations, so this won't prove to be much of a disadvantage.

A second disadvantage to the *date.daterec* format is that the resolution is only one day. Some calculations need to maintain the time of day (down to some fraction of a second) as well as the calendar date. The HLA Standard Library also provides a TIME data structure. By combining these two structures together you should be able handle any problem that comes along.

Before going on and discussing the functions available in the HLA Standard Library's Date module, it's probably worthwhile to briefly discuss some other date formats that find common use. Perhaps the most common date format is to use an integer value that specifies the number of days since an *epoch*, or starting, date. The advantage to this scheme is that it's very easy to do certain kinds of date arithmetic (e.g., to compute the number of days between two dates you simply subtract them) and it's also very easy to compare these dates. The disadvantages to this scheme include the fact that it is difficult to convert between the internal representation and an external representation like "xx/yy/zzzz." Another problem with this scheme, which it shares with the HLA scheme, is that the granularity is one day. You cannot represent time with any more precision than one day.

Another popular format combines dates and times into the same value. For example, the representation of time on most UNIX systems measures the number of seconds that have passed since Jan 1, 1970. Unfortunately, many UNIX systems only use a 32-bit signed integer; therefore, those UNIX systems will experience their own "Y2.038K" problem in the year 2038 when these signed integers roll over from 2,147,483,637 seconds to -2,147,483,638 seconds. Although this format does maintain time down to seconds, it does not handle fractions of a second very well. Most UNIX system include an extra field in their date/time format to handle milliseconds, but this extra field is a kludge. One could just as easily add a time field to an existing date format if you're willing to kludge.

For those who want to be able to accurately measure dates and times, a good solution is to use a 64-bit unsigned integer to count the number of microseconds since some epoch date. A 64-bit unsigned integer will provide microsecond accuracy for a little better than 278,000 years. Probably sufficient for most needs. If you need better than microsecond accuracy, you can get nanosecond accuracy that is good for about 275 years (beyond the epoch date) with a 64-bit integer. Of course, if you want to use such a date/time format, you will have to write the routines that manipulate such dates yourself; the HLA Standard Library's Date/Time module doesn't use that format.

6.3 A Brief History of the Calendar

Man has been interested in keeping track of time since the time man became interested in keeping track of history. To understand why we need to perform various calculations, you'll need to know a little bit about the history of the calendar. So this section will digress a bit from computers and discuss that history.

What exactly is time? Time is a concept that we are all intuitively familiar with, but try and state a concrete definition that does not define time in terms of itself. Before you run off and grab a dictionary, you should note that many of the definitions of time in a typical dictionary contain a circular reference (that is, they define time in terms of itself). The *American Heritage Dictionary of the English Language* provides the following definition:

A nonspatial continuum in which events occur in apparently irreversible succession from the past through the present to the future.

As horrible as this definition sounds, it is one of the few that doesn't define time by how we measure it or by a sequence of observable events.

Why are we so obsessed with keeping track of time? This question is much more easily answered. We need to keep track of time so we can predict certain future events. Historically, important events the human race has needed to predict include the arrival of spring (for planting), the observance of religious anniversaries (e.g., Christmas, Passover), or the gestation period for livestock (or even humans). Of course, modern life may seem much more complex and tracking time more important, but we track time for the same reasons the human race always has, to predict the future. Today, we predict business meetings, when a department store will open to the public, the start of a college lecture, periods of high traffic on the highways, and the start of our favorite television shows by using time. The better we are able to measure time, the better we will be able to predict when certain types of events will occur (e.g., the start of spring so we can begin planting).

To measure time, we need some predictable, periodic, event. Since ancient times, there have been three celestial events that suit this purpose: the solar *day*, the lunar *month*, and the solar *year*. The solar day (or *tropical* day) consists of one complete rotation of the Earth on its axis. The lunar month consists of one complete set of moon phases. The solar year is one complete orbit of the Earth around the Sun. Since these periodic events are easy to measure (crudely, at least), they have become the primary basis by which we measure time.

Since these three celestial events were obvious even in prehistoric times, it should come as no surprise that one society would base their measurement of time on one cyclic standard such as the lunar month while another social group would base their time unit on a different cycle such as the solar year. Clearly, such fundamentally different time keeping schemes would complicate business transactions between the two societies effectively erecting an artificial barrier between them. Nevertheless, until about the year 46 BC (by our modern calendar), most countries used their own system for time keeping.

One major problem with reconciling the different calendars is that the celestial cycles are not integral. That is, there are not an even number of solar days in a lunar month, there are not an integral number of solar days in a solar year, and there are not an integral number of lunar months in a solar year. Indeed, there are approximately 365.2422 days in a solar year and approximately 29.5 days in a lunar month. Twelve lunar months are 354 days, a little over a week short of a full year. Therefore, it is very difficult to reconcile these three periodic events if you want to use two of them or all three of them in your calendar.

In 46 BC (or BCE, for *Before Common Era*, as it is more modernly written) Julius Caesar introduced the calendar upon which our modern calendar is based. He decreed that each year would be exactly $365 \frac{1}{4}$ days long by having three successive years having 365 days each and every fourth year having 366 days. He also abolished reliance upon the lunar cycle from the calendar. However, $365 \frac{1}{4}$ is just a little bit more than 365.2422, so Julius Caesar's calendar lost a day every 128 years or so.

Around 700 AD (or CE, for *Common Era*, as it is more modernly written) it was common to use the birth of Jesus Christ as the *Epoch* year. Unfortunately, the equinox kept losing a full day every 128 years and by the year 1500 the equinoxes occurred on March 12th, and September 12th. This was of increasing concern to the Church since it was using the Calendar to predict Easter, the most important Christian holiday³. In 1582 CE, Pope Gregory XIII dropped ten days from the Calendar so that the equinoxes would fall on March 21st and September 21st, as before, and as advised by Christoph Clavius, he dropped three leap years every 400 years. From that point forward, century years were leap years only if divisible by 400. Hence 1700, 1800, 1900 are *not* leap years, but 2000 *is* a leap year. This new calendar is known as the Gregorian Calendar (named after Pope Gregory XIII) and with the exception of the change from BC/AD to BCE/CE is, essentially, the calendar in common use today⁴.

The Gregorian Calendar wasn't accepted universally until well into the twentieth century. Largely Roman Catholic countries (e.g., Spain and France) adopted the Gregorian Calendar the same year as Rome. Other countries followed later. For example, portions of Germany did not adopt the Gregorian Calendar until the year 1700 AD while England held out until 1750. For this reason, many of the American founding fathers have *two* birthdates listed. The first date is the date in force at the time of their birth, the second date

3. Easter is especially important since the Church computed all other holidays relative to Easter. If the date of Easter was off, then all holidays would be off.

4. One can appreciate that non-Christian cultures might be offended at by the abbreviations BC (Before Christ) and AD (Anno Domini [day of our Lord]).

is their birthdate using the Gregorian Calendar. For example, George Washington was actually born on February 11th by the English Calendar, but after England adopted the Gregorian Calendar, this date changed to February 22nd. Note that George Washington's birthday didn't actually change, only the calendar used to measure dates at the time changed.

The Gregorian Calendar still isn't correct, though the error is very small. After approximately 3323 years it will be off by a day. Although there has been some proposals thrown around to adjust for this in the year 4000, that is such a long time off that it's hardly worth contemporary concern (with any luck, mankind will be a spacefaring race by then and the concept of a year, month, or day, may be a quaint anachronism).

There is one final problem with the calendar- the length of the solar day is constantly changing. Ocean tidal forces, meteors burning up in our atmosphere, and other effects are slowing down the Earth's rotation resulting in longer days. The effect is small, but compared to the length of a day, but it amounts to a loss of one to three milliseconds (that is, about 1/500th of a second) every 100 years since the defining Epoch (Jan 1, 1900). That means that Jan 1, 2000 is about two seconds longer than Jan 1, 1900. Since there are 86,400 seconds in a day, it will probably take on the order of 100,000 years before we lose a day due to the Earth's rotation slowing down. However, those who want to measure especially small time intervals have a problem: hours and seconds have been defined as submultiples of a single day. If the length of a day is constantly changing, that means that the definition of a second is constantly changing as well. In other words, two very precise measurements of equivalent events taken 10 years apart may show measurable differences.

To solve this problem scientists have developed the *Cesium-133 Atomic Clock*, the most accurate timing device ever invented. The Cesium atom, under special conditions, vibrates at exactly 9,192,631,770 cycles per second, for the year 1900. Because the clock is so accurate, it has to be adjusted periodically (about every 500 days, currently) so that its time (known as Universal Coordinated Time or UTC) matches that of the Earth (UT1). A high-quality Cesium Clock (like the one at the National Institute of Standards and Technology in Boulder, Colorado, USA) is very large (about the size of a large truck) and can keep accurate time to about one second in a million and a half years. Commercial units (about the size of a large suitcase) are available and they keep time accurate to about one second every 5-10,000 years.

The wall calendar you purchase each year is a device that is very similar to the Cesium Atomic Clock- it lets you measure time. The Cesium clock, clearly, lets time two discrete events that are very close to one another, but either device will probably let you predict that you start two week's vacation in Mexico starting next Monday (and the wall calendar does it for a whole lot less money). Most people don't think of a calendar as a time keeping device, but the only difference between it and a watch is the *granularity*, that is, the finest amount of time one can measure with the device. With a typical electronic watch, you can probably measure (accurately) to as little as 1/100 seconds. With a calendar, the minimum interval you can measure is one day. While the watch is appropriate for measuring the 100 meter dash, it is inappropriate for measuring the duration of the Second World War; the calendar, however, is perfect for this latter task.

Time measurement devices, be they a Cesium Clock, a wristwatch, or a Calendar, do not measure time in an absolute sense. Instead, these devices measure time between two events. For the Gregorian Calendar, the (intended) Epoch event that marks year one was the birth of Christ. Unfortunately in 1582, the use of negative numbers was not widespread and even the use of zero was not common. Therefore, 1 AD was (supposed to be) the first year of Christ's life. The year prior to that point was considered 1BC. This unfortunate choice created some mathematical problems that tend to bother people 2,000 years later. For example, the first decade was the first 10 years of Christ's life, that is, 1 AD through 10 AD. Likewise, the first century was considered the first 100 years after Christ's birth, that is, 1 AD through 100 AD. Likewise, the first millennium was the first 1,000 years after Christ's birth, specifically 1 AD through 1000 AD. Similarly, the second millennium is the next 1,000 years, specifically 1001 AD through 2000 AD. The third, millennium, contrary to popular belief, began on January 1, 2001 (Hence the title of Clark's book: "2001: A Space Odyssey"). It is an unfortunately accident of human psychology that people attach special significance to round numbers; there were many people mistakenly celebrating the turn of the millennium on December 31st, 1999 when, in fact, the actual date was still a year away.

Now you're probably wondering what this has to do with computers and the representation of dates in the computer... The reason for taking a close look at the history of the Calendar is so that you don't misuse the date and time representations found in the HLA Standard Library. In particular, note that the HLA date format is based on the Gregorian Calendar. Since the Gregorian Calendar was "born" in October of 1582, it

makes absolutely no sense to represent any date earlier than about Jan 1, 1583 using the HLA date format. Granted, the data type can represent earlier dates numerically, but any date computations would be severely off if one or both of the dates in the computation are pre-1583 (remember, Pope Gregory dropped 10 days from the calendar; right off the bat your “days between two dates” computation would be off by 10 real days if the two dates crossed the date that Rome adopted the Gregorian Calendar).

In fact, you should be wary of any dates prior to about January 1, 1800. Prior to this point there were a couple of different (though similar) calendars in use in various countries. Unless you’re a historian and have the appropriate tables to convert between these dates, you should not use dates prior to this point in calculations. Fortunately, by the year 1800, most countries that had a calendar based on Juilus Caesar’s calendar fell into line and adopted the Gregorian Calendar. Some other calendars (most notably, the Chinese Calendar) were in common use into the middle of the 20th century. However, it is unlikely you would ever confuse a Chinese date with a Gregorian date.

6.4 HLA Date Functions

HLA provides a wide array of date functions you can use to manipulate date objects. The following subsections describe many of these functions and how you use them.

6.4.1 `date.IsValid` and `date.validate`

When storing data directly into the fields of a `date.daterec` object, you must be careful to ensure that the resulting date is correct. The HLA date procedures will raise an `ex.InvalidDate` exception if the date values are out of range. The `date.IsValid` and `date.validate` procedures provide some handy code to check the validity of a date object. These two routines use either of the following calling sequences:

```
date.IsValid( dateVar ); // dateVar is type date.daterec
date.IsValid( m, d, y ); // m, d, y are uns8, uns8, uns16, respectively

date.validate( dateVar ); // See comments above.
date.validate( m, d, y );
```

The `date.IsValid` procedure checks the date to see if it is a valid date. This procedure returns true or false in the AL register to indicate whether the date is valid. The `date.validate` procedure also checks the validity of the date; however, it raises the `ex.InvalidDate` exception if the date is invalid. The following sample program demonstrates the use of these two routines:

```
program DateTimeDemo;
#include( "stdlib.hhf" );

static
    m:          uns8;
    d:          uns8;
    y:          uns16;

    theDate:    date.daterec;

begin DateTimeDemo;

    try
```

```
stdout.put( "Enter the month (1-12):" );
stdin.get( m );

stdin.flushInput();
stdout.put( "Enter the day (1-31):" );
stdin.get( d );

stdin.flushInput();
stdout.put( "Enter the year (1583-9999): " );
stdin.get( y );

if( date.isValid( m, d, y )) then

    stdout.put( m, "/", d, "/", y, " is a valid date." nl );

endif;

// Assign the fields to a date variable.

mov( m, al );
mov( al, theDate.month );
mov( d, al );
mov( al, theDate.day );
mov( y, ax );
mov( ax, theDate.year );

// Force an exception if the date is illegal.

date.validate( theDate );

exception( ex.ConversionError )

stdout.put
(
    "One of the input values contained illegal characters" nl
);

exception( ex.ValueOutOfRange )

stdout.put
(
    "One of the input values was too large" nl
);

exception( ex.InvalidDate )

stdout.put
(
    "The input date (", m, "/", d, "/", y, ") was invalid" nl
);

endtry;

end DateTimeDemo;
```

Program 6.1 Date Validation Example

6.4.2 Checking for Leap Years

Determining whether a given year is a leap year is somewhat complex. The exact algorithm is “any year that is evenly divisible by four and is not evenly divisible by 100 or is evenly divisible by 400 is a leap year⁵.” The HLA “datetime.hhf” module provides a convenient function, *date.IsLeapYear*, that efficiently determines whether a given year is a leap year. There are two different ways you can call this function; either of the following will work:

```
date.IsLeapYear( dateVar );    // dateVar is a date.dateRec variable.
date.IsLeapYear( y );         // y is a word value.
```

The following code demonstrates the use of this routine.

```
program DemoIsLeapYear;
#include( "stdlib.hhf" );

static
    m:          uns8;
    d:          uns8;
    y:          uns16;

    theDate:    date.daterec;

begin DemoIsLeapYear;

    try

        stdout.put( "Enter the month (1-12):" );
        stdin.get( m );

        stdin.flushInput();
        stdout.put( "Enter the day (1-31):" );
        stdin.get( d );

        stdin.flushInput();
        stdout.put( "Enter the year (1583-9999): " );
        stdin.get( y );

        // Assign the fields to a date variable.

        mov( m, al );
        mov( al, theDate.month );
        mov( d, al );
        mov( al, theDate.day );
        mov( y, ax );
        mov( ax, theDate.year );

        // Force an exception if the date is illegal.

        date.validate( theDate );
```

5. The Gregorian Calendar does not account for the fact that sometime between the years 3,000 and 4,000 we will have to add an extra leap day to keep the Calendar in sync with the Earth’s rotation around the Sun. The HLA *date.IsLeapYear* does not handle this situation either. Keep this in mind if you are doing date calculations that involve dates after the year 3,000. This is a defect in the current definition of the Gregorian Calendar, which HLA’s routines faithfully reproduce.

```
// Okay, report whether this is a leap year:

if( date.isLeapYear( theDate ) ) then

    stdout.put( "The year ", y, " is a leap year." nl );

else

    stdout.put( "The year ", y, " is not a leap year." nl );

endif;

// Technically, the leap day is Feb 29, but most people don't
// realize this, so use the following output to keep them happy:

if( date.isLeapYear( y ) ) then

    if( m = 2 ) then

        if( d = 29 ) then

            stdout.put( m, "/", d, "/", y, " is the leap day." nl );

        endif;

    endif;

endif;

exception( ex.ConversionError )

    stdout.put
    (
        "One of the input values contained illegal characters" nl
    );

exception( ex.ValueOutOfRange )

    stdout.put
    (
        "One of the input values was too large" nl
    );

exception( ex.InvalidDate )

    stdout.put
    (
        "The input date (", m, "/", d, "/", y, ") was invalid" nl
    );

endtry;

end DemoIsLeapYear;
```

Program 6.2 Calling the date.IsLeapYear Function

6.4.3 Obtaining the System Date

The *date.today* function returns the current system date in the *date.daterec* variable you pass as a parameter⁶. The following program demonstrates how to call this routine:

```

program DemoToday;
#include( "stdlib.hhf" );

static
    TodaysDate: date.daterec;

begin DemoToday;

    date.today( TodaysDate );

    stdout.put
    (
        "Today is ",
        (type uns8 TodaysDate.month), "/",
        (type uns8 TodaysDate.day), "/",
        (type uns16 TodaysDate.year),
        nl
    );

    // Okay, report whether this is a leap year:

    if( date.isLeapYear( TodaysDate ) ) then

        stdout.put( "This is a leap year." nl );

    else

        stdout.put( "This is not a leap year." nl );

    endif;

end DemoToday;

```

Program 6.3 Reading the System Date

Linux users should be aware that *date.today* returns the current date based on Universal Coordinated Time (UTC). Depending upon your time zone, *date.today* may return yesterday's or tomorrow's date within your particular timezone.

6. This function was not available in the Linux version of the HLA Standard Library as this was written. It may have been added by the time you read this, however.

6.4.4 Date to String Conversions and Date Output

The HLA date module provides a set of routines that will convert a *date.dateRec* object to the string representation of that date. HLA provides a mechanism that lets you select from one of several different conversion formats when translating dates to strings. The date package defines an enumerated data type, *date.OutputFormat*, that specifies the different conversion mechanisms. The possible conversions are (these examples assume you are converting the date January 2, 2033):

```

date.mdy      - Outputs date as 1/2/33.
date.mdyyyy   - Outputs date as 1/2/2033.
date.mmddyy   - Outputs date as 01/02/33.
date.mmddyyyy - Outputs date as 01/02/2033.
date.yynd     - Outputs date as 33/1/2.
date.yyynd    - Outputs date as 2033/1/2.
date.yymmdd   - Outputs date as 33/01/02.
date.yyyymmdd - Outputs date as 2033/01/02.
date.MONdyyyy - Outputs date as Jan 1, 2033.
date.MONTHdyyyy - Outputs date as January 1, 2033.

```

To set the conversion format, you must call the *date.SetFormat* procedure and pass one of the above values as the single parameter⁷. For all but the last two formats above, the default month/day/year separator is the slash (“/”) character. You can call the *date.SetSeparator* procedure, passing it a single character parameter, to change the separator character.

The *date.toString* and *date.a_toString* procedures convert a date to string data. Like the other string routines this chapter discusses, the difference between the *date.toString* and *date.a_toString* procedures is that *date.a_toString* automatically allocates storage for the string whereas you must supply a string with sufficient storage to the *date.toString* procedure. Note that a string containing 20 characters is sufficient for all the different date formats. The *date.toString* and *date.a_toString* procedures use the following calling sequences:

```

date.toString( m, d, y, s );
date.toString( dateVar, s );

date.a_toString( m, d, y );
date.a_toString( dateVar );

```

Note that *m* and *d* are byte values, *y* is a word value, *dateVar* is a *date.dateRec* value, and *s* is a string variable that must point at a string that holds at least 20 characters.

The *date.Print* procedure uses the *date.toString* function to print a date to the standard output device. This is a convenient function to use to display a date after some date calculation.

The following program demonstrates the use of the procedures this section discusses:

```

program DemoStrConv;
#include( "stdlib.hhf" );

static
    TodaysDate: date.dateRec;
    s:         string;

begin DemoStrConv;

```

⁷ the *date.SetFormat* routine raises the *ex.InvalidDateFormat* exception if the parameter is not one of these values.


```

date.today( TodaysDate );
stdout.put( "Today's date is " );
date.print( TodaysDate );
stdout.newln();

// Convert the date using various formats
// and display the results:

date.setFormat( date.mdyy );
date.a_toString( TodaysDate );
mov( eax, s );
stdout.put( "Date in mdyy format: `", s, "`" nl );
strfree( s );

date.setFormat( date.mmddyy );
date.a_toString( TodaysDate );
mov( eax, s );
stdout.put( "Date in mmddyy format: `", s, "`" nl );
strfree( s );

date.setFormat( date.mdyyyy );
date.a_toString( TodaysDate );
mov( eax, s );
stdout.put( "Date in mdyyyy format: `", s, "`" nl );
strfree( s );

date.setFormat( date.mmddyyyy );
date.a_toString( TodaysDate );
mov( eax, s );
stdout.put( "Date in mmddyyyy format: `", s, "`" nl );
strfree( s );

date.setFormat( date.MONdyyyy );
date.a_toString( TodaysDate );
mov( eax, s );
stdout.put( "Date in MONdyyyy format: `", s, "`" nl );
strfree( s );

date.setFormat( date.MONTHdyyyy );
date.a_toString( TodaysDate );
mov( eax, s );
stdout.put( "Date in MONTHdyyyy format: `", s, "`" nl );
strfree( s );

end DemoStrConv;

```

Program 6.4 Date <-> String Conversion and Date Output Routines

6.4.5 date.unpack and data.pack

The *date.pack* and *date.unpack* functions pack and unpack date data. The calling syntax for these functions is the following:

```
date.pack( y, m, d, dr );
```

```
date.unpack( dr, y, m, d );
```

Note: *y*, *m*, *d* must be *uns32* or *dword* variables; *dr* must be a *date.daterec* object.

The *date.pack* function takes the *y*, *m*, and *d* values and packs them into a *date.daterec* format and stores the result into *dr*. The *date.unpack* function does just the opposite. Neither of these routines check their parameters for proper range. It is the caller's responsibility to ensure that *d*'s value is in the range 1..31 (as appropriate for the month and year), *m*'s value is in the range 1..12, and *y*'s value is in the range 1583..9999.

6.4.6 date.Julian, date.fromJulian

These two functions convert a Gregorian date to and from a Julian day number⁸. Julian day numbers specify January 1, 4713 BCE as day zero and number the days consecutively from that point⁹. One nice thing about Julian day numbers is that date calculations are very easy. You can compute the number of days between two dates by simply subtracting them, you can compute new dates by adding an integer number of days to a Julian day number, etc. The biggest problem with Julian day numbers is converting them to and from the Gregorian Calendar with which we're familiar. Fortunately, these two functions handle that chore. The syntax for calling these two functions is:

```
date.fromJulian( julian, dateRecVar );
date.Julian( m, d, y );
date.Julian( dateRecVar );
```

The first call above converts the Julian day number that you pass in the first parameter to a Gregorian date and stores the result into the *date.daterec* variable you pass as the second parameter. Keep in mind that Julian day numbers that correspond to dates before Jan 1, 1582, will not produce accurate calendar dates since the Gregorian calendar did not exist prior to that point.

The second two calls above compute the Julian day number and return the value in the EAX register. They differ only in the types of parameters they expect. The first call to *date.Julian* above expects three parameters, *m* and *b* being byte values and *y* being a word value. The second call expects a *date.daterec* parameter; it extracts those three fields and converts them to the Julian day number.

6.4.7 date.datePlusDays, date.datePlusMonths, and date.daysBetween

These two functions provide some simple date arithmetic operations. They compute a new date by adding some number of days or months to an existing date. The calling syntax for these functions is

```
date.datePlusDays( numDays, dateRecVar );
date.datePlusMonths( numMonths, dateRecVar );
```

Note: *numDays* and *numMonths* are *uns32* values, *dateRecVar* must be a *date.daterec* variable.

The *date.datePlusDays* function computes a new date that is *numDays* days beyond the date that *dateRecVar* specifies. This function leaves the resulting date in *dateRecVar*. This function automatically compensates for the differing number of days in each month as well as the differing number of days in leap years. The *date.datePlusMonths* function does a similar calculation except it adds *numMonths* months, rather than days to *dateRecVar*.

The *date.datePlusDays* function is not particularly efficient if the *numDays* parameter is large. There is a more efficient way to calculate a new date if *numDays* exceeds 1,000: convert the date to a Julian Day Number, add the *numDays* value directly to the Julian Number, and then convert the result back to a date.

8. Note that a Julian date and a Julian day number are not the same thing. Julian dates are based on the Julian Calendar, commissioned by Julius Caesar, which is very similar to the Gregorian Calendar; Julian day numbers were invented in the 1800's and are primarily used by astronomers.

9. Jan 1, 4713 BCE was chosen as a date that predates recorded history.

The *date.daysBetween* function computes the number of days between two dates. Like *date.datePlus-Days*, this function is not particularly efficient if the two dates are more than about three years apart; it is more efficient to compute the Julian day numbers of the two dates and subtract those values. For spans of less than three years, this function is probably more efficient. The calling sequence for this function is the following:

```
date.daysBetween( m1, d1, y1, m2, d2, y2 );
date.daysBetween( m1, d1, y1, dateRecVar2 );
date.daysBetween( dateRecVar1, m2, d2, y2 );
date.daysBetween( dateRecVar1, dateRecVar2 );
```

The four different calls allow you to specify either date as a m/d/y value or as a *date.daterec* value. The *m* and *d* parameters in these calls must be byte values and the *y* parameter must be a word value. The *dateRecVar1* and *dateRecVar2* parameters must, obviously, be *date.daterec* values. These functions return the number of days between the two dates in the EAX register. Note that the dates must be valid, but there is no requirement that the first date be less than the second date.

6.4.8 date.dayNumber, date.daysLeft, and date.dayOfWeek

The *date.dayNumber* function computes the day number into the current year (with Jan 1 being day number one) and returns this value in EAX. This value is always in the range 1..365 (or 1..366 for leap years). A call to this function uses the following syntax:

```
date.dayNumber( m, d, y );
date.dayNumber( dateRecVar );
```

The two forms differ only in the way you pass the date. The first call above expects two byte values (*m* and *d*) and a word value (*y*). The second form above expects a *date.daterec* value.

The *date.daysLeft* function computes the number of days left in a year. This function returns the number of days left in a year *counting the date you pass as a parameter*. Therefore, this function returns one for Dec 31st. Like *date.dayNumber*, this function always returns a value in the range 1..365/366 (regular/leap year). The calling syntax for this function is similar to *date.dayNumber*, it is

```
date.daysLeft( m, d, y );
date.daysLeft( dateRecVar );
```

The parameters have the same meaning as for *date.dayNumber*.

The *date.dayOfWeek* function accepts a date and returns a day of week value in the EAX register. A call to this function uses the following syntax:

```
date.dayOfWeek( m, d, y );
date.dayOfWeek( dateRecVar );
```

The parameters have their usual meanings.

These function calls return a value in the range 0..7 (in EAX) as follows:

- 0: Sunday
- 1: Monday
- 2: Tuesday
- 3: Wednesday
- 4: Thursday
- 5: Friday
- 6: Saturday

6.5 Times

The HLA Standard Library provides a simple time module that lets you manipulate times in an HHMMSS (hours/minutes/seconds) format. The *time* namespace in the date/time module defines the time data type as follows:

```
type
  timerec:
    record
      secs:uns8;
      mins:uns8;
      hours:uns16;
    endrecord;
```

This format easily handles 60 seconds per minute and 60 minutes per hour. It also handles up to 65,535 hours (just over 2730 days or about 7- $\frac{1}{2}$ years).

The advantages to this time format parallel the advantages of the date format: it is easy to convert the time format to/from external representation (i.e., HH:MM:SS) and the storage format lets you compare times by treating them as *uns32* objects. Another advantage to this format is that it supports more than 24 hours, so you can use it to maintain timings for events that are not calendar based (up to seven years).

There are a couple of disadvantages to this format. The primary disadvantage is that the minimum granularity is one second; if you want to work with fractions of a second then you will need to use a different format and you will have to write the functions for that format. Another disadvantage is that time calculations are somewhat inconvenient. It is difficult to add *n* seconds to a time variable.

Before discussing the HLA Standard Library Time functions, a quick discussion of other possible time formats is probably wise. The only reasonable alternative to the HH:MM:SS format that the HLA Standard Library Time module uses is to use an integer value to represent some number of time units. The only question is “what time units do you want to use?” Whatever time format you use, you should be able to represent at least 86,400 seconds (24 hours) with the format. Furthermore, the granularity should be one second or less. This effectively means you will need at least 32 bits since 16 bits only provides for 65,536 seconds at one second granularity (okay, 24 bits would work, but it’s much easier to work with four-byte objects than three-byte objects).

With 32-bits, we can easily represent more than 24 hours’ worth of milliseconds (in fact, you can represent almost 50 days before the format rolls over). We could represent five days with a $\frac{1}{10,000}$ second granularity, but this is not a common timing to use (most people want microseconds if they need better than millisecond granularity), so millisecond granularity is probably the best choice for a 32-bit format. If you need better than millisecond granularity, you should use a combined date/time 64-bit format that measures microseconds since Julian Day Number zero (Jan 1, 4713 BCE). That’s good for about a half million years. If you need finer granularity than microseconds, well, you’re own your own! You’ll have to carefully weigh the issues of granularity vs. years covered vs. the size of your data.

6.5.1 time.curTime

This function returns the current time as read from the system’s time of day clock. The calling syntax for this function is the following:

```
time.curTime( timeRecVar );
```

This function call stores the current system time in the *time.timerec* variable you pass as a parameter. On Windows systems, the current time is the wall clock time for your particular time zone; under Linux, the current time is always given in UTC (Universal Coordinated Time) and you must adjust according to your particular time zone to get the local time. Keep this difference in mind when porting programs between Windows and Linux.

6.5.2 `time.hmsToSecs` and `time.secstoHMS`

These two functions convert between the HLA internal time format and a pure seconds format. Generally, when doing time arithmetic (e.g., time plus seconds, minutes, or hours), it's easiest to convert your times to seconds, do the calculations with seconds, and then translate the time back to the HLA internal format. This lets you avoid the headaches of modulo-60 arithmetic.

The calling sequences for the `time.hmsToSecs` function are

```
time.hmsToSecs( timeRecValue );
time.hmsToSecs( h, m, s );
```

Both functions return the number of seconds in the EAX register. They differ only in the type of parameters they expect. The first form above expects an HLA `time.timerec` value. The second call above lets you directly specify the hours, minutes, and seconds as separate parameters. The `h` parameter must be a word value, the `m` and `s` parameters must be byte values.

The `time.secsToHMS` function uses the following calling sequence:

```
time.secsToHMS( seconds, timeRecVar );
```

The first parameter must be an `uns32` value specifying some number of seconds less than 235,939,600 seconds (which corresponds to 65,536 hours). The second parameter in this call must be a `time.timerec` variable. This function converts the seconds parameter to the HLA internal time format and stores the value into the `timeRecVar` variable.

6.5.3 Time Input/Output

The HLA Standard Library doesn't provide any specific I/O routines for time data. However, reading and writing time data in ASCII form is a fairly trivial process. This section will provide some examples of time I/O using the HLA Standard Input and Standard Output modules.

To output time in a standard HH:MM:SS format, just use the `stdout.putisize` routines with a width value of two and a fill character of '0' for the three fields of the HLA `time.timerec` data type. The following code demonstrates this:

```
static
    t:time.timerec;
    .
    .
    .
    stdout.putisize( t.h, 2, '0' );
    stdout.put( ':' );
    stdout.putisize( t.m, 2, '0' );
    stdout.put( ':' );
    stdout.putisize( t.s, 2, '0' );
```

If this seems like too much typing, well fear not; in a later chapter you will learn how to create your own functions and you can put this code into a function that will print the time with a single function call.

Time input is only a little more complicated. As it turns out, HLA accepts the colon (":") character as a delimiter when reading numbers from the user. Therefore, reading a time value is no more difficult than reading any other three integer values; you can do it with a single call like the following:

```
stdin.get( t.hours, t.mins, t.secs );
```

There is one remaining problem with the time input code: it does not validate the input. To do this, you must manually check the seconds and minutes fields to ensure they are values in the range 0..59. If you wish to enforce a limit on the hours field, you should check that value as well. The following code offers one possible solution:

```
stdin.get( t.hours, t.mins, t.secs );
if( t.m >= 60 ) then

    raise( ex.ValueOutOfRange );

endif;
if( t.s >= 60 ) then

    raise( ex.ValueOutOfRange );

endif;
```

6.6 Putting It All Together

Date and time data types do not get anywhere near the consideration they deserve in modern programs. To help ensure that you calculate dates properly in your HLA programs, the HLA Standard Library provides a set of date and time functions that ease the use of dates and times in your programs.

7.1 Chapter Overview

In this chapter you will learn about the *file* persistent data type. In most assembly languages, file I/O is a major headache. Not so in HLA with the HLA Standard Library. File I/O is no more difficult than writing data to the standard output device or reading data from the standard input device. In this chapter you will learn how to create and manipulate sequential and random-access files.

7.2 File Organization

A file is a collection of data that the system maintains in persistent storage. Persistent means that the storage is non-volatile – that is, the system maintains the data even after the program terminates; indeed, even if you shut off system power. For this reason, plus the fact that different programs can access the data in a file, applications typically use files to maintain data across executions of the application and to share data with other applications.

The operating system typically saves file data on a disk drive or some other form of secondary storage device. As you may recall from the chapter on the memory hierarchy (see “The Memory Hierarchy” on page 303), secondary storage (disk drives) is much slower than main memory. Therefore, you generally do not store data that a program commonly accesses in files during program execution unless that data is far too large to fit into main memory (e.g., a large database).

Under Linux and Windows, a standard file is simply a stream of bytes that the operating system does not interpret in any way. It is the responsibility of the application to interpret this information, much the same as it is your application’s responsibility to interpret data in memory. The stream of bytes in a file could be a sequence of ASCII characters (e.g., a text file) or they could be pixel values that form a 24-bit color photograph.

Files generally take one of two different forms: *sequential files* or *random access files*. Sequential files are great for data you read or write all at once; random access files work best for data you read and write in pieces (or rewrite, as the case may be). For example, a typical text file (like an HLA source file) is usually a sequential file. Usually your text editor will read or write the entire file at once. Similarly, the HLA compiler will read the data from the file in a sequential fashion without skipping around in the file. A database file, on the other hand, requires random access since the application can read data from anywhere in the file in response to a query.

7.2.1 Files as Lists of Records

A good view of a file is as a list of records. That is, the file is broken down into a sequential string of records that share a common structure. A list is simply an open-ended single dimensional array of items, so we can view a file as an array of records. As such, we can index into the file and select record number zero, record number one, record number two, etc. Using common file access operations, it is quite possible to skip around to different records in a file. Under Windows and Linux, the principle difference between a sequential file and a random access file is the organization of the records and how easy it is to locate a specific record within the file. In this section we’ll take a look at the issues that differentiate these two types of files.

The easiest file organization to understand is the random access file. A random access file is a list of records whose lengths are all identical (i.e., random access files require fixed length records). If the record length is n bytes, then the first record appears at byte offset zero in the file, the second record appears at byte offset n in the file, the third record appears at byte offset $n*2$ in the file, etc. This organization is virtually identical to that of an array of records in main memory; you use the same computation to locate an “ele-

ment” of this list in the file as you would use to locate an element of an array in memory; the only difference is that a file doesn’t have a “base address” in memory, you simply compute the zero-based offset of the record in the file. This calculation is quite simple, and using some file I/O functions you will learn about a little later, you can quickly locate and manipulate any record in a random access file.

Sequential files also consist of a list of records. However, these records do not all have to be the same length¹. If a sequential file does not use fixed length records then we say that the file uses variable-length records. If a sequential file uses variable-length records, then the file must contain some kind of marker or other mechanism to separate the records in the file. Typical sequential files use one of two mechanisms: a length prefix or some special terminating value. These two schemes should sound quite familiar to those who have read the chapter on strings. Character strings use a similar scheme to determine the bounds of a string in memory.

A text file is the best example of a sequential file that uses variable-length records. Text files use a special marker at the end of each record to delineate the records. In a text file, a record corresponds to a single line of text. Under Windows, the line feed character marks the end of each record. Other operating systems may use a different sequence; e.g., Windows uses a carriage return/line feed sequence while the Mac OS uses a single carriage return. Since we’re working with Windows here, we’ll adopt the line feed end of line marker.

Accessing records in a file containing variable-length records is problematic. Unless you have an array of offsets to each record in a variable-length file, the only practical way to locate record n in a file is to read the first $n-1$ records. This is why variable-length files are sequential-access – you have to read the file sequentially from the start in order to locate a specific record in the file. This will be much slower than accessing the file in a random access fashion. Generally, you would not use a variable-length record organization for files you need to access in a random fashion.

At first blush it would seem that fixed-length random access files offer all the advantages here. After all, you can access records in a file with fixed-length records much more rapidly than files using the variable-length record organization. However, there is a cost to this: your fixed-length records have to be large enough to hold the largest possible data object you want to store in a record. To store a sequence of lines in a text file, for example, your record sizes would have to be large enough to hold the longest possible input line. This could be quite large (for example, HLA allows lines up to 256 characters). Each record in the file will consume this many bytes even if the record uses substantially less data. For example, an empty line only requires one or a single byte (for the line feed character). If your record size is 256 bytes, then you’re wasting 255 or 255 bytes for that blank line in your file. If the average line length is around 60 characters, then each line wastes an average of about 200 characters. This problem, known as *internal fragmentation*, can waste a tremendous amount of space on your disk, especially as your files get larger or you create lots of files. File organizations that use variable-length records generally don’t suffer from this problem.

7.2.2 Binary vs. Text Files

Another important thing to realize about files is that they don’t all contain human readable text. Object and executable files are good examples of files that contain binary information rather than text. A text file is a very special kind of variable-length sequential file that uses special end of line markers (line feeds) at the end of each record (line) in the file. Binary files are everything else.

Binary files are often more compact than text files and they are usually more efficient to access. Consider a text file that contains the following set of two-byte integer values:

```
1234
543
3645
32000
```

1. There is nothing preventing a sequential file from using fixed length records. However, they don’t require fixed length records.

1
87
0

As a text file, this file consumes at least 27 bytes (assuming a single byte line feed at the end of each line). However, were we to store the data in a fixed-record length binary file, with two bytes per integer value, this file would only consume 14 bytes – half the space. Furthermore, since the file now uses fixed-length records (two bytes per record) we can efficiently access it in a random fashion. Finally, there is one additional, though hidden, efficiency aspect to the binary format: when a program reads and writes binary data it doesn't have to convert between the binary and string formats. This is an expensive process (with respect to computer time). If a human being isn't going to read this file with a separate program (like a text editor) then converting to and from text format on every I/O operation is a wasted effort.

Consider the following HLA record type:

```
type
  person:
    record
      name:string;
      age:int16;
      ssn:char[11];
      salary:real64;
    endrecord;
```

If we were to write this record as text to a text file, a typical record would take the following form (<nl> indicates the end of line marker, a line feed or line feed):

```
Hyde, Randall<nl>
45<nl>
555-55-5555<nl>
123456.78<nl>
```

Presumably, the next *person* record in the file would begin with the next line of text in the text file.

The binary version of this file (using a fixed length record, reserving 64 bytes for the *name* string) would look, schematically, like the following:

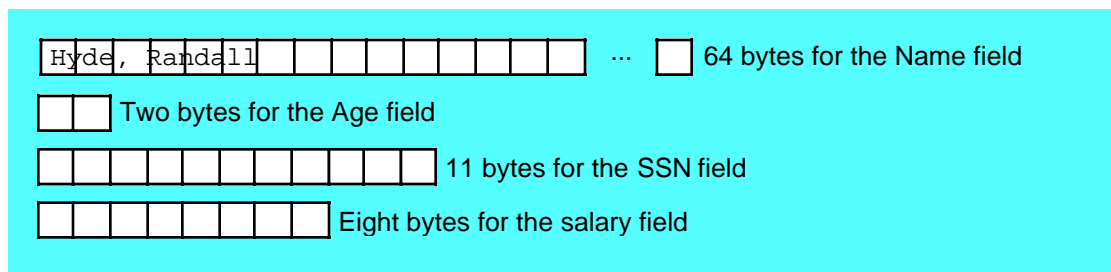


Figure 7.1 Fixed-lengthFormat for Person Record

Don't get the impression that binary files must use fixed length record sizes. We could create a variable-length version of this record by using a zero byte to terminate the string, as follows:

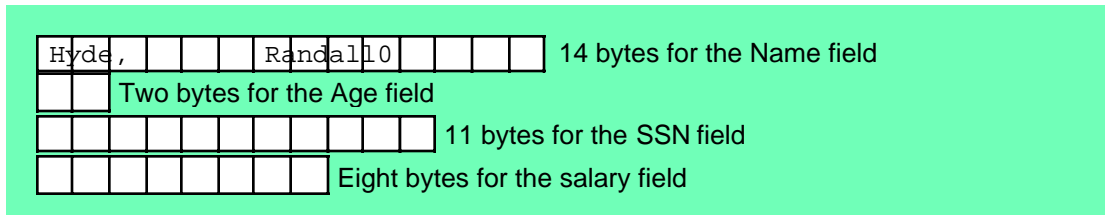


Figure 7.2 Variable-length Format for Person Record

In this particular record format the *age* field starts at offset 14 in the record (since the name field and the “end of field” marker [the zero byte] consume 14 bytes). If a different name were chosen, then the *age* field would begin at a different offset in the record. In order to locate the age, ssn, and salary fields of this record, the program would have to scan past the name and find the zero terminating byte. The remaining fields would follow at fixed offsets from the zero terminating byte. As you can see, it’s a bit more work to process this variable-length record than the fixed-length record. Once again, this demonstrates the performance difference between random access (fixed-length) and sequential access (variable length, in this case) files.

Although binary files are often more compact and more efficient to access, they do have their drawbacks. In particular, only applications that are aware of the binary file’s record format can easily access the file. If you’re handed an arbitrary binary file and asked to decipher its contents, this could be very difficult. Text files, on the other hand, can be read by just about any text editor or filter program out there. Hence, your data files will be more interchangeable with other programs if you use text files. Furthermore, it is easier to debug the output of your programs if they produce text files since you can load a text file into the same editor you use to edit your source files.

7.3 Sequential Files

Sequential files are perfect for three types of persistent data: ASCII text files, “memory dumps”, and stream data. Since you’re probably familiar with ASCII text files, we’ll skip their discussion. The other two methods of writing sequential files deserve more explanation.

A “memory dump” is a file that consists of data you transfer from data structures in memory directly to a file. Although the term “memory dump” suggests that you sequentially transfer data from consecutive memory locations to the file, this isn’t necessarily the case. Memory access can, an often does, occur in a random access fashion. However, once the application constructs a record to write to the file, it writes that record in a sequential fashion (i.e., each record is written in order to the file). A “memory dump” is what most applications do when you request that they save the program’s current data to a file or read data from a file into application memory. When writing, they gather all the important data from memory and write it to the file in a sequential fashion; when reading (loading) data from a file, they read the data from the file in a sequential fashion and store the data into appropriate memory-based data structures. Generally, when loading or saving file data in this manner, the program opens a file, reads/writes data from/to the file, and then it closes the file. Very little processing takes place during the data transfer and the application does not leave the file open for any length of time beyond what is necessary to read or write the file’s data.

Stream data on input is like data coming from a keyboard. The program reads the data at various points in the application where it needs new input to continue. Similarly, stream data on output is like a write to the console device. The application writes data to the file at various points in the program after important computations have taken place and the program wishes to report the results of the calculation. Note that when reading data from a sequential file, once the program reads a particular piece of data, that data is no longer available in future reads (unless, of course, the program closes and reopens the file). When writing data to a

sequential file, once data is written, it becomes a permanent part of the output file. When processing this kind of data the program typically opens a file and then continues execution. As program execution continues, the application can read or write data in the file. At some point, typically towards the end of the application's execution, the program closes the file and commits the data to disk.

Although disk drives are generally thought of as random access devices, the truth is that they are only pseudo-random access; in fact, they perform much better when writing data sequentially on the disk surface. Therefore, sequential access files tend to provide the highest performance (for sequential data) since they match the highest performance access mode of the disk drive.

Working with sequential files in HLA is very easy. In fact, you already know most of the functions you need in order to read or write sequential files. All that's left to learn is how to open and close files and perform some simple tests (like "have we reached the end of a file when reading data from the file?").

The file I/O functions are nearly identical to the *stdin* and *stdout* functions. Indeed, *stdin* and *stdout* are really nothing more than special file I/O functions that read data from the standard input device (a file) or write data to the standard output device (which is also a file). You use the file I/O functions in a manner analogous to *stdin* and *stdout* except you use the *fileio* prefix rather than *stdin* or *stdout*. For example, to write a string to an output file, you could use the *fileio.puts* function almost the same way you use the *stdout.puts* routine. Similarly, if you wanted to read a string from a file, you would use *fileio.gets*. The only real difference between these function calls and their *stdin* and *stdout* counterparts is that you must supply an extra parameter to tell the function what file to use for the transfer. This is a double word value known as the *file handle*. You'll see how to initialize this file handle in a moment, but assuming you have a dword variable that holds a file handle value, you can use calls like the following to read and write data to sequential files:

```
fileio.get( inputHandle, i, j, k ); // Reads i, j, k, from file inputHandle.
fileio.put( outputHandle, "I = ", i, "J = ", j, " K = ", k, nl );
```

Although this example only demonstrates the use of *get* and *put*, be aware that almost all of the *stdin* and *stdout* functions are available as *fileio* functions, as well (in fact, most of the *stdin* and *stdout* functions simply call the appropriate *fileio* function to do the real work).

There is, of course, the issue of this file handle variable. You're probably wondering what a file handle is and how you tell the *fileio* routines to work with data in a specific file on your disk. Well, the definition of the file handle object is the easiest to explain – it's just a dword variable that the operating system initializes and uses to keep track of your file. To declare a file handle, you'd just create a dword variable, e.g.,

```
static
    myFileHandle:dword;
```

You should never explicitly manipulate the value of a file handle variable. The operating system will initialize this variable for you (via some calls you'll see in a moment) and the OS expects you to leave this value alone as long as you're working with the file the OS associates with that handle. If you're curious, both Linux and Windows store small integer values into the handle variable. Internally, the OS uses this value as an index into an array that contains pertinent information about open files. If you mess with the file handle's value, you will confuse the OS greatly the next time you attempt to access the file. Moral of the story – leave this value alone while the file is open.

Before you can read or write a file you must open that file and associate a filename with it. The HLA Standard Library provides a couple of functions that provide this service: *fileio.open* and *fileio.openNew*. The *fileio.open* function opens an existing file for reading, writing, or both. Generally, you open sequential files for reading or writing, but not both (though there are some special cases where you can open a sequential file for reading and writing). The syntax for the call to this function is

```
fileio.open( "filename", access );
```

The first parameter is a string value that specifies the filename of the file to open. This can be a string constant, a register that contains the address of a string value, or a string variable. The second parameter is a constant that specifies how you want to open the file. You may use any of the three predefined constants for the second parameter:

```
fileio.r
```

```
fileio.w
fileio.rw
```

fileio.r obviously specifies that you want to open an existing file in order to read the data from that file; likewise, *fileio.w* says that you want to open an existing file and overwrite the data in that file. The *fileio.rw* option lets you open a file for both reading and writing.

The *fileio.open* routine, if successful, returns a *file handle* in the EAX register. Generally, you will want to save the return value into a double word variable for use by the other HLA *fileio* routines (i.e., the *MyFileHandle* variable in the earlier example).

If the OS cannot open the file, *fileio.open* will raise an *ex.FileOpenFailure* exception. This usually means that it could not find the specified file on the disk.

The *fileio.open* routine requires that the file exist on the disk or it will raise an exception. If you want to create a new file, that might not already exist, the *fileio.openNew* function will do the job for you. This function uses the following syntax:

```
fileio.openNew( "filename" );
```

Note that this call has only a single parameter, a string specifying the filename. When you open a file with *fileio.openNew*, the file is always opened for writing. If a file by the specified filename already exists, then this function will delete the existing file and the new data will be written over the top of the old file (*so be careful!*).

Like *fileio.open*, *fileio.openNew* returns a file handle in the EAX register if it successfully opens the file. You should save this value in a file handle variable. This function raises the *ex.FileOpenFailure* exception if it cannot open the file.

Once you open a sequential file with *fileio.open* or *fileio.openNew* and you save the file handle value away, you can begin reading data from an input file (*fileio.r*) or writing data to an output file (*fileio.w*). To do this, you would use functions like *fileio.put* as noted above.

When the file I/O is complete, you must close the file to commit the file data to the disk. You should always close all files you open as soon as you are through with them so that the program doesn't consume excess system resources. The syntax for *fileio.close* is very simple, it takes a single parameter, the file handle value returned by *fileio.open* or *fileio.openNew*:

```
fileio.close( file_handle );
```

If there is an error closing the file, *fileio.close* will raise the *ex.FileCloseError* exception. Note that Linux and Windows automatically close all open files when an application terminates; however, it is very bad programming style to depend on this feature. If the system crashes (or the user turns off the power) before the application terminates, file data may be lost. So you should always close your files as soon as you are done accessing the data in that file.

The last function of interest to us right now is the *fileio.eof* function. This function returns true (1) or false (0) in the AL register depending on whether the current file pointer is at the end of the file. Generally you would use this function when reading data from an input file to determine if there is more data to read from the file. You would not normally call this function for output files; it always returns false². Since the *fileio* routines will raise an exception if the disk is full, there is no need to waste time checking for end of file (EOF) when writing data to a file. The syntax for *fileio.eof* is

```
fileio.eof( file_handle );
```

The following program example demonstrates a complete program that opens and writes a simple text file:

```
program SimpleFileOutput;
```

2. Actually, it will return true under Windows if the disk is full.

```

#include( "stdlib.hhf" )

static
    outputHandle:dword;

begin SimpleFileOutput;

    fileio.openNew( "myfile.txt" );
    mov( eax, outputHandle );

    for( mov( 0, ebx ); ebx < 10; inc( ebx )) do

        fileio.put( outputHandle, (type uns32 ebx ), nl );

    endfor;
    fileio.close( outputHandle );

end SimpleFileOutput;

```

Program 7.1 A Simple File Output Program

The following sample program reads the data that Program 7.1 produces and writes the data to the standard output device:

```

program SimpleFileInput;
#include( "stdlib.hhf" )

static
    inputHandle:dword;
    u:uns32;

begin SimpleFileInput;

    fileio.open( "myfile.txt", fileio.r );
    mov( eax, inputHandle );

    for( mov( 0, ebx ); ebx < 10; inc( ebx )) do

        fileio.get( inputHandle, u );
        stdout.put( "ebx=", ebx, " u=", u, nl );

    endfor;
    fileio.close( inputHandle );

end SimpleFileInput;

```

Program 7.2 A Sample File Input Program

There are a couple of interesting functions that you can use when working with sequential files. They are the following:

```
fileio.rewind( fileHandle );
fileio.append( fileHandle );
```

The *fileio.rewind* function resets the “file pointer” (the cursor into the file where the next read or write will take place) back to the beginning of the file. This name is a carry-over from the days of files on tape drives when the system would rewind the tape on the tape drive to move the read/write head back to the beginning of the file.

If you’ve opened a file for reading, then *fileio.rewind* lets you begin reading the file from the start (i.e., make a second pass over the data). If you’ve opened the file for writing, then *fileio.rewind* will cause future writes to overwrite the data you’ve previously written; you won’t normally use this function with files you’ve opened only for writing. If you’ve opened the file for reading and writing (using the *fileio.rw* option) then you can write the data after you’ve first opened the file and then rewind the file and read the data you’ve written. The following is a modification to Program 7.2 that reads the data file twice. This program also demonstrates the use of *fileio.eof* to test for the end of the file (rather than just counting the records).

```
program SimpleFileInput2;
#include( "stdlib.hhf" )

static
    inputHandle:dword;
    u:uns32;

begin SimpleFileInput2;

    fileio.open( "myfile.txt", fileio.r );
    mov( eax, inputHandle );

    for( mov( 0, ebx ); ebx < 10; inc( ebx ) ) do

        fileio.get( inputHandle, u );
        stdout.put( "ebx=", ebx, " u=", u, nl );

    endfor;
    stdout.newln();

    // Rewind the file and reread the data from the beginning.
    // This time, use fileio.eof() to determine when we've
    // reached the end of the file.

    fileio.rewind( inputHandle );
    while( fileio.eof( inputHandle ) = false ) do

        // Read and display the next item from the file:

        fileio.get( inputHandle, u );
        stdout.put( "u=", u, nl );

        // Note: after we read the last numeric value, there is still
        // a newline sequence left in the file, if we don't read the
        // newline sequence after each number then EOF will be false
        // at the start of the loop and we'll get an EOF exception
        // when we try to read the next value. Calling fileio.ReadLn
        // "eats" the newline after each number and solves this problem.

        fileio.readLn( inputHandle );

    endwhile;
```

```

    fileio.close( inputHandle );
end SimpleFileInput2;

```

Program 7.3 Another Sample File Input Program

The *fileio.append* function moves the file pointer to the end of the file. This function is really only useful for files you've opened for writing (or reading and writing). After executing *fileio.append*, all data you write to the file will be written after the data that already exists in the file (i.e., you use this call to append data to the end of a file you've opened). The following program demonstrates how to use this program to append data to the file created by Program 7.1:

```

program AppendDemo;
#include( "stdlib.hhf" )

static
    fileHandle:dword;
    u:uns32;

begin AppendDemo;

    fileio.open( "myfile.txt", fileio.rw );
    mov( eax, fileHandle );
    fileio.append( eax );

    for( mov( 10, ecx ); ecx < 20; inc( ecx ) do

        fileio.put( fileHandle, (type uns32 ecx), nl );

    endfor;

    // Okay, let's rewind to the beginning of the file and
    // display all the data from the file, including the
    // new data we just wrote to it:

    fileio.rewind( fileHandle );
    while( !fileio.eof( fileHandle ) do

        // Read and display the next item from the file:

        fileio.get( fileHandle, u );
        stdout.put( "u=", u, nl );
        fileio.readLine( fileHandle );

    endwhile;
    fileio.close( fileHandle );

end AppendDemo;

```

Program 7.4 Demonstration of the fileio.Append Routine

Another function, similar to *fileio.eof*, that will prove useful when reading data from a file is the *fileio.eoln* function. This function returns true if the next character(s) to be read from the file are the end of line sequence (carriage return, linefeed, or the sequence of these two characters under Windows, just a line feed under Linux). This function returns true or false in the EAX register if it detects an end of line sequence. The calling sequence for this function is

```
fileio.eoln( fileHandle );
```

If *fileio.eoln* detects an end of line sequence, it will read those characters from the file (so the next read from the file will not read the end of line characters). If *fileio.eoln* does not detect the end of line sequence, it does not modify the file pointer position. The following sample program demonstrates the use of *fileio.eoln* in the AppendDemo program, replacing the call to *fileio.readLn* (since *fileio.eoln* reads the end of line sequence, there is no need for the call to *fileio.readLn*):

```
program EolnDemo;
#include( "stdlib.hhf" )

static
  fileHandle:dword;
  u:uns32;

begin EolnDemo;

  fileio.open( "myfile.txt", fileio.rw );
  mov( eax, fileHandle );
  fileio.append( eax );

  for( mov( 10, ecx ); ecx < 20; inc( ecx ) ) do

    fileio.put( fileHandle, (type uns32 ecx), nl );

  endfor;

  // Okay, let's rewind to the beginning of the file and
  // display all the data from the file, including the
  // new data we just wrote to it:

  fileio.rewind( fileHandle );
  while( !fileio.eof( fileHandle ) ) do

    // Read and display the next item from the file:

    fileio.get( fileHandle, u );
    stdout.put( "u=", u, nl );
    if( !fileio.eoln( fileHandle ) ) then

      stdout.put( "Hmmm, expected the end of the line", nl );

    endif;

  endwhile;
  fileio.close( fileHandle );

end EolnDemo;
```


7.4 Random Access Files

The problem with sequential files is that they are, well, sequential. They are great for dumping and retrieving large blocks of data all at once, but they are not suitable for applications that need to read, write, and rewrite the same data in a file multiple times. In those situations random access files provide the only reasonable alternative.

Windows and Linux don't differentiate sequential and random access files anymore than the CPU differentiates byte and character values in memory; it's up to your application to treat the files as sequential or random access. As such, you use many of the same functions to manipulate random access files as you use to manipulate sequential access files; you just use them differently is all.

You still open files with *fileio.open* and *fileio.openNew*. Random access files are generally opened for reading or reading and writing. You rarely open a random access file as write-only since a program typically needs to read data if it's jumping around in the file.

You still close the files with *fileio.close*.

You can read and write the files with *fileio.get* and *fileio.put*, although you would not normally use these functions for random access file I/O because each record you read or write has to be exactly the same length and these functions aren't particularly suited for fixed-length record I/O. Most of the time you will use one of the following functions to read and write fixed-length data:

```
fileio.write( fileHandle, buffer, count );
fileio.read( fileHandle, buffer, count );
```

The *fileHandle* parameter is the usual file handle value (a dword variable). The *count* parameter is an `uns32` object that specifies how many bytes to read or write. The *buffer* parameter must be an array object with at least *count* bytes. This parameter supplies the address of the first byte in memory where the I/O transfer will take place. These functions return the number of bytes read or written in the EAX register. For *fileio.read*, if the return value in EAX does not equal *count's* value, then you've reached the end of the file. For *fileio.write*, if EAX does not equal *count* then the disk is full.

Here is a typical call to the *fileio.read* function that will read a record from a file:

```
fileio.read( myHandle, myRecord, @size( myRecord ) );
```

If the return value in EAX does not equal `@size(myRecord)` and it does not equal zero (indicating end of file) then there is something seriously wrong with the file since the file should contain an integral number of records.

Writing data to a file with *fileio.write* uses a similar syntax to *fileio.read*.

You can use *fileio.read* and *fileio.write* to read and write data from/to a sequential file, just as you can use routines like *fileio.get* and *fileio.put* to read/write data from/to a random access file. You'd typically use these routines to read and write data from/to a binary sequential file.

The functions we've discussed to this point don't let you randomly access records in a file. If you call *fileio.read* several times in a row, the program will read those records sequentially from the text file. To do true random access I/O we need the ability to jump around in the file. Fortunately, the HLA Standard Library's file module provides several functions you can use to accomplish this.

The *fileio.position* function returns the current offset into the file in the EAX register. If you call this function immediately before reading or writing a record to a file, then this function will tell you the exact

position of that record. You can use this value to quickly locate that record for a future access. The calling sequence for this function is

```
fileio.position( fileHandle ); // Returns current file position in EAX.
```

The *fileio.seek* function repositions the file pointer to the offset you specify as a parameter. The following is the calling sequence for this function:

```
fileio.seek( fileHandle, offset ); // Repositions file to specified offset.
```

The function call above will reposition the file pointer to the byte offset specified by the *offset* parameter. If you feed this function the value returned by *fileio.position*, then the next read or write operation will access the record written (or read) immediately after the *fileio.position* call.

You can pass any arbitrary offset value as a parameter to the *fileio.seek* routine; this value does not have to be one that the *fileio.position* function returns. For random access file I/O you would normally compute this offset file by specifying the index of the record you wish to access multiplied by the size of the record. For example, the following code computes the byte offset of record *index* in the file, repositions the file pointer to that record, and then reads the record:

```
intmul( @size( myRecord ), index, ebx );
fileio.seek( fileHandle, ebx );
fileio.read( fileHandle, (type byte myRecord), @size( myRecord ) );
```

You can use essentially this same code sequence to select a specific record in the file for writing.

Note that it is not an error to seek beyond the current end of file and then write data. If you do this, the OS will automatically fill in the intervening records with uninitialized data. Generally, this isn't a great way to create files, but it is perfectly legal. On the other hand, be aware that if you do this by accident, you may wind up with garbage in the file and no error to indicate that this has happened.

The *fileio* module provides another routine for repositioning the file pointer: *fileio.rSeek*. This function's calling sequence is very similar to *fileio.seek*, it is

```
fileio.rSeek( fileHandle, offset );
```

The difference between this function and the regular *fileio.seek* function is that this function repositions the file pointer offset bytes from the end of the file (rather than offset bytes from the start of the file). The "r" in "rSeek" stands for "reverse" seek.

Repositioning the file pointer, especially if you reposition it a fair distance from its current location, can be a time-consuming process. If you reposition the file pointer and then attempt to read a record from the file, the system may need to reposition a disk arm (a very slow process) and wait for the data to rotate underneath the disk read/write head. This is why random access I/O is much less efficient than sequential I/O.

The following program demonstrates random access I/O by writing and reading a file of records:

```
program RandomAccessDemo;
#include( "stdlib.hhf" )

type
  fileRec:
    record
      x:int16;
      y:int16;
      magnitude:uns8;
    endrecord;

const

  // Some arbitrary data we can use to initialize the file:
```

```

fileData:=
[
    fileRec:[ 2000, 1, 1 ],
    fileRec:[ 1000, 10, 2 ],
    fileRec:[ 750, 100, 3 ],
    fileRec:[ 500, 500, 4 ],
    fileRec:[ 100, 1000, 5 ],
    fileRec:[ 62, 2000, 6 ],
    fileRec:[ 32, 2500, 7 ],
    fileRec:[ 10, 3000, 8 ]
];

static
    fileHandle:          dword;
    RecordFromFile:     fileRec;
    InitialFileData:    fileRec[ 8 ] := fileData;

begin RandomAccessDemo;

    fileio.openNew( "fileRec.bin" );
    mov( eax, fileHandle );

    // Okay, write the initial data to the file in a sequential fashion:

    for( mov( 0, ebx ); ebx < 8; inc( ebx ) ) do

        intmul( @size( fileRec ), ebx, ecx ); // Compute index into fileData
        fileio.write
        (
            fileHandle,
            (type byte InitialFileData[ecx]),
            @size( fileRec )
        );

    endfor;

    // Okay, now let's demonstrate a random access of this file
    // by reading the records from the file backwards.

    stdout.put( "Reading the records, backwards:" nl );
    for( mov( 7, ebx ); (type int32 ebx) >= 0; dec( ebx ) ) do

        intmul( @size( fileRec ), ebx, ecx ); // Compute file offset
        fileio.seek( fileHandle, ecx );
        fileio.read
        (
            fileHandle,
            (type byte RecordFromFile),
            @size( fileRec )
        );
        if( eax = @size( fileRec ) ) then

            stdout.put
            (
                "Read record #",
                (type uns32 ebx),
                ", values:" nl
                "  x: ", RecordFromFile.x, nl
            );
        endif;
    endfor;
end RandomAccessDemo;

```

```

        " y: ", RecordFromFile.y, nl
        " magnitude: ", RecordFromFile.magnitude, nl nl
    );

else

    stdout.put( "Error reading record number ", (type uns32 ebx), nl );

endif;

endfor;
fileio.close( fileHandle );

end RandomAccessDemo;

```

Program 7.6 Random Access File I/O Example

7.5 ISAM (Indexed Sequential Access Method) Files

ISAM is a trick that attempts to allow random access to variable-length records in a sequential file. This is a technique employed by IBM on their mainframe data bases in the 1960's and 1970's. Back then, disk space was very precious (remember why we wound up with the Y2K problem?) and IBM's engineers did everything they could to save space. At that time disks held about five megabytes, or so, were the size of washing machines, and cost tens of thousands of dollars. You can appreciate why they wanted to make every byte count. Today, data base designers have disk drives with hundreds of gigabytes per drive and RAID³ devices with dozens of these drives installed. They don't bother trying to conserve space at all ("Heck, I don't know how big the person's name can get, so I'll allocate 256 bytes for it!"). Nevertheless, even with large disk arrays, saving space is often a wise idea. Not everyone has a terabyte (1,000 gigabytes) at their disposal and a user of your application may not appreciate your decision to waste their disk space. Therefore, techniques like ISAM that can reduce disk storage requirements are still important today.

ISAM is actually a very simple concept. Somewhere, the program saves the offset to the start of every record in a file. Since offsets are four bytes long, an array of dwords will work quite nicely⁴. Generally, as you construct the file you fill in the list (array) of offsets and keep track of the number of records in the file. For example, if you were creating a text file and you wanted to be able to quickly locate any line in the file, you would save the offset into the file of each line you wrote to the file. The following code fragment shows how you could do this:

```

static
    outputLine: string;
    ISAMarray: dword[ 128*1024 ]; // allow up to 128K records.
    .
    .
    .
    mov( 0, ecx ); // Keep record count here.
    forever

        << create a line of text in "outputLine" >>

        fileio.position( fileHandle );

```

3. Redundant array of inexpensive disks. RAID is a mechanism for combining lots of cheap disk drives together to form the equivalent of a really large disk drive.

4. This assumes, of course, that your files have a maximum size of four gigabytes.

```

mov( eax, ISAMarray[ecx*4] ); // Save away current record offset.
fileio.put( fileHandle, outputLine, nl ); // Write the record.
inc( ecx ); // Advance to next element of ISAMarray.

```

```
<< determine if we're done and BREAK if we are >>
```

```
endfor;
```

```

<< At this point, ECX contains the number of records and >>
<< ISAMarray[0]..ISAMarray[ecx-1] contain the offsets to >>
<< each of the records in the file. >>

```

After building the file using the code above, you can quickly jump to an arbitrary line of text by fetching the index for that line from the *ISAMarray* list. The following code demonstrates how you could read line *recordNumber* from the file:

```

mov( recordNumber, ebx );
fileio.seek( fileHandle, ISAMarray[ ebx*4 ] );
fileio.a_gets( fileHandle, inputString );

```

As long as you've precalculated the *ISAMarray* list, accessing an arbitrary line in this text file is a trivial matter.

Of course, back in the days when IBM programmers were trying to squeeze every byte from their databases as possible so they would fit on a five megabyte disk drive, they didn't have 512 kilobytes of RAM to hold 128K entries in the *ISAMarray* list. Although a half a megabyte is no big deal today, there are a couple of reasons why keeping the *ISAMarray* list in a memory-based array might not be such a good idea. First, databases are much larger these days. Some databases have hundreds of millions of entries. While setting aside a half a megabyte for an ISAM table might not be a bad thing, few people are willing to set aside a half a gigabyte for this purpose. Even if your database isn't amazingly big, there is another reason why you might not want to keep your *ISAMarray* in main memory – it's the same reason you don't keep the file in memory – memory is volatile and the data is lost whenever the application quits or the user removes power from the system. The solution is exactly the same as for the file data: you store the *ISAMarray* data in its own file. A program that builds the ISAM table while writing the file is a simple modification to the previous ISAM generation program. The trick is to open two files concurrently and write the ISAM data to one file while you're writing the text to the other file:

```

static
fileHandle: dword; // file handle for the text file.
outputLine: string; // file handle for the ISAM file.
CurrentOffset: dword; // Holds the current offset into the text file.
.
.
.
forever

<< create a line of text in "outputLine" >>

// Get the offset of the next record in the text file
// and write this offset (sequentially) to the ISAM file.

fileio.position( fileHandle );
mov( eax, CurrentOffset );
fileio.write( isamHandle, (type byte CurrentOffset), 4 );

// Okay, write the actual text data to the text file:

fileio.put( fileHandle, outputLine, nl ); // Write the record.

<< determine if we're done and BREAK if we are >>

```

```
endfor;
```

If necessary, you can count the number of records as before. You might write this value to the first record of the ISAM file (since you know the first record of the text file is always at offset zero, you can use the first element of the ISAM list to hold the count of ISAM/text file records).

Since the ISAM file is just a sequence of four-byte integers, each record in the file (i.e., an integer) has the same length. Therefore, we can easily access any value in the ISAM file using the random access file I/O mechanism. In order to read a particular line of text from the text file, the first task is to read the offset from the ISAM file and then use that offset to read the desired line from the text file. The code to accomplish this is as follows:

```
// Assume we want to read the line specified by the "lineNumber" variable.

if( lineNumber <> 0 ) then

    // If not record number zero, then fetch the offset to the desired
    // line from the ISAM file:

    intmul( 4, lineNumber, eax ); // Compute the index into the ISAM file.
    fileio.seek( isamHandle, eax );
    fileio.read( isamHandle, (type byte CurrentOffset), 4 ); // Read offset

else

    mov( 0, eax ); // Special case for record zero because the file
                  // contains the record count in this position.

endif;
fileio.seek( fileHandle, CurrentOffset ); // Set text file position.
fileio.a_gets( fileHandle, inputLine ); // Read the line of text.
```

This operation runs at about half the speed of having the ISAM array in memory (since it takes four file accesses rather than two to read the line of text from the file), but the data is non-volatile and is not limited by the amount of available RAM.

If you decide to use a memory-based array for your ISAM table, it's still a good idea to keep that data in a file somewhere so you don't have to recompute it (by reading the entire file) every time your application starts. If the data is present in a file, all you've got to do is read that file data into your *ISAMarray* list. Assuming you've stored the number of records in element number zero of the ISAM array, you could use the following code to read your ISAM data into the *ISAMarray* variable:

```
static
    isamSize: uns32;
    isamHandle: dword;
    fileHandle: dword;
    ISAMarray: dword[ 128*1024 ];
    .
    .
    .
// Read the first record of the ISAM file into the isamSize variable:

fileio.read( isamHandle, (type byte isamSize), 4 );

// Now read the remaining data from the ISAM file into the ISAMarray
// variable:

if( isamSize >= 128*1024 ) then

    raise( ex.ValueOutOfRange );
```

```

endif;
intmul( 4, isamSize, ecx ); // #records * 4 is number of bytes to read.
fileio.read( isamHandle, (type byte ISAMarray), ecx );

// At this point, ISAMarray[0]..ISAMarray[isamSize-1] contain the indexes
// into the text file for each line of text.

```

7.6 Truncating a File

If you open an existing file (using *fileio.open*) for output and write data to that file, it overwrites the existing data from the start of the file. However, if the new data you write to the file is shorter than the data originally appearing in the file, the excess data from the original file, beyond the end of the new data you've written, will still appear at the end of the new data. Sometimes this might be desirable, but most of the time you'll want to delete the old data after writing the new data.

One way to delete the old data is to use the *fileio.openNew* function to open the file. The *fileio.openNew* function automatically deletes any existing file so only the data you write to the file will be present in the file. However, there may be times when you may want to read the old data first, rewind the file, and then overwrite the data. In this situation, you'll need a function that will *truncate* the old data at the end of the file after you've written the new data. The *fileio.truncate* function accomplishes this task. This function uses the following calling syntax:

```
fileio.truncate( fileHandle );
```

Note that this function does not close the file. You still have to call *fileio.close* to commit the data to the disk.

The following sample program demonstrates the use of the *fileio.truncate* function:

```

program TruncateDemo;
#include( "stdlib.hhf" )

static
    fileHandle:dword;
    u:uns32;

begin TruncateDemo;

    fileio.openNew( "myfile.txt" );
    mov( eax, fileHandle );
    for( mov( 0, ecx ); ecx < 20; inc( ecx ) ) do

        fileio.put( fileHandle, (type uns32 ecx), nl );

    endfor;

    // Okay, let's rewind to the beginning of the file and
    // rewrite the first ten lines and then truncate the
    // file at that point.

    fileio.rewind( fileHandle );
    for( mov( 0, ecx ); ecx < 10; inc( ecx ) ) do

        fileio.put( fileHandle, (type uns32 ecx), nl );

```

```

endfor;
fileio.truncate( fileHandle );

// Rewind and display the file contents to ensure that
// the file truncation has worked.

fileio.rewind( fileHandle );
while( !fileio.eof( fileHandle ) ) do

    // Read and display the next item from the file:

    fileio.get( fileHandle, u );
    stdout.put( "u=", u, nl );
    fileio.readLn( fileHandle );

endwhile;
fileio.close( fileHandle );

end TruncateDemo;

```

Program 7.7 Using fileio.truncate to Eliminate Old Data From a File

7.7 File Utility Routines

The following subsections describe *fileio* functions that manipulate files or return meta-information about files (e.g., the file size and attributes).

```

program CopyDemo;
#include( "stdlib.hhf" )

begin CopyDemo;

    // Make a copy of myfile.txt to itself to demonstrate
    // a true "failsIfExists" parameter.

    if( !fileio.copy( "myfile.txt", "myfile.txt", true ) ) then

        stdout.put( "Did not copy 'myfile.txt' over itself" nl );

    else

        stdout.put( "Whoa! The failsIfExists parameter didn't work." nl );

    endif;

    // Okay, make a copy of the file to a different file, to verify
    // that this works properly:

    if( fileio.copy( "myfile.txt", "copyOfMyFile.txt", false ) ) then

        stdout.put( "Successfully copied the file" nl );

    else

```



```

        stdout.put( "Failed to copy the file (maybe it doesn't exist?)" nl );

    endif;

end CopyDemo;

program FileMoveDemo;
#include( "stdlib.hhf" )

begin FileMoveDemo;

    // Rename the "myfile.txt" file to the name "renamed.txt".

    if( !fileio.move( "myfile.txt", "renamed.txt" ) ) then

        stdout.put
        (
            "Could not rename 'myfile.txt' (maybe it doesn't exist?)" nl
        );

    else

        stdout.put( "Successfully renamed the file" nl );

    endif;

end FileMoveDemo;

```

7.7.1 Computing the File Size

Another useful function to have is one that computes the size of an existing file on the disk. The *fileio.size* function provides this capability. The calling sequences for this function are

```

fileio.size( filenameString );
fileio.size( fileHandle );

```

The first form above expects you to pass the filename as a string parameter. The second form expects a handle to a file you've opened with *fileio.open* or *fileio.openNew*. These two calls return the size of the file in EAX. If an error occurs, these functions return -1 (\$FFFF_FFFF) in EAX. Note that the files must be less than four gigabytes in length when using this function (if you need to check the size of larger files, you will have to call the appropriate OS function rather than these functions; however, since files larger than four gigabytes are rather rare, you probably won't have to worry about this problem).

One interesting use for this function is to determine the number of records in a fixed-length-record random access file. By getting the size of the file and dividing by the size of a record, you can determine the number of records in the file.

Another use for this function is to allow you to determine the size of a (smaller) file, allocate sufficient storage to hold the entire file in memory (by using *malloc*), and then read the entire file into memory using the *fileio.read* function. This is generally the fastest way to read data from a file into memory.

Program 7.10 demonstrates the use of the two forms of the *fileio.size* function by displaying the size of the "myfile.txt" file created by other sample programs in this chapter.

```

program FileSizeDemo;
#include( "stdlib.hhf" )

```

```

static
    handle:dword;

begin FileSizeDemo;

    // Display the size of the "FileSizeDemo.hla" file:

    fileio.size( "FileSizeDemo.hla" );
    if( eax <> -1 ) then

        stdout.put( "Size of file: ", (type uns32 eax), nl );

    else

        stdout.put( "Error calculating file size" nl );

    endif;

    // Same thing, using the file handle as a parameter:

    fileio.open( "FileSizeDemo.hla", fileio.r );
    mov( eax, handle );
    fileio.size( handle );
    if( eax <> -1 ) then

        stdout.put( "Size of file(2): ", (type uns32 eax), nl );

    else

        stdout.put( "Error calculating file size" nl );

    endif;
    fileio.close( handle );

end FileSizeDemo;

```

Program 7.8 Sample Program That Demonstrates the fileio.size Function

7.7.2 Deleting Files

Another useful file utility function is the fileio.delete function. As its name suggests, this function deletes a file that you specify as the function's parameter. The calling sequence for this function is

```
fileio.delete( filenameToDelete );
```

The single parameter is a string containing the pathname of the file you wish to delete. This function returns true/false in the EAX register to denote success/failure.

Program 7.11 provides an example of the use of the *fileio.delete* function.

```
program DeleteFileDemo;
```

```

#include( "stdlib.hhf" )

static
    handle:dword;

begin DeleteFileDemo;

    // Delete the "myfile.txt" file:

    fileio.delete( "xyz" );
    if( eax ) then

        stdout.put( "Deleted the file", nl );

    else

        stdout.put( "Error deleting the file" nl );

    endif;

end DeleteFileDemo;

```

Program 7.9 Example Usage of the fileio.delete Procedure

7.8 Directory Operations

In addition to manipulating files, you can also manipulate directories with some of the *fileio* functions. The HLA Standard Library includes several functions that let you create and use subdirectories. These functions are *fileio.cd* (change directory), *fileio.gwd* (get working directory), and *fileio.mkdir* (make directory). Their calling sequences are

```

fileio.cd( pathnameString );
fileio.gwd( stringToHoldPathname );
fileio.mkdir( newDirectoryName );

```

The *fileio.cd* and *fileio.mkdir* functions return success or failure (true or false, respectively) in the EAX register. For the *fileio.gwd* function, the string parameter is a destination string where the system will store the pathname to the current directory. You must allocate sufficient storage for the string prior to passing the string to this function (260 characters⁵ is a good default amount if you're unsure how long the pathname could be). If the actual pathname is too long to fit in the destination string you supply as a parameter, the *fileio.gwd* function will raise the *ex.StringOverflow* exception.

The *fileio.cd* function sets the current working directory to the pathname you specify. After calling this function, the OS will assume that all future "unadorned" file references (those without any "\", or "/" characters in the pathname) will default to the directory you specify as the *fileio.cd* parameter. Proper use of this function can help make your program much more convenient to use by your program's users since they won't have to enter full pathnames for every file they manipulate.

The *fileio.gwd* function lets you query the system to determine the current working directory. After a call to *fileio.cd*, the string that *fileio.gwd* returns should be the same as *fileio.cd*'s parameter. Typically, you would use this function to keep track of the default directory when your program first starts running. You

5. This is the default MAX_PATH value in Windows. This is probably sufficient for most Linux applications, too.

program will exhibit good manners by switching back to this default directory when your program terminates.

The *fileio.mkdir* function lets your program create a new subdirectory. If your program creates data files and stores them in a default directory somewhere, it's good etiquette to let the user specify the subdirectory where your program should put these files. If you do this, you should give your users the option to create a new directory (in case they want the data placed in a brand-new directory). You can use *fileio.mkdir* for this purpose.

7.9 Putting It All Together

This chapter began with a discussion of the basic file operations. That section was rather short because you've already learned most of what you need to know about file I/O when learning the *stdout* and *stdin* functions. So the introductory material concentrated on a few general file concepts (like the differences between sequential and random access files and the differences between binary and text files). After teaching you the few extra routines you need in order to open and close files, the remainder of this chapter simply concentrated on providing a few examples (like ISAM) of file access and a discussion of the *fileio* routines available in the HLA Standard Library.

While this chapter demonstrates the mechanics of file I/O, how you efficiently use files is well beyond the scope of this chapter. In future volumes you will see how to search for data in files, sort data in files, and even create databases. So keep on reading if you're interested in more information about file operations.

Introduction to Procedures

Chapter Eight

8.1 Chapter Overview

In a procedural programming language the basic unit of code is the *procedure*. A procedure is a set of instructions that compute some value or take some action (such as printing or reading a character value). The definition of a procedure is very similar to the definition of an *algorithm*. A procedure is a set of rules to follow which, if they conclude, produce some result. An algorithm is also such a sequence, but an algorithm is guaranteed to terminate whereas a procedure offers no such guarantee.

This chapter discusses how HLA implements procedures. This is actually the first of three chapters on this subject in this text. This chapter presents HLA procedures from a high level language perspective. A later chapter, Intermediate Procedures, discusses procedures at the machine language level. A whole volume in this sequence, Advanced Procedures, covers advanced programming topics of interest to the very serious assembly language programmer. This chapter, however, provides the foundation for all that follows.

8.2 Procedures

Most procedural programming languages implement procedures using the call/return mechanism. That is, some code calls a procedure, the procedure does its thing, and then the procedure returns to the caller. The call and return instructions provide the 80x86's *procedure invocation mechanism*. The calling code calls a procedure with the CALL instruction, the procedure returns to the caller with the RET instruction. For example, the following 80x86 instruction calls the HLA Standard Library *stdout.newln* routine¹:

```
call stdout.newln;
```

The *stdout.newln* procedure prints a newline sequence to the console device and returns control to the instruction immediately following the “call stdout.newln;” instruction.

Alas, the HLA Standard Library does not supply all the routines you will need. Most of the time you'll have to write your own procedures. To do this, you will use HLA's procedure declaration facilities. A basic HLA procedure declaration takes the following form:

```
procedure ProcName;
    << Local declarations >>
begin ProcName;
    << procedure statements >>
end ProcName;
```

Procedure declarations appear in the declaration section of your program. That is, anywhere you can put a STATIC, CONST, TYPE, or other declaration section, you may place a procedure declaration. In the syntax example above, *ProcName* represents the name of the procedure you wish to define. This can be any valid HLA identifier. Whatever identifier follows the PROCEDURE reserved word must also follow the BEGIN and END reserved words in the procedure. As you've probably noticed, a procedure declaration looks a whole lot like an HLA program. In fact, the only difference (so far) is the use of the PROCEDURE reserved word rather than the PROGRAM reserved word.

Here is a concrete example of an HLA procedure declaration. This procedure stores zeros into the 256 double words that EBX points at upon entry into the procedure:

```
procedure zeroBytes;
begin zeroBytes;

    mov( 0, eax );
```

1. Normally you would call newln using the “newln();” statement, but the CALL instruction works as well.

```

mov( 256, ecx );
repeat

    mov( eax, [ebx] );
    add( 4, ebx );
    dec( ecx );

until( @z ); // That is, until ECX=0.

end zeroBytes;

```

You can use the 80x86 CALL instruction to call this procedure. When, during program execution, the code falls into the “end zeroBytes;” statement, the procedure returns to whomever called it and begins executing the first instruction beyond the CALL instruction. The following program provides an example of a call to the *zeroBytes* routine:

```

program zeroBytesDemo;
#include( "stdlib.hhf" );

procedure zeroBytes;
begin zeroBytes;

    mov( 0, eax );
    mov( 256, ecx );
    repeat

        mov( eax, [ebx] ); // Zero out current dword.
        add( 4, ebx );    // Point ebx at next dword.
        dec( ecx );      // Count off 256 dwords.

    until( ecx = 0 );    // Repeat for 256 dwords.

end zeroBytes;

static
    dwArray: dword[256];

begin zeroBytesDemo;

    lea( ebx, dwArray );
    call zeroBytes;

end zeroBytesDemo;

```

Program 8.1 Example of a Simple Procedure

As you may have noticed when calling HLA Standard Library procedures, you don’t always need to use the CALL instruction to call HLA procedures. There is nothing special about the HLA Standard Library procedures versus your own procedures. Although the formal 80x86 mechanism for calling procedures is to use the CALL instruction, HLA provides a HLL extension that lets you call a procedure by simply specifying that procedure’s name followed by an empty set of parentheses². For example, either of the following statements will call the HLA Standard Library *stdout.newln* procedure:

2. This assumes that the procedure does not have any parameters.

```
call stdout.newln;
stdout.newln();
```

Likewise, either of the following statements will call the *zeroBytes* procedure in Program 8.1:

```
call zeroBytes;
zeroBytes();
```

The choice of calling mechanism is strictly up to you. Most people, however, find the HLL syntax easier to read.

8.3 Saving the State of the Machine

Take a look at the following program:

```
program nonWorkingProgram;
#include( "stdlib.hhf" );

procedure PrintSpaces;
begin PrintSpaces;

    mov( 40, ecx );
    repeat

        stdout.put( ' ' ); // Print 1 of 40 spaces.
        dec( ecx );        // Count off 40 spaces.

    until( ecx = 0 );

end PrintSpaces;

begin nonWorkingProgram;

    mov( 20, ecx );
    repeat

        PrintSpaces();
        stdout.put( '*', nl );
        dec( ecx );

    until( ecx = 0 );

end nonWorkingProgram;
```

Program 8.2 Program with an Unintended Infinite Loop

This section of code attempts to print 20 lines of 40 spaces and an asterisk. Unfortunately, there is a subtle bug that causes it to print 40 spaces per line and an asterisk in an infinite loop. The main program uses the REPEAT..UNTIL loop to call *PrintSpaces* 20 times. *PrintSpaces* uses ECX to count off the 40 spaces it prints. *PrintSpaces* returns with ECX containing zero. The main program then prints an asterisk, a newline, decrements ECX, and then repeats because ECX isn't zero (it will always contain \$FFFF_FFFF at this point).

The problem here is that the *PrintSpaces* subroutine doesn't preserve the ECX register. Preserving a register means you save it upon entry into the subroutine and restore it before leaving. Had the *PrintSpaces* subroutine preserved the contents of the ECX register, the program above would have functioned properly.

Use the 80x86's PUSH and POP instructions to preserve register values while you need to use them for something else. Consider the following code for *PrintSpaces*:

```

procedure PrintSpaces;
begin PrintSpaces;

    push( eax );
    push( ecx );
    mov( 40, ecx );
    repeat

        stdout.put( ' ' ); // Print 1 of 40 spaces.
        dec( ecx );        // Count off 40 spaces.

    until( ecx = 0 );
    pop( ecx );
    pop( eax );

end PrintSpaces;

```

Note that *PrintSpaces* saves and restores EAX and ECX (since this procedure modifies these registers). Also, note that this code pops the registers off the stack in the reverse order that it pushed them. The last-in, first-out, operation of the stack imposes this ordering.

Either the caller (the code containing the CALL instruction) or the callee (the subroutine) can take responsibility for preserving the registers. In the example above, the callee preserved the registers. The following example shows what this code might look like if the caller preserves the registers:

```

program callerPreservation;
#include( "stdlib.hhf" );

procedure PrintSpaces;
begin PrintSpaces;

    mov( 40, ecx );
    repeat

        stdout.put( ' ' ); // Print 1 of 40 spaces.
        dec( ecx );        // Count off 40 spaces.

    until( ecx = 0 );

end PrintSpaces;

begin callerPreservation;

    mov( 20, ecx );
    repeat

        push( eax );
        push( ecx );
        PrintSpaces();
        pop( ecx );
        pop( eax );
        stdout.put( '*', nl );
        dec( ecx );
    repeat

```



```

until( ecx = 0 );

end callerPreservation;

```

Program 8.3 Demonstration of Caller Register Preservation

There are two advantages to callee preservation: space and maintainability. If the callee preserves all affected registers, then there is only one copy of the PUSH and POP instructions, those the procedure contains. If the caller saves the values in the registers, the program needs a set of PUSH and POP instructions around every call. Not only does this make your programs longer, it also makes them harder to maintain. Remembering which registers to push and pop on each procedure call is not something easily done.

On the other hand, a subroutine may unnecessarily preserve some registers if it preserves all the registers it modifies. In the examples above, the code needn't save EAX. Although *PrintSpaces* changes AL, this won't affect the program's operation. If the caller is preserving the registers, it doesn't have to save registers it doesn't care about:

```

program callerPreservation2;
#include( "stdlib.hhf" );

procedure PrintSpaces;
begin PrintSpaces;

    mov( 40, ecx );
    repeat

        stdout.put( ' ' ); // Print 1 of 40 spaces.
        dec( ecx );       // Count off 40 spaces.

    until( ecx = 0 );

end PrintSpaces;

begin callerPreservation2;

    mov( 10, ecx );
    repeat

        push( ecx );
        PrintSpaces();
        pop( ecx );
        stdout.put( '*', nl );
        dec( ecx );

    until( ecx = 0 );

    mov( 5, ebx );
    while( ebx > 0 ) do

        PrintSpaces();

        stdout.put( ebx, nl );
        dec( ebx );
    end while;
end callerPreservation2;

```

```

endwhile;

mov( 110, ecx );
for( mov( 0, eax );  eax < 7; inc( eax ) ) do

    PrintSpaces();

    stdout.put( eax, " ", ecx, nl );
    dec( ecx );

endfor;

end callerPreservation2;

```

Program 8.4 Demonstrating that Caller Preservation Need not Save All Registers

This example provides three different cases. The first loop (REPEAT..UNTIL) only preserves the ECX register. Modifying the AL register won't affect the operation of this loop. Immediately after the first loop, this code calls *PrintSpaces* again in the WHILE loop. However, this code doesn't save EAX or ECX because it doesn't care if *PrintSpaces* changes them. Since the final loop (FOR) uses EAX and ECX, it saves them both.

One big problem with having the caller preserve registers is that your program may change. You may modify the calling code or the procedure so that they use additional registers. Such changes, of course, may change the set of registers that you must preserve. Worse still, if the modification is in the subroutine itself, you will need to locate *every* call to the routine and verify that the subroutine does not change any registers the calling code uses.

Preserving registers isn't all there is to preserving the environment. You can also push and pop variables and other values that a subroutine might change. Since the 80x86 allows you to push and pop memory locations, you can easily preserve these values as well.

8.4 Prematurely Returning from a Procedure

The HLA EXIT and EXITIF statements let you return from a procedure without having to fall into the corresponding END statement in the procedure. These statements behave a whole lot like the BREAK and BREAKIF statements for loops, except they transfer control to the bottom of the procedure rather than out of the current loop. These statements are quite useful in many cases.

The syntax for these two statements is the following:

```

exit procedurename;
exitif( boolean_expression ) procedurename;

```

The *procedurename* operand is the name of the procedure you wish to exit. If you specify the name of your main program, the EXIT and EXITIF statements will terminate program execution (even if you're currently inside a procedure rather than the body of the main program).

The EXIT statement immediately transfers control out of the specified procedure or program. The conditional exit, EXITIF, statement first tests the boolean expression and exits if the result is true. It is semantically equivalent to the following:

```

if( boolean_expression ) then

    exit procedurename;

endif;

```

```
endif;
```

Although the EXIT and EXITIF statements are invaluable in many cases, you should try to avoid using them without careful consideration. If a simple IF statement will let you skip the rest of the code in your procedure, by all means use the IF statement. Procedures that contain lots of EXIT and EXITIF statements will be harder to read, understand, and maintain than procedures without these statements (after all, the EXIT and EXITIF statements are really nothing more than GOTO statements and you've probably heard already about the problems with GOTOS). EXIT and EXITIF are convenient when you got to return from a procedure inside a sequence of nested control structures and slapping an IF..ENDIF around the remaining code in the procedure is not possible.

8.5 Local Variables

HLA procedures, like procedures and functions in most high level languages, let you declare *local variables*. Local variables are generally accessible only within the procedure, they are not accessible by the code that calls the procedure. Local variable declarations are identical to variable declarations in your main program except, of course, you declare the variables in the procedure's declaration section rather than the main program's declaration section. Actually, you may declare anything in the procedure's declaration section that is legal in the main program's declaration section, including constants, types, and even other procedures³. In this section, however, we'll concentrate on local variables.

Local variables have two important attributes that differentiate them from the variables in your main program (i.e., *global* variables): *lexical scope* and *lifetime*. Lexical scope, or just *scope*, determines when an identifier is usable in your program. Lifetime determines when a variable has memory associated with it and is capable of storing data. Since these two concepts differentiate local and global variables, it is wise to spend some time discussing these two attributes.

Perhaps the best place to start when discussing the scope and lifetimes of local variables is with the scope and lifetimes of global variables -- those variables you declare in your main program. Until now, the only rule you've had to follow concerning the declaration of your variables has been "you must declare all variables that you use in your programs." The position of the HLA declaration section with respect to the program statements automatically enforces the other major rule which is "you must declare all variables before their first use." With the introduction of procedures, it is now possible to violate this rule since (1) procedures may access global variables, and (2) procedure declarations may appear anywhere in a declaration section, even before some variable declarations. The following program demonstrates this source code organization:

```
program demoGlobalScope;
#include( "stdlib.hhf" );

static
    AccessibleInProc: char;

    procedure aProc;
    begin aProc;

        mov( 'a', AccessibleInProc );

    end aProc;
```

3. The chapter on Advanced Procedures discusses the concept of local procedures in greater detail.

```

static
    InaccessibleInProc: char;

begin demoGlobalScope;

    mov( 'b', InaccessibleInProc );
    aProc();
    stdout.put
    (
        "AccessibleInProc = `", AccessibleInProc, "`" nl
        "InaccessibleInProc = `", InaccessibleInProc, "`" nl
    );

end demoGlobalScope;

```

Program 8.5 Demonstration of Global Scope

This example demonstrates that a procedure can access global variables in the main program as long as you declare those global variables before the procedure. In this example, the *aProc* procedure cannot access the *InaccessibleInProc* variable because its declaration appears after the procedure declaration. However, *aProc* may reference *AccessibleInProc* since its declaration appears before the *aProc* procedure in the source code.

A procedure can access any `STATIC`, `STORAGE`, or `READONLY` object exactly the same way the main program accesses such variables -- by simply referencing the name. Although a procedure may access global `VAR` objects, a different syntax is necessary and you need to learn a little more before you will understand the purpose of the additional syntax. Therefore, we'll defer the discussion of accessing `VAR` objects until the chapters dealing with Advanced Procedures.

Accessing global objects is convenient and easy. Unfortunately, as you've probably learned when studying high level language programming, accessing global objects makes your programs harder to read, understand, and maintain. Like most introductory programming texts, this text will discourage the use of global variables within procedures. Accessing global variables within a procedure is sometimes the best solution to a given problem. However, such (legitimate) access typically occurs only in advanced programs involving multiple threads of execution or in other complex systems. Since it is unlikely you would be writing such code at this point, it is equally unlikely that you will absolutely need to access global variables in your procedures so you should carefully consider your options before accessing global variables within your procedures⁴.

Declaring local variables in your procedures is very easy, you use the same declaration sections as the main program: `STATIC`, `READONLY`, `STORAGE`, and `VAR`. The same rules and syntax for the declaration sections and the access of variables you declare in these sections applies in your procedure. The following example code demonstrates the declaration of a local variable.

```

program demoLocalVars;
#include( "stdlib.hhf" );

```

4. Note that this argument against accessing global variables does not apply to other global symbols. It is perfectly reasonable to access global constants, types, procedures, and other objects in your programs.

```

// Simple procedure that displays 0..9 using
// a local variable as a loop control variable.

procedure CntTo10;
var
    i: int32;

begin CntTo10;

    for( mov( 0, i ); i < 10; inc( i ) ) do

        stdout.put( "i=", i, nl );

    endfor;

end CntTo10;

begin demoLocalVars;

    CntTo10();

end demoLocalVars;

```

Program 8.6 Example of a Local Variable in a Procedure

Local variables you declare in a procedure are accessible only within that procedure⁵. Therefore, the variable *i* in procedure *CntTo10* in Program 8.6 is not accessible in the main program.

HLA relaxes, somewhat, the rule that identifiers must be unique in a program for local variables. In an HLA program, all identifiers must be unique within a given *scope*. Therefore, all global names must be unique with respect to one another. Similarly, all local variables within a given procedure must have unique names *but only with respect to other local symbols in that procedure*. In particular, a local name may be the same as a global name. When this occurs, HLA creates two separate variables for the two objects. Within the scope of the procedure any reference to the common name accesses the local variable; outside that procedure, any reference to the common name references the global identifier. Although the quality of the resultant code is questionable, it is perfectly legal to have a global identifier named *MyVar* with the same local name in two or more different procedures. The procedures each have their own local variant of the object which is independent of *MyVar* in the main program. Program 8.7 provides an example of an HLA program that demonstrates this feature.

```

program demoLocalVars2;
#include( "stdlib.hhf" );

static
    i: uns32 := 10;
    j: uns32 := 20;

// The following procedure declares "i" and "j"
// as local variables, so it does not have access
// to the global variables by the same name.

```

5. Strictly speaking, this is not true. The chapter on Advanced Procedures will present an exception.

```
procedure First;
var
    i: int32;
    j: uns32;

begin First;

    mov( 10, j );
    for( mov( 0, i ); i < 10; inc( i ) ) do

        stdout.put( "i=", i, " j=", j, nl );
        dec( j );

    endfor;

end First;

// This procedure declares only an "i" variable.
// It cannot access the value of the global "i"
// variable but it can access the value of the
// global "j" object since it does not provide
// a local variant of "j".

procedure Second;
var
    i: uns32;

begin Second;

    mov( 10, j );
    for( mov( 0, i ); i < 10; inc( i ) ) do

        stdout.put( "i=", i, " j=", j, nl );
        dec( j );

    endfor;

end Second;

begin demoLocalVars2;

    First();
    Second();

    // Since the calls to First and Second have not
    // modified variable "i", the following statement
    // should print "i=10". However, since the Second
    // procedure manipulated global variable "j", this
    // code will print "j=0" rather than "j=20".

    stdout.put( "i=", i, " j=", j, nl );

end demoLocalVars2;
```

Program 8.7 Local Variables Need Not Have Globally Unique Names

There are good and bad points to be made about reusing global names within a procedure. On the one hand, there is the potential for confusion. If you use a name like *ProfitsThisYear* as a global symbol and you reuse that name within a procedure, someone reading the procedure might think that the procedure refers to the global symbol rather than the local symbol. On the other hand, simple names like *i*, *j*, and *k* are nearly meaningless (almost everyone expects the program to use them as loop control variables or for other local uses), so reusing these names as local objects is probably a good idea. From a software engineering perspective, it is probably a good idea to keep all variables names that have a very specific meaning (like *ProfitsThisYear*) unique throughout your program. General names, that have a nebulous meaning (like *index*, *counter*, and names like *i*, *j*, or *k*) will probably be okay to reuse as global variables

There is one last point to make about the scope of identifiers in an HLA program: variables in separate procedures (that is, two procedures where one procedure is not declared in the declaration section of the second procedure) are separate, even if they have the same name. The *First* and *Second* procedures in Program 8.7, for example, share the same name (*i*) for a local variable. However, the *i* in *First* is a completely different variable than the *i* in *Second*.

The second major attribute that differentiates (certain) local variables from global variables is *lifetime*. The lifetime of a variable spans from the point the program first allocates storage for a variable to the point the program deallocates the storage for that variable. Note that lifetime is a dynamic attribute (controlled at run time) whereas scope is a static attribute (controlled at compile time). In particular, a variable can actually have several lifetimes if the program repeatedly allocates and then deallocates the storage for that variable.

Global variables always have a single lifetime that spans from the moment the main program first begins execution to the point the main program terminates. Likewise, all static objects have a single lifetime that spans the execution of the program (remember, static objects are those you declare in the *STATIC*, *READ-ONLY*, or *STORAGE* sections). This is true even for procedures. So there is no difference between the lifetime of a local static object and the lifetime of a global static object. Variables you declare in the *VAR* section, however, are a different matter. *VAR* objects use *automatic storage allocation*. Automatic storage allocation means that the procedure automatically allocates storage for a local variable upon entry into a procedure. Similarly, the program deallocates storage for automatic objects when the procedure returns to its caller. Therefore, the lifetime of an automatic object is from the point the procedure is first called to the point it returns to its caller.

Perhaps the most important thing to note about automatic variables is that you cannot expect them to maintain their values between calls to the procedure. Once the procedure returns to its caller, the storage for the automatic variable is lost and, therefore, the value is lost as well. Therefore, *you must always assume that a local VAR object is uninitialized upon entry into a procedure*; even if you know you've called the procedure before and the previous procedure invocation initialized that variable. Whatever value the last call stored into the variable was lost when the procedure returned to its caller. If you need to maintain the value of a variable between calls to a procedure, you should use one of the static variable declaration types.

Given that automatic variables cannot maintain their values across procedure calls, you might wonder why you would want to use them at all. However, there are several benefits to automatic variables that static variables do not have. The biggest disadvantage to static variables is that they consume memory even when the (only) procedure that references them is not running. Automatic variables, on the other hand, only consume storage while there associated procedure is executing. Upon return, the procedure returns any automatic storage it allocated back to the system for reuse by other procedures. You'll see some additional advantages to automatic variables later in this chapter.

8.6 Other Local and Global Symbol Types

As mentioned in the previous section, HLA lets you declare constants, values, types, and anything else legal in the main program's declaration section within a procedure's declaration section. The same rules for scope apply to these identifiers. Therefore, you can reuse constant names, procedure names, type names, etc. in local declarations (although this is almost always a bad idea).

Referencing global constants, values, and types, does not present the same software engineering problems that occur when you reference global variables. The problem with referencing global variable is that a procedure can change the value of a global variable in a non-obvious way. This makes programs more difficult to read, understand, and maintain since you can't often tell that a procedure is modifying memory by looking only at the call to that procedure. Constants, values, types, and other non-variable objects, don't suffer from this problem because you cannot change them at run-time. Therefore, the pressure to avoid global objects at nearly all costs doesn't apply to non-variable objects.

Having said that it's okay to access global constants, types, etc., it's also worth pointing out that you should declare these objects locally within a procedure if the only place your program references such objects is within that procedure. Doing so will make your programs a little easier to read since the person reading your code won't have to search all over the place for the symbol's definition.

8.7 Parameters

Although there is a large class of procedures that are totally self-contained, most procedures require some input data and return some data to the caller. Parameters are values that you pass to and from a procedure. In straight assembly language, passing parameters can be a real chore. Fortunately, HLA provides a HLL-like syntax for procedure declarations and for procedure calls involving parameters. This chapter will present HLA's HLL parameter syntax. Later chapters on Intermediate Procedures and Advanced Procedures will deal with the low-level mechanisms for passing parameters in pure assembly code.

The first thing to consider when discussing parameters is *how* we pass them to a procedure. If you are familiar with Pascal or C/C++ you've probably seen two ways to pass parameters: pass by value and pass by reference. HLA certainly supports these two parameter passing mechanisms. However, HLA also supports pass by value/result, pass by result, pass by name, and pass by lazy evaluation. Of course, HLA is assembly language so it is possible to pass parameters in HLA using any scheme you can dream up (at least, any scheme that is possible at all on the CPU). However, HLA provides special HLL syntax for pass by value, reference, value/result, result, name, and lazy evaluation.

Because pass by value/result, result, name, and lazy evaluation are somewhat advanced, this chapter will not deal with those parameter passing mechanisms. If you're interested in learning more about these parameter passing schemes, see the chapters on Intermediate and Advanced Procedures.

Another concern you will face when dealing with parameters is *where* you pass them. There are lots of different places to pass parameters; the chapter on Intermediate Procedures will consider these places in greater detail. In this chapter, since we're using HLA's HLL syntax for declaring and calling procedures, we'll wind up passing procedure parameters on the stack. You don't really need to concern yourself with the details since HLA abstracts them away for you; however, do keep in mind that procedure calls and procedure parameters make use of the stack. Therefore, something you push on the stack immediately before a procedure call is not going to be immediately on the top of the stack upon entry into the procedure.

8.7.1 Pass by Value

A parameter passed by value is just that – the caller passes a value to the procedure. Pass by value parameters are input-only parameters. That is, you can pass them to a procedure but the procedure cannot return them. In HLA the idea of a pass by value parameter being an input only parameter makes a lot of sense. Given the HLA procedure call:

```
CallProc(I);
```

If you pass *I* by value, then *CallProc* does not change the value of *I*, regardless of what happens to the parameter inside *CallProc*.

Since you must pass a copy of the data to the procedure, you should only use this method for passing small objects like bytes, words, and double words. Passing arrays and records by value is very inefficient (since you must create and pass a copy of the object to the procedure).

HLA, like Pascal and C/C++, passes parameters by value unless you specify otherwise. Here's what a typical function looks like with a single pass by value parameter:

```
procedure PrintNSpaces( N:uns32 );
begin PrintNSpaces;

    push( ecx );
    mov( N, ecx );
    repeat

        stdout.put( ' ' ); // Print 1 of N spaces.
        dec( ecx );       // Count off N spaces.

    until( ecx = 0 );
    pop( ecx );

end PrintNSpaces;
```

The parameter *N* in *PrintNSpaces* is known as a formal parameter. Anywhere the name *N* appears in the body of the procedure the program references the value passed through *N* by the caller.

The calling sequence for *PrintNSpaces* can be any of the following:

```
PrintNSpaces( constant );
PrintNSpaces( reg32 );
PrintNSpaces( uns32_variable );
```

Here are some concrete examples of calls to *PrintNSpaces*:

```
PrintNSpaces( 40 );
PrintNSpaces( EAX );
PrintNSpaces( SpacesToPrint );
```

The parameter in the calls to *PrintNSpaces* is known as an *actual parameter*. In the examples above, 40, EAX, and *SpacesToPrint* are the actual parameters.

Note that pass by value parameters behave exactly like local variables you declare in the VAR section with the single exception that the procedure's caller initializes these local variables before it passes control to the procedure.

HLA uses positional parameter notation just like most high level languages. Therefore, if you need to pass more than one parameter, HLA will associate the actual parameters with the formal parameters by their position in the parameter list. The following *PrintNChars* procedure demonstrates a simple procedure that has two parameters.

```
procedure PrintNChars( N:uns32; c:char );
begin PrintNChars;

    push( ecx );
    mov( N, ecx );
    repeat

        stdout.put( c ); // Print 1 of N characters.
        dec( ecx );     // Count off N characters.

    until( ecx = 0 );
    pop( ecx );

end PrintNChars;
```

The following is an invocation of the `PrintNChars` procedure that will print 20 asterisk characters:

```
PrintNChars( 20, '*' );
```

Note that HLA uses semicolons to separate the formal parameters in the procedure declaration and it uses commas to separate the actual parameters in the procedure invocation (Pascal programmers should be comfortable with this notation). Also note that each HLA formal parameter declaration takes the following form:

$$\textit{parameter_identifier} : \textit{type_identifier}$$

In particular, note that the parameter type has to be an identifier. None of the following are legal parameter declarations because the data type is not a single identifier:

```
PtrVar: pointer to uns32
ArrayVar: uns32[10]
recordVar: record i:int32; u:uns32; endrecord
DynArray: array.dArray( uns32, 2 )
```

However, don't get the impression that you cannot pass pointer, array, record, or dynamic array variables as parameters. The trick is to declare a data type for each of these types in the `TYPE` section. Then you can use a single identifier as the type in the parameter declaration. The following code fragment demonstrates how to do this with the four data types above:

```
type
  uPtr:      pointer to uns32;
  uArray10:  uns32[10];
  recType:   record i:int32; u:uns32; endrecord
  dType:     array.dArray( uns32, 2 );

procedure FancyParms
(
  PtrVar: uPtr;
  ArrayVar:uArray10;
  recordVar:recType;
  DynArray: dtype
);
begin FancyParms;
.
.
.
end FancyParms;
```

By default, HLA assumes that you intend to pass a parameter by value. HLA also lets you explicitly state that a parameter is a value parameter by prefacing the formal parameter declaration with the `VAL` keyword. The following is a version of the `PrintNSpaces` procedure that explicitly states that `N` is a pass by value parameter:

```
procedure PrintNSpaces( val N:uns32 );
begin PrintNSpaces;

  push( ecx );
  mov( N, ecx );
  repeat

    stdout.put( ' ' ); // Print 1 of N spaces.
    dec( ecx );       // Count off N spaces.

  until( ecx = 0 );
  pop( ecx );

end PrintNSpaces;
```

Explicitly stating that a parameter is a pass by value parameter is a good idea if you have multiple parameters in the same procedure declaration that use different passing mechanisms.

When you pass a parameter by value and call the procedure using the HLA high level language syntax, HLA will automatically generate code that will make a copy of the actual parameter's value and copy this data into the local storage for that parameter (i.e., the formal parameter). For small objects pass by value is probably the most efficient way to pass a parameter. For large objects, however, HLA must generate code that copies each and every byte of the actual parameter into the formal parameter. For large arrays and records this can be a very expensive operation⁶. Unless you have specific semantic concerns that require you to pass an array or record by value, you should use pass by reference or some other parameter passing mechanism for arrays and records.

When passing parameters to a procedure, HLA checks the type of each actual parameter and compares this type to the corresponding formal parameter. If the types do not agree, HLA then checks to see if either the actual or formal parameter is a byte, word, or dword object and the other parameter is one, two, or four bytes in length (respectively). If the actual parameter does not satisfy either of these conditions, HLA reports a parameter type mismatch error. If, for some reason, you need to pass a parameter to a procedure using a different type than the procedure calls for, you can always use the HLA type coercion operator to override the type of the actual parameter.

8.7.2 Pass by Reference

To pass a parameter by reference, you must pass the address of a variable rather than its value. In other words, you must pass a pointer to the data. The procedure must dereference this pointer to access the data. Passing parameters by reference is useful when you must modify the actual parameter or when you pass large data structures between procedures.

To declare a pass by reference parameter you must preface the formal parameter declaration with the VAR keyword. The following code fragment demonstrates this:

```
procedure UsePassByReference( var PBRvar: int32 );
begin UsePassByReference;
.
.
.
end UsePassByReference;
```

Calling a procedure with a pass by reference parameter uses the same syntax as pass by value except that the parameter has to be a memory location; it cannot be a constant or a register. Furthermore, the type of the memory location must exactly match the type of the formal parameter. The following are legal calls to the procedure above (assuming *i32* is an *int32* variable):

```
UsePassByReference( i32 );
UsePassByReference( (type int32 [ebx] ) );
```

The following are all illegal *UsePassbyReference* invocations (assumption: *charVar* is of type *char*):

```
UsePassByReference( 40 );           // Constants are illegal.
UsePassByReference( EAX );         // Bare registers are illegal.
UsePassByReference( charVar );    // Actual parameter type must match
// the formal parameter type.
```

Unlike the high level languages Pascal and C++, HLA does not completely hide the fact that you are passing a pointer rather than a value. In a procedure invocation, HLA will automatically compute the

6. Note to C/C++ programmers: HLA does not automatically pass arrays by reference. If you specify an array type as a formal parameter, HLA will emit code that makes a copy of each and every byte of that array when you call the associated procedure.

address of a variable and pass that address to the procedure. Within the procedure itself, however, you cannot treat the variable like a value parameter (as you could in most HLLs). Instead, you treat the parameter as a dword variable containing a pointer to the specified data. You must explicitly dereference this pointer when accessing the parameter's value. The following example provides a simple demonstration of this:

```

program PassByRefDemo;
#include( "stdlib.hhf" );

var
  i: int32;
  j: int32;

procedure pbr( var a:int32; var b:int32 );
const
  aa: text := "(type int32 [ebx])";
  bb: text := "(type int32 [ebx])";

begin pbr;

  push( eax );
  push( ebx );          // Need to use EBX to dereference a and b.

  // a = -1;

  mov( a, ebx );       // Get ptr to the "a" variable.
  mov( -1, aa );       // Store -1 into the "a" parameter.

  // b = -2;

  mov( b, ebx );       // Get ptr to the "b" variable.
  mov( -2, bb );       // Store -2 into the "b" parameter.

  // Print the sum of a+b.
  // Note that ebx currently contains a pointer to "b".

  mov( bb, eax );
  mov( a, ebx );       // Get ptr to "a" variable.
  add( aa, eax );
  stdout.put( "a+b=", (type int32 eax), nl );

end pbr;

begin PassByRefDemo;

  // Give i and j some initial values so
  // we can see that pass by reference will
  // overwrite these values.

  mov( 50, i );
  mov( 25, j );

  // Call pbr passing i and j by reference

  pbr( i, j );

  // Display the results returned by pbr.

  stdout.put
  (

```

```

        "i= ", i, nl,
        "j= ", j, nl
    );

end PassByRefDemo;

```

Program 8.8 Accessing Pass by Reference Parameters

Passing parameters by reference can produce some peculiar results in some rare circumstances. Consider the *pbr* procedure in Program 8.8. Were you to modify the call in the main program to be “pbr(i,i)” rather than “pbr(i,j);” the program would produce the following non-intuitive output:

```

a+b=-4
i= -2;
j= 25;

```

The reason this code displays “a+b=-4” rather than the expected “a+b=-3” is because the “pbr(i,i);” call passes the same actual parameter for *a* and *b*. As a result, the *a* and *b* reference parameters both contain a pointer to the same memory location- that of the variable *i*. In this case, *a* and *b* are *aliases* of one another. Therefore, when the code stores -2 at the location pointed at by *b*, it overwrites the -1 stored earlier at the location pointed at by *a*. When the program fetches the value pointed at by *a* and *b* to compute their sum, both *a* and *b* point at the same value, which is -2. Summing -2 + -2 produces the -4 result that the program displays. This non-intuitive behavior is possible anytime you encounter aliases in a program. Passing the same variable as two different parameters probably isn’t very common. But you could also create an alias if a procedure references a global variable and you pass that same global variable by reference to the procedure (this is a good example of yet one more reason why you should avoid referencing global variables in a procedure).

Pass by reference is usually less efficient than pass by value. You must dereference all pass by reference parameters on each access; this is slower than simply using a value since it typically requires at least two instructions. However, when passing a large data structure, pass by reference is faster because you do not have to copy a large data structure before calling the procedure. Of course, you’d probably need to access elements of that large data structure (e.g., an array) using a pointer, so very little efficiency is lost when you pass large arrays by reference.

8.8 Functions and Function Results

Functions are procedures that return a result. In assembly language, there are very few syntactical differences between a procedure and a function which is why HLA doesn’t provide a specific declaration for a function. Nevertheless, although there is very little *syntactical* difference between assembly procedures and functions, there are considerable *semantic* differences. That is, although you can declare them the same way in HLA, you use them differently.

Procedures are a sequence of machine instructions that fulfill some activity. The end result of the execution of a procedure is the accomplishment of that activity. Functions, on the other hand, execute a sequence of machine instructions specifically to compute some value to return to the caller. Of course, a function can perform some activity as well and procedures can undoubtedly compute some values, but the main difference is that the purpose of a function is to return some computed result; procedures don’t have this requirement.

A good example of a procedure is the *stdout.puti32* procedure. This procedure requires a single *int32* parameter. The purpose of this procedure is to print the decimal conversion of this integer value to the standard output device. Note that *stdout.puti32* doesn’t return any kind of value that is usable by the calling program.

A good example of a function is the *cs.member* function. This function expects two parameters: the first is a character value and the second is a character set value. This function returns true (1) in EAX if the character is a member of the specified character set. It returns false if the character parameter is not a member of the character set.

Logically, the fact that *cs.member* returns a usable value to the calling code (in EAX) while *std-out.puti32* does not is a good example of the main difference between a function and a procedure. So, in general, a procedure becomes a function by virtue of the fact that you explicitly decide to return a value somewhere upon procedure return. No special syntax is needed to declare and use a function. You still write the code as a procedure.

8.8.1 Returning Function Results

The 80x86's registers are the most popular place to return function results. The *cs.member* routine in the HLA Standard Library is a good example of a function that returns a value in one of the CPU's registers. It returns true (1) or false (0) in the EAX register. By convention, programmers try to return eight, sixteen, and thirty-two bit (non-real) results in the AL, AX, and EAX registers, respectively⁷. For example, this is where most high level languages return these types of results.

Of course, there is nothing particularly sacred about the AL/AX/EAX register. You could return function results in any register if it is more convenient to do so. However, if you don't have a good reason for not using AL/AX/EAX, then you should follow the convention. Doing so will help others understand your code better since they will generally assume that your functions return small results in the AL/AX/EAX register set.

If you need to return a function result that is larger than 32 bits, you obviously must return it somewhere besides in EAX (which can hold values 32 bits or less). For values slightly larger than 32 bits (e.g., 64 bits or maybe even as many as 128 bits) you can split the result into pieces and return those parts in two or more registers. For example, it is very common to see programs returning 64-bit values in the EDX:EAX register pair (e.g., the HLA Standard Library *stdin.geti64* function returns a 64-bit integer in the EDX:EAX register pair).

If you need to return a really large object as a function result, say an array of 1,000 elements, you obviously are not going to be able to return the function result in the registers. There are two common ways to deal with really large function return results: either pass the return value as a reference parameter or allocate storage on the heap (using *malloc*) for the object and return a pointer to it in a 32-bit register. Of course, if you return a pointer to storage you've allocated on the heap, the calling program must free this storage when it is done with it.

8.8.2 Instruction Composition in HLA

Several HLA Standard Library functions allow you to call them as operands of other instructions. For example, consider the following code fragment:

```
if( cs.member( al, {'a'..'z'}) ) then
    .
    .
    .
endif;
```

As your high level language experience (and HLA experience) should suggest, this code calls the *cs.member* function to check to see if the character in AL is a lower case alphabetic character. If the *cs.member* function returns true then this code fragment executes the then section of the IF statement; however, if *cs.member* returns false, this code fragment skips the IF.THEN body. There is nothing spectacular here

7. In the next chapter you'll see where most programmers return real results.

except for the fact that HLA doesn't support function calls as boolean expressions in the IF statement (look back at Chapter Two in Volume One to see the complete set of allowable expressions). How then, does this program compile and run producing the intuitive results?

The very next section will describe how you can tell HLA that you want to use a function call in a boolean expression. However, to understand how this works, you need to first learn about *instruction composition* in HLA.

Instruction composition lets you use one instruction as the operand of another. For example, consider the MOV instruction. It has two operands, a source operand and a destination operand. Instruction composition lets you substitute a valid 80x86 machine instruction for either (or both) operands. The following is a simple example:

```
mov( mov( 0, eax ), ebx );
```

Of course the immediate question is “what does this mean?” To understand what is going on, you must first realize that most instructions “return” a value to the compiler while they are being compiled. For most instructions, the value they “return” is their destination operand. Therefore, “mov(0, eax);” returns the string “eax” to the compiler during compilation since EAX is the destination operand. Most of the time, specifically when an instruction appears on a line by itself, the compiler ignores the string result the instruction returns. However, HLA uses this string result whenever you supply an instruction in place of some operand; specifically, HLA uses that string in place of the instruction as the operand. Therefore, the MOV instruction above is equivalent to the following two instruction sequence:

```
mov( 0, eax );    // HLA compiles interior instructions first.
mov( eax, ebx );
```

When processing composed instructions (that is, instruction sequences that have other instructions as operands), HLA always works in an “left-to-right then depth-first (inside-out)” manner. To make sense of this, consider the following instructions:

```
add( sub( mov( i, eax ), mov( j, ebx ) ), mov( k, ecx ) );
```

To interpret what is happening here, begin with the source operand. It consists of the following:

```
sub( mov( i, eax ), mov( j, ebx ) )
```

The source operand for this instruction is “mov(i, eax)” and this instruction does not have any composition, so HLA emits this instruction and returns its destination operand (EAX) for use as the source to the SUB instruction. This effectively gives us the following:

```
sub( eax, mov( j, ebx ) )
```

Now HLA compiles the instruction that appears as the destination operand (“mov(j, ebx)”) and returns its destination operand (EBX) to substitute for this MOV in the SUB instruction. This yields the following:

```
sub( eax, ebx )
```

This is a complete instruction, without composition, that HLA can compile. So it compiles this instruction and returns its destination operand (EBX) as the string result to substitute for the SUB in the original ADD instruction. So the original ADD instruction now becomes:

```
add( ebx, mov( i, ecx ) );
```

HLA next compiles the MOV instruction appearing in the destination operand. It returns its destination operand as a string that HLA substitutes for the MOV, finally yielding the simple instruction:

```
add( ebx, ecx );
```

The compilation of the original ADD instruction, therefore, yields the following instruction sequence:

```
mov( i, eax );
mov( j, ebx );
sub( eax, ebx );
```

```
mov( k, ecx );
add( ebx, ecx );
```

Whew! It's rather difficult to look at the original instruction and easily see that this sequence is the result. As you can easily see in this example, *overzealous use of instruction composition can produce nearly unreadable programs*. You should be very careful about using instruction composition in your programs. With only a few exceptions, writing a composed instruction sequence makes your program harder to read.

Note that the excessive use of instruction composition may make errors in your program difficult to decipher. Consider the following HLA statement:

```
add( mov( eax, i ), mov( ebx, j ) );
```

This instruction composition yields the 80x86 instruction sequence:

```
mov( eax, i );
mov( ebx, j );
add( i, j );
```

Of course, the compiler will complain that you're attempting to add one memory location to another. However, the instruction composition effectively masks this fact and makes it difficult to comprehend the cause of the error message. Moral of the story: avoid using instruction composition unless it really makes your program easier to read. The few examples in this section demonstrate how *not* to use instruction composition.

There are two main areas where using instruction composition can help make your programs more readable. The first is in HLA's high level language control structures. The other is in procedure parameters. Although instruction composition is useful in these two cases (and probably a few others as well), this doesn't give you a license to use extremely convoluted instructions like the ADD instruction in the previous example. Instead, most of the time you will use a single instruction or a function call in place of a single operand in a high level language boolean expression or in a procedure/function parameter.

While we're on the subject, exactly what does a procedure call return as the string that HLA substitutes for the call in an instruction composition? For that matter, what do statements like IF..ENDIF return? How about instructions that don't have a destination operand? Well, function return results are the subject of the very next section so you'll read about that in a few moments. As for all the other statements and instructions, you should check out the HLA reference manual. It lists each instruction and its "RETURNS" value. The "RETURNS" value is the string that HLA will substitute for the instruction when it appears as the operand to another instruction. Note that many HLA statements and instructions return the empty string as their "RETURNS" value (by default, so do procedure calls). If an instruction returns the empty string as its composition value, then HLA will report an error if you attempt to use it as the operand of another instruction. For example, the IF..ENDIF statement returns the empty string as its "RETURNS" value, so you may not bury an IF..ENDIF inside another instruction.

8.8.3 The HLA RETURNS Option in Procedures

HLA procedure declarations allow a special option that specifies the string to use when a procedure invocation appears as the operand of another instruction: the RETURNS option. The syntax for a procedure declaration with the RETURNS option is as follows:

```
procedure ProcName ( optional parameters ); RETURNS( string_constant );
  << Local declarations >>
begin ProcName;
  << procedure statements >>
end ProcName;
```

If the RETURNS option is not present, HLA associates the empty string with the RETURNS value for the procedure. This effectively makes it illegal to use that procedure invocation as the operand to another instruction.

The RETURNS option requires a single string parameter surrounded by parentheses. This must be a string constant⁸. HLA will substitute this string constant for the procedure call if it ever appears as the operand of another instruction. Typically this string constant is a register name; however, any text that would be legal as an instruction operand is okay here. For example, you could specify memory address or constants. For purposes of clarity, you should always specify the location of a function's return value in the RETURNS parameter.

As an example, consider the following boolean function that returns true or false in the EAX register if the single character parameter is an alphabetic character⁹:

```
procedure IsAlphabeticChar( c:char ); RETURNS( "EAX" );
begin IsAlphabeticChar;

    // Note that cs.member returns true/false in EAX

    cs.member( c, { 'a'..'z', 'A'..'Z' } );

end IsAlphabeticChar;
```

Once you tack the RETURNS option on the end of this procedure declaration you can legally use a call to *IsAlphabeticChar* as an operand to other HLA statements and instructions:

```
mov( IsAlphabeticChar( al ), EBX );
.
.
.
if( IsAlphabeticChar( ch ) ) then
.
.
.
endif;
```

The last example above demonstrates that, via the RETURNS option, you can embed calls to your own functions in the boolean expression field of various HLA statements. Note that the code above is equivalent to

```
IsAlphabeticChar( ch );
if( EAX ) then
.
.
.
endif;
```

Not all HLA high level language statements expand composed instructions before the statement. For example, consider the following WHILE statement:

```
while( IsAlphabeticChar( ch ) ) do
.
.
.
endwhile;
```

This code does not expand to the following:

```
IsAlphabeticChar( ch );
while( EAX ) do
.
.
.
```

8. Do note, however, that it doesn't have to be a string literal constant. A CONST string identifier or even a constant string expression is legal here.

9. Before you run off and actually use this function in your own programs, note that the HLA Standard Library provides the *char.isAlpha* function that provides this test. See the HLA documentation for more details.

```
endwhile;
```

Instead, the call to *IsAlphabeticChar* expands inside the WHILE's boolean expression so that the program calls this function on each iteration of the loop.

You should exercise caution when entering the RETURNS parameter. HLA does not check the syntax of the string parameter when it is compiling the procedure declaration (other than to verify that it is a string constant). Instead, HLA checks the syntax when it replaces the function call with the RETURNS string. So if you had specified "EAZ" instead of "EAX" as the RETURNS parameter for *IsAlphabeticChar* in the previous examples, HLA would not have reported an error until you actually used *IsAlphabeticChar* as an operand. Then of course, HLA complains about the illegal operand and it's not at all clear what the problem is by looking at the *IsAlphabeticChar* invocation. So take special care not to introduce typographical errors in the RETURNS string; figuring out such errors later can be very difficult.

8.9 Side Effects

A *side effect* is any computation or operation by a procedure that isn't the primary purpose of that procedure. For example, if you elect not to preserve all affected registers within a procedure, the modification of those registers is a side effect of that procedure. Side effect programming, that is, the practice of using a procedure's side effects, is very dangerous. All too often a programmer will rely on a side effect of a procedure. Later modifications may change the side effect, invalidating all code relying on that side effect. This can make your programs hard to debug and maintain. Therefore, you should avoid side effect programming.

Perhaps some examples of side effect programming will help enlighten you to the difficulties you may encounter. The following procedure zeros out an array. For efficiency reasons, it makes the caller responsible for preserving necessary registers. As a result, one side effect of this procedure is that the EBX and ECX registers are modified. In particular, the ECX register contains zero upon return.

```
procedure ClrArray;
begin ClrArray;

    lea( ebx, array );
    mov( 32, ecx );
    while( ecx > 0 ) do

        mov( 0, (type dword [ebx]));
        add( 4, ebx );
        dec( ecx );

    endwhile;

end ClrArray;
```

If your code expects ECX to contain zero after the execution of this subroutine, you would be relying on a side effect of the *ClrArray* procedure. The main purpose behind this code is zeroing out an array, not setting the ECX register to zero. Later, if you modify the *ClrArray* procedure to the following, your code that depends upon ECX containing zero would no longer work properly:

```
procedure ClrArray;
begin ClrArray;

    mov( 0, ebx );
    while( ebx < 32 ) do

        mov( 0, array[ebx*4] );
        inc( ebx );

    endwhile;
```

```

endwhile;

end ClrArray;

```

So how can you avoid the pitfalls of side effect programming in your procedures? By carefully structuring your code and paying close attention to exactly how your calling code and the subservient procedures interface with one another. These rules can help you avoid problems with side effect programming:

- Always properly document the input and output conditions of a procedure. Never rely on any other entry or exit conditions other than these documented operations.
- Partition your procedures so that they compute a single value or execute a single operation. Subroutines that do two or more tasks are, by definition, producing side effects unless every invocation of that subroutine requires all the computations and operations.
- When updating the code in a procedure, make sure that it still obeys the entry and exit conditions. If not, either modify the program so that it does or update the documentation for that procedure to reflect the new entry and exit conditions.
- Avoid passing information between routines in the CPU's flag register. Passing an error status in the carry flag is about as far as you should ever go. Too many instructions affect the flags and it's too easy to foul up a return sequence so that an important flag is modified on return.
- Always save and restore all registers a procedure modifies.
- Avoid passing parameters and function results in global variables.
- Avoid passing parameters by reference (with the intent of modifying them for use by the calling code).

These rules, like all other rules, were meant to be broken. Good programming practices are often sacrificed on the altar of efficiency. There is nothing wrong with breaking these rules as often as you feel necessary. However, your code will be difficult to debug and maintain if you violate these rules often. But such is the price of efficiency¹⁰. Until you gain enough experience to make a judicious choice about the use of side effects in your programs, you should avoid them. More often than not, the use of a side effect will cause more problems than it solves.

8.10 Recursion

Recursion occurs when a procedure calls itself. The following, for example, is a recursive procedure:

```

procedure Recursive;
begin Recursive;

    Recursive();

end Recursive;

```

Of course, the CPU will never return from this procedure. Upon entry into *Recursive*, this procedure will immediately call itself again and control will never pass to the end of the procedure. In this particular case, run away recursion results in an infinite loop¹¹.

Like a looping structure, recursion requires a termination condition in order to stop infinite recursion. *Recursive* could be rewritten with a termination condition as follows:

```

procedure Recursive;
begin Recursive;

    dec( eax );
    if( @nz ) then

```

10. This is not just a snide remark. Expert programmers who have to wring the last bit of performance out of a section of code often resort to poor programming practices in order to achieve their goals. They are prepared, however, to deal with the problems that are often encountered in such situations and they are a lot more careful when dealing with such code.

11. Well, not really infinite. The stack will overflow and Windows or Linux will raise an exception at that point.

```

        Recursive();

    endif;

end Recursive;

```

This modification to the routine causes *Recursive* to call itself the number of times appearing in the EAX register. On each call, *Recursive* decrements the EAX register by one and calls itself again. Eventually, *Recursive* decrements EAX to zero and returns. Once this happens, each successive call returns back to *Recursive* until control returns to the original call to *Recursive*.

So far, however, there hasn't been a real need for recursion. After all, you could efficiently code this procedure as follows:

```

procedure Recursive;
begin Recursive;

    repeat

        dec( eax );

    until( @z );

end Recursive;

```

Both examples would repeat the body of the procedure the number of times passed in the EAX register¹². As it turns out, there are only a few recursive algorithms that you cannot implement in an iterative fashion. However, many recursively implemented algorithms are more efficient than their iterative counterparts and most of the time the recursive form of the algorithm is much easier to understand.

The quicksort algorithm is probably the most famous algorithm that usually appears in recursive form (see a "Data Structures and Algorithms" textbook for a discussion of this algorithm). An HLA implementation of this algorithm follows:

```

program QSDemo;
#include( "stdlib.hhf" );

type
    ArrayType:  uns32[ 10 ];

static
    theArray:   ArrayType := [1,10,2,9,3,8,4,7,5,6];

procedure quicksort( var a:ArrayType; Low:int32; High: int32 );
const
    i:          text := "(type int32 edi)";
    j:          text := "(type int32 esi)";
    Middle:     text := "(type uns32 edx)";
    ary:        text := "[ebx]";

begin quicksort;

    push( eax );
    push( ebx );
    push( ecx );
    push( edx );

```

12. Although the latter version will do it considerably faster since it doesn't have the overhead of the CALL/RET instructions.

```

push( esi );
push( edi );

mov( a, ebx );      // Load BASE address of "a" into EBX

mov( Low, edi);     // i := Low;
mov( High, esi );  // j := High;

// Compute a pivotal element by selecting the
// physical middle element of the array.

mov( i, eax );
add( j, eax );
shr( 1, eax );
mov( ary[eax*4], Middle ); // Put middle value in EDX

// Repeat until the EDI and ESI indicies cross one
// another (EDI works from the start towards the end
// of the array, ESI works from the end towards the
// start of the array).

repeat

    // Scan from the start of the array forward
    // looking for the first element greater or equal
    // to the middle element).

    while( Middle > ary[i*4] ) do

        inc( i );

    endwhile;

    // Scan from the end of the array backwards looking
    // for the first element that is less than or equal
    // to the middle element.

    while( Middle < ary[j*4] ) do

        dec( j );

    endwhile;

    // If we've stopped before the two pointers have
    // passed over one another, then we've got two
    // elements that are out of order with respect
    // to the middle element. So swap these two elements.

    if( i <= j ) then

        mov( ary[i*4], eax );
        mov( ary[j*4], ecx );
        mov( eax, ary[j*4] );
        mov( ecx, ary[i*4] );
        inc( i );
        dec( j );

    endif;

until( i > j );

```

```

// We have just placed all elements in the array in
// their correct positions with respect to the middle
// element of the array. So all elements at indices
// greater than the middle element are also numerically
// greater than this element. Likewise, elements at
// indices less than the middle (pivotal) element are
// now less than that element. Unfortunately, the
// two halves of the array on either side of the pivotal
// element are not yet sorted. Call quicksort recursively
// to sort these two halves if they have more than one
// element in them (if they have zero or one elements, then
// they are already sorted).

if( Low < j ) then

    quicksort( a, Low, j );

endif;
if( i < High ) then

    quicksort( a, i, High );

endif;

pop( edi );
pop( esi );
pop( edx );
pop( ecx );
pop( ebx );
pop( eax );

end quicksort;

begin QSDemo;

stdout.put( "Data before sorting: " nl );
for( mov( 0, ebx ); ebx < 10; inc( ebx ) ) do

    stdout.put( theArray[ebx*4]:5 );

endfor;
stdout.newln();

quicksort( theArray, 0, 9 );

stdout.put( "Data after sorting: " nl );
for( mov( 0, ebx ); ebx < 10; inc( ebx ) ) do

    stdout.put( theArray[ebx*4]:5 );

endfor;
stdout.newln();

end QSDemo;

```

Program 8.9 Recursive Quicksort Program

Note that this quicksort procedure uses registers for all non-parameter local variables. Also note how Quicksort uses TEXT constant definitions to provide more readable names for the registers. This technique can often make an algorithm easier to read; however, one must take care when using this trick not to forget that those registers are being used.

8.11 Forward Procedures

As a general rule HLA requires that you declare all symbols before their first use in a program¹³. Therefore, you must define all procedures before their first call. There are two reasons this isn't always practical: mutual recursion (two procedures call each other) and source code organization (you prefer to place a procedure in your code after the point you've first called it). Fortunately, HLA lets you use a *forward procedure definition* to declare a procedure *prototype*. Forward declarations let you define a procedure before you actually supply the code for that procedure.

A forward procedure declaration is a familiar procedure declaration that uses the reserved word FORWARD in place of the procedure's declaration section and body. The following is a forward declaration for the quicksort procedure appearing in the last section:

```
procedure quicksort( var a:ArrayType; Low:int32; High: int32 ); forward;
```

A forward declaration in an HLA program is a promise to the compiler that the actual procedure declaration will appear, exactly as stated in the forward declaration, at a later point in the source code. "Exactly as stated" means exactly that. The forward declaration must have the same parameters, they must be passed the same way, and they must all have the same types as the formal parameters in the procedure¹⁴.

Routines that are mutually recursive (that is, procedure A calls procedure B and procedure B calls procedure A) require at least one forward declaration since only one of procedure A or B can be declared before the other. In practice, however, mutual recursion (direct or indirect) doesn't occur very frequently, so the need for forward declarations is not that great.

In the absence of mutual recursion, it is always possible to organize your source code so that each procedure declaration appears before its first invocation. What's possible and what's desired are two different things, however. You might want to group a related set of procedures at the beginning of your source code and a different set of procedures towards the end of your source code. This logical grouping, by function rather than by invocation, may make your programs much easier to read and understand. However, this organization may also yield code that attempts to call a procedure before its declaration. No sweat. Just use a forward procedure definition to resolve the problem.

One major difference between the forward definition and the actual procedure declaration has to do with the procedure options. Some options, like RETURNS may appear only in the forward declaration (if a FORWARD declaration is present). Other options may only appear in the actual procedure declaration (we haven't covered any of the other procedure options yet, so don't worry about them just yet). If your procedure requires a RETURNS option, the RETURNS option must appear before the FORWARD reserved word. E.g.,

```
procedure IsItReady( valueToTest: dword ); returns( "EAX" ); forward;
```

The RETURNS option must not also appear in the actual procedure declaration later in your source file.

8.12 Putting It All Together

This chapter has filled in one of the critical elements missing from your assembly language knowledge: how to create user-defined procedures in an HLA program. This chapter discussed HLA's high level proce-

13. There are a few minor exceptions to this rule, but it is certainly true for procedure calls.

14. Actually, "exactly" is too strong a word. You will see some exceptions in a moment.

dure declaration and calling syntax. It also described how to pass parameters by value and by reference as well as the use of local variables in HLA procedures. This chapter also provided information about instruction composition and the RETURNS option for procedures. Finally, this chapter explained recursion and the use of forward procedure declarations (prototypes).

The one thing this chapter did not discuss was how procedures are written in “pure” assembly language. This chapter presents just enough information to let you start using procedures in your HLA programs. The “real stuff” will have to wait for a few chapters. Fear not, however; later chapters will teach you far more than you probably care to know about procedures in assembly language programs.

Managing Large Programs

Chapter Nine

9.1 Chapter Overview

When writing larger HLA programs you do not typically write the whole program as a single source file. This chapter discusses how to break up a large project into smaller pieces and assemble the pieces separately. This radically reduces development time on large projects.

9.2 Managing Large Programs

Most assembly language programs are not totally stand alone programs. In general, you will call various standard library or other routines that are not defined in your main program. For example, you've probably noticed by now that the 80x86 doesn't provide any machine instructions like "read", "write", or "printf" for doing I/O operations. Of course, you can write your own procedures to accomplish this. Unfortunately, writing such routines is a complex task, and beginning assembly language programmers are not ready for such tasks. That's where the HLA Standard Library comes in. This is a package of procedures you can call to perform simple I/O operations like *stdout.put*.

The HLA Standard Library contains tens of thousands of lines of source code. Imagine how difficult programming would be if you had to merge these thousands of lines of code into your simple programs! imagine how slow compiling your programs would be if you had to compile those tens of thousands of lines with each program you write. Fortunately, you don't have to.

For small programs, working with a single source file is fine. For large programs this gets very cumbersome (consider the example above of having to include the entire HLA Standard Library into each of your programs). Furthermore, once you've debugged and tested a large section of your code, continuing to assemble that same code when you make a small change to some other part of your program is a waste of time. The HLA Standard Library, for example, takes several minutes to assemble, even on a fast machine. Imagine having to wait five or ten minutes on a fast Pentium machine to assemble a program to which you've made a one line change!

As with high level languages, the solution is *separate compilation*. First, you break up your large source files into manageable chunks. Then you compile the separate files into object code modules. Finally, you link the object modules together to form a complete program. If you need to make a small change to one of the modules, you only need to reassemble that one module, you do not need to reassemble the entire program.

The HLA Standard Library works in precisely this way. The Standard Library is already compiled and ready to use. You simply call routines in the Standard Library and link your code with the Standard Library using a *linker* program. This saves a tremendous amount of time when developing a program that uses the Standard Library code. Of course, you can easily create your own object modules and link them together with your code. You could even add new routines to the Standard Library so they will be available for use in future programs you write.

"Programming in the large" is a term software engineers have coined to describe the processes, methodologies, and tools for handling the development of large software projects. While everyone has their own idea of what "large" is, separate compilation, and some conventions for using separate compilation, are among the more popular techniques that support "programming in the large." The following sections describe the tools HLA provides for separate compilation and how to effectively employ these tools in your programs.

9.3 The #INCLUDE Directive

The #INCLUDE directive, when encountered in a source file, switches program input from the current file to the file specified in the parameter list of the include directive. This allows you to construct text files containing common constants, types, source code, and other HLA items, and include such a file into the assembly of several separate programs. The syntax for the include directive is

```
#include( "filename" )
```

Filename must be a valid filename. HLA merges the specified file into the compilation at the point of the #INCLUDE directive. Note that you can nest #INCLUDE statements inside files you include. That is, a file being included into another file during assembly may itself include a third file. In fact, the “stdlib.hhf” header file you see in most example programs contains the following¹:

```
#include( "hla.hhf" )
#include( "x86.hhf" )
#include( "misctypes.hhf" )
#include( "hll.hhf" )

#include( "excepts.hhf" )
#include( "memory.hhf" )

#include( "args.hhf" )
#include( "conv.hhf" )
#include( "strings.hhf" )
#include( "cset.hhf" )
#include( "patterns.hhf" )
#include( "tables.hhf" )
#include( "arrays.hhf" )
#include( "chars.hhf" )

#include( "math.hhf" )
#include( "rand.hhf" )

#include( "stdio.hhf" )
#include( "stdin.hhf" )
#include( "stdout.hhf" )
```

Program 9.1 The stdlib.hhf Header File, as of 01/01/2000

By including “stdlib.hhf” in your source code, you automatically include all the HLA library modules. It’s often more efficient (in terms of compile time and size of code generated) to provide only those #INCLUDE statements for the modules you actually need in your program. However, including “stdlib.hhf” is extremely convenient and takes up less space in this text, which is why most programs appearing in this text use “stdlib.hhf”.

Note that the #INCLUDE directive does not need to end with a semicolon. If you put a semicolon after the #INCLUDE, that semicolon becomes part of the source file and is the first character following the included file during compilation. HLA generally allows spare semicolons in various parts of the program, so you will often see a #INCLUDE statement ending with a semicolon that produces no harm. In general,

1. Note that this file changes over time as new library modules appear in the HLA Standard Library, so this file is probably not up to date. Furthermore, there are some minor differences between the Linux and Windows version of this file. The OS-specific entries do not appear in this example.

though, you should not get in the habit of putting semicolons after `#INCLUDE` statements because there is the slight possibility this could create a syntax error in certain circumstances.

Using the `#include` directive by itself does not provide separate compilation. You *could* use the `include` directive to break up a large source file into separate modules and join these modules together when you compile your file. The following example would include the `PRINTF.HLA` and `PUTC.HLA` files during the compilation of your program:

```
#include( "printf.hla" )
#include( "putc.hla" )
```

Now your program *will* benefit from the modularity gained by this approach. Alas, you will not save any development time. The `#INCLUDE` directive inserts the source file at the point of the `#INCLUDE` during compilation, exactly as though you had typed that code in yourself. HLA still has to compile the code and that takes time. Were you to include all the files for the Standard Library routines in this manner, your compilations would take *forever*.

In general, you should *not* use the `include` directive to include source code as shown above². Instead, you should use the `#INCLUDE` directive to insert a common set of constants, types, external procedure declarations, and other such items into a program. Typically an assembly language include file does *not* contain any machine code (outside of a macro, see the chapter on Macros and the Compile-Time Language for details). The purpose of using `#INCLUDE` files in this manner will become clearer after you see how the external declarations work.

9.4 Ignoring Duplicate Include Operations

As you begin to develop sophisticated modules and libraries, you eventually discover a big problem: some header files will need to include other header files (e.g., the `stdlib.hhf` header file includes all the other Standard Library Header files). Well, this isn't actually a big problem, but a problem will occur when one header file includes another, and that second header file includes another, and that third header file includes another, and ..., and that last header file includes the first header file. Now *this* is a big problem.

There are two problems with a header file indirectly including itself. First, this creates an infinite loop in the compiler. The compiler will happily go on about its business including all these files over and over again until it runs out of memory or some other error occurs. Clearly this is not a good thing. The second problem that occurs (usually before the problem above) is that the second time HLA includes a header file, it starts complaining bitterly about duplicate symbol definitions. After all, the first time it reads the header file it processes all the declarations in that file, the second time around it views all those symbols as duplicate symbols.

HLA provides a special include directive that eliminates this problem: `#INCLUDEONCE`. You use this directive exactly like you use the `#include` directive, e.g.,

```
#includeonce( "myHeaderFile.hhf" )
```

If `myHeaderFile.hhf` directly or indirectly includes itself (with a `#INCLUDEONCE` directive), then HLA will ignore the new request to include the file. Note, however, that if you use the `#INCLUDE` directive, rather than `#INCLUDEONCE`, HLA will include the file a second name. This was done in case you really do need to include a header file twice, for some reason (though it is hard to imagine needing to do this).

The bottom line is this: you should always use the `#INCLUDEONCE` directive to include header files you've created. In fact, you should get in the habit of always using `#INCLUDEONCE`, even for header files created by others (the HLA Standard Library already has provisions to prevent recursive includes, so you don't have to worry about using `#INCLUDEONCE` with the Standard Library header files).

There is another technique you can use to prevent recursive includes – using conditional compilation. For details on this technique, see the chapter on the HLA Compile-Time Language in a later volume.

2. There is nothing wrong with this, other than the fact that it does not take advantage of separate compilation.

9.5 UNITS and the EXTERNAL Directive

Technically, the #INCLUDE directive provides you with all the facilities you need to create modular programs. You can create several modules, each containing some specific routine, and include those modules, as necessary, in your assembly language programs using #INCLUDE. However, HLA provides a better way: external and public symbols.

One major problem with the include mechanism is that once you've debugged a routine, including it into a compilation still wastes a lot of time since HLA must recompile bug-free code every time you assemble the main program. A much better solution would be to preassemble the debugged modules and link the object code modules together rather than reassembling the entire program every time you change a single module. This is what the EXTERNAL directive allows you to do.

To use the external facilities, you must create at least two source files. One file contains a set of variables and procedures used by the second. The second file uses those variables and procedures without knowing how they're implemented. The only problem is that if you create two separate HLA programs, the linker will get confused when you try to combine them. This is because both HLA programs have their own main program. Which main program does the OS run when it loads the program into memory? To resolve this problem, HLA uses a different type of compilation module, the UNIT, to compile programs without a main program. The syntax for an HLA UNIT is actually simpler than that for an HLA program, it takes the following form:

```
unit unitname;

    << declarations >>

end unitname;
```

With one exception (the VAR section), anything that can go in the declaration section of an HLA program can go into the declaration section of an HLA unit. Notice that a unit does not have a BEGIN clause and there are no program statements in the unit³; a unit only contains declarations.

In addition to the fact that a unit does not contain any executable statements, there is one other difference between units and programs. Units cannot have a VAR section. This is because the VAR section declares variables that are local to the main program's source code. Since there is no source code associated with a unit, VAR sections are illegal⁴.

To demonstrate, consider the following two modules:

```
unit Number1;

static
    Var1:  uns32;
    Var2:  uns32;

    procedure Add1and2;
    begin Add1and2;

        push( eax );
        mov( Var2, eax );
        add( eax, Var1 );

    end Add1and2;
```

3. Of course, units may contain procedures and those procedures may have statements, but the unit itself does not have any executable instructions associated with it.

4. Of course, procedures in the unit may have their own VAR sections, but the procedure's declaration section is separate from the unit's declaration section.

```
end Number1;
```

Program 9.2 Example of a Simple HLA Unit

```
program main;
#include( "stdlib.hhf" );

begin main;

    mov( 2, Var2 );
    mov( 3, Var1 );
    Addland2();
    stdout.put( "Var1=", Var1, nl );

end main;
```

Program 9.3 Main Program that References External Objects

The main program references `Var1`, `Var2`, and `Add1and2`, yet these symbols are external to this program (they appear in unit *Number1*). If you attempt to compile the main program as it stands, HLA will complain that these three symbols are undefined.

Therefore, you must declare them external with the `EXTERNAL` option. An external procedure declaration looks just like a forward declaration except you use the reserved word `EXTERNAL` rather than `FORWARD`. To declare external static variables, simply follow those variables' declarations with the reserved word `EXTERNAL`. The following is a modification to the previous main program that includes the external declarations:

```
program main;
#include( "stdlib.hhf" );

    procedure Addland2; external;

static
    Var1: uns32; external;
    Var2: uns32; external;

begin main;

    mov( 2, Var2 );
    mov( 3, Var1 );
    Addland2();
    stdout.put( "Var1=", Var1, nl );

end main;
```

Program 9.4 Modified Main Program with EXTERNAL Declarations

If you attempt to compile this second version of *main*, using the typical HLA compilation command “HLA main2.hla” you will be somewhat disappointed. This program will actually compile without error. However, when HLA attempts to link this code it will report that the symbols *Var1*, *Var2*, and *Add1and2* are undefined. This happens because you haven’t compiled and linked in the associated unit with this main program. Before you try that, and discover that it still doesn’t work, you should know that all symbols in a unit, by default, are *private* to that unit. This means that those symbols are inaccessible in code outside that unit unless you explicitly declare those symbols as *public* symbols. To declare symbols as public, you simply put external declarations for those symbols in the unit before the actual symbol declarations. If an external declaration appears in the same source file as the actual declaration of a symbol, HLA assumes that the name is needed externally and makes that symbol a public (rather than private) symbol. The following is a correction to the *Number1* unit that properly declares the external objects:

```
unit Number1;

static
    Var1:  uns32; external;
    Var2:  uns32; external;

    procedure Add1and2; external;

static
    Var1:  uns32;
    Var2:  uns32;

    procedure Add1and2;
    begin Add1and2;

        push( eax );
        mov( Var2, eax );
        add( eax, Var1 );

    end Add1and2;

end Number1;
```

Program 9.5 Correct Number1 Unit with External Declarations

It may seem redundant declaring these symbols twice as occurs in Program 9.5, but you’ll soon see that you don’t normally write the code this way.

If you attempt to compile the *main* program or the *Number1* unit using the typical HLA statement, i.e.,

```
HLA main2.hla
HLA unit2.hla
```

You’ll quickly discover that the linker still returns errors. It returns an error on the compilation of *main2.hla* because you still haven’t told HLA to link in the object code associated with *unit2.hla*. Likewise, the linker complains if you attempt to compile *unit2.hla* by itself because it can’t find a main program. The simple solution is to compile both of these modules together with the following single command:

```
HLA main2.hla unit2.hla
```

This command will properly compile both modules and link together their object code.

Unfortunately, the command above defeats one of the major benefits of separate compilation. When you issue this command it will compile both `main2` and `unit2` prior to linking them together. Remember, a major reason for separate compilation is to reduce compilation time on large projects. While the above command is convenient, it doesn't achieve this goal.

To separately compile the two modules you must run HLA separately on them. Of course, we saw earlier that attempting to compile these modules separately produced linker errors. To get around this problem, you need to compile the modules without linking them. The “-c” (compile-only) HLA command line option achieves this. To compile the two source files without running the linker, you would use the following commands:

```
HLA -c main2.hla
HLA -c unit2.hla
```

This produces two object code files, `main2.obj` and `unit2.obj`, that you can link together to produce a single executable. You could run the linker program directly, but an easier way is to use the HLA compiler to link the object modules together for you:

```
HLA main2.obj unit2.obj
```

Under Windows, this command produces an executable file named `main2.exe`⁵; under Linux, this command produces a file named `main2`. You could also type the following command to compile the main program and link it with a previously compiled `unit2` object module:

```
HLA main2.hla unit2.obj
```

In general, HLA looks at the suffixes of the filenames following the HLA commands. If the filename doesn't have a suffix, HLA assumes it to be “.HLA”. If the filename has a suffix, then HLA will do the following with the file:

- If the suffix is “.HLA”, HLA will compile the file with the HLA compiler.
- If the suffix is “.ASM”, HLA will assemble the file with MASM.
- If the suffix is “.OBJ” or “.LIB”(Windows), or “.o” or “.a” (Linux), then HLA will link that module with the rest of the compilation.

9.5.1 Behavior of the EXTERNAL Directive

Whenever you declare a symbol EXTERNAL using the external directive, keep in mind several limitations of EXTERNAL objects:

- Only one EXTERNAL declaration of an object may appear in a given source file. That is, you cannot define the same symbol twice as an EXTERNAL object.
- Only PROCEDURE, STATIC, READONLY, and STORAGE variable objects can be external. VAR and parameter objects cannot be external.
- External objects must be at the global declaration level. You cannot declare EXTERNAL objects within a procedure or other nested structure.
- EXTERNAL objects publish their name globally. Therefore, you must carefully choose the names of your EXTERNAL objects so they do not conflict with other symbols.

This last point is especially important to keep in mind. As this text is being written, the HLA compiler translates your HLA source code into assembly code. HLA assembles the output by using MASM (the Microsoft Macro Assembler), Gas (Gnu's as), or some other assembler. Finally, HLA links your modules using a linker. At each step in this process, your choice of external names could create problems for you.

5. If you want to explicitly specify the name of the output file, HLA provides a command-line option to achieve this. You can get a menu of all legal command line options by entering the command “HLA -?”.

Consider the following HLA external/public declaration:

```
static
    extObj:      uns32; external;
    extObj:      uns32;
    localObject: uns32;
```

When you compile a program containing these declarations, HLA automatically generates a “munged” name for the *localObject* variable that probably isn’t ever going to have any conflicts with system-global external symbols⁶. Whenever you declare an external symbol, however, HLA uses the object’s name as the default external name. This can create some problems if you inadvertently use some global name as your variable name. Worse still, the assembler will not be able to properly process HLA’s output if you happen to choose an identifier that is legal in HLA but is one of the assembler’s reserved word. For example, if you attempt to compile the following code fragment as part of an HLA program (producing MASM output), it will compile properly but MASM will not be able to assemble the code:

```
static
    c: char; external;
    c: char;
```

The reason MASM will have trouble with this is because HLA will write the identifier “c” to the assembly language output file and it turns out that “c” is a MASM reserved word (MASM uses it to denote C-language linkage).

To get around the problem of conflicting external names, HLA supports an additional syntax for the EXTERNAL option that lets you explicitly specify the external name. The following example demonstrates this extended syntax:

```
static
    c: char; external( "var_c" );
    c: char;
```

If you follow the EXTERNAL keyword with a string constant enclosed by parentheses, HLA will continue to use the declared name (*c* in this example) as the identifier within your HLA source code. Externally (i.e., in the assembly code) HLA will substitute the name *var_c* whenever you reference *c*. This feature helps you avoid problems with the misuse of assembler reserved words, or other global symbols, in your HLA programs.

You should also note that this feature of the EXTERNAL option lets you create *aliases*. For example, you may want to refer to an object by the name *StudentCount* in one module while refer to the object as *PersonCount* in another module (you might do this because you have a general library module that deals with counting people and you want to use the object in a program that deals only with students). Using a declaration like the following lets you do this:

```
static
    StudentCount: uns32; external( "PersonCount" );
```

Of course, you’ve already seen some of the problems you might encounter when you start creating aliases. So you should use this capability sparingly in your programs. Perhaps a more reasonable use of this feature is to simplify certain OS APIs. For example, Win32 uses some really long names for certain procedure calls. You can use the EXTERNAL directive to provide a more meaningful name than the standard one supplied by the operating system.

9.5.2 Header Files in HLA

HLA’s technique of using the same EXTERNAL declaration to define public as well as external symbols may seem somewhat counter-intuitive. Why not use a PUBLIC reserved word for public symbols and

6. Typically, HLA creates a name like *?001A_localObject* out of *localObject*. This is a legal MASM identifier but it is not likely it will conflict with any other global symbols when HLA compiles the program with MASM.

the `EXTERNAL` keyword for external definitions? Well, as counter-intuitive as HLA's external declarations may seem, they are founded on decades of solid experience with the C/C++ programming language that uses a similar approach to public and external symbols⁷. Combined with a *header file*, HLA's external declarations make large program maintenance a breeze.

An important benefit of the `EXTERNAL` directive (versus separate `PUBLIC` and `EXTERNAL` directives) is that it lets you minimize duplication of effort in your source files. Suppose, for example, you want to create a module with a bunch of support routines and variables for use in several different programs (e.g., the HLA Standard Library). In addition to sharing some routines and some variables, suppose you want to share constants, types, and other items as well.

The `#INCLUDE` file mechanism provides a perfect way to handle this. You simply create a `#INCLUDE` file containing the constants, macros, and external definitions and include this file in the module that implements your routines and in the modules that use those routines (see Figure 9.1).

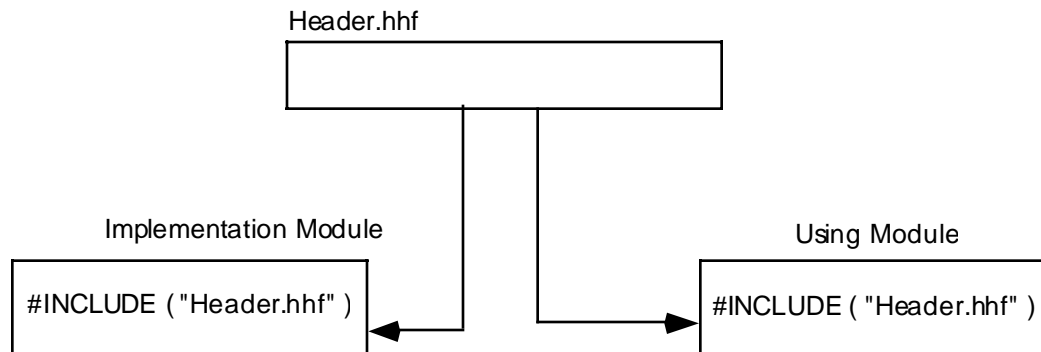


Figure 9.1 Using Header Files in HLA Programs

A typical header file contains only `CONST`, `VAL`, `TYPE`, `STATIC`, `READONLY`, `STORAGE`, and procedure prototypes (plus a few others we haven't look at yet, like macros). Objects in the `STATIC`, `READONLY`, and `STORAGE` sections, as well as all procedure declarations, are always `EXTERNAL` objects. In particular, you generally should not put any `VAR` objects in a header file, nor should you put any non-external variables or procedure bodies in a header file. If you do, HLA will make duplicate copies of these objects in the different source files that include the header file. Not only will this make your programs larger, but it will cause them to fail under certain circumstances. For example, you generally put a variable in a header file so you can share the value of that variable amongst several different modules. However, if you fail to declare that symbol as external in the header file and just put a standard variable declaration there, each module that includes the source file will get its own separate variable - the modules will not share a common variable.

If you create a standard header file, containing `CONST`, `VAL`, and `TYPE` declarations, and external objects, you should always be sure to include that file in the declaration section of all modules that need the definitions in the header file. Generally, HLA programs include all their header files in the first few statements after the `PROGRAM` or `UNIT` header.

This text adopts the HLA Standard Library convention of using an “.hhf” suffix for HLA header files (“HHF” stands for HLA Header File).

7. Actually, C/C++ is a little different. All global symbols in a module are assumed to be public unless explicitly declared private. HLA's approach (forcing the declaration of public items via `EXTERNAL`) is a little safer.

9.6 Make Files

Although using separate compilation reduces assembly time and promotes code reuse and modularity, it is not without its own drawbacks. Suppose you have a program that consists of two modules: `pgma.hla` and `pgmb.hla`. Also suppose that you've already compiled both modules so that the files `pgma.obj` and `pgmb.obj` exist. Finally, you make changes to `pgma.hla` and `pgmb.hla` and compile the `pgma.hla` file *but forget to compile the `pgmb.hla` file*. Therefore, the `pgmb.obj` file will be *out of date* since this object file does not reflect the changes made to the `pgmb.hla` file. If you link the program's modules together, the resulting executable file will only contain the changes to the `pgma.hla` file, it will not have the updated object code associated with `pgmb.hla`. As projects get larger they tend to have more modules associated with them, and as more programmers begin working on the project, it gets very difficult to keep track of which object modules are up to date.

This complexity would normally cause someone to recompile *all* modules in a project, even if many of the object files are up to date, simply because it might seem too difficult to keep track of which modules are up to date and which are not. Doing so, of course, would eliminate many of the benefits that separate compilation offers. Fortunately, there is a tool that can help you manage large projects: *make*⁸. The make program, with a little help from you, can figure out which files need to be reassemble and which files have up to date `.obj` files. With a properly defined *make file*, you can easily assemble only those modules that absolutely must be assembled to generate a consistent program.

A make file is a text file that lists compile-time dependencies between files. An `.exe` file, for example, is *dependent* on the source code whose assembly produce the executable. If you make any changes to the source code you will (probably) need to reassemble or recompile the source code to produce a new executable file⁹.

Typical dependencies include the following:

- An executable file generally depends only on the set of object files that the linker combines to form the executable.
- A given object code file depends on the assembly language source files that were assembled to produce that object file. This includes the assembly language source files (`.hla`) and any files included during that assembly (generally `.hhf` files).
- The source files and include files generally don't depend on anything.

A make file generally consists of a dependency statement followed by a set of commands to handle that dependency. A dependency statement takes the following form:

dependent-file : *list of files*

Example :

```
pgm.exe: pgma.obj pgmb.obj          --Windows/nmake example
```

This statement says that "pgm.exe" is dependent upon `pgma.obj` and `pgmb.obj`. Any changes that occur to `pgma.obj` or `pgmb.obj` will require the generation of a new `pgm.exe` file. This example is Windows-specific, here's the same makefile statement in a Linux-friendly form:

Example :

```
pgm: pgma.o pgmb.o                --Linux/make example
```

The make program uses a *time/date stamp* to determine if a dependent file is out of date with respect to the files it depends upon. Any time you make a change to a file, the operating system will update a *modification time and date* associated with the file. The make program compares the modification date/time stamp of the dependent file against the modification date/time stamp of the files it depends upon. If the dependent

8. Under Windows, Microsoft calls this program *nmake*. This text will use the more generic name "make" when referring to this program. If you are using Microsoft tools under Windows, just substitute "nmake" for "make" throughout this chapter.

9. Obviously, if you only change comments or other statements in the source file that do not affect the executable file, a recompile or reassembly will not be necessary. To be safe, though, we will assume *any* change to the source file will require a reassembly.

file's modification date/time is earlier than one or more of the files it depends upon, or one of the files it depends upon is not present, then make assumes that some operation must be necessary to update the dependent file.

When an update is necessary, make executes the set of commands following the dependency statement. Presumably, these commands would do whatever is necessary to produce the updated file.

The dependency statement *must* begin in column one. Any commands that must execute to resolve the dependency must start on the line immediately following the dependency statement and each command must be indented one tabstop. The `pgm.exe` statement above (the Windows example) would probably look something like the following:

```
pgm.exe: pgma.obj pgmb.obj
    hla -opgm.exe pgma.obj pgmb.obj
```

(The “-opgm.exe” option tells HLA to name the executable file “pgm.exe.”) Here's the same example for Linux users:

```
pgm: pgma.o pgmb.o
    hla -opgm pgma.obj pgmb.obj
```

If you need to execute more than one command to resolve the dependencies, you can place several commands after the dependency statement in the appropriate order. Note that you must indent all commands one tab stop. The make program ignores any blank lines in a make file. Therefore, you can add blank lines, as appropriate, to make the file easier to read and understand.

There can be more than a single dependency statement in a make file. In the example above, for example, executable (`pgm` or `pgm.exe`) depends upon the object files (`pgma.obj` or `pgma.o` and `pgmb.obj` or `pgmb.o`). Obviously, the object files depend upon the source files that generated them. Therefore, before attempting to resolve the dependencies for the executable, make will first check out the rest of the make file to see if the object files depend on anything. If they do, make will resolve those dependencies first. Consider the following (Windows) make file:

```
pgm.exe: pgma.obj pgmb.obj
    hla -opgm.exe pgma.obj pgmb.obj

pgma.obj: pgma.hla
    hla -c pgma.hla

pgmb.obj: pgmb.hla
    hla -c pgmb.hla
```

The make program will process the first dependency line it finds in the file. However, the files that `pgm.exe` depends upon themselves have dependency lines. Therefore, make will first ensure that `pgma.obj` and `pgmb.obj` are up to date before attempting to execute HLA to link these files together. Therefore, if the only change you've made has been to `pgmb.hla`, make takes the following steps (assuming `pgma.obj` exists and is up to date).

1. The make program processes the first dependency statement. It notices that dependency lines for `pgma.obj` and `pgmb.obj` (the files on which `pgm.exe` depends) exist. So it processes those statements first.
2. the make program processes the `pgma.obj` dependency line. It notices that the `pgma.obj` file is newer than the `pgma.hla` file, so it does *not* execute the command following this dependency statement.
3. The make program processes the `pgmb.obj` dependency line. It notes that `pgmb.obj` is older than `pgmb.hla` (since we just changed the `pgmb.hla` source file). Therefore, make executes the command following on the next line. This generates a new `pgmb.obj` file that is now up to date.
4. Having processed the `pgma.obj` and `pgmb.obj` dependencies, make now returns its attention to the first dependency line. Since make just created a new `pgmb.obj` file, its date/time stamp will be newer than `pgm.exe`'s. Therefore, make will execute the HLA command that links `pgma.obj` and `pgmb.obj` together to form the new `pgm.exe` file.

Note that a properly written make file will instruct the make program to assemble only those modules absolutely necessary to produce a consistent executable file. In the example above, make did not bother to assemble `pgma.hla` since its object file was already up to date.

There is one final thing to emphasize with respect to dependencies. Often, object files are dependent not only on the source file that produces the object file, but any files that the source file includes as well. In the previous example, there (apparently) were no such include files. Often, this is not the case. A more typical make file might look like the following (Linux example):

```
pgm: pgma.o pgmb.o
    hla -opgm pgma.o pgmb.o

pgma.o: pgma.hla pgm.hhf
    hla -c pgma.hla

pgmb.o: pgmb.hla pgm.hhf
    hla -c pgmb.hla
```

Note that any changes to the `pgm.hhf` file will force the make program to recompile *both* `pgma.hla` and `pgmb.hla` since the `pgma.o` and `pgmb.o` files both depend upon the `pgm.hhf` include file. Leaving include files out of a dependency list is a common mistake programmers make that can produce inconsistent executable files.

Note that you would not normally need to specify the HLA Standard Library include files nor the Standard Library “.lib” (Windows) or “.a” (Linux) files in the dependency list. True, your resulting executable file does depend on this code, but the Standard Library rarely changes, so you can safely leave it out of your dependency list. Should you make a modification to the Standard Library, simply delete any old executable and object files to force a reassembly of the entire system.

The make program, by default, assumes that it will be processing a make file named “makefile”. When you run the make program, it looks for “makefile” in the current directory. If it doesn’t find this file, it complains and terminates¹⁰. Therefore, it is a good idea to collect the files for each project you work on into their own subdirectory and give each project its own makefile. Then to create an executable, you need only change into the appropriate subdirectory and run the make program.

Although this section discusses the make program in sufficient detail to handle most projects you will be working on, keep in mind that the make program provides considerable functionality that this chapter does not discuss. To learn more about the `nmake.exe` program, consult the the appropriate documentation. Note that several versions of MAKE exist. Microsoft produces `nmake.exe`, Borland has their own MAKE.EXE program and various versions of MAKE have been ported to Windows from UNIX systems (e.g., GMAKE). Linux users will typically employ the GNU make program. While these various make programs are not equivalent, they all do a pretty good job of handling the simple make syntax that this chapter describes.

9.7 Code Reuse

One of the principle goals of Software Engineering is to reduce program development time. Although the techniques we’ve studied in this chapter will certainly reduce development effort, there are bigger prizes to be had here. Consider for a moment a simple program that reads an integer from the user and then displays the value of that integer on the standard output device. You can easily write a trivial version of this program with about eight lines of HLA code. That’s not too difficult. However, suppose you did not have the HLA Standard Library at your disposal. Now, instead of an eight line program, you’d be faced with writing a program that hundreds if not thousands of lines long. Obviously, this program will take a lot longer to write than the original eight-line version. The difference between these two applications is the fact that in the first version of this program you got to reuse some code that was already written; in the second version of the program you had to write everything from scratch. This concept of code reuse is very important when

10. There is a command line option that lets you specify the name of the makefile. See the `nmake` documentation in the MASM manuals for more details.

writing large programs – you can get large programs working much more quickly if you reuse code from previous projects.

The idea behind code reuse is that many code sequences you write will be usable in future programs. As time passes and you write more code, progress on your projects will be faster since you can reuse code you've written (or others have written) on previous projects. The HLA Standard Library functions are the classic example, somebody had to write those functions so you could use them. And use them you do. As of this writing, the Standard Library represented about 50,000 lines of HLA source code. Imagine having to write a fair portion of that everytime you wanted to write an HLA program!

Although the HLA Standard Library contains lots of very useful routines and functions, this code base cannot possibly predict the type of code you will need in every future project. The HLA Standard Library provides some of the more common routines you'll need when writing programs, but you're certainly going to have need for routines that the HLA Standard Library cannot satisfy. Unless you can find a source for the code you need from some third party, you're probably going to have to write the new routines yourself.

The trick when writing a program is to try and figure out which routines are general purpose and could be used in future programs; once you make this determination, you should write such routines separately from the rest of your application (i.e., put them in a separate source file for compilation). By keeping them separate, you can use them in future projects. If "try and figure out which routines are general purpose..." sounds a bit difficult, well, you're right it is. Even after 30 years of Software Engineering research, no one has really figured out how to effectively reuse code. There are some obvious routines we can reuse (that's why there are "standard libraries") but it is quite difficult for the practicing engineer to successfully predict which routines s/he will need in the future and write these as separate modules.

Attempting to teach you how to decide which routines are worthy of saving for future programs and which are specific to your current application is well beyond the scope of this text. There are several Software Engineering texts out there that try to explain how to do this, but keep in mind that even after the publication of these texts, practicing engineers still have problems picking the right routines to save. Hopefully, as you gain experience, you will begin to recognize those routines that are worth keeping for future programs and those that aren't worth bothering with. This text will take the easy way out and assume that you know which routines you want to keep and which you don't.

9.8 Creating and Managing Libraries

Imagine that you've created a few hundred routines over the past couple of years and you would like to have the object code ready to link with any new projects you begin. You could move all this code into a single source file, stick in a bunch of EXTERNAL declarations, and then link the resulting object file with any new programs you write that can use the routines in your "library". Unfortunately, there are a couple of problems with this approach. Let's take a look at some of these problems.

Problem number one is that your library will grow to a fairly good size with time; if you put the source code to every routine in a single source file, small additions or changes to the file will require a complete recompilation of the whole library. That's clearly not what we want to do, based on what you've learned from this chapter.

Another problem with this "solution" is that whenever you link this object file to your new applications, you link in the entire library, not just the routines you want to use. This makes your applications unnecessarily large, especially if your library has grown. Were you to link your simple projects with the entire HLA Standard library, for example, the result would be positively huge.

A solution to both of the above problems is to compile each routine in a separate file and produce a unique object file for it. Unfortunately, with hundreds of routines you're going to wind up with hundreds of object files; any time you want to call a dozen or so library routines, you'd have to link your main application with a dozen or so object modules from your library. Clearly, this isn't acceptable either.

You may have noticed by now that when you link your applications with the HLA Standard Library, you only link with a single file: *hllib.lib* (Windows) or *hllib.a* (Linux). .LIB (library) and ".a" (archive) files are a collection of object files. When the linker processes a library file, it pulls out only the object files it

needs, it does not link the entire file with your application. Hence you get to work with a single file and your applications don't grow unnecessarily large.

Linux provides the “ar” (archiver) program to manage library files. To use this program to combine several object files into a single “.a” file, you'd use a command line like the following:

```
ar -q library.a list_of_.o_files
```

For more information on this command, check out the man page on the “ar” program (“man ar”).

9.9 Name Space Pollution

One problem with creating libraries with lots of different modules is name space pollution. A typical library module will have a #INCLUDE file associated with it that provides external definitions for all the routines, constants, variables, and other symbols provided in the library. Whenever you want to use some routines or other objects from the library, you would typically #INCLUDE the library's header file in your project. As your libraries get larger and you add more declarations in the header file, it becomes more and more likely that the names you've chosen for your library's identifiers will conflict with names you want to use in your current project. This conflict is what is meant by name space pollution: library header files pollute the name space with many names you typically don't need in order to gain easy access to the few routines in the library you actually use. Most of the time those names don't harm anything – unless you want to use those names yourself in your program.

HLA requires that you declare all external symbols at the global (PROGRAM/UNIT) level. You cannot, therefore, include a header file with external declarations within a procedure¹¹. As such, there will be no naming conflicts between external library symbols and symbols you declare locally within a procedure; the conflicts will only occur between the external symbols and your global symbols. While this is a good argument for avoiding global symbols as much as possible in your program, the fact remains that most symbols in an assembly language program will have global scope. So another solution is necessary.

HLA's solution, which it certainly uses in the Standard Library, is to put most of the library names in a NAMESPACE declaration section. A NAMESPACE declaration encapsulates all declarations and exposes only a single name (the NAMESPACE identifier) at the global level. You access the names within the NAMESPACE by using the familiar dot-notation (see “Namespaces” on page 496). This reduces the effect of namespace pollution from many dozens or hundreds of names down to a single name.

Of course, one disadvantage of using a NAMESPACE declaration is that you have to type a longer name in order to reference a particular identifier in that name space (i.e., you have to type the NAMESPACE identifier, a period, and then the specific identifier you wish to use). For a few identifiers you use frequently, you might elect to leave those identifiers outside of any NAMESPACE declaration. For example, the HLA Standard Library does not define the symbols *malloc*, *free*, or *nl* (among others) within a NAMESPACE. However, you want to minimize such declarations in your libraries to avoid conflicts with names in your own programs. Often, you can choose a NAMESPACE identifier to complement your routine names. For example, the HLA Standard Libraries string copy routine was named after the equivalent C Standard Library function, *strcpy*. HLA's version is *str.cpy*. The actual function name is *cpy*; it happens to be a member of the *str* NAMESPACE, hence the full name *str.cpy* which is very similar to the comparable C function. The HLA Standard Library contains several examples of this convention. The *arg.c* and *arg.v* functions are another pair of such identifiers (corresponding to the C identifiers *argc* and *argv*).

Using a NAMESPACE in a header file is no different than using a NAMESPACE in a PROGRAM or UNIT. Here's an example of a typical header file containing a NAMESPACE declaration:

```
// myHeader.hhf -
//
// Routines supported in the myLibrary.lib file.

namespace myLib;
```

11. Or within an Iterator or Method, as you will see in later chapters.

```

procedure func1; external;
procedure func2; external;
procedure func3; external;

end myLib;

```

Typically, you would compile each of the functions (*func1*..*func3*) as separate units (so each has its own object file and linking in one function doesn't link them all). Here's what a sample UNIT declaration for one of these functions:

```

unit func1Unit;
#includeonce( "myHeader.hhf" )

procedure myLib.func1;
begin func1;

    << code for func1 >>

end func1;

end func1Unit;

```

You should notice two important things about this unit. First, you do not put the actual *func1* procedure code within a NAMESPACE declaration block. By using the identifier *myLib.func1* as the procedure's name, HLA automatically realizes that this procedure declaration belongs in a name space. The second thing to note is that you do not preface *func1* with "myLib." after the BEGIN and END clauses in the procedure. HLA automatically associates the BEGIN and END identifiers with the PROCEDURE declaration, so it knows that these identifiers are part of the *myLib* name space and it doesn't make you type the whole name again.

Important note: when you declare external names within a name space, as was done in *func1Unit* above, HLA uses only the function name (*func1* in this example) as the external name. This creates a name space pollution problem in the external name space. For example, if you have two different name spaces, *myLib* and *yourLib* and they both define a *func1* procedure, the linker will complain about a duplicate definition for *func1* if you attempt to use functions from both these library modules. There is an easy work-around to this problem: use the extended form of the EXTERNAL directive to explicitly supply an external name for all external identifiers appearing in a NAMESPACE declaration. For example, you could solve this problem with the following simple modification to the *myHeader.hhf* file above:

```

// myHeader.hhf -
//
// Routines supported in the myLibrary.lib file.

namespace myLib;

    procedure func1; external( "myLib_func1" );
    procedure func2; external( "myLib_func2" );
    procedure func3; external( "myLib_func3" );

end myLib;

```

This example demonstrates an excellent convention you should adopt: when exporting names from a name space, always supply an explicit external name and construct that name by concatenating the NAMESPACE identifier with an underscore and the object's internal name.

The use of NAMESPACE declarations does not completely eliminate the problems of name space pollution (after all, the name space identifier is still a global object, as anyone who has included *stdlib.hhf* and attempted to define a "cs" variable can attest), but NAMESPACE declarations come pretty close to eliminating this problem. Therefore, you should use NAMESPACE everywhere practical when creating your own libraries.

9.10 Putting It All Together

Managing large projects is considerably easier if you break your program up into separate modules and work on them independently. In this chapter you learned about HLA's `UNITs`, `include` files, and the `EXTERNAL` directive. These provide the tools you need to break a program up into smaller modules. In addition to HLA's facilities, you'll also use a separate tool, `nmake.exe`, to automatically compile and link only those files that are necessary in a large project.

This chapter provided a very basic introduction to the use of makefiles and the make utility. Note that the `MAKE` programs are quite sophisticated. The presentation of the make program in this chapter barely scratches the surface. If you're interested in more information about `MAKE` facilities you should consult one of the excellent texts available on this subject. Lots of good information is also available on the Internet (just use the usual search tools).

In addition to breaking up large HLA projects, `UNITs` are also the basis for letting you write assembly language functions that you can call from high level languages like `C/C++` and `Delphi/Kylix`. A later volume in this text will describe how you can use `UNITs` for this purpose.

Integer Arithmetic

Chapter Ten

10.1 Chapter Overview

This chapter discusses the implementation of arithmetic computation in assembly language. By the conclusion of this chapter you should be able to translate (integer) arithmetic expressions and assignment statements from high level languages like Pascal and C/C++ into 80x86 assembly language.

10.2 80x86 Integer Arithmetic Instructions

Before describing how to encode arithmetic expressions in assembly language, it would be a good idea to first discuss the remaining arithmetic instructions in the 80x86 instruction set. Previous chapters have covered most of the arithmetic and logical instructions, so this section will cover the few remaining instructions you'll need.

10.2.1 The MUL and IMUL Instructions

The multiplication instructions provide you with another taste of irregularity in the 80x86's instruction set. Instructions like ADD, SUB, and many others in the 80x86 instruction set support two operands. Unfortunately, there weren't enough bits in the 80x86's opcode byte to support all instructions, so the 80x86 treats the MUL (unsigned multiply) and IMUL (signed integer multiply) instructions as single operand instructions, like the INC, DEC, and NEG instructions.

Of course, multiplication *is* a two operand function. To work around this fact, the 80x86 always assumes the accumulator (AL, AX, or EAX) is the destination operand. This irregularity makes using multiplication on the 80x86 a little more difficult than other instructions because one operand has to be in the accumulator. Intel adopted this unorthogonal approach because they felt that programmers would use multiplication far less often than instructions like ADD and SUB.

Another problem with the MUL and IMUL instructions is that you cannot multiply the accumulator by a constant using these instructions. Intel quickly discovered the need to support multiplication by a constant and added the INTMUL instruction to overcome this problem. Nevertheless, you must be aware that the basic MUL and IMUL instructions do not support the full range of operands that INTMUL does.

There are two forms of the multiply instruction: unsigned multiplication (MUL) and signed multiplication (IMUL). Unlike addition and subtraction, you need separate instructions for these two operations.

The multiply instructions take the following forms:

Unsigned Multiplication:

```
mul( reg8 );           // returns "ax"
mul( reg16 );          // returns "dx:ax"
mul( reg32 );          // returns "edx:eax"

mul( mem8 );           // returns "ax"
mul( mem16 );          // returns "dx:ax"
mul( mem32 );          // returns "edx:eax"
```

Signed (Integer) Multiplication:

```
imul( reg8 );           // returns "ax"
imul( reg16 );          // returns "dx:ax"
imul( reg32 );          // returns "edx:eax"
```

```

imul( mem8 );      // returns "ax"
imul( mem16 );     // returns "dx:ax"
imul( mem32 );     // returns "edx:eax"

```

The “returns” values above are the strings these instructions return for use with instruction composition in HLA (see “Instruction Composition in HLA” on page 558).

(I)MUL, available on all 80x86 processors, multiplies eight, sixteen, or thirty-two bit operands. Note that when multiplying two n-bit values, the result may require as many as 2*n bits. Therefore, if the operand is an eight bit quantity, the result could require sixteen bits. Likewise, a 16 bit operand produces a 32 bit result and a 32 bit operand requires 64 bits to hold the result.

The (I)MUL instruction, with an eight bit operand, multiplies the AL register by the operand and leaves the 16 bit product in AX. So

```

mul( operand8 );
or  imul( operand8 );

```

computes:

```
AX := AL * operand8
```

“*” represents an unsigned multiplication for MUL and a signed multiplication for IMUL.

If you specify a 16 bit operand, then MUL and IMUL compute:

```
DX:AX := AX * operand16
```

“*” has the same meanings as above and DX:AX means that DX contains the H.O. word of the 32 bit result and AX contains the L.O. word of the 32 bit result. If you’re wondering why Intel didn’t put the 32-bit result in EAX, just note that Intel introduced the MUL and IMUL instructions in the earliest 80x86 processors, before the advent of 32-bit registers in the 80386 CPU.

If you specify a 32 bit operand, then MUL and IMUL compute the following:

```
EDX:EAX := EAX * operand32
```

“*” has the same meanings as above and EDX:EAX means that EDX contains the H.O. double word of the 64 bit result and EAX contains the L.O. double word of the 64 bit result.

If an 8x8, 16x16, or 32x32 bit product requires more than eight, sixteen, or thirty-two bits (respectively), the MUL and IMUL instructions set the carry and overflow flags. MUL and IMUL scramble the sign, and zero flags. **Especially note that the sign and zero flags do not contain meaningful values after the execution of these two instructions.**

To help reduce some of the problems with the use of the MUL and IMUL instructions, HLA provides an extended syntax that allows the following two-operand forms:

Unsigned Multiplication:

```

mul( reg8, al );
mul( reg16, ax );
mul( reg32, eax );

mul( mem8, al );
mul( mem16, ax );
mul( mem32, eax );

mul( constant8, al );
mul( constant16, ax );
mul( constant32, eax );

```

Signed (Integer) Multiplication:

```

imul( reg8, al );
imul( reg16, ax );
imul( reg32, eax );

imul( mem8, al );
imul( mem16, ax );
imul( mem32, eax );

imul( constant8, al );
imul( constant16, ax );
imul( constant32, eax );

```

The two operand forms let you specify the (L.O.) destination register. The instructions whose first operand is a register or memory location are completely identical to the instructions above. By specifying the destination register, however, you can make your programs easier to read; therefore, it's probably a good idea to go ahead and specify the destination register. Note that just because HLA allows two operands here, you can't specify an arbitrary register. The destination operand must always be AL, AX, or EAX, depending on the source operand.

Note that HLA allows a form that lets you specify a constant. The 80x86 doesn't actually support a MUL or IMUL instruction that has a constant operand. HLA will take the constant you specify and create a "variable" in the special "const" segment in memory and initialize that variable with this value. Then HLA converts the instruction to the "(IMUL(memory);" instruction. Generally, you won't need to use this special form since the INTMUL instruction will multiply a register by a constant.

You'll use the MUL and IMUL instructions quite a bit when you learn about extended precision arithmetic in the chapter on Advanced Arithmetic. Until you get to that chapter, you'll probably just want to use the INTMUL instruction in place of the MUL or IMUL since it is more general. However, INTMUL is not a complete replacement for these two instructions. Besides the number of operands, there are several differences between the INTMUL instruction you've learned about earlier and the MUL and IMUL instructions. Specifically for the INTMUL instruction:

- There isn't an 8x8 bit INTMUL instruction available (the immediate₈ operands simply provide a shorter form of the instruction. Internally, the CPU sign extends the operand to 16 or 32 bits as necessary).
- The INTMUL instruction does not produce a 2*n bit result. That is, a 16x16 multiply produces a 16 bit result. Likewise, a 32x32 bit multiply produces a 32 bit result. These instructions set the carry and overflow flags if the result does not fit into the destination register.

10.2.2 The DIV and IDIV Instructions

The 80x86 divide instructions perform a 64/32 division, a 32/16 division or a 16/8 division. These instructions take the form:

```

div( reg8 );           // returns "al"
div( reg16 );          // returns "ax"
div( reg32 );          // returns "eax"

div( reg8, AX );       // returns "al"
div( reg16, DX:AX );   // returns "ax"
div( reg32, EDX:EAX ); // returns "eax"

div( mem8 );           // returns "al"
div( mem16 );          // returns "ax"
div( mem32 );          // returns "eax"

```

```

div( mem8, AX );           // returns "al"
div( mem16, DX:AX );       // returns "ax"
div( mem32, EDX:EAX );     // returns "eax"

div( constant8, AX );      // returns "al"
div( constant16, DX:AX );  // returns "ax"
div( constant32, EDX:EAX ); // returns "eax"

idiv( reg8 );              // returns "al"
idiv( reg16 );            // returns "ax"
idiv( reg32 );           // returns "eax"

idiv( reg8, AX );         // returns "al"
idiv( reg16, DX:AX );    // returns "ax"
idiv( reg32, EDX:EAX );  // returns "eax"

idiv( mem8 );             // returns "al"
idiv( mem16 );          // returns "ax"
idiv( mem32 );          // returns "eax"

idiv( mem8, AX );         // returns "al"
idiv( mem16, DX:AX );    // returns "ax"
idiv( mem32, EDX:EAX );  // returns "eax"

idiv( constant8, AX );    // returns "al"
idiv( constant16, DX:AX ); // returns "ax"
idiv( constant32, EDX:EAX ); // returns "eax"

```

The DIV instruction computes an unsigned division. If the operand is an eight bit operand, DIV divides the AX register by the operand leaving the quotient in AL and the remainder (modulo) in AH. If the operand is a 16 bit quantity, then the DIV instruction divides the 32 bit quantity in DX:AX by the operand leaving the quotient in AX and the remainder in DX. With 32 bit operands DIV divides the 64 bit value in EDX:EAX by the operand leaving the quotient in EAX and the remainder in EDX.

You cannot, on the 80x86, simply divide one eight bit value by another. If the denominator is an eight bit value, the numerator must be a sixteen bit value. If you need to divide one unsigned eight bit value by another, you must zero extend the numerator to sixteen bits. You can accomplish this by loading the numerator into the AL register and then moving zero into the AH register. Then you can divide AX by the denominator operand to produce the correct result. *Failing to zero extend AL before executing DIV may cause the 80x86 to produce incorrect results!*

When you need to divide two 16 bit unsigned values, you must zero extend the AX register (which contains the numerator) into the DX register. To do this, just load zero into the DX register. If you need to divide one 32-bit value by another, you must zero extend the EAX register into EDX (by loading a zero into EDX) before the division.

When dealing with signed integer values, you will need to sign extend AL into AX, AX into DX or EAX into EDX before executing IDIV. To do so, use the CBW, CWD, CDQ, or MOVSX instructions. If the H.O. byte or word does not already contain significant bits, then you must sign extend the value in the accumulator (AL/AX/EAX) before doing the IDIV operation. Failure to do so may produce incorrect results.

There is one other catch to the 80x86's divide instructions: you can get a fatal error when using this instruction. First, of course, you can attempt to divide a value by zero. Second, the quotient may be too large to fit into the EAX, AX, or AL register. For example, the 16/8 division "\$8000 / 2" produces the quotient \$4000 with a remainder of zero. \$4000 will not fit into eight bits. If this happens, or you attempt to divide by zero, the 80x86 will generate an *ex.DivisionError* exception or integer overflow error (*ex.IntoInstr*). This usually means your program will display the appropriate dialog box and abort your program. If this happens

to you, chances are you didn't sign or zero extend your numerator before executing the division operation. Since this error will cause your program to crash, you should be very careful about the values you select when using division. Of course, you can use the TRY..ENDTRY block with the *ex.DivisionError* and *ex.IntoInstr* to trap this problem in your program.

The carry, overflow, sign, and zero flags are undefined after a division operation. Like MUL and IMUL, HLA provides special syntax to allow the use of constant operands even though these instructions don't really support them.

The 80x86 does not provide a separate instruction to compute the remainder of one number divided by another. The DIV and IDIV instructions automatically compute the remainder at the same time they compute the quotient. HLA, however, provides mnemonics (instructions) for the MOD and IMOD instructions. These special HLA instructions compile into the exact same code as their DIV and IDIV counterparts. The only difference is the "returns" value for the instruction (since these instructions return the remainder in a different location than the quotient). The MOD and IMOD instructions that HLA supports are

```

mod( reg8 );           // returns "ah"
mod( reg16 );          // returns "dx"
mod( reg32 );          // returns "edx"

mod( reg8, AX );       // returns "ah"
mod( reg16, DX:AX );   // returns "dx"
mod( reg32, EDX:EAX ); // returns "edx"

mod( mem8 );           // returns "ah"
mod( mem16 );          // returns "dx"
mod( mem32 );          // returns "edx"

mod( mem8, AX );       // returns "ah"
mod( mem16, DX:AX );   // returns "dx"
mod( mem32, EDX:EAX ); // returns "edx"

mod( constant8, AX ); // returns "ah"
mod( constant16, DX:AX ); // returns "dx"
mod( constant32, EDX:EAX ); // returns "edx"

imod( reg8 );           // returns "ah"
imod( reg16 );          // returns "dx"
imod( reg32 );          // returns "edx"

imod( reg8, AX );       // returns "ah"
imod( reg16, DX:AX );   // returns "dx"
imod( reg32, EDX:EAX ); // returns "edx"

imod( mem8 );           // returns "ah"
imod( mem16 );          // returns "dx"
imod( mem32 );          // returns "edx"

imod( mem8, AX );       // returns "ah"
imod( mem16, DX:AX );   // returns "dx"
imod( mem32, EDX:EAX ); // returns "edx"

imod( constant8, AX ); // returns "ah"
imod( constant16, DX:AX ); // returns "dx"
imod( constant32, EDX:EAX ); // returns "edx"

```

10.2.3 The CMP Instruction

The CMP (compare) instruction is identical to the SUB instruction with one crucial difference – it does not store the difference back into the destination operand. The syntax for the CMP instruction is similar to SUB (though the operands are reversed so it reads better), the generic form is

```
cmp( LeftOperand, RightOperand );
```

This instruction computes “LeftOperand - RightOperand” (note the reversal from SUB). The specific forms are

```
cmp( reg, reg );      // Registers must be the same size (8, 16, or 32 bits)
cmp( reg, mem );     // Sizes must match.
cmp( reg, constant );
cmp( mem, constant );
```

Note that both operands are “source” operands, so the fact that a constant appears as the second operand is okay.

The CMP instruction updates the 80x86’s flags according to the result of the subtraction operation (LeftOperand - RightOperand). The flags are generally set in an appropriate fashion so that we can read this instruction as “compare LeftOperand to RightOperand”. You can test the result of the comparison by checking the appropriate flags in the flags register using the conditional set instructions (see the next section) or the conditional jump instructions.

Probably the first place to start when exploring the CMP instruction is to take a look at exactly how the CMP instruction affects the flags. Consider the following CMP instruction:

```
cmp( ax, bx );
```

This instruction performs the computation AX - BX and sets the flags depending upon the result of the computation. The flags are set as follows:

- Z: The zero flag is set if and only if AX = BX. This is the only time AX - BX produces a zero result. Hence, you can use the zero flag to test for equality or inequality.
- S: The sign flag is set to one if the result is negative. At first glance, you might think that this flag would be set if AX is less than BX but this isn’t always the case. If AX=\$7FFF and BX=-1 (\$FFFF) subtracting AX from BX produces \$8000, which is negative (and so the sign flag will be set). So, for signed comparisons anyway, the sign flag doesn’t contain the proper status. For unsigned operands, consider AX=\$FFFF and BX=1. AX is greater than BX but their difference is \$FFFE which is still negative. As it turns out, the sign flag and the overflow flag, taken together, can be used for comparing two signed values.
- O: The overflow flag is set after a CMP operation if the difference of AX and BX produced an overflow or underflow. As mentioned above, the sign flag and the overflow flag are both used when performing signed comparisons.
- C: The carry flag is set after a CMP operation if subtracting BX from AX requires a borrow. This occurs only when AX is less than BX where AX and BX are both unsigned values.

Given that the CMP instruction sets the flags in this fashion, you can test the comparison of the two operands with the following flags:

```
cmp( Left, Right );
```

Table 1: Condition Code Settings After CMP

Unsigned operands:	Signed operands:
Z: equality/inequality	Z: equality/inequality
C: Left < Right (C=1) Left >= Right (C=0)	C: no meaning
S: no meaning	S: see below
O: no meaning	O: see below

For signed comparisons, the S (sign) and O (overflow) flags, taken together, have the following meaning:

If ((S=0) and (O=1)) or ((S=1) and (O=0)) then Left < Right when using a signed comparison.

If ((S=0) and (O=0)) or ((S=1) and (O=1)) then Left >= Right when using a signed comparison.

Note that (S xor O) is one if the left operand is less than the right operand. Conversely, (S xor O) is zero if the left operand is greater or equal to the right operand.

To understand why these flags are set in this manner, consider the following examples:

Left	minus	Right	S	O
-----		-----	-	-
\$FFFF (-1)	-	\$FFFE (-2)	0	0
\$8000	-	\$0001	0	1
\$FFFE (-2)	-	\$FFFF (-1)	1	0
\$7FFF (32767)	-	\$FFFF (-1)	1	1

Remember, the CMP operation is really a subtraction, therefore, the first example above computes (-1)-(-2) which is (+1). The result is positive and an overflow did not occur so both the S and O flags are zero. Since (S xor O) is zero, Left is greater than or equal to Right.

In the second example, the CMP instruction would compute (-32768)-(+1) which is (-32769). Since a 16-bit signed integer cannot represent this value, the value wraps around to \$7FFF (+32767) and sets the overflow flag. The result is positive (at least as a 16 bit value) so the CPU clears the sign flag. (S xor O) is one here, so *Left* is less than *Right*.

In the third example above, CMP computes (-2)-(-1) which produces (-1). No overflow occurred so the O flag is zero, the result is negative so the sign flag is one. Since (S xor O) is one, Left is less than Right.

In the fourth (and final) example, CMP computes (+32767)-(-1). This produces (+32768), setting the overflow flag. Furthermore, the value wraps around to \$8000 (-32768) so the sign flag is set as well. Since (S xor O) is zero, Left is greater than or equal to Right.

10.2.4 The SETcc Instructions

The *set on condition* (or SETcc) instructions set a single byte operand (register or memory location) to zero or one depending on the values in the flags register. The general formats for the SETcc instructions are

```
setcc( reg8 );
setcc( mem8 );
```

SETcc represents a mnemonic appearing in the following tables. These instructions store a zero into the corresponding operand if the condition is false, they store a one into the eight bit operand if the condition is true.

Table 2: SETcc Instructions That Test Flags

Instruction	Description	Condition	Comments
SETC	Set if carry	Carry = 1	Same as SETB, SETNAE
SETNC	Set if no carry	Carry = 0	Same as SETNB, SETAE
SETZ	Set if zero	Zero = 1	Same as SETE
SETNZ	Set if not zero	Zero = 0	Same as SETNE
SETS	Set if sign	Sign = 1	
SETNS	Set if no sign	Sign = 0	
SETO	Set if overflow	Ovrflw=1	
SETNO	Set if no overflow	Ovrflw=0	
SETP	Set if parity	Parity = 1	Same as SETPE
SETPE	Set if parity even	Parity = 1	Same as SETP
SETNP	Set if no parity	Parity = 0	Same as SETPO
SETPO	Set if parity odd	Parity = 0	Same as SETNP

The SETcc instructions above simply test the flags without any other meaning attached to the operation. You could, for example, use SETC to check the carry flag after a shift, rotate, bit test, or arithmetic operation. You might notice the SETP, SETPE, and SETNP instructions above. They check the *parity* flag. These instructions appear here for completeness, but this text will not consider the uses of the parity flag.

The CMP instruction works synergistically with the SETcc instructions. Immediately after a CMP operation the processor flags provide information concerning the relative values of those operands. They allow you to see if one operand is less than, equal to, greater than, or any combination of these.

There are two additional groups of SETcc instructions that are very useful after a CMP operation. The first group deals with the result of an *unsigned* comparison, the second group deals with the result of a *signed* comparison.

Table 3: SETcc Instructions for Unsigned Comparisons

Instruction	Description	Condition	Comments
SETA	Set if above (>)	Carry=0, Zero=0	Same as SETNBE
SETNBE	Set if not below or equal (not <=)	Carry=0, Zero=0	Same as SETA
SETAE	Set if above or equal (>=)	Carry = 0	Same as SETNC, SETNB
SETNB	Set if not below (not <)	Carry = 0	Same as SETNC, SETAE
SETB	Set if below (<)	Carry = 1	Same as SETC, SETNAE
SETNAE	Set if not above or equal (not >=)	Carry = 1	Same as SETC, SETB
SETBE	Set if below or equal (<=)	Carry = 1 or Zero = 1	Same as SETNA
SETNA	Set if not above (not >)	Carry = 1 or Zero = 1	Same as SETBE
SETE	Set if equal (=)	Zero = 1	Same as SETZ
SETNE	Set if not equal (≠)	Zero = 0	Same as SETNZ

The corresponding table for signed comparisons is

Table 4: SETcc Instructions for Signed Comparisons

Instruction	Description	Condition	Comments
SETG	Set if greater (>)	Sign = Ovrflw and Zero=0	Same as SETNLE
SETNLE	Set if not less than or equal (not <=)	Sign = Ovrflw or Zero=0	Same as SETG
SETGE	Set if greater than or equal (>=)	Sign = Ovrflw	Same as SETNL
SETNL	Set if not less than (not <)	Sign = Ovrflw	Same as SETGE
SETL	Set if less than (<)	Sign ≠ Ovrflw	Same as SETNGE

Table 4: SETcc Instructions for Signed Comparisons

Instruction	Description	Condition	Comments
SETNGE	Set if not greater or equal (not >=)	Sign \neq Ovrflw	Same as SETL
SETLE	Set if less than or equal (<=)	Sign \neq Ovrflw or Zero = 1	Same as SETNG
SETNG	Set if not greater than (not >)	Sign \neq Ovrflw or Zero = 1	Same as SETLE
SETE	Set if equal (=)	Zero = 1	Same as SETZ
SETNE	Set if not equal (\neq)	Zero = 0	Same as SETNZ

The SETcc instructions are particularly valuable because they can convert the result of a comparison to a boolean value (false/true or 0/1). This is especially important when translating statements from a high level language like Pascal or C/C++ into assembly language. The following example shows how to use these instructions in this manner:

```
// Bool := A <= B

mov( A, eax );
cmp( eax, B );
setle( bool ); // bool is a boolean or byte variable.
```

Since the SETcc instructions always produce zero or one, you can use the results with the AND and OR instructions to compute complex boolean values:

```
// Bool := ((A <= B) and (D = E))

mov( A, eax );
cmp( eax, B );
setle( bl );
mov( D, eax );
cmp( eax, E );
sete( bh );
and( bl, bh );
mov( bh, Bool );
```

For more examples, see “Logical (Boolean) Expressions” on page 604.

10.2.5 The TEST Instruction

The 80x86 TEST instruction is to the AND instruction what the CMP instruction is to SUB. That is, the TEST instruction computes the logical AND of its two operands and sets the condition code flags based on the result; it does not, however, store the result of the logical AND back into the destination operand. The syntax for the TEST instruction is similar to AND, it is

```
test( operand1, operand2 );
```

The TEST instruction sets the zero flag if the result of the logical AND operation is zero. It sets the sign flag if the H.O. bit of the result contains a one. TEST always clears the carry and overflow flags.

The primary use of the TEST instruction is to check to see if an individual bit contains a zero or a one. Consider the instruction “test(1, AL);” This instruction logically ANDs AL with the value one; if bit one of AL contains zero, the result will be zero (setting the zero flag) since all the other bits in the constant one are

zero. Conversely, if bit one of AL contains one, then the result is not zero so TEST clears the zero flag. Therefore, you can test the zero flag after this TEST instruction to see if bit zero contains a zero or a one.

The TEST instruction can also check to see if all the bits in a specified set of bits contain zero. The instruction “test(\$F, AL);” sets the zero flag if and only if the L.O. four bits of AL all contain zero.

One very important use of the TEST instruction is to check to see if a register contains zero. The instruction “TEST(reg, reg);” where both operands are the same register will logically AND that register with itself. If the register contains zero, then the result is zero and the CPU will set the zero flag. However, if the register contains a non-zero value, logically ANDing that value with itself produces that same non-zero value, so the CPU clears the zero flag. Therefore, you can test the zero flag immediately after the execution of this instruction (e.g., using the SETZ or SETNZ instructions) to see if the register contains zero. E.g.,

```
test( eax, eax );
setz( bl );           // BL is set to one if EAX contains zero.
```

10.3 Arithmetic Expressions

Probably the biggest shock to beginners facing assembly language for the very first time is the lack of familiar arithmetic expressions. Arithmetic expressions, in most high level languages, look similar to their algebraic equivalents, e.g.,

```
X:=Y*Z;
```

In assembly language, you’ll need several statements to accomplish this same task, e.g.,

```
mov( y, eax );
intmul( z, eax );
mov( eax, x );
```

Obviously the HLL version is much easier to type, read, and understand. This point, more than any other, is responsible for scaring people away from assembly language.

Although there is a lot of typing involved, converting an arithmetic expression into assembly language isn’t difficult at all. By attacking the problem in steps, the same way you would solve the problem by hand, you can easily break down any arithmetic expression into an equivalent sequence of assembly language statements. By learning how to convert such expressions to assembly language in three steps, you’ll discover there is little difficulty to this task.

10.3.1 Simple Assignments

The easiest expressions to convert to assembly language are the simple assignments. Simple assignments copy a single value into a variable and take one of two forms:

```
variable := constant
or
variable := variable
```

Converting the first form to assembly language is trivial, just use the assembly language statement:

```
mov( constant, variable );
```

This MOV instruction copies the constant into the variable.

The second assignment above is slightly more complicated since the 80x86 doesn’t provide a memory-to-memory MOV instruction. Therefore, to copy one memory variable into another, you must move the data through a register. By convention (and for slight efficiency reasons), most programmers tend to use AL/AX/EAX for this purpose. If the AL, AX, or EAX register is available, you should use it for this operation. For example,

```
var1 := var2;
```

becomes

```
mov( var2, eax );
mov( eax, var1 );
```

This is assuming, of course, that *var1* and *var2* are 32-bit variables. Use *AL* if they are eight bit variables, use *AX* if they are 16-bit variables.

Of course, if you're already using *AL*, *AX*, or *EAX* for something else, one of the other registers will suffice. Regardless, you must use a register to transfer one memory location to another.

Although the 80x86 does not support a memory-to-memory move, HLA does provide an extended syntax for the *MOV* instruction that allows two memory operands. However, both operands have to be 16-bit or 32-bit values; eight-bit values won't work. Assuming you want to copy the value of a word or dword object to another variable, you can use the following syntax:

```
mov( var2, var1 );
```

HLA translates this "instruction" into the following two instruction sequence:

```
push( var2 );
pop( var1 );
```

Although this is slightly slower than the two *MOV* instructions, it is convenient.

10.3.2 Simple Expressions

The next level of complexity up from a simple assignment is a simple expression. A simple expression takes the form:

```
var1 := term1 op term2;
```

Var1 is a variable, *term1* and *term2* are variables or constants, and *op* is some arithmetic operator (addition, subtraction, multiplication, etc.).

As simple as this expression appears, most expressions take this form. It should come as no surprise then, that the 80x86 architecture was optimized for just this type of expression.

A typical conversion for this type of expression takes the following form:

```
mov( term1, eax );
op( term2, eax );
mov( eax, var1 );
```

Op is the mnemonic that corresponds to the specified operation (e.g., "+" = add, "-" = sub, etc.).

There are a few inconsistencies you need to be aware of. Of course, when dealing with the multiply and divide instructions on the 80x86, you must use the *AL/AX/EAX* and *DX/EDX* registers. You cannot use arbitrary registers as you can with other operations. Also, don't forget the sign extension instructions if you're performing a division operation and you're dividing one 16/32 bit number by another. Finally, don't forget that some instructions may cause overflow. You may want to check for an overflow (or underflow) condition after an arithmetic operation.

Examples of common simple expressions:

```
x := y + z;
```

```
mov( y, eax );
add( z, eax );
mov( eax, x );
```

```
x := y - z;
```

```

    mov( y, eax );
    sub( z, eax );
    mov( eax, x );

x := y * z; {unsigned}

    mov( y, eax );
    mul( z, eax );    // Don't forget this wipes out EDX.
    mov( eax, x );

x := y div z; {unsigned div}

    mov( y, eax );
    mov( 0, edx );    // Zero extend EAX into EDX.
    div( z, edx:eax );
    mov( eax, x );

x := y idiv z; {signed div}

    mov( y, eax );
    cdq();            // Sign extend EAX into EDX.
    idiv( z, edx:eax );
    mov( eax, z );

x := y mod z; {unsigned remainder}

    mov( y, eax );
    mov( 0, edx );    // Zero extend EAX into EDX.
    mod( z, edx:eax );
    mov( edx, x );    // Note that remainder is in EDX.

x := y imod z; {signed remainder}

    mov( y, eax );
    cdq();            // Sign extend EAX into EDX.
    imod( z, edx:eax );
    mov( edx, x );    // Remainder is in EDX.

```

Certain unary operations also qualify as simple expressions. A good example of a unary operation is negation. In a high level language negation takes one of two possible forms:

```
var := -var or var1 := -var2
```

Note that `var := -constant` is really a simple assignment, not a simple expression. You can specify a negative constant as an operand to the MOV instruction:

```
mov( -14, var );
```

To handle “`var = -var;`” use the single assembly language statement:

```
// var = -var;
```

```
neg( var );
```

If two different variables are involved, then use the following:

```
// var1 = -var2;

mov( var2, eax );
neg( eax );
mov( eax, var1 );
```

10.3.3 Complex Expressions

A complex expression is any arithmetic expression involving more than two terms and one operator. Such expressions are commonly found in programs written in a high level language. Complex expressions may include parentheses to override operator precedence, function calls, array accesses, etc. While the conversion of some complex expressions to assembly language is fairly straight-forward, others require some effort. This section outlines the rules you use to convert such expressions.

A complex expression that is easy to convert to assembly language is one that involves three terms and two operators, for example:

$$w := w - y - z;$$

Clearly the straight-forward assembly language conversion of this statement will require two SUB instructions. However, even with an expression as simple as this one, the conversion is not trivial. There are actually *two ways* to convert this from the statement above into assembly language:

```

mov( w, eax );
sub( y, eax );
sub( z, eax );
mov( eax, w );
and
mov( y, eax );
sub( z, eax );
sub( eax, w );

```

The second conversion, since it is shorter, looks better. However, it produces an incorrect result (assuming Pascal-like semantics for the original statement). Associativity is the problem. The second sequence above computes $W := W - (Y - Z)$ which is not the same as $W := (W - Y) - Z$. How we place the parentheses around the subexpressions can affect the result. Note that if you are interested in a shorter form, you can use the following sequence:

```

mov( y, eax );
add( z, eax );
sub( eax, w );

```

This computes $W := W - (Y + Z)$. This is equivalent to $W := (W - Y) - Z$.

Precedence is another issue. Consider the Pascal expression:

$$X := W * Y + Z;$$

Once again there are two ways we can evaluate this expression:

```

X := (W * Y) + Z;
or
X := W * (Y + Z);

```

By now, you're probably thinking that this text is crazy. Everyone knows the correct way to evaluate these expressions is the second form provided in these two examples. However, you're wrong to think that way. The APL programming language, for example, evaluates expressions solely from right to left and does not give one operator precedence over another.

Most high level languages use a fixed set of precedence rules to describe the order of evaluation in an expression involving two or more different operators. Such programming languages usually compute multiplication and division before addition and subtraction. Those that support exponentiation (e.g., FORTRAN and BASIC) usually compute that before multiplication and division. These rules are intuitive since almost everyone learns them before high school. Consider the expression:

$$X \text{ op}_1 Y \text{ op}_2 Z$$

If op_1 takes precedence over op_2 then this evaluates to $(X \text{ op}_1 Y) \text{ op}_2 Z$ otherwise if op_2 takes precedence over op_1 then this evaluates to $X \text{ op}_1 (Y \text{ op}_2 Z)$. Depending upon the operators and operands involved, these two computations could produce different results.

When converting an expression of this form into assembly language, you must be sure to compute the subexpression with the highest precedence first. The following example demonstrates this technique:

```
// w := x + y * z;

mov( x, ebx );
mov( y, eax );      // Must compute y*z first since "*"
intmul( z, eax );   // has higher precedence than "+".
add( ebx, eax );
mov( eax, w );
```

If two operators appearing within an expression have the same precedence, then you determine the order of evaluation using *associativity* rules. Most operators are *left associative* meaning that they evaluate from left to right. Addition, subtraction, multiplication, and division are all left associative. A *right associative* operator evaluates from right to left. The exponentiation operator in FORTRAN and BASIC is a good example of a right associative operator:

2^{2^3} is equal to $2^{(2^3)}$ *not* $(2^2)^3$

The precedence and associativity rules determine the order of evaluation. Indirectly, these rules tell you where to place parentheses in an expression to determine the order of evaluation. Of course, you can always use parentheses to override the default precedence and associativity. However, the ultimate point is that your assembly code must complete certain operations before others to correctly compute the value of a given expression. The following examples demonstrate this principle:

```
// w := x - y - z

mov( x, eax );      // All the same operator, so we need
sub( y, eax );      // to evaluate from left to right
sub( z, eax );      // because they all have the same
mov( eax, w );      // precedence and are left associative.

// w := x + y * z

mov( y, eax );      // Must compute Y * Z first since
intmul( z, eax );   // multiplication has a higher
add( x, eax );      // precedence than addition.
mov( eax, w );

// w := x / y - z

mov( x, eax );      // Here we need to compute division
cdq();              // first since it has the highest
idiv( y, edx:eax ); // precedence.
sub( z, eax );
mov( eax, w );

// w := x * y * z

mov( y, eax );      // Addition and multiplication are
intmul( z, eax );   // commutative, therefore the order
intmul( x, eax );   // of evaluation does not matter
mov( eax, w );
```

There is one exception to the associativity rule. If an expression involves multiplication and division it is generally better to perform the multiplication first. For example, given an expression of the form:

$W := X/Y * Z$ // Note: this is $\frac{x}{y} \times z$ not $\frac{x}{y \times z}$!

It is usually better to compute $X*Z$ and then divide the result by Y rather than divide X by Y and multiply the quotient by Z . There are two reasons this approach is better. First, remember that the `IMUL` instruction always produces a 64 bit result (assuming 32 bit operands). By doing the multiplication first, you automatically *sign extend* the product into the `EDX` register so you do not have to sign extend `EAX` prior to the division. This saves the execution of the `CDQ` instruction. A second reason for doing the multiplication first is to

increase the accuracy of the computation. Remember, (integer) division often produces an inexact result. For example, if you compute $5/2$ you will get the value two, not 2.5. Computing $(5/2)*3$ produces six. However, if you compute $(5*3)/2$ you get the value seven which is a little closer to the real quotient (7.5). Therefore, if you encounter an expression of the form:

```
w := x/y*z;
```

You can usually convert it to the assembly code:

```
mov( x, eax );
imul( z, eax );      // Note the use of IMUL, not INTMUL!
idiv( y, edx:eax );
mov( eax, w );
```

Of course, if the algorithm you're encoding depends on the truncation effect of the division operation, you cannot use this trick to improve the algorithm. Moral of the story: always make sure you fully understand any expression you are converting to assembly language. Obviously if the semantics dictate that you must perform the division first, do so.

Consider the following Pascal statement:

```
w := x - y * x;
```

This is similar to a previous example except it uses subtraction rather than addition. Since subtraction is not commutative, you cannot compute $y * z$ and then subtract x from this result. This tends to complicate the conversion a tiny amount. Rather than a straight forward multiply and addition sequence, you'll have to load x into a register, multiply y and z leaving their product in a different register, and then subtract this product from x , e.g.,

```
mov( x, ebx );
mov( y, eax );
intmul( x, eax );
sub( eax, ebx );
mov( ebx, w );
```

This is a trivial example that demonstrates the need for *temporary variables* in an expression. This code uses the EBX register to temporarily hold a copy of x until it computes the product of y and z . As your expressions increase in complexity, the need for temporaries grows. Consider the following Pascal statement:

```
w := (a + b) * (y + z);
```

Following the normal rules of algebraic evaluation, you compute the subexpressions inside the parentheses (i.e., the two subexpressions with the highest precedence) first and set their values aside. When you computed the values for both subexpressions you can compute their sum. One way to deal with complex expressions like this one is to reduce it to a sequence of simple expressions whose results wind up in temporary variables. For example, we can convert the single expression above into the following sequence:

```
Temp1 := a + b;
Temp2 := y + z;
w := Temp1 * Temp2;
```

Since converting simple expressions to assembly language is quite easy, it's now a snap to compute the former, complex, expression in assembly. The code is

```
mov( a, eax );
add( b, eax );
mov( eax, Temp1 );
mov( y, eax );
add( z, eax );
mov( eax, Temp2 );
mov( Temp1, eax );
intmul( Temp2, eax );
mov( eax, w );
```


Of course, this code is grossly inefficient and it requires that you declare a couple of temporary variables in your data segment. However, it is very easy to optimize this code by keeping temporary variables, as much as possible, in 80x86 registers. By using 80x86 registers to hold the temporary results this code becomes:

```
mov( a, eax );
add( b, eax );
mov( y, ebx );
add( z, ebx );
intmul( ebx, eax );
mov( eax, w );
```

Yet another example:

```
x := (y+z) * (a-b) / 10;
```

This can be converted to a set of four simple expressions:

```
Temp1 := (y+z)
Temp2 := (a-b)
Temp1 := Temp1 * Temp2
X := Temp1 / 10
```

You can convert these four simple expressions into the assembly language statements:

```
mov( y, eax );      // Compute eax = y+z
add( z, eax );
mov( a, ebx );      // Compute ebx = a-b
sub( b, ebx );
imul( ebx, eax );   // This also sign extends eax into edx.
idiv( 10, edx:eax );
mov( eax, x );
```

The most important thing to keep in mind is that temporary values, if possible, should be kept in registers. Remember, accessing an 80x86 register is much more efficient than accessing a memory location. Use memory locations to hold temporaries only if you've run out of registers to use.

Ultimately, converting a complex expression to assembly language is little different than solving the expression by hand. Instead of actually computing the result at each stage of the computation, you simply write the assembly code that computes the results. Since you were probably taught to compute only one operation at a time, this means that manual computation works on "simple expressions" that exist in a complex expression. Of course, converting those simple expressions to assembly is fairly trivial. Therefore, anyone who can solve a complex expression by hand can convert it to assembly language following the rules for simple expressions.

10.3.4 Commutative Operators

If "@" represents some operator, that operator is *commutative* if the following relationship is always true:

$$(A @ B) = (B @ A)$$

As you saw in the previous section, commutative operators are nice because the order of their operands is immaterial and this lets you rearrange a computation, often making that computation easier or more efficient. Often, rearranging a computation allows you to use fewer temporary variables. Whenever you encounter a commutative operator in an expression, you should always check to see if there is a better sequence you can use to improve the size or speed of your code. The following tables list the commutative and non-commutative operators you typically find in high level languages:

Table 5: Some Common Commutative Binary Operators

Pascal	C/C++	Description
+	+	Addition
*	*	Multiplication
AND	&& or &	Logical or bitwise AND
OR	or	Logical or bitwise OR
XOR	^	(Logical or) Bitwise exclusive-OR
=	==	Equality
<>	!=	Inequality

Table 6: Some Common Noncommutative Binary Operators

Pascal	C/C++	Description
-	-	Subtraction
/ or DIV	/	Division
MOD	%	Modulo or remainder
<	<	Less than
<=	<=	Less than or equal
>	>	Greater than
>=	>=	Greater than or equal

10.4 Logical (Boolean) Expressions

Consider the following expression from a Pascal program:

```
B := ((X=Y) and (A <= C)) or ((Z-A) <> 5);
```

B is a boolean variable and the remaining variables are all integers.

How do we represent boolean variables in assembly language? Although it takes only a single bit to represent a boolean value, most assembly language programmers allocate a whole byte or word for this purpose (as such, HLA also allocates a whole byte for a BOOLEAN variable). With a byte, there are 256 possible values we can use to represent the two values *true* and *false*. So which two values (or which two sets of values) do we use to represent these boolean values? Because of the machine's architecture, it's much easier to test for conditions like zero or not zero and positive or negative rather than to test for one of two particular boolean values. Most programmers (and, indeed, some programming languages like "C") choose zero to represent false and anything else to represent true. Some people prefer to represent true and false with one

and zero (respectively) and not allow any other values. Others select all one bits (\$FFFF_FFFF, \$FFFF, or \$FF) for true and 0 for false. You could also use a positive value for true and a negative value for false. All these mechanisms have their own advantages and drawbacks.

Using only zero and one to represent false and true offers two very big advantages: (1) The SETcc instructions produce these results so this scheme is compatible with those instructions; (2) the 80x86 logical instructions (AND, OR, XOR and, to a lesser extent, NOT) operate on these values exactly as you would expect. That is, if you have two boolean variables A and B, then the following instructions perform the basic logical operations on these two variables:

```
// c = a AND b;

    mov( a, al );
    and( b, al );
    mov( al, c );

// c = a OR b;

    mov( a, al );
    or( b, al );
    mov( al, c );

// c = a XOR b;

    mov( a, al );
    xor( b, al );
    mov( al, c );

// b = not a;

    mov( a, al );      // Note that the NOT instruction does not
    not( al );         // properly compute al = not al by itself.
    and( 1, al );     // I.e., (not 0) does not equal one. The AND
    mov( al, b );     // instruction corrects this problem.

    mov( a, al );     // Another way to do b = not a;
    xor( 1, al );     // Inverts bit zero.
    mov( al, b );
```

Note, as pointed out above, that the NOT instruction will not properly compute logical negation. The bitwise not of zero is \$FF and the bitwise not of one is \$FE. Neither result is zero or one. However, by ANDing the result with one you get the proper result. Note that you can implement the NOT operation more efficiently using the “xor(1, ax);” instruction since it only affects the L.O. bit.

As it turns out, using zero for false and anything else for true has a lot of subtle advantages. Specifically, the test for true or false is often implicit in the execution of any logical instruction. However, this mechanism suffers from a very big disadvantage: you cannot use the 80x86 AND, OR, XOR, and NOT instructions to implement the boolean operations of the same name. Consider the two values \$55 and \$AA. They’re both non-zero so they both represent the value true. However, if you logically AND \$55 and \$AA together using the 80x86 AND instruction, the result is zero. True AND true should produce true, not false. Although you can account for situations like this, it usually requires a few extra instructions and is somewhat less efficient when computing boolean operations.

A system that uses non-zero values to represent true and zero to represent false is an *arithmetic logical system*. A system that uses the two distinct values like zero and one to represent false and true is called a *boolean logical system*, or simply a boolean system. You can use either system, as convenient. Consider again the boolean expression:

```
B := ((X=Y) and (A <= D)) or ((Z-A) <> 5);
```

The simple expressions resulting from this expression might be:

```
mov( x, eax );
cmp( y, eax );
```

```

sete( al );          // AL := x = y;

mov( a, ebx );
cmp( ebx, d );
setle( bl );        // BL := a <= d;
and( al, bl );      // BL := (x=y) and (a <= d);

mov( z, eax );
sub( a, eax );
cmp( eax, 5 );
setne( al );
or( bl, al );       // AL := ((X=Y) and (A <= D)) or ((Z-A) <> 5);
mov( al, b );

```

When working with boolean expressions don't forget that you might be able to optimize your code by simplifying those boolean expressions. You can use algebraic transformations (especially DeMorgan's theorems) to help reduce the complexity of an expression. In the chapter on low-level control structures you'll also see how to use control flow to calculate a boolean result. This is generally quite a bit more efficient than using *complete boolean evaluation* as the examples in this section teach.

10.5 Machine and Arithmetic Idioms

An idiom is an idiosyncrasy. Several arithmetic operations and 80x86 instructions have idiosyncrasies that you can take advantage of when writing assembly language code. Some people refer to the use of machine and arithmetic idioms as “tricky programming” that you should always avoid in well written programs. While it is wise to avoid tricks just for the sake of tricks, many machine and arithmetic idioms are well-known and commonly found in assembly language programs. Some of them can be really tricky, but a good number of them are simply “tricks of the trade.” This text cannot even begin to present all of the idioms in common use today; they are too numerous and the list is constantly changing. Nevertheless, there are some very important idioms that you will see all the time, so it makes sense to discuss those.

10.5.1 Multiplying without MUL, IMUL, or INTMUL

If you take a quick look at the timing for the multiply instruction, you'll notice that the execution time for this instruction is often long¹. When multiplying by a constant, you can sometimes avoid the performance penalty of the MUL, IMUL, and INTMUL instructions by using shifts, additions, and subtractions to perform the multiplication.

Remember, a SHL instruction computes the same result as multiplying the specified operand by two. Shifting to the left two bit positions multiplies the operand by four. Shifting to the left three bit positions multiplies the operand by eight. In general, shifting an operand to the left n bits multiplies it by 2^n . Any value can be multiplied by some constant using a series of shifts and adds or shifts and subtractions. For example, to multiply the AX register by ten, you need only multiply it by eight and then add in two times the original value. That is, $10*AX = 8*AX + 2*AX$. The code to accomplish this is

```

shl( 1, ax );       // Multiply AX by two.
mov( ax, bx );      // Save 2*AX for later.
shl( 2, ax );       // Multiply ax by eight (*4 really, but it contains *2).
add( bx, ax );      // Add in AX*2 to AX*8 to get AX*10.

```

The AX register (or just about any register, for that matter) can often be multiplied by many constant values much faster using SHL than by using the MUL instruction. This may seem hard to believe since it only takes one instruction to compute this product:

1. Actually, this is specific to a given processor. Some processors execute the INTMUL instruction fairly fast.

```
intmul( 10, ax );
```

However, if you look at the timings, the shift and add example above requires fewer clock cycles on many processors in the 80x86 family than the MUL instruction. Of course, the code is somewhat larger (by a few bytes), but the performance improvement is usually worth it. Of course, on the later 80x86 processors, the multiply instructions are quite a bit faster than the earlier processors, but the shift and add scheme is often faster on these processors as well.

You can also use subtraction with shifts to perform a multiplication operation. Consider the following multiplication by seven:

```
mov( eax, ebx );    // Save EAX * 1
shl( 3, eax );      // EAX = EAX * 8
sub( ebx, eax );    // EAX*8 - EAX*1 is EAX*7
```

This follows directly from the fact that $EAX*7 = (EAX*8) - EAX$.

A common error made by beginning assembly language students is subtracting or adding one or two rather than $EAX*1$ or $EAX*2$. The following does *not* compute $eax*7$:

```
shl( 3, eax );
sub( 1, eax );
```

It computes $(8*EAX)-1$, something entirely different (unless, of course, $EAX = 1$). Beware of this pitfall when using shifts, additions, and subtractions to perform multiplication operations.

You can also use the LEA instruction to compute certain products. The trick is to use the scaled index addressing modes. The following examples demonstrate some simple cases:

```
lea( eax, [ecx][ecx] );    // EAX := ECX * 2
lea( eax, [eax]eax*2 ] );  // EAX := EAX * 3
lea( eax, [eax*4] );       // EAX := EAX * 4
lea( eax, [ebx][ebx*4] );  // EAX := EBX * 5
lea( eax, [eax*8] );       // EAX := EAX * 8
lea( eax, [edx][edx*8] );  // EAX := EDX * 9
```

10.5.2 Division Without DIV or IDIV

Much as the SHL instruction can be used for simulating a multiplication by some power of two, you may use the SHR and SAR instructions to simulate a division by a power of two. Unfortunately, you cannot use shifts, additions, and subtractions to perform a division by an arbitrary constant as easily as you can use these instructions to perform a multiplication operation.

Another way to perform division is to use the multiply instructions. You can divide by some value by multiplying by its reciprocal. Since the multiply instruction is faster than the divide instruction; multiplying by a reciprocal is usually faster than division.

Now you're probably wondering "how does one multiply by a reciprocal when the values we're dealing with are all integers?" The answer, of course, is that we must cheat to do this. If you want to multiply by one tenth, there is no way you can load the value $1/10^{\text{th}}$ into an 80x86 register prior to performing the multiplication. However, we could multiply $1/10^{\text{th}}$ by 10, perform the multiplication, and then divide the result by ten to get the final result. Of course, this wouldn't buy you anything at all, in fact it would make things worse since you're now doing a multiplication by ten as well as a division by ten. However, suppose you multiply $1/10^{\text{th}}$ by 65,536 (6553), perform the multiplication, and then divide by 65,536. This would still perform the correct operation and, as it turns out, if you set up the problem correctly, you can get the division operation for free. Consider the following code that divides AX by ten:

```
mov( 6554, dx );    // 6554 = round( 65,536/10 ).
mul( dx, ax );
```

This code leaves $AX/10$ in the DX register.

To understand how this works, consider what happens when you multiply AX by 65,536 (\$10000). This simply moves AX into DX and sets AX to zero (a multiply by \$10000 is equivalent to a shift left by sixteen bits). Multiplying by 6,554 (65,536 divided by ten) puts AX divided by ten into the DX register. Since MUL is faster than DIV, this technique runs a little faster than using a straight division.

Multiplying by the reciprocal works well when you need to divide by a constant. You could even use it to divide by a variable, but the overhead to compute the reciprocal only pays off if you perform the division many, many times (by the same value).

10.5.3 Implementing Modulo-N Counters with AND

If you want to implement a counter variable that counts up to 2^n-1 and then resets to zero, simply using the following code:

```
inc( CounterVar );
and( nBits, CounterVar );
```

where *nBits* is a binary value containing n one bits right justified in the number. For example, to create a counter that cycles between zero and fifteen, you could use the following:

```
inc( CounterVar );
and( %00001111, CounterVar );
```

10.5.4 Careless Use of Machine Idioms

One problem with using machine idioms is that the machines change over time. The DOS/16-bit version of this text recommends the use of several machine idioms in addition to those this chapter presents. Unfortunately, as time passed Intel improved the processor and tricks that used to provide a performance benefit are actually slower on the newer processors. Therefore, you should be careful about employing common “tricks” you pick up; they may not actually improve the code.

10.6 The HLA (Pseudo) Random Number Unit

The HLA *rand.hhf* module provides a set of pseudo-random generators that returns seemingly random values on each call. These pseudo-random number generator functions are great for writing games and other simulations that require a sequence of values that the user can not easily guess. These functions return a 32-bit value in the EAX register. You can treat the result as a signed or unsigned value as appropriate for your application.

The *rand.hhf* library module includes the following functions:

```
procedure rand.random; returns( "eax" );
procedure rand.range( startRange:dword; endRange:dword ); returns( "eax" );

procedure rand.uniform; returns( "eax" );
procedure rand.urange( startRange:dword; endRange:dword ); returns( "eax" );

procedure rand.randomize;
```

The *rand.random* and *rand.uniform* procedures are both functions that return a 32-bit pseudo-random number in the EAX register. They differ only in the algorithm they use to compute the random number sequence (*rand.random* uses a standard linear congruential generator, *rand.uniform* uses an additive generator. See Knuth’s “The Art of Computer Programming” Volume Two for details on these two algorithms).

The *rand.range* and *rand.urange* functions return a pseudo-random number that falls between two values passed as parameters (inclusive). These routines use better algorithms than the typical “mod the result

by the range of values and add the starting value” algorithm that naive users often employ to limit random numbers to a specific range (that naive algorithm generally produces a stream of numbers that is somewhat less than random).

By default, the random number generators in the HLA Standard Library generate the same sequence of numbers every time you run a program. While this may not seem random at all (and statistically, it certainly is not random), this is generally what you want in a random number generator. The numbers should appear to be random but you usually need to be able to generate the same sequence over and over again when testing your program. After all, a defect you encounter with one random sequence may not be apparent when using a different random number sequence. By emitting the same sequence over and over again, your programs become deterministic so you can properly test them across several runs of the program.

Once your program is tested and operational, you might want your random number generator to generate a different sequence every time you run the program. For example, if you write a game and that game uses a pseudo-random sequence to control the action, the end user may detect a pattern and play the game accordingly if the random number generator always returns the same sequence of numbers.

To alleviate this problem, the HLA Standard Library `rand` module provides the `rand.randomize` procedure. This procedure reads the current date and time (in milliseconds) and, on processors that support it, reads the CPU’s timestamp counter to generate an almost random set of bits as the starting random number generator value. Calling the `rand.randomize` procedure at the beginning of your program essentially guarantees that different executions of the program will produce a different sequence of random numbers.

Note that you cannot make the sequence “more random” by calling `rand.randomize` multiple times. In fact, since `rand.randomize` generates a new seed based on the date and time, calling `rand.randomize` multiple times in your program will actually generate a less random sequence (since time is an ever increasing value, not a random value). So make at most one call to `rand.randomize` and leave it up to the random number generators to take it from there.

Note that `rand.randomize` will randomize both the `rand.random` and `rand.uniform` random number generators. You do not need separate calls for the two different generators nor can you randomize one without randomizing the other.

One attribute of a random number generator is “how uniform are the results the generator returns.” A uniform random number generator² that produces a 32-bit result returns a sequence of values that are evenly distributed throughout the 32-bit range of values. That is, any return result is as equally likely as any other return result. Good random number generators don’t tend to bunch numbers up in groups.

The following program code provides a simple test of the random number generators by plotting asterisks at random positions on the screen. This program works by choosing two random numbers, one between zero and 79, the other between zero and 23. Then the program uses the `console.puts` function to print a single asterisk at the (X,Y) coordinate on the screen specified by these two random numbers (therefore, this code runs only under Windows). After 10,000 iterations of this process the program stops and lets you observe the result. **Note:** since random number generators generate random numbers, you should not expect this program to fill the entire screen with asterisks in only 10,000 iterations.

```

program testRandom;
#include( "stdlib.hhf" );

begin testRandom;

    console.cls();
    mov( 10_000, ecx );
    repeat

        // Generate a random X-coordinate

```

2. Despite their names, both `rand.uniform` and `rand.random` generate a uniformly distributed set of pseudo-random numbers.

```
// using rand.range.

rand.range( 0, 79 );
mov( eax, ebx );           // Save the X-coordinate for now.

// Generate a random Y-coordinate
// using rand.urange.

rand.urange( 0, 23 );

// Print an asterisk at
// the specified coordinate on the screen.

console.puts( ax, bx, "*" );

// Repeat this 10,000 times to get
// a good distribution of values.

dec( ecx );

until( @z );

// Position the cursor at the bottom of the
// screen so we can observe the results.

console.gotoxy( 24, 0 );

end testRandom;
```

Program 10.1 Screen Plot Test of the HLA Random Number Generators

The `rand.hhf` module also provides an *iterator* that generates a random sequence of value in the range $0..n-1$. However, a discussion of this function must wait until we cover iterators in a later chapter.

10.7 Putting It All Together

This chapter finished the presentation of the integer arithmetic instructions on the 80x86. Then it demonstrated how to convert expressions from a high level language syntax into assembly language. This chapter concluded by teaching you a few assembly language tricks you will commonly find in programs. By the conclusion of this chapter you are (hopefully) in a position where you can easily evaluate arithmetic expressions in your assembly language programs.

11.1 Chapter Overview

This chapter discusses the implementation of floating point arithmetic computation in assembly language. By the conclusion of this chapter you should be able to translate arithmetic expressions and assignment statements involving floating point operands from high level languages like Pascal and C/C++ into 80x86 assembly language.

11.2 Floating Point Arithmetic

When the 8086 CPU first appeared in the late 1970's, semiconductor technology was not to the point where Intel could put floating point instructions directly on the 8086 CPU. Therefore, they devised a scheme whereby they could use a second chip to perform the floating point calculations – the floating point unit (or FPU)¹. They released their original floating point chip, the 8087, in 1980. This particular FPU worked with the 8086, 8088, 80186, and 80188 CPUs. When Intel introduced the 80286 CPU, they released a redesigned 80287 FPU chip to accompany it. Although the 80287 was compatible with the 80386 CPU, Intel designed a better FPU, the 80387, for use in 80386 systems. The 80486 CPU was the first Intel CPU to include an on-chip floating point unit. Shortly after the release of the 80486, Intel introduced the 80486sx CPU that was an 80486 without the built-in FPU. To get floating point capabilities on this chip, you had to add an 80487 chip, although the 80487 was really nothing more than a full-blown 80486 which took over for the “sx” chip in the system. Intel's Pentium chips provide a high-performance floating point unit directly on the CPU. There is no (Intel) floating point coprocessor available for the Pentium chip.

Collectively, we will refer to all these chips as the 80x87 FPU. Given the obsolescence of the 8086, 80286, 8087, 80287, 80387, and 80487 chips, this text will concentrate on the Pentium and later chips. There are some differences between the Pentium floating point units and the earlier FPUs. If you need to write code that will execute on those earlier machines, you should consult the appropriate Intel documentation for those devices.

11.2.1 FPU Registers

The 80x86 FPUs add 13 registers to the 80x86 and later processors: eight floating point data registers, a control register, a status register, a tag register, an instruction pointer, and a data pointer. The data registers are similar to the 80x86's general purpose register set insofar as all floating point calculations take place in these registers. The control register contains bits that let you decide how the FPU handles certain degenerate cases like rounding of inaccurate computations, it contains bits that control precision, and so on. The status register is similar to the 80x86's flags register; it contains the condition code bits and several other floating point flags that describe the state of the FPU. The tag register contains several groups of bits that determine the state of the value in each of the eight general purpose registers. The instruction and data pointer registers contain certain state information about the last floating point instruction executed. We will not consider the last three registers in this text, see the Intel documentation for more details.

1. Intel has also referred to this device as the Numeric Data Processor (NDP), Numeric Processor Extension (NPX), and math coprocessor.

11.2.1.1 FPU Data Registers

The FPUs provide eight 80 bit data registers organized as a stack. This is a significant departure from the organization of the general purpose registers on the 80x86 CPU that comprise a standard general-purpose register set. HLA refers to these registers as ST0, ST1, ..., ST7.

The biggest difference between the FPU register set and the 80x86 register set is the stack organization. On the 80x86 CPU, the AX register is always the AX register, no matter what happens. On the FPU, however, the register set is an eight element stack of 80 bit floating point values (see Figure 11.1).

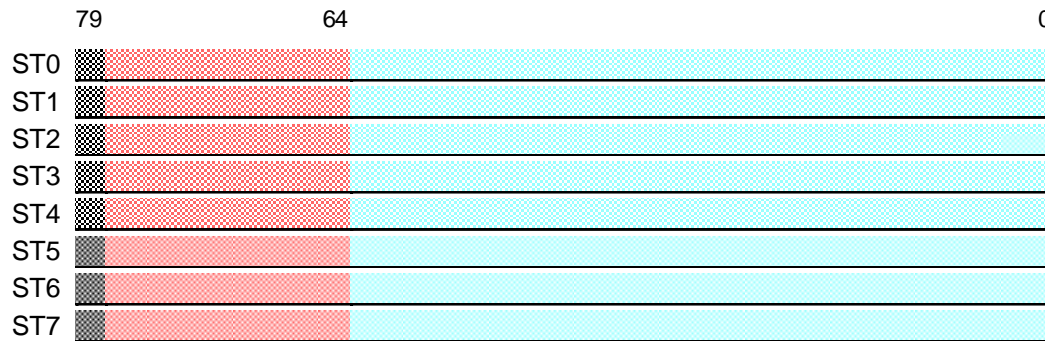


Figure 11.1 FPU Floating Point Register Stack

ST0 refers to the item on the top of the stack, ST1 refers to the next item on the stack, and so on. Many floating point instructions push and pop items on the stack; therefore, ST1 will refer to the previous contents of ST0 after you push something onto the stack. It will take some thought and practice to get used to the fact that the registers are changing under you, but this is an easy problem to overcome.

11.2.1.2 The FPU Control Register

When Intel designed the 80x87 (and, essentially, the IEEE floating point standard), there were no standards in floating point hardware. Different (mainframe and mini) computer manufacturers all had different and incompatible floating point formats. Unfortunately, much application software had been written taking into account the idiosyncrasies of these different floating point formats. Intel wanted to design an FPU that could work with the majority of the software out there (keep in mind, the IBM PC was three to four years away when Intel began designing the 8087, they couldn't rely on that "mountain" of software available for the PC to make their chip popular). Unfortunately, many of the features found in these older floating point formats were mutually incompatible. For example, in some floating point systems rounding would occur when there was insufficient precision; in others, truncation would occur. Some applications would work with one floating point system but not with the other. Intel wanted as many applications as possible to work with as few changes as possible on their 80x87 FPUs, so they added a special register, the FPU *control register*, that lets the user choose one of several possible operating modes for their FPU.

The 80x87 control register contains 16 bits organized as shown in Figure 11.2.

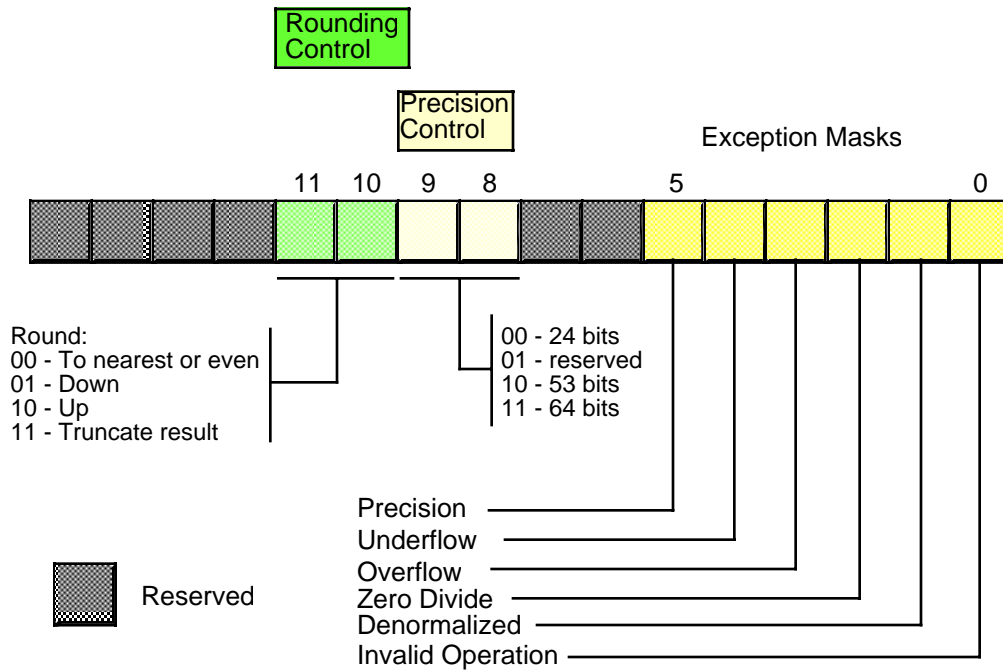


Figure 11.2 FPU Control Register

Bits 10 and 11 provide rounding control according to the following values:

Table 1: Rounding Control

Bits 10 & 11	Function
00	To nearest or even
01	Round down
10	Round up
11	Truncate

The “00” setting is the default. The FPU rounds values above one-half of the least significant bit up. It rounds values below one-half of the least significant bit down. If the value below the least significant bit is exactly one-half of the least significant bit, the FPU rounds the value towards the value whose least significant bit is zero. For long strings of computations, this provides a reasonable, automatic, way to maintain maximum precision.

The round up and round down options are present for those computations where it is important to keep track of the accuracy during a computation. By setting the rounding control to round down and performing the operation, then repeating the operation with the rounding control set to round up, you can determine the minimum and maximum ranges between which the true result will fall.

The truncate option forces all computations to truncate any excess bits during the computation. You will rarely use this option if accuracy is important to you. However, if you are porting older software to the FPU, you might use this option to help when porting the software. One place where this option is extremely useful is when converting a floating point value to an integer. Since most software expects floating point to integer conversions to truncate the result, you will need to use the truncation rounding mode to achieve this.

Bits eight and nine of the control register specify the precision during computation. This capability is provided to allow compatibility with older software as required by the IEEE 754 standard. The precision control bits use the following values:

Table 2: Mantissa Precision Control Bits

Bits 8 & 9	Precision Control
00	24 bits
01	Reserved
10	53 bits
11	64 bits

Some CPUs may operate faster with floating point values whose precision is 53 bits (i.e., 64-bit floating point format) rather than 64 bits (i.e., 80-bit floating point format). Please see the documentation for your specific processor for details. Generally, the CPU defaults these bits to %11 to select the 64-bit mantissa precision.

Bits zero through five are the *exception masks*. These are similar to the interrupt enable bit in the 80x86's flags register. If these bits contain a one, the corresponding condition is ignored by the FPU. However, if any bit contains zero, and the corresponding condition occurs, then the FPU immediately generates an interrupt so the program can handle the degenerate condition.

Bit zero corresponds to an invalid operation error. This generally occurs as the result of a programming error. Problems which raise the invalid operation exception include pushing more than eight items onto the stack or attempting to pop an item off an empty stack, taking the square root of a negative number, or loading a non-empty register.

Bit one masks the *denormalized* interrupt that occurs whenever you try to manipulate denormalized values. Denormalized exceptions occur when you load arbitrary extended precision values into the FPU or work with very small numbers just beyond the range of the FPU's capabilities. Normally, you would probably *not* enable this exception. If you enable this exception and the FPU generates this interrupt, the HLA run-time system raises the *ex.fDenormal* exception.

Bit two masks the *zero divide* exception. If this bit contains zero, the FPU will generate an interrupt if you attempt to divide a nonzero value by zero. If you do not enable the zero division exception, the FPU will produce NaN (not a number) whenever you perform a zero division. It's probably a good idea to enable this exception by programming a zero into this bit. Note that if your program generates this interrupt, the HLA run-time system will raise the *ex.fDivByZero* exception.

Bit three masks the *overflow* exception. The FPU will raise the overflow exception if a calculation overflows or if you attempt to store a value which is too large to fit into a destination operand (e.g., storing a large extended precision value into a single precision variable). If you enable this exception and the FPU generates this interrupt, the HLA run-time system raises the *ex.fOverflow* exception.

Bit four, if set, masks the *underflow* exception. Underflow occurs when the result is too *small* to fit in the destination operand. Like overflow, this exception can occur whenever you store a small extended precision value into a smaller variable (single or double precision) or when the result of a computation is too small for extended precision. If you enable this exception and the FPU generates this interrupt, the HLA run-time system raises the *ex.fUnderflow* exception.

Bit five controls whether the *precision* exception can occur. A precision exception occurs whenever the FPU produces an imprecise result, generally the result of an internal rounding operation. Although many operations will produce an exact result, many more will not. For example, dividing one by ten will produce an inexact result. Therefore, this bit is usually one since inexact results are very common. If you enable this exception and the FPU generates this interrupt, the HLA run-time system raises the *ex.InexactResult* exception.

Bits six and thirteen through fifteen in the control register are currently undefined and reserved for future use. Bit seven is the interrupt enable mask, but it is only active on the 8087 FPU; a zero in this bit enables 8087 interrupts and a one disables FPU interrupts.

The FPU provides two instructions, FLDCW (load control word) and FSTCW (store control word), that let you load and store the contents of the control register. The single operand to these instructions must be a 16 bit memory location. The FLDCW instruction loads the control register from the specified memory location, FSTCW stores the control register into the specified memory location. The syntax for these instructions is

```
fldcw( mem16 );
fstcw( mem16 );
```

Here's some example code that sets the rounding control to "truncate result" and sets the rounding precision to 24 bits:

```
static
fcw16: word;
.
.
.
fstcw( fcw16 );
mov( fcw16, ax );
and( $f0ff, ax );      // Clears bits 8-11.
or( $0c00, ax );      // Rounding control=%11, Precision = %00.
mov( ax, fcw16 );
fldcw( fcw16 );
```

11.2.1.3 The FPU Status Register

The FPU status register provides the status of the coprocessor at the instant you read it. The FSTSW instruction stores the 16 bit floating point status register into a word variable. The status register is a 16 bit register, its layout appears in Figure 11.3.

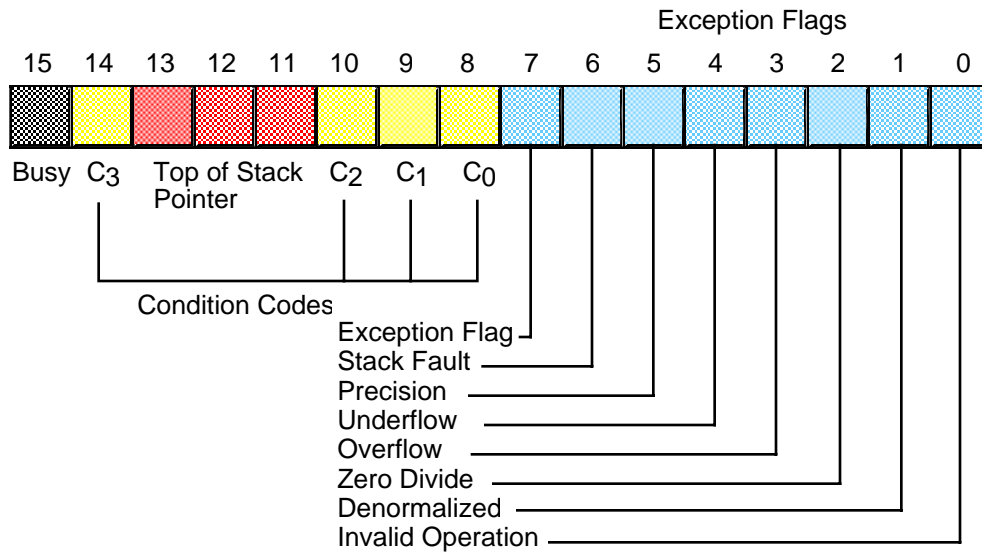


Figure 11.3 The FPU Status Register

Bits zero through five are the exception flags. These bits appear in the same order as the exception masks in the control register. If the corresponding condition exists, then the bit is set. These bits are independent of the exception masks in the control register. The FPU sets and clears these bits regardless of the corresponding mask setting.

Bit six indicates a *stack fault*. A stack fault occurs whenever there is a stack overflow or underflow. When this bit is set, the C₁ condition code bit determines whether there was a stack overflow (C₁=1) or stack underflow (C₁=0) condition.

Bit seven of the status register is set if *any* error condition bit is set. It is the logical OR of bits zero through five. A program can test this bit to quickly determine if an error condition exists.

Bits eight, nine, ten, and fourteen are the coprocessor condition code bits. Various instructions set the condition code bits as shown in the following table:

Table 3: FPU Condition Code Bits

Instruction	Condition Code Bits				Condition
	C3	C2	C1	C0	
fcom,	0	0	X	0	ST > source
fcomp,	0	0	X	1	ST < source
fcompp,	1	0	X	0	ST = source
ficom,	1	1	X	1	ST or source undefined
ficompp					
X = Don't care					

Table 3: FPU Condition Code Bits

Instruction	Condition Code Bits				Condition
	C3	C2	C1	C0	
ftst	0	0	X	0	ST is positive
	0	0	X	1	ST is negative
	1	0	X	0	ST is zero (+ or -)
	1	1	X	1	ST is uncomparable
fxam	0	0	0		+ Unnormalized
	0	0	1	0	-Unnormalized
	0	1	0	0	+Normalized
	0	1	1	0	-Normalized
	1	0	0	0	+0
	1	0	1	0	-0
	1	1	0	0	+Denormalized
	1	1	1	0	-Denormalized
	0	0	0	1	+NaN
	0	0	1	1	-NaN
	0	1	0	1	+Infinity
	0	1	1	1	-Infinity
	1	X	X	1	Empty register
fucom, fucomp, fucompp	0	0	X	0	ST > source
	0	0	X	1	ST < source
	1	0	X	0	ST = source
	1	1	X	1	Unordered
X = Don't care					

Table 4: Condition Code Interpretations

Instruction(s)	Condition Code Bits			
	C ₀	C ₃	C ₂	C ₁
fcom, fcomp, fcmp, ftst, fucom, fucomp, fucompp, ficom, ficomp	Result of comparison. See previous table.	Result of comparison. See previous table.	Operands are not comparable	Result of comparison. See previous table. Also denotes stack overflow/underflow if stack exception bit is set.

Table 4: Condition Code Interpretations

Instruction(s)	Condition Code Bits			
	C ₀	C ₃	C ₂	C ₁
fxam	See previous table.	See previous table.	See previous table.	Sign of result, or stack overflow/underflow (if stack exception bit is set).
fprem, fprem1	Bit 2 of remainder	Bit 0 of remainder	0- reduction done. 1- reduction incomplete.	Bit 1 of remainder or stack overflow/underflow (if stack exception bit is set).
fist, fbstp, frndint, fst, fstp, fadd, fmul, fdiv, fdivr, fsub, fsubr, fscale, fsqrt, fpatan, f2xm1, fyl2x, fyl2xp1	Undefined	Undefined	Undefined	Round up occurred or stack overflow/underflow (if stack exception bit is set).
fptan, fsin, fcos, fsincos	Undefined	Undefined	0- reduction done. 1- reduction incomplete.	Round up occurred or stack overflow/underflow (if stack exception bit is set).
fchs, fabs, fxch, fincstp, fdecstp, <i>constant loads</i> , fextract, fld, fild, fbld, fstp (80 bit)	Undefined	Undefined	Undefined	Zero result or stack overflow/underflow (if stack exception bit is set).
fldenv, fstor	Restored from memory operand.	Restored from memory operand.	Restored from memory operand.	Restored from memory operand.
fldcw, fstenv, fstcw, fstsw, fclex	Undefined	Undefined	Undefined	Undefined
finit, fsave	Cleared to zero.	Cleared to zero.	Cleared to zero.	Cleared to zero.

Bits 11-13 of the FPU status register provide the register number of the top of stack. During computations, the FPU adds (modulo eight) the *logical* register numbers supplied by the programmer to these three bits to determine the *physical* register number at run time.

Bit 15 of the status register is the *busy* bit. It is set whenever the FPU is busy. Most programs will have little reason to access this bit.

11.2.2 FPU Data Types

The FPU supports seven different data types: three integer types, a packed decimal type, and three floating point types. The integer type provides for 64-bit integers, although it is often faster to do the 64-bit arithmetic using the integer unit of the CPU (see the chapter on Advanced Arithmetic). Certainly it is often faster to do 16-bit and 32-bit integer arithmetic using the standard integer registers. The packed decimal type provides a 17 digit signed decimal (BCD) integer. The primary purpose of the BCD format is to convert between strings and floating point values. The remaining three data types are the 32 bit, 64 bit, and 80 bit floating point data types we've looked at so far. The 80x87 data types appear in Figure 11.4, Figure 11.5, and Figure 11.6.

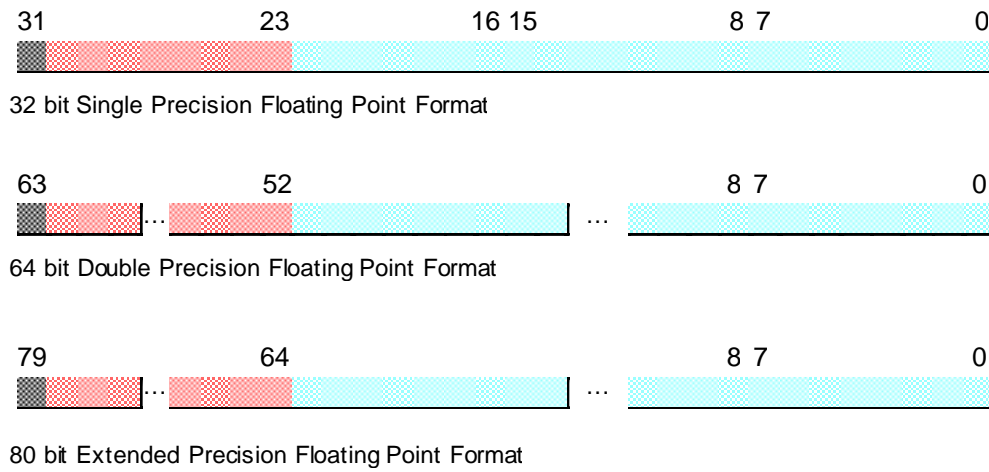


Figure 11.4 FPU Floating Point Formats

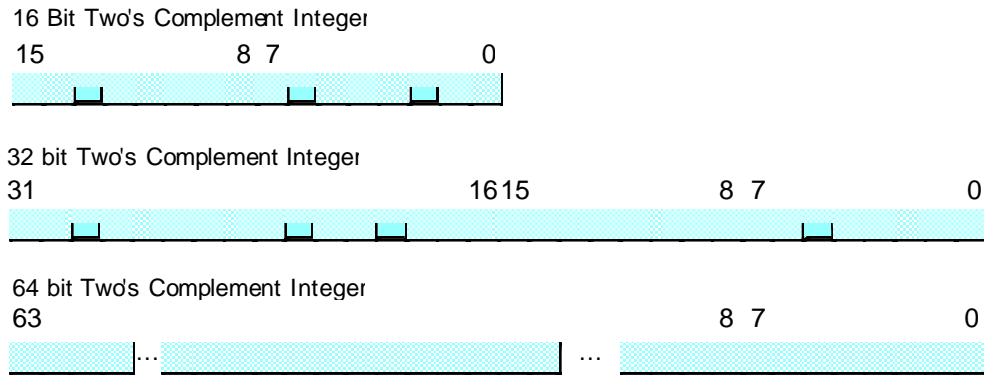


Figure 11.5 FPU Integer Formats

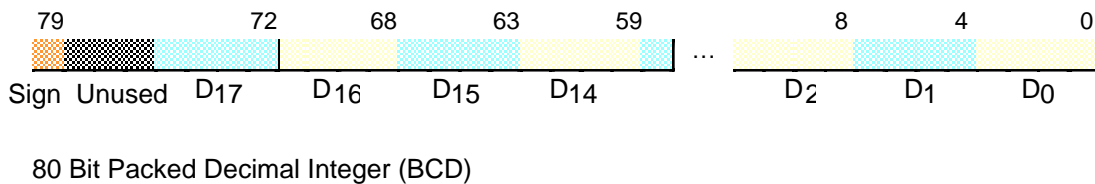


Figure 11.6 FPU Packed Decimal Format

The FPU generally stores values in a *normalized* format. When a floating point number is normalized, the H.O. bit of the mantissa is always one. In the 32 and 64 bit floating point formats, the FPU does not actually store this bit, the FPU always assumes that it is one. Therefore, 32 and 64 bit floating point numbers are always normalized. In the extended precision 80 bit floating point format, the FPU does *not* assume that the H.O. bit of the mantissa is one, the H.O. bit of the mantissa appears as part of the string of bits.

Normalized values provide the greatest precision for a given number of bits. However, there are a large number of non-normalized values which we *cannot* represent with the 80-bit format. These values are very close to zero and represent the set of values whose mantissa H.O. bit is not zero. The FPUs support a special 80-bit form known as *denormalized* values. Denormalized values allow the FPU to encode very small values it cannot encode using normalized values, but at a price. Denormalized values offer fewer bits of precision than normalized values. Therefore, using denormalized values in a computation may introduce some slight inaccuracy into a computation. Of course, this is always better than underflowing the denormalized value to zero (which could make the computation even less accurate), but you must keep in mind that if you work with very small values you may lose some accuracy in your computations. Note that the FPU status register contains a bit you can use to detect when the FPU uses a denormalized value in a computation.

11.2.3 The FPU Instruction Set

The FPU adds over 80 new instructions to the 80x86 instruction set. We can classify these instructions as *data movement instructions*, *conversions*, *arithmetic instructions*, *comparisons*, *constant instructions*, *transcendental instructions*, and *miscellaneous instructions*. The following sections describe each of the instructions in these categories.

11.2.4 FPU Data Movement Instructions

The data movement instructions transfer data between the internal FPU registers and memory. The instructions in this category are FLD, FST, FSTP, and FXCH. The FLD instruction always pushes its operand onto the floating point stack. The FSTP instruction always pops the top of stack after storing the top of stack (tos). The remaining instructions do not affect the number of items on the stack.

11.2.4.1 The FLD Instruction

The FLD instruction loads a 32 bit, 64 bit, or 80 bit floating point value onto the stack. This instruction converts 32 and 64 bit operands to an 80 bit extended precision value before pushing the value onto the floating point stack.

The FLD instruction first decrements the top of stack (TOS) pointer (bits 11-13 of the status register) and then stores the 80 bit value in the physical register specified by the new TOS pointer. If the source operand of the FLD instruction is a floating point data register, ST_i , then the actual register the FPU uses for the load operation is the register number *before* decrementing the tos pointer. Therefore, “fld(st0);” duplicates the value on the top of the stack.

The FLD instruction sets the stack fault bit if stack overflow occurs. It sets the denormalized exception bit if you load an 80-bit denormalized value. It sets the invalid operation bit if you attempt to load an empty floating point register onto the stop of stack (or perform some other invalid operation).

Examples:

```
fld( st1 );
fld( real32_variable );
fld( real64_variable );
fld( real80_variable );
fld( real_constant );
```

Note that there is no way to directly load a 32-bit integer register onto the floating point stack, even if that register contains a REAL32 value. To accomplish this, you must first store the integer register into a memory location then you can push that memory location onto the FPU stack using the FLD instruction. E.g.,

```

mov( eax, tempReal32 );    // Save REAL32 value in EAX to memory.
fld( tempReal32 );        // Push that real value onto the FPU stack.

```

Note: loading a constant via FLD is actually an HLA extension. The FPU doesn't support this instruction type. HLA creates a REAL80 object in the "constants" segment and uses the address of this memory object as the true operand for FLD.

11.2.4.2 The FST and FSTP Instructions

The FST and FSTP instructions copy the value on the top of the floating point register stack to another floating point register or to a 32, 64, or 80 bit memory variable. When copying data to a 32 or 64 bit memory variable, the 80 bit extended precision value on the top of stack is rounded to the smaller format as specified by the rounding control bits in the FPU control register.

The FSTP instruction pops the value off the top of stack when moving it to the destination location. It does this by incrementing the top of stack pointer in the status register after accessing the data in ST0. If the destination operand is a floating point register, the FPU stores the value at the specified register number *before* popping the data off the top of the stack.

Executing an "fstp(st0);" instruction effectively pops the data off the top of stack with no data transfer. Examples:

```

fst( real32_variable );
fst( real64_variable );
fst( realArray[ ebx*8 ] );
fst( real80_variable );
fst( st2 );
fstp( st1 );

```

The last example above effectively pops ST1 while leaving ST0 on the top of the stack.

The FST and FSTP instructions will set the stack exception bit if a stack underflow occurs (attempting to store a value from an empty register stack). They will set the precision bit if there is a loss of precision during the store operation (this will occur, for example, when storing an 80 bit extended precision value into a 32 or 64 bit memory variable and there are some bits lost during conversion). They will set the underflow exception bit when storing an 80 bit value into a 32 or 64 bit memory variable, but the value is too small to fit into the destination operand. Likewise, these instructions will set the overflow exception bit if the value on the top of stack is too big to fit into a 32 or 64 bit memory variable. The FST and FSTP instructions set the denormalized flag when you try to store a denormalized value into an 80 bit register or variable². They set the invalid operation flag if an invalid operation (such as storing into an empty register) occurs. Finally, these instructions set the C₁ condition bit if rounding occurs during the store operation (this only occurs when storing into a 32 or 64 bit memory variable and you have to round the mantissa to fit into the destination).

Note: Because of an idiosyncrasy in the FPU instruction set related to the encoding of the instructions, you cannot use the FST instruction to store data into a real80 memory variable. You may, however, store 80-bit data using the FSTP instruction.

11.2.4.3 The FXCH Instruction

The FXCH instruction exchanges the value on the top of stack with one of the other FPU registers. This instruction takes two forms: one with a single FPU register as an operand, the second without any operands. The first form exchanges the top of stack (tos) with the specified register. The second form of FXCH swaps the top of stack with ST1.

Many FPU instructions, e.g., FSQRT, operate only on the top of the register stack. If you want to perform such an operation on a value that is not on the top of stack, you can use the FXCH instruction to swap

2. Storing a denormalized value into a 32 or 64 bit memory variable will always set the underflow exception bit.

that register with `tos`, perform the desired operation, and then use the `FXCH` to swap the `tos` with the original register. The following example takes the square root of `ST2`:

```
fxch( st2 );
fsqrt();
fxch( st2 );
```

The `FXCH` instruction sets the stack exception bit if the stack is empty. It sets the invalid operation bit if you specify an empty register as the operand. This instruction always clears the C_1 condition code bit.

11.2.5 Conversions

The FPU performs all arithmetic operations on 80 bit real quantities. In a sense, the `FLD` and `FST/FSTP` instructions are conversion instructions as well as data movement instructions because they automatically convert between the internal 80 bit real format and the 32 and 64 bit memory formats. Nonetheless, we'll simply classify them as data movement operations, rather than conversions, because they are moving real values to and from memory. The FPU provides five other instructions that convert to or from integer or binary coded decimal (BCD) format when moving data. These instructions are `FILD`, `FIST`, `FISTP`, `FBLD`, and `FBSTP`.

11.2.5.1 The FILD Instruction

The `FILD` (integer load) instruction converts a 16, 32, or 64 bit two's complement integer to the 80 bit extended precision format and pushes the result onto the stack. This instruction always expects a single operand. This operand must be the address of a word, double word, or quad word integer variable. You cannot specify one of the 80x86's 16 or 32 bit general purpose registers. If you want to push an 80x86 general purpose register onto the FPU stack, you must first store it into a memory variable and then use `FILD` to push that value of that memory variable.

The `FILD` instruction sets the stack exception bit and C_1 (accordingly) if stack overflow occurs while pushing the converted value. Examples:

```
filed( word_variable );
filed( dword_val[ ecx*4 ] );
filed( qword_variable );
```

11.2.5.2 The FIST and FISTP Instructions

The `FIST` and `FISTP` instructions convert the 80 bit extended precision variable on the top of stack to a 16, 32, or 64 bit integer and store the result away into the memory variable specified by the single operand. These instructions convert the value on `tos` to an integer according to the rounding setting in the FPU control register (bits 10 and 11). As for the `FILD` instruction, the `FIST` and `FISTP` instructions will not let you specify one of the 80x86's general purpose 16 or 32 bit registers as the destination operand.

The `FIST` instruction converts the value on the top of stack to an integer and then stores the result; it does not otherwise affect the floating point register stack. The `FISTP` instruction pops the value off the floating point register stack after storing the converted value.

These instructions set the stack exception bit if the floating point register stack is empty (this will also clear C_1). They set the precision (imprecise operation) and C_1 bits if rounding occurs (that is, if there is any fractional component to the value in `ST0`). These instructions set the underflow exception bit if the result is too small (i.e., less than one but greater than zero or less than zero but greater than -1). Examples:

```
fist( word_var[ ebx*2 ] );
fist( qword_var );
fistp( dword_var );
```

Don't forget that these instructions use the rounding control settings to determine how they will convert the floating point data to an integer during the store operation. By default, the rounding control is usually set to "round" mode; yet most programmers expect FIST/FISTP to truncate the decimal portion during conversion. If you want FIST/FISTP to truncate floating point values when converting them to an integer, you will need to set the rounding control bits appropriately in the floating point control register, e.g.,

```
static
fcw16:      word;
fcw16_2:    word;
IntResult:  int32;
.
.
.
fstcw( fcw16 );
mov( fcw16, ax );
or( $0c00, ax );      // Rounding control=%11 (truncate).
mov( ax, fcw16_2 );   // Store into memory and reload the ctrl word.
fldcw( fcw16_2 );

fistp( IntResult );   // Truncate ST0 and store as int32 object.

fldcw( fcw16 );      // Restore original rounding control
```

11.2.5.3 The FBLD and FBSTP Instructions

The FBLD and FBSTP instructions load and store 80 bit BCD values. The FBLD instruction converts a BCD value to its 80 bit extended precision equivalent and pushes the result onto the stack. The FBSTP instruction pops the extended precision real value on TOS, converts it to an 80 bit BCD value (rounding according to the bits in the floating point control register), and stores the converted result at the address specified by the destination memory operand. Note that there is no FBST instruction which stores the value on tos without popping it.

The FBLD instruction sets the stack exception bit and C₁ if stack overflow occurs. It sets the invalid operation bit if you attempt to load an invalid BCD value. The FBSTP instruction sets the stack exception bit and clears C₁ if stack underflow occurs (the stack is empty). It sets the underflow flag under the same conditions as FIST and FISTP. Examples:

```
// Assuming fewer than eight items on the stack, the following
// code sequence is equivalent to an fbst instruction:

    fld( st0 );
    fbstp( tbyte_var );

// The following example easily converts an 80 bit BCD value to
// a 64 bit integer:

    fbld( tbyte_var );
    fist( qword_var );
```

11.2.6 Arithmetic Instructions

The arithmetic instructions make up a small, but important, subset of the FPU's instruction set. These instructions fall into two general categories – those which operate on real values and those which operate on a real and an integer value.

11.2.6.1 The FADD and FADDP Instructions

These two instructions take the following forms:

```
fadd()
faddp()
fadd( st0, sti );
fadd( sti, st0 );
faddp( st0, sti );
fadd( mem_32_64 );
fadd( real_constant );
```

The first two forms are equivalent. They pop the two values on the top of stack, add them, and push their sum back onto the stack.

The next two forms of the FADD instruction, those with two FPU register operands, behave like the 80x86's ADD instruction. They add the value in the source register operand to the value in the destination register operand. Note that one of the register operands must be ST0.

The FADDP instruction with two operands adds ST0 (which must always be the source operand) to the destination operand and then pops ST0. The destination operand must be one of the other FPU registers.

The last form above, FADD with a memory operand, adds a 32 or 64 bit floating point variable to the value in ST0. This instruction will convert the 32 or 64 bit operands to an 80 bit extended precision value before performing the addition. Note that this instruction does *not* allow an 80 bit memory operand.

These instructions can raise the stack, precision, underflow, overflow, denormalized, and illegal operation exceptions, as appropriate. If a stack fault exception occurs, C₁ denotes stack overflow or underflow.

Like FLD(real_constant), the FADD(real_constant) instruction is an HLA extension. Note that it creates a 64-bit variable holding the constant value and emits the FADD(mem64) instruction, specifying the read-only object it creates in the constants segment.

11.2.6.2 The FSUB, FSUBP, FSUBR, and FSUBRP Instructions

These four instructions take the following forms:

```
fsub()
fsubp()
fsubr()
fsubrp()

fsub( st0, sti )
fsub( sti, st0 );
fsubp( st0, sti );
fsub( mem_32_64 );
fsub( real_constant );

fsubr( st0, sti )
fsubr( sti, st0 );
fsubrp( st0, sti );
fsubr( mem_32_64 );
fsubr( real_constant );
```

With no operands, the FSUB and FSUBP instructions operate identically. They pop ST0 and ST1 from the register stack, compute ST1-ST0, and then push the difference back onto the stack. The FSUBR and FSUBRP instructions (reverse subtraction) operate in an almost identical fashion except they compute ST0-ST1 and push that difference.

With two register operands (*source*, *destination*) the FSUB instruction computes *destination := destination - source*. One of the two registers must be ST0. With two registers as operands, the FSUBP also com-

puts $destination := destination - source$ and then it pops ST0 off the stack after computing the difference. For the FSUBP instruction, the source operand must be ST0.

With two register operands, the FSUBR and FSUBRP instruction work in a similar fashion to FSUB and FSUBP, except they compute $destination := source - destination$.

The FSUB(mem) and FSUBR(mem) instructions accept a 32 or 64 bit memory operand. They convert the memory operand to an 80 bit extended precision value and subtract this from ST0 (FSUB) or subtract ST0 from this value (FSUBR) and store the result back into ST0.

These instructions can raise the stack, precision, underflow, overflow, denormalized, and illegal operation exceptions, as appropriate. If a stack fault exception occurs, C₁ denotes stack overflow or underflow.

Note: the instructions that have real constants as operands aren't true FPU instructions. These are extensions provided by HLA. HLA generates a constant segment memory object initialized with the constant's value.

11.2.6.3 The FMUL and FMULP Instructions

The FMUL and FMULP instructions multiply two floating point values. These instructions allow the following forms:

```
fmul()
fmulp()

fmul( sti, st0 );
fmul( st0, sti );
fmul( mem_32_64 );
fmul( real_constant );

fmulp( st0, sti );
```

With no operands, FMUL and FMULP both do the same thing – they pop ST0 and ST1, multiply these values, and push their product back onto the stack. The FMUL instructions with two register operands compute $destination := destination * source$. One of the registers (source or destination) must be ST0.

The FMULP(ST0, ST*i*) instruction computes $ST*i* := ST*i* * ST0$ and then pops ST0. This instruction uses the value for *i* before popping ST0. The FMUL(mem) instruction requires a 32 or 64 bit memory operand. It converts the specified memory variable to an 80 bit extended precision value and the multiplies ST0 by this value.

These instructions can raise the stack, precision, underflow, overflow, denormalized, and illegal operation exceptions, as appropriate. If rounding occurs during the computation, these instructions set the C₁ condition code bit. If a stack fault exception occurs, C₁ denotes stack overflow or underflow.

Note: the instruction that has a real constant as its operand isn't a true FPU instruction. It is an extension provided by HLA (see the note at the end of the previous section for details).

11.2.6.4 The FDIV, FDIVP, FDIVR, and FDIVRP Instructions

These four instructions allow the following forms:

```
fdiv()
fdivp()
fdivr()
fdivrp()

fdiv( sti, st0 );
fdiv( st0, sti );
fdivp( st0, sti );
```



```

fdivr( sti, st0 );
fdivr( st0, sti );
fdivrp( st0, sti );

fdiv( mem_32_64 );
fdivr( mem_32_64 );
fdiv( real_constant );
fdivr( real_constant );

```

With no operands, the FDIV and FDIVP instructions pop ST0 and ST1, compute ST1/ST0, and push the result back onto the stack. The FDIVR and FDIVRP instructions also pop ST0 and ST1 but compute ST0/ST1 before pushing the quotient onto the stack.

With two register operands, these instructions compute the following quotients:

```

fdiv( sti, st0 );      // ST0 := ST0/STi
fdiv( st0, sti );     // STi := STi/ST0
fdivp( st0, sti );    // STi := STi/ST0 then pop ST0
fdivr( st0, sti );   // ST0 := ST0/STi
fdivrp( st0, sti );  // STi := ST0/STi then pop ST0

```

The FDIVP and FDIVRP instructions also pop ST0 after performing the division operation. The value for *i* in these two instructions is computed before popping ST0.

These instructions can raise the stack, precision, underflow, overflow, denormalized, zero divide, and illegal operation exceptions, as appropriate. If rounding occurs during the computation, these instructions set the C₁ condition code bit. If a stack fault exception occurs, C₁ denotes stack overflow or underflow.

Note: the instructions that have real constants as operands aren't true FPU instructions. These are extensions provided by HLA.

11.2.6.5 The FSQRT Instruction

The FSQRT routine does not allow any operands. It computes the square root of the value on top of stack (TOS) and replaces ST0 with this result. The value on TOS must be zero or positive, otherwise FSQRT will generate an invalid operation exception.

This instruction can raise the stack, precision, denormalized, and invalid operation exceptions, as appropriate. If rounding occurs during the computation, FSQRT sets the C₁ condition code bit. If a stack fault exception occurs, C₁ denotes stack overflow or underflow.

Example:

```

// Compute Z := sqrt(x**2 + y**2);

fld( x );      // Load X.
fld( st0 );    // Duplicate X on TOS.
fmul();        // Compute X**2.

fld( y );      // Load Y
fld( st0 );    // Duplicate Y.
fmul();        // Compute Y**2.

fadd();        // Compute X**2 + Y**2.
fsqrt();       // Compute sqrt( X**2 + Y**2 ).
fstp( z );     // Store result away into Z.

```

11.2.6.6 The FPREM and FPREM1 Instructions

The FPREM and FPREM1 instructions compute a *partial remainder*. Intel designed the FPREM instruction before the IEEE finalized their floating point standard. In the final draft of the IEEE floating point standard, the definition of FPREM was a little different than Intel's original design. Unfortunately, Intel needed to maintain compatibility with the existing software that used the FPREM instruction, so they designed a new version to handle the IEEE partial remainder operation, FPREM1. You should always use FPREM1 in new software you write, therefore we will only discuss FPREM1 here, although you use FPREM in an identical fashion.

FPREM1 computes the *partial* remainder of ST0/ST1. If the difference between the exponents of ST0 and ST1 is less than 64, FPREM1 can compute the exact remainder in one operation. Otherwise you will have to execute the FPREM1 two or more times to get the correct remainder value. The C₂ condition code bit determines when the computation is complete. Note that FPREM1 does *not* pop the two operands off the stack; it leaves the partial remainder in ST0 and the original divisor in ST1 in case you need to compute another partial product to complete the result.

The FPREM1 instruction sets the stack exception flag if there aren't two values on the top of stack. It sets the underflow and denormal exception bits if the result is too small. It sets the invalid operation bit if the values on tos are inappropriate for this operation. It sets the C₂ condition code bit if the partial remainder operation is not complete. Finally, it loads C₃, C₁, and C₀ with bits zero, one, and two of the quotient, respectively.

Example:

```
// Compute Z := X mod Y

    fld( y );
    fld( x );
    repeat

        fprem1();
        fstsw( ax );    // Get condition code bits into AX.
        and( 1, ah );  // See if C2 is set.

    until( @z );      // Repeat until C2 is clear.
    fstp( z );        // Store away the remainder.
    fstp( st0 );      // Pop old Y value.
```

11.2.6.7 The FRNDINT Instruction

The FRNDINT instruction rounds the value on the top of stack (TOS) to the nearest integer using the rounding algorithm specified in the control register.

This instruction sets the stack exception flag if there is no value on the TOS (it will also clear C₁ in this case). It sets the precision and denormal exception bits if there was a loss of precision. It sets the invalid operation flag if the value on the tos is not a valid number. Note that the result on tos is still a floating point value, it simply does not have a fractional component.

11.2.6.8 The FABS Instruction

FABS computes the absolute value of ST0 by clearing the mantissa sign bit of ST0. It sets the stack exception bit and invalid operation bits if the stack is empty.

Example:

```
// Compute X := sqrt(abs(x));
```

```
fld( x );
fabs();
fsqrt();
fstp( x );
```

11.2.6.9 The FCHS Instruction

FCHS changes the sign of ST0's value by inverting the mantissa sign bit (that is, this is the floating point negation instruction). It sets the stack exception bit and invalid operation bits if the stack is empty. Example:

```
// Compute X := -X if X is positive, X := X if X is negative.
```

```
fld( x );
fabs();
fchs();
fstp( x );
```

11.2.7 Comparison Instructions

The FPU provides several instructions for comparing real values. The FCOM, FCOMP, and FCOMPP instructions compare the two values on the top of stack and set the condition codes appropriately. The FTST instruction compares the value on the top of stack with zero.

Generally, most programs test the condition code bits immediately after a comparison. Unfortunately, there are no conditional jump instructions that branch based on the FPU condition codes. Instead, you can use the FSTSW instruction to copy the floating point status register (see “The FPU Status Register” on page 615) into the AX register; then you can use the SAHF instruction to copy the AH register into the 80x86's condition code bits. After doing this, you can use the conditional jump instructions to test some condition. This technique copies C_0 into the carry flag, C_2 into the parity flag, and C_3 into the zero flag. The SAHF instruction does not copy C_1 into any of the 80x86's flag bits.

Since the SAHF instruction does not copy any FPU status bits into the sign or overflow flags, you cannot use signed comparison instructions. Instead, use unsigned operations (e.g., SETA, SETB) when testing the results of a floating point comparison. *Yes, these instructions normally test unsigned values and floating point numbers are signed values.* However, use the unsigned operations anyway; the FSTSW and SAHF instructions set the 80x86 flags register as though you had compared unsigned values with the CMP instruction.

The Pentium II and (upwards) compatible processors provide an extra set of floating point comparison instructions that directly affect the 80x86 condition code flags. These instructions circumvent having to use FSTSW and SAHF to copy the FPU status into the 80x86 condition codes. These instructions include FCOMI and FCOMIP. You use them just like the FCOM and FCOMP instructions except, of course, you do not have to manually copy the status bits to the FLAGS register. Do be aware that these instructions are not available on many processors in common use today (as of 1/1/2000). However, as time passes it may be safe to begin assuming that everyone's CPU supports these instructions. Since this text assumes a minimum Pentium CPU, it will not discuss these two instructions any further.

11.2.7.1 The FCOM, FCOMP, and FCOMPP Instructions

The FCOM, FCOMP, and FCOMPP instructions compare ST0 to the specified operand and set the corresponding FPU condition code bits based on the result of the comparison. The legal forms for these instructions are

```

fcom( )
fcomp( )
fcompp( )

fcom( sti )
fcomp( sti )

fcom( mem_32_64 )
fcomp( mem_32_64 )
fcom( real_constant )
fcomp( real_constant )

```

With no operands, FCOM, FCOMP, and FCOMPP compare ST0 against ST1 and set the processor flags accordingly. In addition, FCOMP pops ST0 off the stack and FCOMPP pops both ST0 and ST1 off the stack.

With a single register operand, FCOM and FCOMP compare ST0 against the specified register. FCOMP also pops ST0 after the comparison.

With a 32 or 64 bit memory operand, the FCOM and FCOMP instructions convert the memory variable to an 80 bit extended precision value and then compare ST0 against this value, setting the condition code bits accordingly. FCOMP also pops ST0 after the comparison.

These instructions set C₂ (which winds up in the parity flag) if the two operands are not comparable (e.g., NaN). If it is possible for an illegal floating point value to wind up in a comparison, you should check the parity flag for an error before checking the desired condition.

These instructions set the stack fault bit if there aren't two items on the top of the register stack. They set the denormalized exception bit if either or both operands are denormalized. They set the invalid operation flag if either or both operands are quite NaNs. These instructions always clear the C₁ condition code.

Note: the instructions that have real constants as operands aren't true FPU instructions. These are extensions provided by HLA. When HLA encounters such an instruction, it creates a real64 read-only variable in the constants segment and initializes this variable with the specified constant. Then HLA translates the instruction to one that specifies a real64 memory operand. *Note that because of the precision differences (64 bits vs. 80 bits), if you use a constant operand in a floating point instruction you may not get results that are as precise as you would expect.*

Example of a floating point comparison:

```

fcompp( );
fstsw( ax );
sahf( );
setb( al ); // AL = true if ST1 < ST0.
.
.
.

```

Note that you cannot compare floating point values in an HLA run-time boolean expression (e.g., within an IF statement).

11.2.7.2 The FTST Instruction

The FTST instruction compares the value in ST0 against 0.0. It behaves just like the FCOM instruction would if ST1 contained 0.0. Note that this instruction does not differentiate -0.0 from +0.0. If the value in ST0 is either of these values, ftst will set C₃ to denote equality. Note that this instruction does *not* pop st(0) off the stack. Example:

```

ftst( );
fstsw( ax );
sahf( );
sete( al ); // Set AL to 1 if TOS = 0.0

```

11.2.8 Constant Instructions

The FPU provides several instructions that let you load commonly used constants onto the FPU's register stack. These instructions set the stack fault, invalid operation, and C_1 flags if a stack overflow occurs; they do not otherwise affect the FPU flags. The specific instructions in this category include:

```

fldz()      ;Pushes +0.0.
fldl()      ;Pushes +1.0.
fldpi()     ;Pushes  $\pi$ .
fldl2t()    ;Pushes  $\log_2(10)$ .
fldl2e()    ;Pushes  $\log_2(e)$ .
fldlg2()    ;Pushes  $\log_{10}(2)$ .
fldln2()    ;Pushes  $\ln(2)$ .

```

11.2.9 Transcendental Instructions

The FPU provides eight transcendental (log and trigonometric) instructions to compute sin, cos, partial tangent, partial arctangent, 2^x-1 , $y * \log_2(x)$, and $y * \log_2(x+1)$. Using various algebraic identities, it is easy to compute most of the other common transcendental functions using these instructions.

11.2.9.1 The F2XM1 Instruction

F2XM1 computes $2^{\text{st}0}-1$. The value in ST0 must be in the range $-1.0 \leq \text{ST}0 \leq +1.0$. If ST0 is out of range F2XM1 generates an undefined result but raises no exceptions. The computed value replaces the value in ST0. Example:

```

; Compute  $10^x$  using the identity:  $10^x = 2^{x \cdot \lg(10)}$  ( $\lg = \log_2$ ).

fld( x );
fldl2t();
fmul();
f2xm1();
fldl();
fadd();

```

Note that F2XM1 computes 2^x-1 , which is why the code above adds 1.0 to the result at the end of the computation.

11.2.9.2 The FSIN, FCOS, and FSINCOS Instructions

These instructions pop the value off the top of the register stack and compute the sine, cosine, or both, and push the result(s) back onto the stack. The FSINCOS pushes the sine followed by the cosine of the original operand, hence it leaves $\cos(\text{ST}0)$ in ST0 and $\sin(\text{ST}0)$ in ST1.

These instructions assume ST0 specifies an angle in radians and this angle must be in the range $-2^{63} < \text{ST}0 < +2^{63}$. If the original operand is out of range, these instructions set the C_2 flag and leave ST0 unchanged. You can use the FPREM1 instruction, with a divisor of 2π , to reduce the operand to a reasonable range.

These instructions set the stack fault/ C_1 , precision, underflow, denormalized, and invalid operation flags according to the result of the computation.

11.2.9.3 The FPTAN Instruction

FPTAN computes the tangent of ST0 and pushes this value and then it pushes 1.0 onto the stack. Like the FSIN and FCOS instructions, the value of ST0 is assumed to be in radians and must be in the range $-2^{63} < ST0 < +2^{63}$. If the value is outside this range, FPTAN sets C₂ to indicate that the conversion did not take place. As with the FSIN, FCOS, and FSINCOS instructions, you can use the FPREM1 instruction to reduce this operand to a reasonable range using a divisor of 2π .

If the argument is invalid (i.e., zero or π radians, which causes a division by zero) the result is undefined and this instruction raises no exceptions. FPTAN will set the stack fault, precision, underflow, denormal, invalid operation, C₂, and C₁ bits as required by the operation.

11.2.9.4 The FPATAN Instruction

This instruction expects two values on the top of stack. It pops them and computes the following:

$$ST0 = \tan^{-1}(ST1 / ST0)$$

The resulting value is the arctangent of the ratio on the stack expressed in radians. If you have a value you wish to compute the tangent of, use FLD1 to create the appropriate ratio and then execute the FPATAN instruction.

This instruction affects the stack fault/C₁, precision, underflow, denormal, and invalid operation bits if a problem occurs during the computation. It sets the C₁ condition code bit if it has to round the result.

11.2.9.5 The FYL2X Instruction

This instruction expects two operands on the FPU stack: y is found in ST1 and x is found in ST0. This function computes:

$$ST0 = ST1 * \log_2(ST0)$$

This instruction has no operands (to the instruction itself). The instruction uses the following syntax:

```
fy12x();
```

Note that this instruction computes the base two logarithm. Of course, it is a trivial matter to compute the log of any other base by multiplying by the appropriate constant.

11.2.9.6 The FYL2XP1 Instruction

This instruction expects two operands on the FPU stack: y is found in ST1 and x is found in ST0. This function computes:

$$ST0 = ST1 * \log_2(ST0 + 1.0)$$

The syntax for this instruction is

```
fy12xp1();
```

Otherwise, the instruction is identical to FYL2X.

11.2.10 Miscellaneous instructions

The FPU includes several additional instructions which control the FPU, synchronize operations, and let you test or set various status bits. These instructions include FINIT/FNINIT, FLDCW, FSTCW, FCLEX/FNCLEX, and FSTSW.

11.2.10.1 The FINIT and FNINIT Instructions

The FINIT instruction initializes the FPU for proper operation. Your applications should execute this instruction before executing any other FPU instructions. This instruction initializes the control register to 37Fh (see “The FPU Control Register” on page 612), the status register to zero (see “The FPU Status Register” on page 615) and the tag word to 0FFFFh. The other registers are unaffected. Examples:

```
FINIT();
FNINIT();
```

The difference between FINIT and FNINIT is that FINIT first checks for any pending floating point exceptions before initializing the FPU; FNINIT does not.

11.2.10.2 The FLDCW and FSTCW Instructions

The FLDCW and FSTCW instructions require a single 16 bit memory operand:

```
fldcw( mem_16 );
fstcw( mem_16 );
```

These two instructions load the control register (see “The FPU Control Register” on page 612) from a memory location (FLDCW) or store the control word to a 16 bit memory location (FSTCW).

When using the FLDCW instruction to turn on one of the exceptions, if the corresponding exception flag is set when you enable that exception, the FPU will generate an immediate interrupt before the CPU executes the next instruction. Therefore, you should use the FCLEX instruction to clear any pending interrupts before changing the FPU exception enable bits.

11.2.10.3 The FCLEX and FNCLEX Instructions

The FCLEX and FNCLEX instructions clear all exception bits the stack fault bit, and the busy flag in the FPU status register (see “The FPU Status Register” on page 615). Examples:

```
fclex();
fnclex();
```

The difference between these instructions is the same as FINIT and FNINIT.

11.2.10.4 The FSTSW and FNSTSW Instructions

```
fstsw( ax )
fnstsw( ax )
fstsw( mem_16 )
fnstsw( mem_16 )
```

These instructions store the FPU status register (see “The FPU Status Register” on page 615) into a 16 bit memory location or the AX register. These instructions are unusual in the sense that they can copy an FPU value into one of the 80x86 general purpose registers (specifically, AX). Of course, the whole purpose

behind allowing the transfer of the status register into AX is to allow the CPU to easily test the condition code register with the SAHF instruction. The difference between FSTSW and FNSTSW is the same as for FCLEX and FNCLEX.

11.2.11 Integer Operations

The 80x87 FPUs provide special instructions that combine integer to extended precision conversion along with various arithmetic and comparison operations. These instructions are the following:

```
fiadd( int_16_32 )
fisub( int_16_32 )
fisubr( int_16_32 )
fimul( int_16_32 )
fidiv( int_16_32 )
fidivr( int_16_32 )

ficom( int_16_32 )
ficomp( int_16_32 )
```

These instructions convert their 16 or 32 bit integer operands to an 80 bit extended precision floating point value and then use this value as the source operand for the specified operation. These instructions use ST0 as the destination operand.

11.3 Converting Floating Point Expressions to Assembly Language

Because the FPU register organization is different than the 80x86 integer register set, translating arithmetic expressions involving floating point operands is a little different than the techniques for translating integer expressions. Therefore, it makes sense to spend some time discussing how to manually translate floating point expressions into assembly language.

In one respect, it's actually easier to translate floating point expressions into assembly language. The stack architecture of the Intel FPU eases the translation of arithmetic expressions into assembly language. If you've ever used a Hewlett-Packard calculator, you'll be right at home on the FPU because, like the HP calculator, the FPU uses *reverse polish notation*, or *RPN*, for arithmetic calculations. Once you get used to using RPN, it's actually a bit more convenient for translating expressions because you don't have to worry about allocating temporary variables - they always wind up on the FPU stack.

RPN, as opposed to standard *infix notation*, places the operands before the operator. The following examples give some simple examples of infix notation and the corresponding RPN notation:

infix notation	RPN notation
5 + 6	5 6 +
7 - 2	7 2 -
x * y	x y *
a / b	a b /

An RPN expression like "5 6 +" says "push five onto the stack, push six onto the stack, then pop the value off the top of stack (six) and add it to the new top of stack." Sound familiar? This is exactly what the FLD and FADD instructions do. In fact, you can calculate this using the following code:

```
fld( 5.0 );
fld( 6.0 );
fadd(); // 11.0 is now on the top of the FPU stack.
```

As you can see, RPN is a convenient notation because it's very easy to translate this code into FPU instructions.

One advantage to RPN (or *postfix notation*) is that it doesn't require any parentheses. The following examples demonstrate some slightly more complex infix to postfix conversions:

infix notation	postfix notation
$(x + y) * 2$	$x y + 2 *$
$x * 2 - (a + b)$	$x 2 * a b + -$
$(a + b) * (c + d)$	$a b + c d + *$

The postfix expression “ $x y + 2 *$ ” says “push x , then push y ; next, add those values on the stack (producing $X+Y$ on the stack). Next, push 2 and then multiply the two values (two and $X+Y$) on the stack to produce two times the quantity $X+Y$.” Once again, we can translate these postfix expressions directly into assembly language. The following code demonstrates the conversion for each of the above expressions:

```
//      x y + 2 *

      fld( x );
      fld( y );
      fadd();
      fld( 2.0 );
      fmul();

//      x 2 * a b + -

      fld( x );
      fld( 2.0 );
      fmul();
      fld( a );
      fld( b );
      fadd();
      fsub();

//      a b + c d + *

      fld( a );
      fld( b );
      fadd();
      fld( c );
      fld( d );
      fadd();
      fmul();
```

11.3.1 Converting Arithmetic Expressions to Postfix Notation

Since the process of translating arithmetic expressions into assembly language involves postfix (RPN) notation, converting arithmetic expressions into postfix notation seems like the right place to start. This section will concentrate on that conversion.

For simple expressions, those involving two operands and a single expression, the translation is trivial. Simply move the operator from the infix position to the postfix position (that is, move the operator from inbetween the operands to after the second operand). For example, “ $5 + 6$ ” becomes “ $5 6 +$ ”. Other than separating your operands so you don't confuse them (i.e., is it “5” and “6” or “56”?) there isn't much to converting simple infix expressions into postfix notation.

For complex expressions, the idea is to convert the simple sub-expressions into postfix notation and then treat each converted subexpression as a single operand in the remaining expression. The following discussion will surround completed conversions in square brackets so it is easy to see which text needs to be treated as a single operand in the conversion.

As for integer expression conversion, the best place to start is in the inner-most parenthetical sub-expression and then work your way outward considering precedence, associativity, and other parenthetical sub-expressions. As a concrete working example, consider the following expression:

$$x = ((y-z)*a) - (a + b * c)/3.14159$$

A possible first translation is to convert the subexpression “(y-z)” into postfix notation. This is accomplished as follows:

$$x = ([y z -] * a) - (a + b * c)/3.14159$$

Square brackets surround the converted postfix code just to separate it from the infix code. These exist only to make the partial translations more readable. Remember, for the purposes of conversion we will treat the text inside the square brackets as a single operand. Therefore, you would treat “[y z -]” as though it were a single variable name or constant.

The next step is to translate the subexpression “([y z -] * a)” into postfix form. This yields the following:

$$x = [y z - a *] - (a + b * c)/3.14159$$

Next, we work on the parenthetical expression “(a + b * c).” Since multiplication has higher precedence than addition, we convert “b*c” first:

$$x = [y z - a *] - (a + [b c *])/3.14159$$

After converting “b*c” we finish the parenthetical expression:

$$x = [y z - a *] - [a b c * +]/3.14159$$

This leaves only two infix operators: subtraction and division. Since division has the higher precedence, we’ll convert that first:

$$x = [y z - a *] - [a b c * + 3.14159 /]$$

Finally, we convert the entire expression into postfix notation by dealing with the last infix operation, subtraction:

$$x = [y z - a *] [a b c * + 3.14159 /] -$$

Removing the square brackets to give us true postfix notation yields the following RPN expression:

$$x = y z - a * a b c * + 3.14159 / -$$

Here is another example of an infix to postfix conversion:

$$a = (x * y - z + t)/2.0$$

Step 1: Work inside the parentheses. Since multiplication has the highest precedence, convert that first:

$$a = ([x y *] - z + t)/2.0$$

Step 2: Still working inside the parentheses, we note that addition and subtraction have the same precedence, so we rely upon associativity to determine what to do next. These operators are left associative, so we must translate the expressions in a left to right order. This means translate the subtraction operator first:

$$a = ([x y * z -] + t)/2.0$$

Step 3: Now translate the addition operator inside the parentheses. Since this finishes the parenthetical operators, we can drop the parentheses:

$$a = [x y * z - t +] / 2.0$$

Step 4: Translate the final infix operator (division). This yields the following:

$$a = [x y * z - t + 2.0 /]$$

Step 5: Drop the square brackets and we're done:

$$a = x y * z - t + 2.0 /$$

11.3.2 Converting Postfix Notation to Assembly Language

Once you've translated an arithmetic expression into postfix notation, finishing the conversion to assembly language is especially easy. All you have to do is issue an FLD instruction whenever you encounter an operand and issue an appropriate arithmetic instruction when you encounter an operator. This section will use the completed examples from the previous section to demonstrate how little there is to this process.

$$x = y z - a * a b c * + 3.14159 / -$$

- Step 1: Convert y to FLD(y);
- Step 2: Convert z to FLD(z);
- Step 3: Convert “-” to FSUB();
- Step 4: Convert a to FLD(a);
- Step 5: Convert “*” to FMUL();
- Steps 6-n: Continuing in a left-to-right fashion, generate the following code for the expression:

```
fld( y );
fld( z );
fsub();
fld( a );
fmul();
fld( a );
fld( b );
fld( c );
fmul();
fadd();
fldpi();      // Loads pi (3.14159)
fdiv();
fsub();

fstp( x );    // Store result away into x.
```

Here's the translation for the second example in the previous section:

$$a = x y * z - t + 2.0 /$$

```
fld( x );
fld( y );
fmul();
fld( z );
fsub();
fld( t );
fadd();
fld( 2.0 );
fdiv();

fstp( a );    // Store result away into a.
```

As you can see, the translation is fairly trivial once you've converted the infix notation to postfix notation. Also note that, unlike integer expression conversion, you don't need any explicit temporaries. It turns out that the FPU stack provides the temporaries for you³. For these reasons, conversion of floating point expressions into assembly language is actually easier than converting integer expressions.

11.3.3 Mixed Integer and Floating Point Arithmetic

Throughout the previous sections on floating point arithmetic an unstated assumption was made: all operands in the expressions were floating point variables or constants. In the real world, you'll often need to mix integer and floating point operands in the same expression. Thanks to the `FILD` instruction, this is a trivial exercise.

Of course, the FPU cannot operate on integer operands. That is, you cannot push an integer operand (in integer format) onto the FPU stack and add this integer to a floating point value that is also on the stack. Instead, you use the `FILD` instruction to load and translate the integer value; this winds up pushing the floating point equivalent of the integer onto the FPU stack. Once the value is converted to a floating point number, you continue the calculation using the standard real arithmetic operations.

Embedding a floating point value in an integer expression is a little more problematic. In this case you must convert the floating point value to an integer value for use in the integer expression. To do this, you must use the `FIST` instruction. `FIST` converts the floating point value on the top of stack into an integer value according to the setting of the rounding bits in the floating point control register (See "The FPU Control Register" on page 612). By default, `FIST` will round the floating point value to the nearest integer before storing the value into memory; if you want to use the more common fraction truncation mode, you will need to change the value in the FPU control register. You compute the integer expression using the techniques from the previous chapter (see "Complex Expressions" on page 600). The FPU participates only to the point of converting the floating point value to an integer.

```
static
    intVal1 : uns32 := 1;
    intVal2 : uns32 := 2;
    realVal : real64;
    .
    .
    .
    fild( intVal1 );
    fild( intVal2 );
    fadd();
    fstp( realVal );
    stdout.put( "realVal = ", realVal, nl );
```

11.4 HLA Standard Library Support for Floating Point Arithmetic

The HLA Standard Library provides several routines that support the use of real number on the FPU. In Volume One you saw, with one exception, how the standard input and output routines operate. This section will not repeat that discussion, see "HLA Support for Floating Point Values" on page 93 for more details. One input function that Volume One only mentioned briefly was the `stdin.getf` function. This section will elaborate on that function. The HLA Standard Library also includes the "math.hhf" module that provides several mathematical functions that the FPU doesn't directly support. This section will discuss those functions, as well.

3. Assuming, of course, that your calculations aren't so complex that you exceed the eight-element limitation of the FPU stack.

11.4.1 The `stdin.getf` and `fileio.getf` Functions

The `stdin.getf` function reads a floating point value from the standard input device. It leaves the converted value in ST0 (i.e., on the top of the floating point stack). The only reason Chapter Two did not discuss this function thoroughly was because you hadn't seen the FPU and FPU registers at that point.

The `stdin.getf` function accepts the same inputs that “`stdin.get(fp_variable);`” would except. The only difference between the two is where these functions store the floating point value.

As you'd probably surmise, there is a corresponding `fileio.getf` function as well. This function reads the floating point value from the file whose file handle is the single parameter in this function call. It, too, leaves the converted result on the top of the FPU stack.

11.4.2 Trigonometric Functions in the HLA Math Library

The FPU provides a small handful of trigonometric functions. It does not, however, support the full range of trig functions. The HLA MATH.HHF module fills in most of the missing functions. The trigonometric functions that HLA provides include

- ACOS(arc cosine)
- ACOT(arc cotangent)
- ACSC(arc cosecant)
- ASEC(arc secant)
- ASIN(arc sin)
- COT(cotangent)
- CSC(cosecant)
- SEC(secant)

The HLA Standard Library actually provides five different routines you can call for each of these functions. For example, the prototypes for the first four COT (cotangent) routines are:

```
procedure cot32( r32: real32 );
procedure cot64( r64: real64 );
procedure cot80( r80: real80 );
procedure _cot();
```

The first three routines push their parameter onto the FPU stack and compute the cotangent of the result. The fourth routine above (`_cot`) computes the cotangent of the value in ST0.

The fifth routine is actually an overloaded procedure that calls one of the four routines above depending on the parameter. This call uses the following syntax:

```
cot();           // Calls _cot() to compute cot(ST0).
cot( r32 );     // Calls cot32 to compute the cotangent of r32.
cot( r64 );     // Calls cot64 to compute the cotangent of r64.
cot( r80 );     // Calls cot80 to compute the cotangent of r80.
```

Using this fifth form is probably preferable since it is much more convenient. Note that there is no efficiency loss when you used `cot` rather than one of the other cotangent routines. HLA actually translates this statement directly into one of the other calls.

The HLA trigonometric functions that require an angle as a parameter expect that angle to be expressed in radians, not degrees. Keep in mind that some of these functions produce undefined results for certain input values. If you've enabled exceptions on the FPU, these functions will raise the appropriate FPU exception if an error occurs.

11.4.3 Exponential and Logarithmic Functions in the HLA Math Library

The HLA MATH.HHF module provides several exponential and logarithmic functions in addition to the trigonometric functions. Like the trig functions, the exponential and logarithmic functions provide five different interfaces to each function depending on the size and location of the parameter. The functions that MATH.HHF supports are

- TwoToX (raise 2.0 to the specified power).
- TenToX (raise 10.0 to the specified power).
- exp (raises e [2.718281828...] to the specified power).
- YtoX (raises first parameter to the power specified by the second parameter).
- log (computes base 10 logarithm).
- ln (computes base e logarithm).

Except for the *YtoX* function, all these functions provide the same sort of interface as the *cot* function mentioned in the previous section. For example, the *exp* function provides the following prototypes:

```
procedure exp32( r32: real32 );
procedure exp64( r64: real64 );
procedure exp80( r80: real80 );
procedure _exp();
```

The *exp* function, by itself, automatically calls one of the above functions depending on the parameter type (and presence of a parameter):

```
exp(); // Calls _exp() to compute exp(ST0).
exp( r32 ); // Calls exp32 to compute the e**r32.
exp( r64 ); // Calls exp64 to compute the e**r64.
exp( r80 ); // Calls exp80 to compute the e**r80.
```

The lone exception to the above is the *YtoX* function. *YtoX* has its own rules because it has two parameters rather than one (Y and X). *YtoX* provides the following function prototypes:

```
procedure YtoX32( y: real32; x: real32 );
procedure YtoX64( y: real64; x: real64 );
procedure YtoX80( y: real80; x: real80 );
procedure _YtoX();
```

The *_YtoX* function computes $ST1^{**}ST0$ (i.e., *ST1* raised to the *ST0* power).

The *YtoX* function provides the following interface:

```
YtoX(); // Calls _YtoX() to compute exp(ST0).
YtoX( y32, x32); // Calls YtoX32 to compute y32**x32.
YtoX( y64, x64 ); // Calls YtoX64 to compute y64**x64.
YtoX( y80, x80 ); // Calls YtoX80 to compute y80**x80.
```

11.5 Sample Program

This chapter presents a simple “Reverse Polish Notation” calculator that demonstrates the use of the 80x86 FPU. In addition to demonstrating the use of the FPU, this sample program also introduces a few new routines from the HLA Standard Library, specifically the *arg.c*, *arg.v*, and *conv.strToFlt* routines.

The HLA Standard Library conversions module (“conv.hhf”) contains dozens of procedures that translate data between various formats. A large percentage of these routines convert data between some internal numeric form and a corresponding string format. The *conv.strToFlt* routine, as its name suggests, converts string data to a floating point value. The prototype for this function is the following:

```
procedure conv.strToFlt( s:string; index:dword );
```

The first parameter is the string containing the textual representation of the floating point value. The second parameter contains an index into the string where the floating point text actually begins (usually the *index* parameter is zero if the string contains nothing but the floating point text). The *conv.strToFlt* procedure will attempt to convert the specified string to a floating point number. If there is a problem, this function will raise an appropriate exception (e.g., *ex.ConversionError*). In fact, the HLA *stdin* routines that read floating point values from the user actually read string data and call this same procedure to convert that data to a floating point value; hence, you should protect this procedure call with a TRY..ENDTRY statement exactly the same way you protect a call to *stdin.get* or *stdin.getf*. If this routine is successful, it leaves the converted floating point value on the top of the FPU stack.

The HLA Standard Library contains numerous other procedures that convert textual data to the corresponding internal format. Examples include *conv.strToi8*, *conv.strToi16*, *conv.strToi32*, *conv.strToi64*, and more. See the HLA Standard Library documentation for more details.

This sample program also uses the *arg.c* and *arg.v* routines from the HLA Standard Library's command line arguments module ("args.hhf"). These functions provide access to the text following the program name when you run a program from the command line prompt. This calculator program, for example, expects the user to supply the desired calculation on the command line immediately after the program name. For example to add the two values 18 and 22 together, you'd specify the following command line:

```
rpncalc 18 22 +
```

The text "18 22 +" is an example of three *command line parameters*. Programs often use command line parameters to communicate filenames and other data to the application. For example, the HLA compiler uses command line parameters to pass the names of the source files to the compiler. The *rpncalc* program uses the command line to pass the RPN expression to the calculator.

The *arg.c* function ("argument count") returns the number of parameters on the command line. This function returns the count in the EAX register. It does not have any parameters. In general, you can probably assume that the maximum possible number of command line arguments is between 64 and 128. Note that the operating system counts the program's name on the command line in this argument count. Therefore, this value will always be one or greater. If *arg.c* returns one, then there are no extra command line parameters; the single item is the program's name.

A program that expects at least one command line parameter should always call *arg.c* and verify that it returns the value two or greater. Programs that process command line parameters typically execute a loop of some sort that executes the number of times specified by *arg.c*'s return value. Of course, when you use *arg.c*'s return value for this purpose, don't forget to subtract one from the return result to account for the program's name (unless you are treating the program name as one more parameter).

The *arg.v* function returns a string containing one of the program's command line arguments. This function has the following prototype:

```
procedure arg.v( index:uns32 );
```

The *index* parameter specifies which command line parameter you wish to retrieve. The value zero returns a string containing the program's name. The value one returns the first command line parameter following the program's name. The value two returns a string containing the second command line parameter following the program's name. Etc. The value you provide as a parameter to this function must fall in the range 0..*arg.c*()-1 or *arg.v* will raise an exception.

The *arg.v* procedure allocates storage for the string it returns on the heap by calling *stralloc*. It returns a pointer to the allocated string in the EAX register. Don't forget to call *strfree* to return the storage to the system after you are done processing the command line parameter.

Well, without further ado, here is the RPN calculator program that uses the aforementioned functions.

```
// This sample program demonstrates how to use
```

```

// the FPU to create a simple RPN calculator.
// This program reads a string from the user
// and "parses" that string to figure out the
// calculation the user is requesting. This
// program assumes that any item beginning
// with a numeric digit is a numeric operand
// to push onto the FPU stack and all other
// items are operators.
//
// Example of typical user input:
//
//   calc 123.45 67.89 +
//
// The program responds by printing
//
//   Result = 1.9134000000000000e+2
//
// Current operators supported:
//
//   + - * /
//
// Current functions supported:
//
//   sin sqrt

program RPNcalculator;
#include( "stdlib.hhf" )

static
  argc:      uns32;
  curOperand: string;
  ItemsOnStk: uns32;
  realRslt:  real80;

// The following function converts an
// angle (in ST0) from degrees to radians.
// It leaves the result in ST0.

procedure DegreesToRadians; @nodisplay;
begin DegreesToRadians;

  fld( 2.0 ); // Radians = degrees*2*pi/360.0
  fmul();
  fldpi();
  fmul();
  fld( 360.0 );
  fdiv();

end DegreesToRadians;

begin RPNcalculator;

  // Initialize the FPU.

```



```

finit();

// Okay, extract the items from the Windows
// CMD.EXE command line and process them.

arg.c();
if( eax <= 1 ) then

    stdout.put( "Usage: `rpnCalc <rpn expression>'" nl );
    exit RPNcalculator;

endif;

// ECX holds the index of the current operand.
// ItemsOnStk keeps track of the number of numeric operands
// pushed onto the FPU stack so we can ensure that each
// operation has the appropriate number of operands.

mov( eax, argc );
mov( 1, ecx );
mov( 0, ItemsOnStk );

// The following loop repeats once for each item on the
// command line:

while( ecx < argc ) do

    // Get the string associated with the current item:

    arg.v( ecx ); // Note that this malloc's storage!
    mov( eax, curOperand );

    // If the operand begins with a numeric digit, assume
    // that it's a floating point number.

    if( (type char [eax]) in '0'..'9' ) then

        try

            // Convert this string representation of a numeric
            // value to the equivalent real value. Leave the
            // result on the top of the FPU stack. Also, bump
            // ItemsOnStk up by one since we're pushing a new
            // item onto the FPU stack.

            conv.strToFlt( curOperand, 0 );
            inc( ItemsOnStk );

        exception( ex.ConversionError )

            stdout.put("Illegal floating point constant" nl );
            exit RPNcalculator;

        anyexception

            stdout.put
            (
                "Exception ",
                (type uns32 eax ),

```

```
        " while converting real constant"
        nl
    );
    exit RPNcalculator;

endtry;

// Handle the addition operation here.

elseif( str.eq( curOperand, "+" )) then

    // The addition operation requires two
    // operands on the stack.  Ensure we have
    // two operands before proceeding.

    if( ItemsOnStk >= 2 ) then

        fadd();
        dec( ItemsOnStk ); // fadd() removes one operand.

    else

        stdout.put( "'+' operation requires two operands." nl );
        exit RPNcalculator;

    endif;

// Handle the subtraction operation here.  See the comments
// for FADD for more details.

elseif( str.eq( curOperand, "-" )) then

    if( ItemsOnStk >= 2 ) then

        fsub();
        dec( ItemsOnStk );

    else

        stdout.put( "'-' operation requires two operands." nl );
        exit RPNcalculator;

    endif;

// Handle the multiplication operation here.  See the comments
// for FADD for more details.

elseif( str.eq( curOperand, "*" )) then

    if( ItemsOnStk >= 2 ) then

        fmul();
        dec( ItemsOnStk );

    else

        stdout.put( "'*' operation requires two operands." nl );
```

```

        exit RPNcalculator;

    endif;

// Handle the division operation here.  See the comments
// for FADD for more details.

elseif( str.eq( curOperand, "/" ) ) then

    if( ItemsOnStk >= 2 ) then

        fdiv();
        dec( ItemsOnStk );

    else

        stdout.put( "`/' operation requires two operands." nl );
        exit RPNcalculator;

    endif;

// Provide a square root operation here.

elseif( str.eq( curOperand, "sqrt" ) ) then

    // Sqrt is a monadic (unary) function.  Therefore
    // we only require a single item on the stack.

    if( ItemsOnStk >= 1 ) then

        fsqrt();

    else

        stdout.put
        (
            "SQRT function requires at least one operand."
            nl
        );
        exit RPNcalculator;

    endif;

// Provide the SINE function here.  See SQRT comments for details.

elseif( str.eq( curOperand, "sin" ) ) then

    if( ItemsOnStk >= 1 ) then

        DegreesToRadians();
        fsin();

    else

        stdout.put( "SIN function requires at least one operand." nl );
        exit RPNcalculator;

```

```
        endif;

    else

        stdout.put( "`", curOperand, "` is an unknown operation." nl );
        exit RPNcalculator;

    endif;

    // Free the storage associated with the current item.

    strfree( curOperand );

    // Move on to the next item on the command line:

    inc( ecx );

endwhile;
if( ItemsOnStk = 1 ) then

    fstp( realRslt );
    stdout.put( "Result = ", realRslt, nl );

else

    stdout.put( "Syntax error in expression. ItemsOnStk=", ItemsOnStk, nl );

endif;

end RPNcalculator;
```

Program 11.1 A Floating Point Calculator Program

11.6 Putting It All Together

Between the FPU and the HLA Standard Library, floating point arithmetic is actually quite simple. In this chapter you learned about the floating point instruction set and you learned how to convert arithmetic expressions involving real arithmetic into a sequence of floating point instructions. This chapter also presented several transcendental functions that the HLA Standard Library provides. Armed with the information from this chapter, you should be able to deal with floating point expressions just as easily as integer expressions.

Calculation Via Table Lookups

Chapter Twelve

12.1 Chapter Overview

This chapter discusses arithmetic computation via table lookup. By the conclusion of this chapter you should be able to use table lookups to quickly compute complex functions. You will also learn how to construct these tables programmatically.

12.2 Tables

The term “table” has different meanings to different programmers. To most assembly language programmers, a table is nothing more than an array that is initialized with some data. The assembly language programmer often uses tables to compute complex or otherwise slow functions. Many very high level languages (e.g., SNOBOL4 and Icon) directly support a table data type. Tables in these languages are essentially arrays whose elements you can access with a non-integer index (e.g., floating point, string, or any other data type). HLA provides a table module that lets you index an array using a string. However, in this chapter we will adopt the assembly language programmer’s view of tables.

A table is an array containing preinitialized values that do not change during the execution of the program. A table can be compared to an array in the same way an integer constant can be compared to an integer variable. In assembly language, you can use tables for a variety of purposes: computing functions, controlling program flow, or simply “looking things up”. In general, tables provide a fast mechanism for performing some operation at the expense of some space in your program (the extra space holds the tabular data). In the following sections we’ll explore some of the many possible uses of tables in an assembly language program.

Note: since tables typically contain preinitialized data that does not change during program execution, the READONLY section is a good place to declare your table objects.

12.2.1 Function Computation via Table Look-up

Tables can do all kinds of things in assembly language. In HLLs, like Pascal, it’s real easy to create a formula which computes some value. A simple looking arithmetic expression can be equivalent to a considerable amount of 80x86 assembly language code. Assembly language programmers tend to compute many values via table look up rather than through the execution of some function. This has the advantage of being easier, and often more efficient as well. Consider the following Pascal statement:

```
if (character >= 'a') and (character <= 'z') then character := chr(ord(character) - 32);
```

This Pascal if statement converts the character variable character from lower case to upper case if character is in the range ‘a’..‘z’. The HLA code that does the same thing is

```
mov( character, al );
if( al in 'a'..'z' ) then

    and( $5f, al );        // Same as SUB( 32, al ) in this code.

endif;
mov( al, character );
```

Note that HLA’s high level IF statement translates into four machine instructions in this particular example. Hence, this code requires a total of seven machine instructions.

Had you buried this code in a nested loop, you'd be hard pressed to improve the speed of this code without using a table look up. Using a table look up, however, allows you to reduce this sequence of instructions to just four instructions:

```
mov( character, al );
lea( ebx, CnvrtLower );
xlat
mov( al, character );
```

You're probably wondering how this code works and what is this new instruction, XLAT? The XLAT, or *translate*, instruction does the following:

```
mov( [ebx+al*1], al );
```

That is, it uses the current value of the AL register as an index into the array whose base address is contained in EBX. It fetches the byte at that index in the array and copies that byte into the AL register. Intel calls this the translate instruction because programmers typically use it to translate characters from one form to another using a lookup table. That's exactly how we are using it here.

In the previous example, *CnvrtLower* is a 256-byte table which contains the values 0..\$60 at indices 0..\$60, \$41..\$5A at indices \$61..\$7A, and \$7B..\$FF at indices \$7Bh..0FF. Therefore, if AL contains a value in the range 0..\$60, the XLAT instruction returns the value 0..\$60, effectively leaving AL unchanged. However, if AL contains a value in the range \$61..\$7A (the ASCII codes for 'a'..'z') then the XLAT instruction replaces the value in AL with a value in the range \$41..\$5A. \$41..\$5A just happen to be the ASCII codes for 'A'..'Z'. Therefore, if AL originally contains an lower case character (\$61..\$7A), the XLAT instruction replaces the value in AL with a corresponding value in the range \$61..\$7A, effectively converting the original lower case character (\$61..\$7A) to an upper case character (\$41..\$5A). The remaining entries in the table, like entries 0..\$60, simply contain the index into the table of their particular element. Therefore, if AL originally contains a value in the range \$7A..\$FF, the XLAT instruction will return the corresponding table entry that also contains \$7A..\$FF.

As the complexity of the function increases, the performance benefits of the table look up method increase dramatically. While you would almost never use a look up table to convert lower case to upper case, consider what happens if you want to swap cases:

Via computation:

```
mov( character, al );
if( al in 'a'..'z' ) then
    and( $5f, al );
elseif( al in 'A'..'Z' ) then
    or( $20, al );
endif;
mov( al, character );
```

The IF and ELSEIF statements generate four and five actual machine instructions, respectively, so this code is equivalent to 13 actual machine instructions.

The table look up code to compute this same function is:

```
mov( character, al );
lea( ebx, SwapUL );
xlat();
mov( al, character );
```

As you can see, when using a table look up to compute a function only the table changes, the code remains the same.

Table look ups suffer from one major problem – functions computed via table look up have a limited domain. The domain of a function is the set of possible input values (parameters) it will accept. For example, the upper/lower case conversion functions above have the 256-character ASCII character set as their domain.

A function such as SIN or COS accepts the set of real numbers as possible input values. Clearly the domain for SIN and COS is much larger than for the upper/lower case conversion function. If you are going to do computations via table look up, you must limit the domain of a function to a small set. This is because each element in the domain of a function requires an entry in the look up table. You won't find it very practical to implement a function via table look up whose domain the set of real numbers.

Most look up tables are quite small, usually 10 to 128 entries. Rarely do look up tables grow beyond 1,000 entries. Most programmers don't have the patience to create (and verify the correctness) of a 1,000 entry table.

Another limitation of functions based on look up tables is that the elements in the domain of the function must be fairly contiguous. Table look ups take the input value for a function, use this input value as an index into the table, and return the value at that entry in the table. If you do not pass a function any values other than 0, 100, 1,000, and 10,000 it would seem an ideal candidate for implementation via table look up, its domain consists of only four items. However, the table would actually require 10,001 different elements due to the range of the input values. Therefore, you cannot efficiently create such a function via a table look up. Throughout this section on tables, we'll assume that the domain of the function is a fairly contiguous set of values.

The best functions you can implement via table look ups are those whose domain and range is always 0..255 (or some subset of this range). You can efficiently implement such functions on the 80x86 via the XLAT instruction. The upper/lower case conversion routines presented earlier are good examples of such a function. Any function in this class (those whose domain and range take on the values 0..255) can be computed using the same two instructions: "lea(table, ebx);" and "xlat();" The only thing that ever changes is the look up table.

You cannot (conveniently) use the XLAT instruction to compute a function value once the range or domain of the function takes on values outside 0..255. There are three situations to consider:

- The domain is outside 0..255 but the range is within 0..255,
- The domain is inside 0..255 but the range is outside 0..255, and
- Both the domain and range of the function take on values outside 0..255.

We will consider each of these cases separately.

If the domain of a function is outside 0..255 but the range of the function falls within this set of values, our look up table will require more than 256 entries but we can represent each entry with a single byte. Therefore, the look up table can be an array of bytes. Next to look ups involving the XLAT instruction, functions falling into this class are the most efficient. The following Pascal function invocation,

```
B := Func(X);
```

where *Func* is

```
function Func(X:dword):byte;
```

consists of the following HLA code:

```
mov( X, ebx );
mov( FuncTable[ ebx ], al );
mov( al, B );
```

This code loads the function parameter into EBX, uses this value (in the range 0..??) as an index into the *FuncTable* table, fetches the byte at that location, and stores the result into *B*. Obviously, the table must contain a valid entry for each possible value of *X*. For example, suppose you wanted to map a cursor position on the video screen in the range 0..1999 (there are 2,000 character positions on an 80x25 video display) to its *X* or *Y* coordinate on the screen. You could easily compute the *X* coordinate via the function:

```
X:=Posn mod 80
```

and the Y coordinate with the formula

$$Y := \text{Posn} \text{ div } 80$$

(where Posn is the cursor position on the screen). This can be easily computed using the 80x86 code:

```
mov( Posn, ax );
div( 80, ax );

// X is now in AH, Y is now in AL
```

However, the DIV instruction on the 80x86 is very slow. If you need to do this computation for every character you write to the screen, you will seriously degrade the speed of your video display code. The following code, which realizes these two functions via table look up, would improve the performance of your code considerably:

```
movzx( Posn, ebx );           // Use a plain MOV instr if Posn is uns32
mov( YCoord[ebx], al );      // rather than an uns16 value.
mov( XCoord[ebx], ah );
```

If the domain of a function is within 0..255 but the range is outside this set, the look up table will contain 256 or fewer entries but each entry will require two or more bytes. If both the range and domains of the function are outside 0..255, each entry will require two or more bytes and the table will contain more than 256 entries.

Recall from the chapter on arrays that the formula for indexing into a single dimensional array (of which a table is a special case) is

$$\text{Address} := \text{Base} + \text{index} * \text{size}$$

If elements in the range of the function require two bytes, then the index must be multiplied by two before indexing into the table. Likewise, if each entry requires three, four, or more bytes, the index must be multiplied by the size of each table entry before being used as an index into the table. For example, suppose you have a function, $F(x)$, defined by the following (pseudo) Pascal declaration:

```
function F(x:dword):word;
```

You can easily create this function using the following 80x86 code (and, of course, the appropriate table named F):

```
mov( X, ebx );
mov( F[ebx*2], ax );
```

Any function whose domain is small and mostly contiguous is a good candidate for computation via table look up. In some cases, non-contiguous domains are acceptable as well, as long as the domain can be coerced into an appropriate set of values. Such operations are called conditioning and are the subject of the next section.

12.2.2 Domain Conditioning

Domain conditioning is taking a set of values in the domain of a function and massaging them so that they are more acceptable as inputs to that function. Consider the following function:

$$\sin x = \langle \sin x | x \in [-2\pi, 2\pi] \rangle$$

This says that the (computer) function $\text{SIN}(x)$ is equivalent to the (mathematical) function $\sin x$ where $-2\pi \leq x \leq 2\pi$

As we all know, sine is a circular function which will accept any real valued input. The formula used to compute sine, however, only accept a small set of these values.

This range limitation doesn't present any real problems, by simply computing $\text{SIN}(X \bmod (2\pi))$ we can compute the sine of any input value. Modifying an input value so that we can easily compute a function is called conditioning the input. In the example above we computed " $X \bmod 2\pi$ " and used the result as the input to the *sin* function. This truncates X to the domain *sin* needs without affecting the result. We can apply input conditioning to table look ups as well. In fact, scaling the index to handle word entries is a form of input conditioning. Consider the following Pascal function:

```
function val(x:word):word; begin
  case x of
    0: val := 1;
    1: val := 1;
    2: val := 4;
    3: val := 27;
    4: val := 256;
    otherwise val := 0;
  end;
end;
```

This function computes some value for x in the range 0..4 and it returns zero if x is outside this range. Since x can take on 65,536 different values (being a 16 bit word), creating a table containing 65,536 words where only the first five entries are non-zero seems to be quite wasteful. However, we can still compute this function using a table look up if we use input conditioning. The following assembly language code presents this principle:

```
mov( 0, ax );           // AX = 0, assume X > 4.
movzx( x, ebx );       // Note that H.O. bits of EBX must be zero!
if( ebx <= 4 ) then

  mov( val[ ebx*2 ], ax );

endif;
```

This code checks to see if x is outside the range 0..4. If so, it manually sets AX to zero, otherwise it looks up the function value through the *val* table. With input conditioning, you can implement several functions that would otherwise be impractical to do via table look up.

12.2.3 Generating Tables

One big problem with using table look ups is creating the table in the first place. This is particularly true if there are a large number of entries in the table. Figuring out the data to place in the table, then laboriously entering the data, and, finally, checking that data to make sure it is valid, is a very time-staking and boring process. For many tables, there is no way around this process. For other tables there is a better way – use the computer to generate the table for you. An example is probably the best way to describe this. Consider the following modification to the sine function:

$$(\sin x) \times r = \left\langle \frac{(r \times (1000 \times \sin x))}{1000} \right\rangle_{x \in [0, 359]}$$

This states that x is an integer in the range 0..359 and r must be an integer. The computer can easily compute this with the following code:

```
movzx( x, ebx );
mov( Sines[ ebx*2 ], eax ); // Get SIN(X) * 1000
imul( r, eax );           // Note that this extends EAX into EDX.
idiv( 1000, edx:eax );    // Compute (R*(SIN(X)*1000)) / 1000
```

Note that integer multiplication and division are not associative. You cannot remove the multiplication by 1000 and the division by 1000 because they seem to cancel one another out. Furthermore, this code must

compute this function in exactly this order. All that we need to complete this function is a table containing 360 different values corresponding to the sine of the angle (in degrees) times 1,000. Entering such a table into an assembly language program containing such values is extremely boring and you'd probably make several mistakes entering and verifying this data. However, you can have the program generate this table for you. Consider the following HLA program:

```

program GenerateSines;
#include( "stdlib.hhf" );

var
    outFile: dword;
    angle:   int32;
    r:      int32;

readonly
    RoundMode: uns16 := $23f;

begin GenerateSines;

    // Open the file:
    mov( fileio.openNew( "sines.hla" ), outFile );

    // Emit the initial part of the declaration to the output file:
    fileio.put
    (
        outFile,
        stdio.tab,
        "sines: int32[360] := " nl,
        stdio.tab, stdio.tab, stdio.tab, "[" nl );

    // Enable rounding control (round to the nearest integer).
    fldcw( RoundMode );

    // Emit the sines table:
    for( mov( 0, angle); angle < 359; inc( angle ) ) do

        // Convert angle in degrees to an angle in radians
        // using "radians := angle * 2.0 * pi / 360.0;"

        fild( angle );
        fld( 2.0 );
        fmul();
        fldpi();
        fmul();
        fld( 360.0 );
        fdiv();

        // Okay, compute the sine of ST0

        fsin();

        // Multiply by 1000 and store the rounded result into
        // the integer variable r.

```

```

fld( 1000.0 );
fmul();
fistp( r );

// Write out the integers eight per line to the source file:
// Note: if (angle AND %111) is zero, then angle is evenly
// divisible by eight and we should output a newline first.

test( %111, angle );
if( @z ) then

    fileio.put
    (
        outFile,
        nl,
        stdio.tab,
        stdio.tab,
        stdio.tab,
        stdio.tab,
        r:5,
        ','
    );

else

    fileio.put( outFile, r:5, ',' );

endif;

endfor;

// Output sine(359) as a special case (no comma following it).
// Note: this value was computed manually with a calculator.

fileio.put
(
    outFile,
    " -17",
    nl,
    stdio.tab,
    stdio.tab,
    stdio.tab,
    "];",
    nl
);
fileio.close( outFile );

end GenerateSines;

```

Program 12.1 An HLA Program that Generates a Table of Sines

The program above produces the following output:

```

sines: int32[360] :=
[
    0,  17,  35,  52,  70,  87, 105, 122,

```

```

139, 156, 174, 191, 208, 225, 242, 259,
276, 292, 309, 326, 342, 358, 375, 391,
407, 423, 438, 454, 469, 485, 500, 515,
530, 545, 559, 574, 588, 602, 616, 629,
643, 656, 669, 682, 695, 707, 719, 731,
743, 755, 766, 777, 788, 799, 809, 819,
829, 839, 848, 857, 866, 875, 883, 891,
899, 906, 914, 921, 927, 934, 940, 946,
951, 956, 961, 966, 970, 974, 978, 982,
985, 988, 990, 993, 995, 996, 998, 999,
999, 1000, 1000, 1000, 999, 999, 998, 996,
995, 993, 990, 988, 985, 982, 978, 974,
970, 966, 961, 956, 951, 946, 940, 934,
927, 921, 914, 906, 899, 891, 883, 875,
866, 857, 848, 839, 829, 819, 809, 799,
788, 777, 766, 755, 743, 731, 719, 707,
695, 682, 669, 656, 643, 629, 616, 602,
588, 574, 559, 545, 530, 515, 500, 485,
469, 454, 438, 423, 407, 391, 375, 358,
342, 326, 309, 292, 276, 259, 242, 225,
208, 191, 174, 156, 139, 122, 105, 87,
70, 52, 35, 17, 0, -17, -35, -52,
-70, -87, -105, -122, -139, -156, -174, -191,
-208, -225, -242, -259, -276, -292, -309, -326,
-342, -358, -375, -391, -407, -423, -438, -454,
-469, -485, -500, -515, -530, -545, -559, -574,
-588, -602, -616, -629, -643, -656, -669, -682,
-695, -707, -719, -731, -743, -755, -766, -777,
-788, -799, -809, -819, -829, -839, -848, -857,
-866, -875, -883, -891, -899, -906, -914, -921,
-927, -934, -940, -946, -951, -956, -961, -966,
-970, -974, -978, -982, -985, -988, -990, -993,
-995, -996, -998, -999, -999, -1000, -1000, -1000,
-999, -999, -998, -996, -995, -993, -990, -988,
-985, -982, -978, -974, -970, -966, -961, -956,
-951, -946, -940, -934, -927, -921, -914, -906,
-899, -891, -883, -875, -866, -857, -848, -839,
-829, -819, -809, -799, -788, -777, -766, -755,
-743, -731, -719, -707, -695, -682, -669, -656,
-643, -629, -616, -602, -588, -574, -559, -545,
-530, -515, -500, -485, -469, -454, -438, -423,
-407, -391, -375, -358, -342, -326, -309, -292,
-276, -259, -242, -225, -208, -191, -174, -156,
-139, -122, -105, -87, -70, -52, -35, -17

```

```
];
```

Obviously it's much easier to write the HLA program that generated this data than to enter (and verify) this data by hand. Of course, you don't even have to write the table generation program in HLA. If you prefer, you might find it easier to write the program in Pascal/Delphi, C/C++, or some other high level language. Obviously, the program will only execute once, so the performance of the table generation program is not an issue. If it's easier to write the table generation program in a high level language, by all means do so. Note, also, that HLA has a built-in interpreter that allows you to easily create tables without having to use an external program. For more details, see the chapter on macros and the HLA compile-time language.

Once you run your table generation program, all that remains to be done is to cut and paste the table from the file (`sines.hla` in this example) into the program that will actually use the table.

12.3 High Performance Implementation of `cs.rangeChar`

Way back in Chapter Three this volume made the comment that the implementation of the `cs.rangeChar` was not very efficient when generating large character sets (see “Character Set Functions That Build Sets” on page 449). That chapter also mentioned that a table lookup would be a better solution for this function if you generate large character sets. That chapter also promised an table lookup implementation of `cs.rangeChar`. This section fulfills that promise.

Program 12.2 provides a table lookup implementation of this function. To understand how this function works, consider the two tables (*StartRange* and *EndRange*) appearing in this program.

Each element in the *StartRange* table is a character set whose binary representation contains all one bits from bit position zero through the index into the table. That is, element zero contains a single ‘1’ bit in bit position zero; element one contains one bits in bit positions zero and one; element two contains one bits in bit positions zero, one, and two; etc.

Each element of the *EndRange* table contains one bits from the bit position specified by the index into the table through to bit position 127. Therefore, element zero of this array contains all one bits from positions zero through 127; element one of this array contains a zero in bit position zero and ones in bit positions one through 127; element two of this array contains zeros in bit positions zero and one and it contains ones in bit positions two through 127; etc.

The *fastRangeChar* function builds a character set containing all the characters between two characters specified as parameters. The calling sequence for this function is

```
fastRangeChar( LowBoundChar, HighBoundChar, CsetVariable );
```

This function constructs the character set “{ LowBoundChar..HighBoundChar }” and stores this character set into *CsetVariable*.

As you may recall from the discussion of `cs.rangeChar`’s low-level implementation, it constructed the character set by running a FOR loop from the *LowBoundChar* through to the *HighBoundChar* and set the corresponding bit in the character set on each iteration of the loop. So to build the character set { ‘a’..’z’ } the loop would have to execute 26 times. The *fastRangeChar* function avoids this iteration by construction a set containing all elements from #0 to *HighBoundChar* and intersecting this set with a second character set containing all the characters from *LowBoundChar* to #127. The *fastRangeChar* function doesn’t actually build these two sets, of course, it uses *HighBoundChar* and *LowBoundChar* as indices into the *StartRange* and *EndRange* tables, respectively. The intersection of these two table elements computes the desired result set. As you’ll see by looking at *fastRangeChar*, this function computes the intersection of these two sets on the fly by using the AND instruction. Without further ado, here’s the program:

```
program csRangeChar;
#include( "stdlib.hhf" )

static

    // Note: the following tables were generated
    // by the genRangeChar program:

    StartRange: cset[128] :=
        [
            {#0},
            {#0..#1},
            {#0..#2},
            {#0..#3},
            {#0..#4},
            {#0..#5},
```

{#0..#6},
{#0..#7},
{#0..#8},
{#0..#9},
{#0..#10},
{#0..#11},
{#0..#12},
{#0..#13},
{#0..#14},
{#0..#15},
{#0..#16},
{#0..#17},
{#0..#18},
{#0..#19},
{#0..#20},
{#0..#21},
{#0..#22},
{#0..#23},
{#0..#24},
{#0..#25},
{#0..#26},
{#0..#27},
{#0..#28},
{#0..#29},
{#0..#30},
{#0..#31},
{#0..#32},
{#0..#33},
{#0..#34},
{#0..#35},
{#0..#36},
{#0..#37},
{#0..#38},
{#0..#39},
{#0..#40},
{#0..#41},
{#0..#42},
{#0..#43},
{#0..#44},
{#0..#45},
{#0..#46},
{#0..#47},
{#0..#48},
{#0..#49},
{#0..#50},
{#0..#51},
{#0..#52},
{#0..#53},
{#0..#54},
{#0..#55},
{#0..#56},
{#0..#57},
{#0..#58},
{#0..#59},
{#0..#60},
{#0..#61},
{#0..#62},
{#0..#63},
{#0..#64},
{#0..#65},
{#0..#66},

{#0..#67},
{#0..#68},
{#0..#69},
{#0..#70},
{#0..#71},
{#0..#72},
{#0..#73},
{#0..#74},
{#0..#75},
{#0..#76},
{#0..#77},
{#0..#78},
{#0..#79},
{#0..#80},
{#0..#81},
{#0..#82},
{#0..#83},
{#0..#84},
{#0..#85},
{#0..#86},
{#0..#87},
{#0..#88},
{#0..#89},
{#0..#90},
{#0..#91},
{#0..#92},
{#0..#93},
{#0..#94},
{#0..#95},
{#0..#96},
{#0..#97},
{#0..#98},
{#0..#99},
{#0..#100},
{#0..#101},
{#0..#102},
{#0..#103},
{#0..#104},
{#0..#105},
{#0..#106},
{#0..#107},
{#0..#108},
{#0..#109},
{#0..#110},
{#0..#111},
{#0..#112},
{#0..#113},
{#0..#114},
{#0..#115},
{#0..#116},
{#0..#117},
{#0..#118},
{#0..#119},
{#0..#120},
{#0..#121},
{#0..#122},
{#0..#123},
{#0..#124},
{#0..#125},
{#0..#126},
{#0..#127}

```
];  
  
EndRange: cset[128] :=  
[  
  {#0..#127},  
  {#1..#127},  
  {#2..#127},  
  {#3..#127},  
  {#4..#127},  
  {#5..#127},  
  {#6..#127},  
  {#7..#127},  
  {#8..#127},  
  {#9..#127},  
  {#10..#127},  
  {#11..#127},  
  {#12..#127},  
  {#13..#127},  
  {#14..#127},  
  {#15..#127},  
  {#16..#127},  
  {#17..#127},  
  {#18..#127},  
  {#19..#127},  
  {#20..#127},  
  {#21..#127},  
  {#22..#127},  
  {#23..#127},  
  {#24..#127},  
  {#25..#127},  
  {#26..#127},  
  {#27..#127},  
  {#28..#127},  
  {#29..#127},  
  {#30..#127},  
  {#31..#127},  
  {#32..#127},  
  {#33..#127},  
  {#34..#127},  
  {#35..#127},  
  {#36..#127},  
  {#37..#127},  
  {#38..#127},  
  {#39..#127},  
  {#40..#127},  
  {#41..#127},  
  {#42..#127},  
  {#43..#127},  
  {#44..#127},  
  {#45..#127},  
  {#46..#127},  
  {#47..#127},  
  {#48..#127},  
  {#49..#127},  
  {#50..#127},  
  {#51..#127},  
  {#52..#127},  
  {#53..#127},  
  {#54..#127},  
  {#55..#127},  
  {#56..#127},
```


{#57..#127},
{#58..#127},
{#59..#127},
{#60..#127},
{#61..#127},
{#62..#127},
{#63..#127},
{#64..#127},
{#65..#127},
{#66..#127},
{#67..#127},
{#68..#127},
{#69..#127},
{#70..#127},
{#71..#127},
{#72..#127},
{#73..#127},
{#74..#127},
{#75..#127},
{#76..#127},
{#77..#127},
{#78..#127},
{#79..#127},
{#80..#127},
{#81..#127},
{#82..#127},
{#83..#127},
{#84..#127},
{#85..#127},
{#86..#127},
{#87..#127},
{#88..#127},
{#89..#127},
{#90..#127},
{#91..#127},
{#92..#127},
{#93..#127},
{#94..#127},
{#95..#127},
{#96..#127},
{#97..#127},
{#98..#127},
{#99..#127},
{#100..#127},
{#101..#127},
{#102..#127},
{#103..#127},
{#104..#127},
{#105..#127},
{#106..#127},
{#107..#127},
{#108..#127},
{#109..#127},
{#110..#127},
{#111..#127},
{#112..#127},
{#113..#127},
{#114..#127},
{#115..#127},
{#116..#127},
{#117..#127},

```

        {#118..#127},
        {#119..#127},
        {#120..#127},
        {#121..#127},
        {#122..#127},
        {#123..#127},
        {#124..#127},
        {#125..#127},
        {#126..#127},
        {#127}
];

/*****
/*
/* fastRangeChar-
/*
/* A fast implementation of cs.rangeChar that uses a table lookup
/* to speed up the generation of the character set for really large
/* sets (note: because of memory latencies, this function is probably
/* slower than cs.rangeChar for small character sets).
/*
/*
*****/

procedure fastRangeChar( LowBound:char; HighBound:char; var csDest:cset );
begin fastRangeChar;

    push( eax );
    push( ebx );
    push( esi );
    push( edi );
    mov( csDest, ebx ); // Get pointer to destination character set.

    // Copy EndRange[ LowBound ] into csDest. This adds all the
    // characters from LowBound to #127 into csDest. Then intersect
    // this set with StartRange[ HighBound ] to trim off all the
    // characters after HighBound.

    movzx( LowBound, esi );
    shl( 4, esi ); // *16 'cause each element is 16 bytes.
    movzx( HighBound, edi );
    shl( 4, edi );

    mov( (type dword EndRange[ esi + 0 ]), eax );
    and( (type dword StartRange[ edi + 0 ]), eax ); // Does the intersection.
    mov( eax, (type dword [ ebx+0 ]));

    mov( (type dword EndRange[ esi + 4 ]), eax );
    and( (type dword StartRange[ edi + 4 ]), eax );
    mov( eax, (type dword [ ebx+4 ]));

    mov( (type dword EndRange[ esi + 8 ]), eax );
    and( (type dword StartRange[ edi + 8 ]), eax );
    mov( eax, (type dword [ ebx+8 ]));

    mov( (type dword EndRange[ esi + 12 ]), eax );
    and( (type dword StartRange[ edi + 12 ]), eax );
    mov( eax, (type dword [ ebx+12 ]));

    pop( edi );
    pop( esi );

```

```

    pop( ebx );
    pop( eax );

end fastRangeChar;

static
    TestCset: cset := {};

begin csRangeChar;

    fastRangeChar( 'a', 'z', TestCset );
    stdout.put( "Result from fastRangeChar: {" , TestCset, "}" nl );

end csRangeChar;

```

Program 12.2 Table Lookup Implementation of cs.rangeChar

Naturally, the *StartRange* and *EndRange* tables were not hand-generated. An HLA program generated these two tables (with a combined 512 elements). Program 12.3 is the program that generated these tables.

```

program GenerateRangeChar;
#include( "stdlib.hhf" );

var
    outFile: dword;

begin GenerateRangeChar;

    // Open the file:

    mov( fileio.openNew( "rangeCharCset.hla" ), outFile );

    // Emit the initial part of the declaration to the output file:

    fileio.put
    (
        outFile,
        stdio.tab,
        "StartRange: cset[128] := " nl,
        stdio.tab, stdio.tab, stdio.tab, "[" nl,
        stdio.tab, stdio.tab, stdio.tab, stdio.tab, "{#0}," nl // element zero
    );

    for( mov( 1, ecx ); ecx < 127; inc( ecx ) ) do

        fileio.put
        (
            outFile,
            stdio.tab, stdio.tab, stdio.tab, stdio.tab,

```

```

        "{#0..#",
        (type uns32 ecx),
        "}," nl
    );

endfor;
fileio.put
(
    outFile,
    stdio.tab, stdio.tab, stdio.tab, stdio.tab,
    "{#0..#127}" nl,
    stdio.tab, stdio.tab, stdio.tab, "]" nl
);

// Now emit the second table to the file:

fileio.put
(
    outFile,
    nl,
    stdio.tab,
    "EndRange: cset[128] := " nl,
    stdio.tab, stdio.tab, stdio.tab, "[" nl
);
for( mov( 0, ecx ); ecx < 127; inc( ecx ) ) do

    fileio.put
    (
        outFile,
        stdio.tab, stdio.tab, stdio.tab, stdio.tab,
        "{#",
        (type uns32 ecx),
        "..#127}," nl
    );

endfor;
fileio.put
(
    outFile,
    stdio.tab, stdio.tab, stdio.tab, stdio.tab, "{#127}" nl,
    stdio.tab, stdio.tab, stdio.tab, "]" nl nl
);
fileio.close( outFile );

end GenerateRangeChar;

```

Program 12.3 Table Generation Program for the csRangeChar Program

Volume Four: Intermediate Assembly Language

This volume completes the material traditionally taught in a 10 or 15 week course on assembly language programming (the difference between such courses is how many chapters they've skipped up to this point). This volume also completes this text's discussion of the essential material you need to know to start using assembly language effectively. Although there is still much for you to learn, after you complete this volume any further study of assembly language tends to be more specialized. In any case, mastery of the material up to the end of this volume is an important milestone. Once you absorb and are able to apply this material, you can start calling yourself an "Assembly Language Programmer."

- Chapter One: Advanced High Level Control Structures I
- This chapter completes the discussion of HLA's high level control structures. It completely discusses TRY..ENDTRY and introduces several new high level control structures.
- Chapter Two: Low Level Control Structures
- This chapter discusses the "real" way to do control structures, using "pure" assembly language. This is a very important chapter; you cannot call yourself an assembly language programmer if you haven't mastered the low-level control structures.
- Chapter Three: Intermediate Procedures
- This chapter extends the information on procedures found in the previous volume. This chapter discusses some of the low-level implementation details of procedures and describes how to call procedures and pass parameters using "pure" assembly language.
- Chapter Four: Advanced Arithmetic
- This chapter discusses multiprecision and binary coded decimal arithmetic. It also describes how to input and output very large values (code included!).
- Chapter Five: Bit Manipulation
- This chapter discusses bit operations in assembly language. You'll learn how to deal with packed data, insert and extract bit strings, count bits in an operand, and do all other sorts of bit-related stuff.
- Chapter Six: The String Instructions
- This chapter discusses the 80x86 string instructions which are convenient for manipulating large blocks of memory.

Volume Four:

Intermediate Assembly Language

Chapter Seven: The HLA Compile-Time Language

This chapter begins the discussion of one of HLA's most powerful features - the HLA compile time language. In this chapter you'll learn about conditional compilation, compile-time loops, compile-time functions, generating tables, and lots of other features that make assembly language programming easier.

Chapter Eight: Macros

This chapter continues the discussion of the HLA compile-time language with a discussion of one of HLA's most powerful features – the HLA macro processor. In this chapter you'll learn how to extend the HLA language and do all those neat things that the HLA Standard Library provides.

Chapter Nine: Domain Specific Languages

This chapter describes how to design and implement your own programming language inside HLA.

Chapter Ten: Classes

This chapter describes classes and object-oriented programming in HLA.

Chapter Eleven: The MMX Instruction Set

This chapter describes the special MMX multimedia extensions on the Pentium and later chips.

Chapter Twelve Mixed Language Programming

This chapter describes how to call HLA procedures and access HLA data from other languages.

Chapter Thirteen: Questions, Projects, and Laboratory Exercises

Test your knowledge and see how well you've learned the material in this chapter!

Advanced High Level Control Structures Chapter One

1.1 Chapter Overview

Volume One introduced some basic HLA control structures like the IF and WHILE statements (see “Some Basic HLA Control Structures” on page 29.). This section elaborates on some of those control structures (discussing features that were a little too advanced to present in Volume One) and it introduces the remaining high level language control structures that HLA provides.

This includes a full discussion of HLA’s boolean expressions, the TRY..ENDTRY statement, the RAISE statement, the BEGIN..END/EXIT/EXITIF statements, and the SWITCH/CASE/ENDSWITCH statement the HLA Standard Library provides.

1.2 Conjunction, Disjunction, and Negation in Boolean Expressions

One obvious omission in HLA’s high level control structures is the ability to use conjunction (logical AND), disjunction (logical OR), and negation (logical NOT) in run-time boolean expressions. This omission, however, has been in this text, not in the HLA language. HLA does provide these facilities, this section will describe their use.

HLA uses the “&&” operator to denote logical AND in a run-time boolean expression. This is a dyadic (two-operand) operator and the two operands must be legal run-time boolean expressions. This operator evaluates true if both operands evaluate to true. Example:

```
if( eax > 0 && ch = 'a' ) then

    mov( eax, ebx );
    mov( ' ', ch );

endif;
```

The two MOV statements above execute only if EAX is greater than zero *and* CH is equal to the character ‘a’. If either of these conditions is false, then program execution skips over the two MOV instructions.

Note that the expressions on either side of the “&&” operator may be any expression that is legal in an IF statement, these expressions don’t have to be comparisons using one of the relational operators. For example, the following are all legal expressions:

```
@z && al in 5..10
al in {'a'..'z'} && ebx
boolVar && !eax
!fileio.eof( fileHandle ) && fileio.getc( fileHandle ) <> ' '
```

HLA uses short circuit evaluation when compiling the “&&” operator. If the left-most operand evaluates false, then the code that HLA generates does not bother evaluating the second operand (since the whole expression must be false at that point). Therefore, in the last expression above, the code will not execute the call to *fileio.getc* if the file pointer is currently pointing at the end of the file.

Note that an expression like “*eax < 0 && ebx <> eax*” is itself a legal boolean expression and, therefore, may appear as the left or right operand of the “&&” operator. Therefore, expressions like the following are perfectly legal:

```
eax < 0 && ebx <> eax && !ecx
```

The “&&” operator is left associative, so the code that HLA generates evaluates the expression above in a left-to-right fashion. Note that if EAX is less than zero, the code will not test either of the remaining expres-

sions. Likewise, if EAX is not less than zero but EBX is equal to EAX, this code will not evaluate the third expression since the whole expression is false regardless of ECX's value.

HLA uses the “||” operator to denote disjunction (logical OR) in a run-time boolean expression. Like the “&&” operator, this operator expects two otherwise legal run-time boolean expressions as operands. This operator evaluates true if either (or both) operands evaluate true. Like the “&&” operator, the disjunction operator uses short-circuit evaluation. If the left operand evaluates true, then the code that HLA generates doesn't bother to test the value of the second operand. Instead, the code will transfer to the location that handles the situation when the boolean expression evaluates true. Examples of legal expressions using the “||” operator:

```
@z || al = 10
al in {'a'..'z'} || ebx
!boolVar || eax
```

As for the “&&” operator, the disjunction operator is left associative so multiple instances of the “||” operator may appear within the same expression. Should this be the case, the code that HLA generates will evaluate the expressions from left to right, e.g.,

```
eax < 0 || ebx <> eax || !ecx
```

The code above executes if either EAX is less than zero, EBX does not equal EAX, or ECX is zero. Note that if the first comparison is true, the code doesn't bother testing the other conditions. Likewise, if the first comparison is false and the second is true, the code doesn't bother checking to see if ECX is zero. The check for ECX equal to zero only occurs if the first two comparisons are false.

If both the conjunction and disjunction operators appear in the same expression then the “&&” operator takes precedence over the “||” operator. Consider the following expression:

```
eax < 0 || ebx <> eax && !ecx
```

The code HLA generates evaluates this as

```
eax < 0 || (ebx <> eax && !ecx)
```

If EAX is less than zero, then the code HLA generates does not bother to check the remainder of the expression, the entire expression evaluates true. However, if EAX is not less than zero, then both of the following conditions must evaluate true in order for the overall expression to evaluate true.

HLA allows you to use parentheses to surround subexpressions involving “&&” and “||” if you need to adjust the precedence of the operators. Consider the following expression:

```
(eax < 0 || ebx <> eax) && !ecx
```

For this expression to evaluate true, ECX must contain zero and either EAX must be less than zero or EBX must not equal EAX. Contrast this to the result obtained without the parentheses.

As you saw in Volume One, HLA uses the “!” operator to denote logical negation. However, the “!” operator may only prefix a register or boolean variable; you may not use it as part of a larger expression (e.g., “!eax < 0”). To achieve logical negative of an existing boolean expression you must surround that expression with parentheses and prefix the parentheses with the “!” operator, e.g.,

```
!( eax < 0 )
```

This expression evaluates true if EAX is not less than zero.

The logical not operator is primarily useful for surrounding complex expressions involving the conjunction and disjunction operators. While it is occasionally useful for short expressions like the one above, it's usually easier (and more readable) to simply state the logic directly rather than convolute it with the logical not operator.

Note that HLA's “|” and “&” operators (compile-time address expressions) are distinct from “||” and “&&” and have completely different meanings. See the chapter on the HLA Run-Time Language in this volume or the chapter on constants in the previous volume for details.

1.3 TRY..ENDTRY

Volume One discusses the TRY..ENDTRY statement, but does not fully discuss all of the features available. This section will complete the discussion of TRY..ENDTRY and discuss some problems that could occur when you use this statement.

As you may recall, the TRY..ENDTRY statement surrounds a block of statements in order to capture any exceptions that occur during the execution of those statements. The system raises exceptions in one of three ways: through a hardware fault (such as a divide by zero error), through an operating system generated exception, or through the execution of the HLA RAISE statement. You can write an exception handler to intercept specific exceptions using the EXCEPTION clause. The following program provides a typical example of the use of this statement:

```

program testBadInput;
#include( "stdlib.hhf" );

static
    u:      uns16;

begin testBadInput;

    try

        stdout.put( "Enter an unsigned integer:" );
        stdin.get( u );
        stdout.put( "You entered: ", u, nl );

        exception( ex.ConversionError )

            stdout.put( "Your input contained illegal characters" nl );

        exception( ex.ValueOutOfRange )

            stdout.put( "The value was too large" nl );

    endtry;

end testBadInput;

```

Program 1.1 TRY..ENDTRY Example

HLA refers to the statements between the TRY clause and the first EXCEPTION clause as the *protected* statements. If an exception occurs within the protected statements, then the program will scan through each of the exceptions and compare the value of the current exception against the value in the parentheses after each of the EXCEPTION clauses¹. This exception value is simply an *uns32* value. The value in the parentheses after each EXCEPTION clause, therefore, must be an unsigned 32-bit value. The HLA "excepts.hhf" header file predefines several exception constants. Other than it would be an incredibly bad style violation,

1. Note that HLA loads this value into the EAX register. So upon entry into an EXCEPTION clause, EAX contains the exception number.

you could substitute the numeric values for the two EXCEPTION clauses above (see the `excepts.hhf` header file for the actual values).

1.3.1 Nesting TRY..ENDTRY Statements

If the program scans through all the exception clauses in a TRY..ENDTRY statement and does not match the current exception value, then the program searches through the EXCEPTION clauses of a *dynamically nested* TRY..ENDTRY block in an attempt to find an appropriate exception handler. For example, consider the following code:

```

program testBadInput2;
#include( "stdlib.hhf" );

static
    u:      uns16;

begin testBadInput2;

    try

        try

            stdout.put( "Enter an unsigned integer:" );
            stdin.get( u );
            stdout.put( "You entered: ", u, nl );

            exception( ex.ConversionError )

                stdout.put( "Your input contained illegal characters" nl );

        endtry;

        stdout.put( "Input did not fail due to a value out of range" nl );

        exception( ex.ValueOutOfRange )

            stdout.put( "The value was too large" nl );

        endtry;

    end testBadInput2;

```

Program 1.2 Nested TRY..ENDTRY Statements

In this example one TRY statement is nested inside another. During the execution of the `stdin.get` statement, if the user enters a value greater than four billion and some change, then `stdin.get` will raise the `ex.ValueOutOfRange` exception. When the HLA run-time system receives this exception, it first searches through all the EXCEPTION clauses in the TRY..ENDTRY statement immediately surrounding the statement that raised the exception (this would be the nested TRY..ENDTRY in the example above). If the HLA run-time system fails to locate an exception handler for `ex.ValueOutOfRange` then it checks to see if the current TRY..ENDTRY is nested inside another TRY..ENDTRY (as is the case in Program 1.2). If so, the HLA

run-time system searches for the appropriate EXCEPTION clause in that TRY..ENDTRY statement. Within this TRY..ENDTRY block the program finds an appropriate exception handler, so control transfers to the statements after the “exception(ex.ValueOutOfRange)” clause.

After leaving a TRY..ENDTRY block, the HLA run-time system no longer considers that block active and will not search through its list of exceptions when the program raises an exception². This allows you to handle the same exception differently in other parts of the program.

If two nested TRY..ENDTRY statements handle the same exception, and the program raises an exception while executing in the innermost TRY..ENDTRY sequence, then HLA transfers control directly to the exception handler provided by that TRY..ENDTRY block. HLA does not automatically transfer control to the exception handler provided by the outer TRY..ENDTRY sequence.

If the program raises an exception for which there is no appropriate EXCEPTION clause active, control transfers to the HLA run-time system. It will stop the program and print an appropriate error message.

In the previous example (Program 1.2) the second TRY..ENDTRY statement was statically nested inside the enclosing TRY..ENDTRY statement³. As mentioned without comment earlier, if the most recently activated TRY..ENDTRY statement does not handle a specific exception, the program will search through the EXCEPTION clauses of any dynamically nesting TRY..ENDTRY blocks. Dynamic nesting does not require the nested TRY..ENDTRY block to physically appear within the enclosing TRY..ENDTRY statement. Instead, control could transfer from inside the enclosing TRY..ENDTRY protected block to some other point in the program. Execution of a TRY..ENDTRY statement at that other point dynamically nests the two TRY statements. Although you will see lots of ways to dynamically nest code a little later in this chapter, there is one method you are familiar with that will let you dynamically nest these statements: the procedure call. The following program provides yet another example of nested TRY..ENDTRY statements, this example demonstrates dynamic nesting via a procedure call:

```

program testBadInput3;
#include( "stdlib.hhf" );

procedure getUns;
static
    u:      uns16;

begin getUns;

    try

        stdout.put( "Enter an unsigned integer:" );
        stdin.get( u );
        stdout.put( "You entered: ", u, nl );

        exception( ex.ConversionError )

            stdout.put( "Your input contained illegal characters" nl );

    endtry;

end getUns;

begin testBadInput3;

```

2. Unless, of course, the program re-enters the TRY..ENDTRY block via a loop or other control structure.

3. Statically nested means that one statement is physically nested within another in the source code. When we say one statement is nested within another, this typically means that the statement is statically nested within the other statement.

```

try

    getUns();
    stdout.put( "Input did not fail due to a value out of range" nl );

    exception( ex.ValueOutOfRange )

        stdout.put( "The value was too large" nl );

endtry;

end testBadInput3;

```

Program 1.3 Dynamic Nesting of TRY..ENDTRY Statements

In Program 1.3 the main program executes the TRY statement that activates a value out of range exception handler, then it calls the *getUns* procedure. Inside the *getUns* procedure, the program executes a second TRY statement. This dynamically nests this TRY..ENDTRY block inside the TRY of the main program. Because the main program has not yet encountered its ENDTRY, the TRY..ENDTRY block in the main program is still active. However, upon execution of the TRY statement in *getUns*, the nested TRY..ENDTRY block takes precedence. If an exception occurs inside the *stdin.get* procedure, control transfers to the most recently activated TRY..ENDTRY block of statements and the program scans through the EXCEPTION clauses looking for a match to the current exception value. In the program above, if the exception is a conversion error exception, then the exception handler inside *getUns* will handle the error and print an appropriate message. After the execution of the exception handler, the program falls through to the bottom of *getUns* and it returns to the main program and prints the message "Input did not fail due to a value out of range". Note that if a nested exception handler processes an exception, the program does not automatically reraise this exception in other active TRY..ENDTRY blocks, even if they handle that same exception (*ex.ConversionError*, in this case).

Suppose, however, that *stdin.get* raises the *ex.ValueOutOfRange* exception rather than the *ex.ConversionError* exception. Since the TRY..ENDTRY statement inside *getUns* does not handle this exception, the program will search through the exception list of the enclosing TRY..ENDTRY statement. Since this statement is in the main program, the exception will cause the program to automatically return from the *getUns* procedure. Since the program will find the value out of range exception handler in the main program, it transfers control directly to the exception handler. Note that the program will not print the string "Input did not fail due to a value out of range" since control transfers directly from *stdin.get* to the exception handler.

1.3.2 The UNPROTECTED Clause in a TRY..ENDTRY Statement

Whenever a program executes the TRY clause, it preserves the current exception environment and sets up the system to transfer control to the EXCEPTION clauses within that TRY..ENDTRY statement should an exception occur. If the program successfully completes the execution of a TRY..ENDTRY protected block, the program restores the original exception environment and control transfers to the first statement beyond the ENDTRY clause. This last step, restoring the execution environment, is very important. If the program skips this step, any future exceptions will transfer control to this TRY..ENDTRY statement even though the program has already left the TRY..ENDTRY block. The following program demonstrates this problem:

```

program testBadInput4;
#include( "stdlib.hhf" );

```

```

static
    input: uns32;

begin testBadInput4;

    // This forever loop repeats until the user enters
    // a good integer and the BREAK statement below
    // exits the loop.

    forever

        try

            stdout.put( "Enter an integer value: " );
            stdin.get( input );
            stdout.put( "The first input value was: ", input, nl );
            break;

            exception( ex.ValueOutOfRange )

                stdout.put( "The value was too large, reenter." nl );

            exception( ex.ConversionError )

                stdout.put( "The input contained illegal characters, reenter." nl );

        endtry;

    endfor;

    // Note that the following code is outside the loop and there
    // is no TRY..ENDTRY statement protecting this code.

    stdout.put( "Enter another number: " );
    stdin.get( input );
    stdout.put( "The new number is: ", input, nl );

end testBadInput4;

```

Program 1.4 Improperly Exiting a TRY..ENDTRY Statement

This example attempts to create a robust input system by putting a loop around the TRY..ENDTRY statement and forcing the user to reenter the data if the *stdin.get* routine raises an exception (because of bad input data). While this is a good idea, there is a big problem with this implementation: the BREAK statement immediately exits the FOREVER..ENDFOR loop without first restoring the exception environment. Therefore, when the program executes the second *stdin.get* statement, at the bottom of the program, the HLA exception handling code still thinks that it's inside the TRY..ENDTRY block. If an exception occurs, HLA transfers control back into the TRY..ENDTRY statement looking for an appropriate exception handler. Assuming the exception was *ex.ValueOutOfRange* or *ex.ConversionError*, Program 1.4 will print an appropriate error message *and then force the user to reenter the first value*. This isn't desirable.

Transferring control to the wrong TRY..ENDTRY exception handlers is only part of the problem. Another big problem with the code in Program 1.4 has to do with the way HLA preserves and restores the exception environment: specifically, HLA saves the old execution environment information on the stack. If you exit a TRY..ENDTRY without restoring the exception environment, this leaves garbage on the stack (the old execution environment information) and this extra data on the stack could cause your program to malfunction.

Although it is quite clear that a program should not exit from a TRY..ENDTRY statement in the manner that Program 1.4 uses, it would be nice if you could use a loop around a TRY..ENDTRY block to force the re-entry of bad data as this program attempts to do. To allow for this, HLA's TRY..ENDTRY provides an UNPROTECTED section. Consider the following program code:

```

program testBadInput5;
#include( "stdlib.hhf" );

static
    input:  uns32;

begin testBadInput5;

    // This forever loop repeats until the user enters
    // a good integer and the BREAK statement below
    // exits the loop. Note that the BREAK statement
    // appears in an UNPROTECTED section of the TRY..ENDTRY
    // statement.

    forever

        try

            stdout.put( "Enter an integer value: " );
            stdin.get( input );
            stdout.put( "The first input value was: ", input, nl );

            unprotected

                break;

            exception( ex.ValueOutOfRange )

                stdout.put( "The value was too large, reenter." nl );

            exception( ex.ConversionError )

                stdout.put( "The input contained illegal characters, reenter." nl );

        endtry;

    endfor;

    // Note that the following code is outside the loop and there
    // is no TRY..ENDTRY statement protecting this code.

    stdout.put( "Enter another number: " );
    stdin.get( input );
    stdout.put( "The new number is: ", input, nl );

end testBadInput5;

```

Program 1.5 The TRY..ENDTRY UNPROTECTED Section

Whenever the TRY..ENDTRY statement hits the UNPROTECTED clause, it immediately restores the exception environment from the stack. As the phrase suggests, the execution of statements in the UNPRO-

TECTED section is no longer protected by the enclosing TRY..ENDTRY block (note, however, that any dynamically nesting TRY..ENDTRY statements will still be active, UNPROTECTED only turns off the exception handling of the TRY..ENDTRY statement that immediately contains the UNPROTECTED clause). Since the BREAK statement in Program 1.5 appears inside the UNPROTECTED section, it can safely transfer control out of the TRY..ENDTRY block without “executing” the ENDTRY since the program has already restored the former exception environment.

Note that the UNPROTECTED keyword must appear in the TRY..ENDTRY statement immediately after the protected block. I.e., it must precede all EXCEPTION keywords.

If an exception occurs during the execution of a TRY..ENDTRY sequence, HLA automatically restores the execution environment. Therefore, you may execute a BREAK statement (or any other instruction that transfers control out of the TRY..ENDTRY block) within an EXCEPTION clause without having to do anything special.

Since the program restores the exception environment upon encountering an UNPROTECTED block or an EXCEPTION block, an exception that occurs within one of these areas immediately transfers control to the previous (dynamically nesting) active TRY..ENDTRY sequence. If there is no nesting TRY..ENDTRY sequence, the program aborts with an appropriate error message.

1.3.3 The ANYEXCEPTION Clause in a TRY..ENDTRY Statement

In a typical situation, you will use a TRY..ENDTRY statement with a set of EXCEPTION clauses that will handle all possible exceptions that can occur in the protected section of the TRY..ENDTRY sequence. Often, it is important to ensure that a TRY..ENDTRY statement handles all possible exceptions to prevent the program from prematurely aborting due to an unhandled exception. If you have written all the code in the protected section, you will know the exceptions it can raise so you can handle all possible exceptions. However, if you are calling a library routine (especially a third-party library routine), making a OS API call, or otherwise executing code that you have no control over, it may not be possible for you to anticipate all possible exceptions this code could raise (especially when considering past, present, and future versions of the code). If that code raises an exception for which you do not have an EXCEPTION clause, this could cause your program to fail. Fortunately, HLA’s TRY..ENDTRY statement provides the ANYEXCEPTION clause that will automatically trap any exception the existing EXCEPTION clauses do not handle.

The ANYEXCEPTION clause is similar to the EXCEPTION clause except it does not require an exception number parameter (since it handles any exception). If the ANYEXCEPTION clause appears in a TRY..ENDTRY statement with other EXCEPTION sections, the ANYEXCEPTION section must be the last exception handler in the TRY..ENDTRY statement. An ANYEXCEPTION section may be the only exception handler in a TRY..ENDTRY statement.

If an otherwise unhandled exception transfers control to an ANYEXCEPTION section, the EAX register will contain the exception number. Your code in the ANYEXCEPTION block can test this value to determine the cause of the exception.

1.3.4 Raising User-Defined Exceptions

Although you typically use the TRY..ENDTRY statement to catch exceptions that the hardware, the OS or the HLA Standard Library raises, it is also possible to create your own exceptions and process them via the TRY..ENDTRY statement. You accomplish this by assigning an unused exception number to your exception and raising this exception with the HLA RAISE statement.

The parameter you supply to the EXCEPTION statement is really nothing more than an unsigned integer value. The HLA “excepts.hhf” header file provides definitions for the standard HLA Standard Library and hardware/OS exception types. These names are nothing more than dword constants that have been given a descriptive name. HLA reserves the values zero through 1023 for HLA and HLA Standard Library exceptions; it also reserves all exception values greater than \$FFFF (65,535) for use by the operating system. The values in the range 1024 through 65,535 are available for user-defined exceptions.

To create a user-defined exception, you would generally begin by defining a descriptive symbolic name for the exception⁴. Then within your code you can use the RAISE statement to raise that exception. The following program provides a short example of some code that uses a user-defined exception to trap empty strings:

```

program userDefinedExceptions;
#include( "stdlib.hhf" );

    // Provide a descriptive name for the
    // user-defined exception.

const   EmptyString:dword := 1024;

    // readAString-
    //
    // This procedure reads a string from the user
    // and returns a pointer to that string in the
    // EAX register. It raises the "EmptyString"
    // exception if the string is empty.

procedure readAString;
begin readAString;

    stdin.a_gets();
    if( (type str.strRec [eax]).length == 0 ) then

        strfree( eax );
        raise( EmptyString );

    endif;

end readAString;

begin userDefinedExceptions;

    try

        stdout.put( "Enter a non-empty string: " );
        readAString();
        stdout.put
        (
            "You entered the string '",
            (type string eax),
            "'\n"
        );
        strfree( eax );

        exception( EmptyString )

        stdout.put( "You entered an empty string", nl );

    endtry;

end userDefinedExceptions;

```

4. Technically, you could use a literal numeric constant, e.g., EXCEPTION(1024), but this is extremely poor programming style.

Program 1.6 User-Defined Exceptions and the RAISE Statement

One important thing to notice in this example: the *readAString* procedure frees the string storage before raising the exception. It has to do this because the RAISE statement loads the EAX register with the exception number (1024 in this case), effectively obliterating the pointer to the string. Therefore, this code frees the storage before the exception and assumes that EAX does not contain a valid string pointer if an exception occurs.

In addition to raising exceptions you've defined, you can also use the RAISE statement to raise any exception. Therefore, if your code encounters an error converting some data, you could raise the *ex.ConversionError* exception to denote this condition. There is nothing sacred about the predefined exception values. Feel free to use their values as exceptions if they are descriptive of the error you need to handle.

1.3.5 Reraising Exceptions in a TRY..ENDTRY Statement

Once a program transfers control to an exception handling section, the exception is effectively dead. That is, after executing the associated EXCEPTION block, control transfers to the first statement after the ENDTRY and program execution continues as though an exception had not occurred. HLA assumes that the exception handler has taken care of the problem and it is okay to continue program execution after the ENDTRY statement. In some instances, this isn't an appropriate response.

Although falling through and executing the statements after the ENDTRY when an exception handler finishes is probably the most common response, another possibility is to reraise the exception at the end of the EXCEPTION sequence. This lets the current TRY..ENDTRY block accommodate the exception as best it can and then pass control to an enclosing TRY..ENDTRY statement to complete the exception handling process. To reraise an exception all you need do is execute a RAISE statement at the end of the exception handler. Although you would typically reraise the same exception, there is nothing preventing you from raising a different exception at the end of your exception handler. For example, after handling a user-defined exception you've defined, you might want to raise a different exception (e.g., *ex.MemoryAllocationFailure*) and let an enclosing TRY..ENDTRY statement finish handling your exception.

1.3.6 A List of the Predefined HLA Exceptions

Appendix G in this text provides a listing of the HLA exceptions and the situations in which the hardware, the operating system, or the HLA Standard Library raises these exceptions. The HLA Standard Library reference in Appendix F also lists the exceptions that each HLA Standard Library routine raises. You should skim over these appendices to familiarize yourself with the types of exceptions HLA raises and refer to these sections when calling Standard Library routines to ensure that you handle all the possible exceptions.

1.3.7 How to Handle Exceptions in Your Programs

When an exception occurs in a program there are four general ways to handle the exception: (1) correct the problem in the exception handler and restart the offending instruction (if this is possible), (2) report an error message and loop back to the offending code and re-execute the entire sequence, preferably with better input data that won't cause an exception, (3) report an error and reraise the exception (or raise a different exception) and leave it up to a dynamically nesting exception handler to deal with the problem, or (4) clean up the program's data as much as possible and abort program execution. The HLA run-time system only supports the last three options (i.e., it does not allow you to restart the offending instruction after some sort of correction), so we will ignore the first option in this text.

Reporting an error and looping back to repeat the offending code is an extremely common solution when the program raises an exception because of bad user input. The following program provides a typical example of this solution that forces a user to enter a valid unsigned integer value:

```
program repeatingBadCode;
#include( "stdlib.hhf" );

static
    u:      uns16;

begin repeatingBadCode;

    forever

        try
            // Protected block.  Read an unsigned integer
            // from the user and display that value if
            // there wasn't an error.

            stdout.put( "Enter an unsigned integer:" );
            stdin.get( u );

            // Clean up the exception and break out of
            // the forever loop if all went well.

            unprotected

                break;

            // If the user entered an illegal character,
            // print an appropriate error message.

            exception( ex.ConversionError )

                stdout.put( "Your input contained illegal characters" nl );

            // If the user entered a value outside the range
            // 0..65535 then print an error message.

            exception( ex.ValueOutOfRange )

                stdout.put( "The value was too large" nl );

        endtry;

        // If we get down here, it's because there was an exception.
        // Loop back and make the user reenter the value.

    endfor;

    // Only by executed the BREAK statement do we wind up down here.
    // That occurs if the user entered a value unsigned integer value.

    stdout.put( "You entered: ", u, nl );

end repeatingBadCode;
```

Program 1.7 Repeating Code via a Loop to Handle an Exception

Another way to handle an exception is to print an appropriate message (or take other corrective action) and then re-raise the exception or raise a different exception. This allows an enclosing exception handler to handle the exception. The big advantage to this scheme is that it minimizes the code you have to write to handle a given exception throughout your code (i.e., passing an exception on to a different handler that contains some complex code is much easier than replicating that complex code everywhere the exception can occur). However, this approach has its own problems. Primary among the problems is ensuring that there is some enclosing TRY..ENDTRY statement that will handle the exception for you. Of course, HLA automatically encloses your entire program in one big TRY..ENDTRY statement, but the default handler simply prints a short message and then stops your program. This is unacceptable behavior in a robust program. At the very least, you should supply your own exception handler that surrounds the code in your main program that attempts to clean up the system before shutting it down if an otherwise unhandled exception comes along. Generally, however, you would like to handle the exception without shutting down the program. Ensuring that this always occurs if you re-raise an exception can be difficult.

The last alternative, and certainly the least desirable of the four, is to clean up the system as much as possible and terminate program execution. Cleaning up the system includes writing transient data in memory to files, closing the files, releasing system resources (i.e., peripheral devices and memory), and, in general, preserving as much of the user's work as possible before quitting the program. Although you would like to continue program execution whenever an exception occurs, sometimes it is impossible to recover from an invalid operation (either on the part of the user or because of an error in your program) and continue execution. In such a situation you want to shut the program down as gracefully as possible so the user can restart the program and continue where they left off.

Of course, the absolute worst thing you can do is allow the program to terminate without attempting to save user data or release system resources. The user of your application will not have kind things to say about your program if they use it for three or four hours and the program aborts and loses all the data they've entered (requiring them to spend another three or four hours entering that data). Telling the user to "save your data often" is not a good substitute for automatically saving their data when an exception occurs.

The easiest way to handle an arbitrary (and unexpected) exception is to place a TRY..ANYEXCEPTION..ENDTRY statement around your main program. If the program raises an exception, you should save the value in EAX upon entry into the ANYEXCEPTION section (this contains the exception number) and then save any important data and release any resources your program is using. After this, you can re-raise the exception and let the default handler print the error message and terminate your program.

1.3.8 Registers and the TRY..ENDTRY Statement

The TRY..ENDTRY statement preserves about 16 bytes of data on the stack whenever you enter a TRY..ENDTRY statement. Upon leaving the TRY..ENDTRY block (or hitting the UNPROTECTED clause), the program restores the exception environment by popping this data off the stack. As long as no exception occurs, the TRY..ENDTRY statement does not affect the values of any registers upon entry to or upon exit from the TRY..ENDTRY statement. However, this claim is not true if an exception occurs during the execution of the protected statements.

Upon entry into an EXCEPTION clause the EAX register contains the exception number and the state of all other general purpose registers is undefined. Since the operating system may have raised the exception in response to a hardware error (and, therefore, has played around with the registers), you can't even assume that the general purpose registers contain whatever values they happened to contain at the point of the exception. The underlying code that HLA generates for exceptions is subject to change in different versions of the compiler, and certainly it changes across operating systems, so it is never a good idea to experimentally determine what values registers contain in an exception handler and depend upon those values in your code.

Since entry into an exception handler can scramble all the register values, you must ensure that you reload important registers if the code following your `ENDTRY` clause assumes that the registers contain valid values (i.e., values set in the protected section or values set prior to executing the `TRY..ENDTRY` statement). Failure to do so will introduce some nasty defects into your program (and these defects may be very intermittent and difficult to detect since exceptions rarely occur and may not always destroy the value in a particular register). The following code fragment provides a typical example of this problem and its solution:

```
static
    array: uns32[8];
    .
    .
    .
for( mov( 0, ebx ); ebx < 8; inc( ebx ) ) do

    push( ebx ); // Must preserve EBX in case there is an exception.
    forever
        try

            stdin.geti32();
            unprotected break;

        exception( ex.ConversionError )

            stdout.put( "Illegal input, please reenter value: " );

        endtry;
    endfor;
    pop( ebx ); // Restore EBX's value.
    mov( eax, array[ ebx*4 ] );

endfor;
```

Because the HLA exception handling mechanism messes with the registers, and because exception handling is a relatively inefficient process, you should never use the `TRY..ENDTRY` statement as a generic control structure (e.g., using it to simulate a `SWITCH/CASE` statement by raising an integer exception value and using the `EXCEPTION` clauses as the cases to process). Doing so will have a very negative impact on the performance of your program and may introduce subtle defects because exceptions scramble the registers.

For proper operation, the `TRY..ENDTRY` statement assumes that you only use the `EBP` register to point at *activation records* (the chapter on intermediate procedures discusses activation records). By default, HLA programs automatically use `EBP` for this purpose; as long as you do not modify the value in `EBP`, your programs will automatically use `EBP` to maintain a pointer to the current activation record. If you attempt to use the `EBP` register as a general purpose register to hold values and compute arithmetic results, HLA's exception handling capabilities will no longer function properly (not to mention you will lose access to procedure parameters and variables in the `VAR` section). Therefore, you should never use the `EBP` register as a general purpose register. Of course, this same discussion applies to the `ESP` register.

1.4 BEGIN..EXIT..EXITIF..END

HLA provides a structured `GOTO` via the `EXIT` and `EXITIF` statements. The `EXIT` and `EXITIF` statements let you exit a block of statements surrounded by a `BEGIN..END` pair. These statements behave much like the `BREAK` and `BREAKIF` statements (that let you exit from an enclosing loop) except, of course, they jump out of a `BEGIN..END` block rather than a loop. The `EXIT` and `EXITIF` statements are *structured gotos*

because they do not let you jump to an arbitrary point in the code, they only let you exit from a block delimited by the BEGIN..END pair.

The EXIT and EXITIF statements take the following forms:

```
exit identifier;
exitif( boolean_expression) identifier;
```

The *identifier* component at the end of these two statements must match the identifier following the BEGIN and END keywords (e.g., a procedure or program name). The EXIT statement immediately transfers control to the “end identifier;” clause. The EXITIF statement evaluates the boolean expression immediately following the EXITIF reserved word and transfers control to the specified END clause only if the expression evaluates true. If the boolean expression evaluates false, the control transfers to the first statement following the EXITIF statement.

If you specify the name of a procedure as the identifier for an EXIT statement, the program will return from the procedure upon encountering the EXIT statement⁵. *Note that the EXIT statement does not automatically restore any registers you pushed on the stack upon entry into the procedure.* If you need to pop data off the stack, you must do this before executing the EXIT statement.

If you specify the name of your main program as the identifier following the EXIT (or EXITIF) statement, your program will terminate upon encountering the EXIT statement. With EXITIF, your program will only terminate if the boolean expression evaluates true. Note that your program will still terminate even if you execute the “exit MainPgmName;” statement within a procedure nested inside your main program. You do not have to execute the EXIT statement in the main program to terminate the main program.

HLA lets you place arbitrary BEGIN..END blocks within your program, they are not limited to surrounding your procedures or main program. The syntax for an arbitrary BEGIN..END block is

```
begin identifier;

    <statements>

end identifier;
```

The identifier following the END clause must match the identifier following the corresponding BEGIN statement. Naturally, you can nest BEGIN..END blocks, but the identifier following an END clause must match the identifier following the previous unmatched BEGIN clause.

One interesting use of the BEGIN..END block is that it lets you easily escape a deeply nested control structure without having to completely restructure the program. Typically, you would use this technique to exit a block of code on some special condition and the TRY..ENDTRY statement would be inappropriate (e.g., you might need to pass values in registers to the outside code, an EXCEPTION clause can’t guarantee register status). The following program demonstrates the use of the BEGIN..EXIT..END sequence to bail out of some deeply nested code.

```
program beginEndDemo;
#include( "stdlib.hhf" );

static
    m:uns8;
    d:uns8;
    y:uns16;

readonly
    DaysInMonth: uns8[13] :=
    [
```

5. This is true for the EXITIF statement as well, though, of course, the program will only exit the procedure if the boolean expression in the EXITIF statement evaluates true.

```

    0, // No month zero.
    31, // Jan
    28, // Feb is a special case, see the code.
    31, // Mar
    30, // Apr
    31, // May
    30, // Jun
    31, // Jul
    31, // Aug
    30, // Sep
    31, // Oct
    30, // Nov
    31 // Dec
];

begin beginEndDemo;

    forever

        try

            stdout.put( "Enter month, day, year: " );
            stdin.get( m, d, y );

            unprotected

                break;

            exception( ex.ValueOutOfRange )

                stdout.put( "Value out of range, please reenter", nl );

            exception( ex.ConversionError )

                stdout.put( "Illegal character in value, please reenter", nl );

        endtry;

    endfor;
begin goodDate;

    mov( y, bx );
    movzx( m, eax );
    mov( d, dl );

    // Verify that the year is legal
    // (this program allows years 2000..2099.)

    if( bx in 2000..2099 ) then

        // Verify that the month is legal

        if( al in 1..12 ) then

            // Quick check to make sure the
            // day is half-way reasonable.

            if( dl <> 0 ) then

                // To verify that the day is legal,
                // we have to handle Feb specially

```

```

// since this could be a leap year.

if( al = 2 ) then

    // If this is a leap year, subtract
    // one from the day value (to convert
    // Feb 29 to Feb 28) so that the
    // last day of Feb in a leap year
    // will pass muster. (This could
    // set dl to zero if the date is
    // Feb 1 in a leap year, but we've
    // already handled dl=0 above, so
    // we don't have to worry about this
    // anymore.)

    date.isLeapYear( bx );
    sub( al, dl );

endif;

// Verify that the number of days in the month
// is valid.

exitif( dl <= DaysInMonth[ eax ] ) goodDate;

endif;

endif;

endif;
stdout.put( "You did not enter a valid date!", nl );

end goodDate;

end beginEndDemo;

```

Program 1.8 Demonstration of BEGIN..EXIT..END Sequence

In this program, the “begin goodDate;” statement surrounds a section of code that checks to see if the date entered by a user is a valid date in the 100 years from 2000..2099. If the user enters an invalid date, it prints an appropriate error message, otherwise the program quits without further user interaction. While you could restructure this code to avoid the use of the EXITIF statement, the resulting code would probably be more difficult to understand. The nice thing about the design of the code is that it uses refinement to test for a legal date. That is, it tests to see if one component is legal, then tests to see if the next component of the date is legal, and works downward in this fashion. In the middle of the tests, this code determines that the date is legal. To restructure this code to work without the EXITIF (or other GOTO type instruction) would require using negative logic at each step (asking is this component *not* a legal date). That logic would be quite a bit more complex and much more difficult to read, understand, and verify. Hence, this example is preferable even if it contains a structured form of the GOTO statement.

Because the BEGIN..END statement uses a label, that the EXIT and EXITIF statements specify, you can nest BEGIN..END blocks and break out of several nested blocks with a single EXIT/EXITIF statement. Figure 1.1 provides a schematic of this capability.

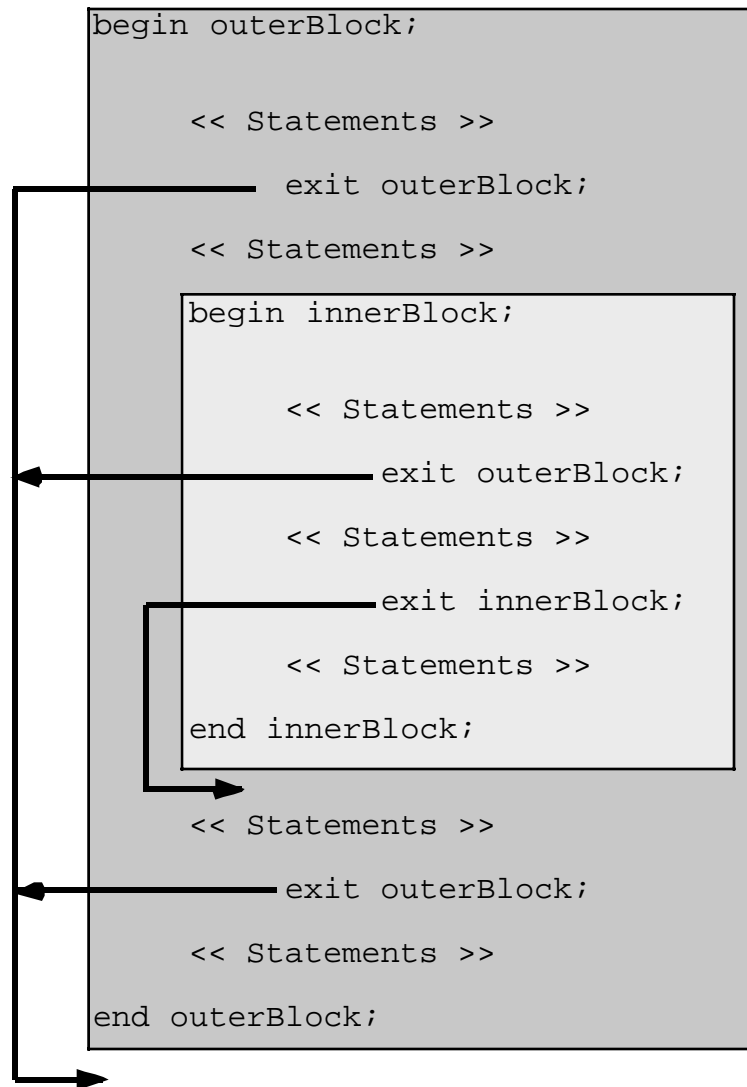


Figure 1.1 Nesting BEGIN..END Blocks

This ability to break out of nested BEGIN..END blocks is very powerful. Contrast this with the BREAK and BREAKIF statements that only let you exit the loop that immediately contains the BREAK or BREAKIF. Of course, if you need to exit out of multiple nested loops you won't be able to use the BREAK/BREAKIF statement to achieve this, but you can surround your loops with a BEGIN..END sequence and use the EXIT/EXITIF statement to leave those loops. The following program demonstrates how this could work, using the EXITIF statement to break out of two nested loops.

```

program brkNestedLoops;
#include( "stdlib.hhf" )

static
  i:int32;

begin brkNestedLoops;

```



```

// DL contains the last value to print on each line.

for( mov(0, dl ); dl <= 7; inc( dl ) ) do

    begin middleLoop;

        // DH ranges over the values to print.

        for( mov( 0, dh ); dh <= 7; inc( dh ) ) do

            // "i" specifies the field width
            // when printing DH, it also specifies
            // the maximum number of times to print DH.

            for( mov( 2, i ); i <= 4; inc( i ) ) do

                // Break out of both inner loops
                // when DH becomes equal to DL.

                exitif( dh >= dl ) middleLoop;

                // The following statement prints
                // a triangular shaped object composed
                // of the values that DH goes through.

                stdout.puti8Size( dh, i, '.' );

            endfor;

        endfor;

    endfor;

end brkNestedLoops;

```

Program 1.9 Breaking Out of Nested Loops Using EXIT/EXITIF

1.5 CONTINUE..CONTINUEIF

The CONTINUE and CONTINUEIF statements are very similar to the BREAK and BREAKIF statements insofar as they affect control flow within a loop. The CONTINUE statement immediately transfers control to the point in the loop where the current iteration completes and the next iteration begins. The CONTINUEIF statement first checks a boolean expression and transfers control if the expression evaluates false.

The phrase “where the current iteration completes and the next iteration begins” has a different meaning for nearly every loop in HLA. For the WHILE..ENDWHILE loop, control transfers to the top of the loop at the start of the test for loop termination. For the FOREVER..ENDFOR loop, control transfers to the top of the loop (no test for loop termination). For the FOR..ENDFOR loop, control transfers to the bottom of the loop where the increment operation occurs (i.e., to execute the third component of the FOR loop). For the REPEAT..UNTIL loop, control transfers to the bottom of the loop, just before the test for loop termination. The following diagrams show how the CONTINUE statement behaves.

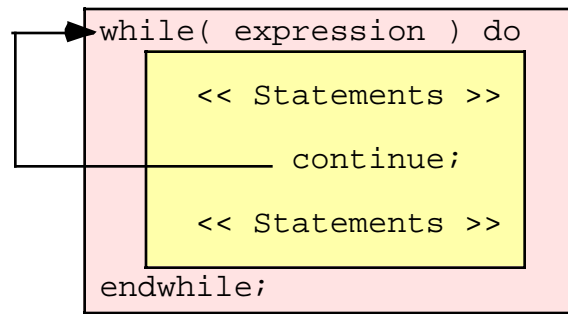


Figure 1.2 Behavior of CONTINUE in a WHILE Loop

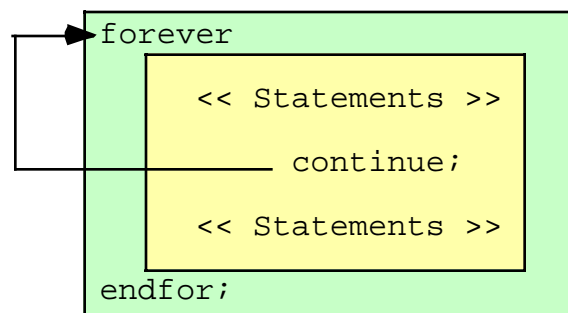


Figure 1.3 Behavior of CONTINUE in a FOREVER Loop

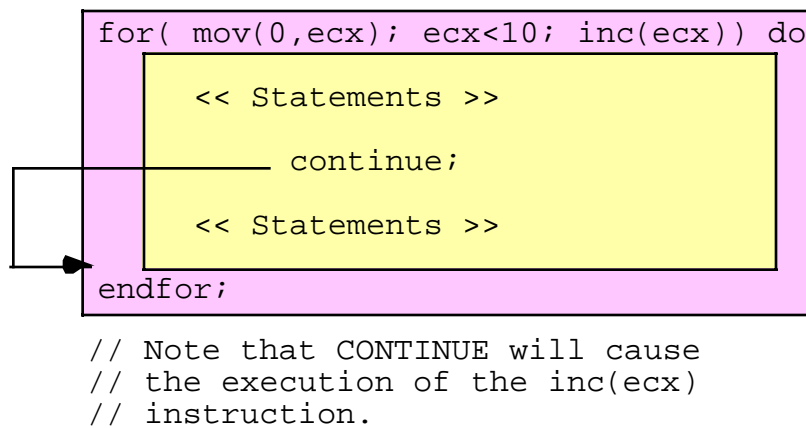


Figure 1.4 Behavior of CONTINUE in a FOR Loop

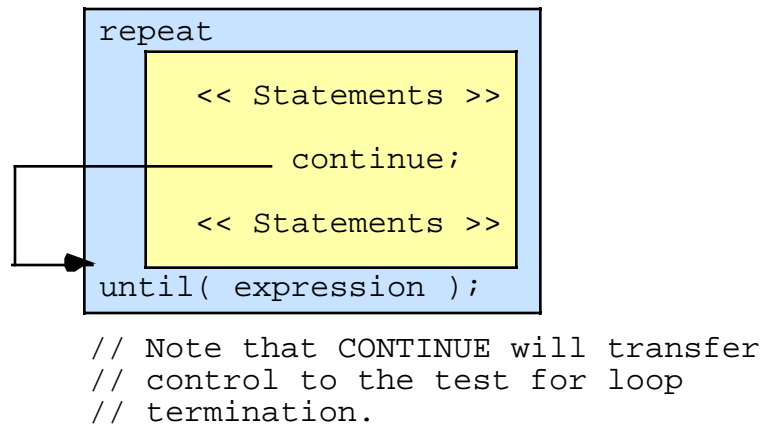


Figure 1.5 Behavior of CONTINUE in a REPEAT..UNTIL Loop

It turns out that CONTINUE is rarely needed in common programs. Most of the time an IF..ENDIF statement provides the same functionality as CONTINUE (or CONTINUEIF) while being much more readable. Nevertheless, there are a few instances you will encounter where the CONTINUE or CONTINUEIF statements provide exactly what you need. However, if you find yourself using the CONTINUE or CONTINUEIF statements on a frequent basis, you should probably reconsider the logic in your programs.

1.6 SWITCH..CASE..DEFAULT..ENDSWITCH

The HLA language does not provide a selection statement similar to SWITCH in C/C++ or CASE in Pascal/Delphi. This omission was intentional; by leaving the SWITCH statement out of the language it is possible to demonstrate how to extend the HLA language by adding new control structures. In the chapters on Macros and the HLA Compile-time Language, this text will demonstrate how you can add your own statements, like SWITCH, to the HLA language. In the meantime, although HLA does not provide a SWITCH statement, the HLA Standard Library provides a macro that provides this capability for you. If you include the “hll.hhf” header file (which “stdlib.hhf” automatically includes for you), then you can use the SWITCH statement exactly as though it were a part of the HLA language.

The HLA Standard Library SWITCH statement has the following syntax:

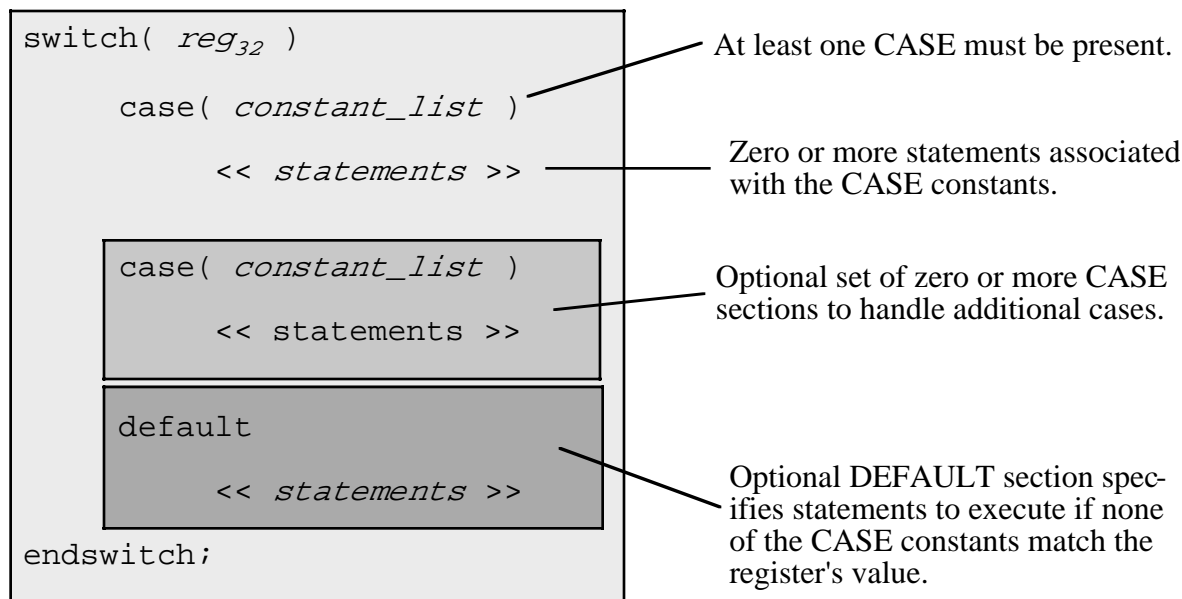


Figure 1.6 Syntax for the SWITCH..CASE..DEFAULT..ENDSWITCH Statement

Like most HLA high level language statements, there are several restrictions on the SWITCH statement. First of all, the SWITCH clause does not allow a general expression as the selection value. The SWITCH clause will only allow a value in a 32-bit general purpose register. In general you should only use EAX, EBX, ECX, EDX, ESI, and EDI since EBP and ESP are reserved for special purposes.

The second restriction is that the HLA SWITCH statement supports a maximum of 256 different case values. Few SWITCH statements use anywhere near this number, so this shouldn't prove to be a problem. Note that each CASE in Figure 1.6 allows a constant list. This could be a single unsigned integer value or a comma separated list of values, e.g.,

```

case( 10 )
-or-
case( 5, 6, 8 )

```

Each value in the list of constants counts as one case constant towards the maximum of 256 possible constants. So the second CASE clause above contributes three constants towards the total maximum of 256 constants.

Another restriction on the HLA SWITCH statement is that the difference between the largest and smallest values in the case list must be 1,024. Therefore, you cannot have CASEs (in the same SWITCH statement) with values like 1, 10, 100, 1,000, and 10,000 since the difference between the smallest and largest values, 9999, exceeds 1,024.

The DEFAULT section, if it appears in a SWITCH statement, must be the last section in the SWITCH statement. If no DEFAULT section is present and the value in the 32-bit register does not match one of the CASE constants, then control transfers to the first statement following the ENDSWITCH clause.

Here is a typical example of a SWITCH..ENDSWITCH statement:

```

switch( eax )

    case( 1 )

        stdout.put( "Selection #1:" nl );
        << Code for case #1 >>

```

```

case( 2, 3 )

    stdout.put( "Selections (2) and (3):" nl );
    << code for cases 2 & 3 >>

case( 5,6,7,8,9 )

    stdout.put( "Selections (5)..(9)" nl );
    << code for cases 5..9 >

default

    stdout.put( "Selection outside range 1..9" nl );
    << default case code >>

endswitch;

```

The SWITCH statement in a program lets your code choose one of several different code paths depending upon the value of the case selection variable. Among other things, the SWITCH statement is ideal for processing user input that selects a menu item and executes different code depending on the user's selection.

Later in this volume you will see how to implement a SWITCH statement using low-level machine instructions. Once you see the implementation you will understand the reasons behind these limitations in the SWITCH statement. You will also see why the CASE constants must be constants and not variables or registers.

1.7 Putting It All Together

This chapter completes the discussion of the high level control structures built into the HLA language or provided by the HLA Standard Library (i.e., SWITCH). First, this chapter gave a complete discussion of the TRY..ENDTRY and RAISE statements. Although Volume One provided a brief discussion of exception handling and the TRY..ENDTRY statement, this particular statement is too complex to fully describe earlier in this text. This chapter completes the discussions of this important statement and suggests ways to use it in your programs that will help make them more robust.

After discussing TRY..ENDTRY and RAISE, this chapter discusses the EXIT and EXITIF statements and describes how to use them to prematurely exit a procedure or the program. This chapter also discusses the BEGIN..END block and describes how to use the EXIT and EXITIF statements to exit such a block. These statements provide a structured GOTO (JMP) in the HLA language.

Although you will not use them as frequently as the BREAK and BREAKIF statements, the CONTINUE and CONTINUEIF statements are helpful once in a while for jumping over the remainder of a loop body and starting the next loop iteration. This chapter discusses the syntax of these statements and warns against overusing them.

This chapter concludes with a discussion of the SWITCH/CASE/DEFAULT/ENDCASE statement. This statement isn't actually a part of the HLA language - instead it is provided by the HLA Standard Library as an example of how you can extend the language. If you would like details on extending the HLA language yourself, see the chapter on "Domain Specific Languages."

Low-Level Control Structures

Chapter Two

2.1 Chapter Overview

This chapter discusses “pure” assembly language control statements. The last section of this chapter discusses *hybrid* control structures that combine the features of HLA’s high level control statements with the 80x86 control instructions.

2.2 Low Level Control Structures

Until now, most of the control structures you’ve seen and have used in your programs have been very similar to the control structures found in high level languages like Pascal, C++, and Ada. While these control structures make learning assembly language easy they are not true assembly language statements. Instead, the HLA compiler translates these control structures into a sequence of “pure” machine instructions that achieve the same result as the high level control structures. This text uses the high level control structures to avoid your having to learn too much all at once. Now, however, it’s time to put aside these high level language control structures and learn how to write your programs in *real* assembly language, using low-level control structures.

2.3 Statement Labels

HLA low level control structures make extensive use of *labels* within your code. A low level control structure usually transfers control from one point in your program to another point in your program. You typically specify the destination of such a transfer using a statement label. A statement label consists of a valid (unique) HLA identifier and a colon, e.g.,

```
aLabel:
```

Of course, like procedure, variable, and constant identifiers, you should attempt to choose descriptive and meaningful names for your labels. The identifier “aLabel” is hardly descriptive or meaningful.

Statement labels have one important attribute that differentiates them from most other identifiers in HLA: you don’t have to declare a label before you use it. This is important, because low-level control structures must often transfer control to a label at some point later in the code, therefore the label may not be defined at the point you reference it.

You can do three things with labels: transfer control to a label via a jump (*goto*) instruction, call a label via the *CALL* instruction, and you can take the address of a label. There is very little else you can directly do with a label (of course, there is very little else you would want to do with a label, so this is hardly a restriction). The following program demonstrates two ways to take the address of a label in your program and print out the address (using the *LEA* instruction and using the “&” address-of operator):

```
program labelDemo;
#include( "stdlib.hhf" );

begin labelDemo;

    lbl1:

        lea( ebx, lbl1 );
        lea( eax, lbl2 );
        stdout.put( "&lbl1=$", ebx, " &lbl2=", eax, nl );
```

```

    lbl2:

end labelDemo;

```

Program 2.1 Displaying the Address of Statement Labels in a Program

HLA also allows you to initialize dword variables with the addresses of statement labels. However, there are some restrictions on labels that appear in the initialization portions of variable declarations. The most important restriction is that you must define the statement label at the same lex level as the variable declaration. That is, if you reference a statement label in the initialization section of a variable declaration appearing in the main program, the statement label must also be in the main program. Conversely, if you take the address of a statement label in a local variable declaration, that symbol must appear in the same procedure as the local variable. The following program demonstrates the use of statement labels in variable initialization:

```

program labelArrays;
#include( "stdlib.hhf" );

static
    labels:dword[2] := [ &lbl1, &lbl2 ];

    procedure hasLabels;
    static
        stmtLbls: dword[2] := [ &label1, &label2 ];

    begin hasLabels;

        label1:

            stdout.put
            (
                "stmtLbls[0]= $", stmtLbls[0], nl,
                "stmtLbls[1]= $", stmtLbls[4], nl
            );

        label2:

    end hasLabels;

begin labelArrays;

    hasLabels();
    lbl1:

        stdout.put( "labels[0]= $", labels[0], " labels[1]=", labels[4], nl );

    lbl2:

end labelArrays;

```

Program 2.2 Initializing DWORD Variables with the Address of Statement Labels

Once in a really great while, you'll need to refer to a label that is not within the current procedure. The need for this is sufficiently rare that this text will not describe all the details. However, you can look up the details on HLA's LABEL declaration section in the HLA documentation should the need to do this ever arise.

2.4 Unconditional Transfer of Control (JMP)

The JMP (jump) instruction unconditionally transfers control to another point in the program. There are three forms of this instruction: a direct jump, and two indirect jumps. These instructions take one of the following three forms:

```

jmp label;
jmp( reg32 );
jmp( mem32 );

```

For the first (direct) jump above, you normally specify the target address using a statement label (see the previous section for a discussion of statement labels). The statement label is usually on the same line as an executable machine instruction or appears by itself on a line preceding an executable machine instruction. The direct jump instruction is the most commonly used of these three forms. It is completely equivalent to a GOTO statement in a high level language¹. Example:

```

<< statements >>
  jmp laterInPgm;
  .
  .
  .
laterInPgm:
  << statements >>

```

The second form of the JMP instruction above, "jmp(reg32);", is a *register indirect* jump instruction. This instruction transfers control to the instruction whose address appears in the specified 32-bit general purpose register. To use this form of the JMP instruction you must load the specified register with the address of some machine instruction prior to the execution of the JMP. You could use this instruction to implement a state machine (see "State Machines and Indirect Jumps" on page 784) by loading a register with the address of some label at various points throughout your program; then, arriving along different paths, a point in the program can determine what path it arrived upon by executing the indirect jump. The following short sample program demonstrates how you could use the JMP in this manner:

```

program regIndJmp;
#include( "stdlib.hhf" );

static
  i:int32;

begin regIndJmp;

  // Read an integer from the user and set EBX to
  // denote the success or failure of the input.

  try

    stdout.put( "Enter an integer value between 1 and 10: " );
    stdin.get( i );
    mov( i, eax );

```

1. Unlike high level languages, where your instructors usually forbid you to use GOTO statements, you will find that the use of the JMP instruction in assembly language is absolutely essential.

```

    if( eax in 1..10 ) then
        mov( &GoodInput, ebx );
    else
        mov( &valRange, ebx );
    endif;

exception( ex.ConversionError )

    mov( &convError, ebx );

exception( ex.ValueOutOfRange )

    mov( &valRange, ebx );

endtry;

// Okay, transfer control to the appropriate
// section of the program that deals with
// the input.

jmp( ebx );

valRange:
    stdout.put( "You entered a value outside the range 1..10" nl );
    jmp Done;

convError:
    stdout.put( "Your input contained illegal characters" nl );
    jmp Done;

GoodInput:
    stdout.put( "You entered the value ", i, nl );

Done:

end regIndJump;

```

Program 2.3 Using Register Indirect JMP Instructions

The third form of the JMP instruction is a memory indirect JMP. This form of the JMP instruction fetches a dword value from the specified memory location and transfers control to the instruction at the address specified by the contents of the memory location. This is similar to the register indirect JMP except the address appears in a memory location rather than in a register. The following program demonstrates a rather trivial use of this form of the JMP instruction:

```

program memIndJump;
#include( "stdlib.hhf" );

static
    LabelPtr:dword := &stmtLabel;

```

```

begin memIndJump;

    stdout.put( "Before the JMP instruction" nl );
    jmp( LabelPtr );

    stdout.put( "This should not execute" nl );

stmtLabel:

    stdout.put( "After the LabelPtr label in the program" nl );

end memIndJump;

```

Program 2.4 Using Memory Indirect JMP Instructions

Warning: unlike the HLA high level control structures, the low-level JMP instructions can get you into a lot of trouble. In particular, if you do not initialize a register with the address of a valid instruction and you jump indirect through that register, the results are undefined (though this will usually cause a general protection fault). Similarly, if you do not initialize a dword variable with the address of a legal instruction, jumping indirect through that memory location will probably crash your program.

2.5 The Conditional Jump Instructions

Although the JMP instruction provides transfer of control, it does not allow you to make any serious decisions. The 80x86's conditional jump instructions handle this task. The conditional jump instructions are the basic tool for creating loops and other conditionally executable statements like the IF..ENDIF statement.

The conditional jumps test one or more flags in the flags register to see if they match some particular pattern (just like the SETcc instructions). If the flag settings match the instruction control transfers to the target location. If the match fails, the CPU ignores the conditional jump and execution continues with the next instruction. Some conditional jump instructions simply test the setting of the sign, carry, overflow, and zero flags. For example, after the execution of a SHL instruction, you could test the carry flag to determine if the SHL shifted a one out of the H.O. bit of its operand. Likewise, you could test the zero flag after a TEST instruction to see if any specified bits were one. Most of the time, however, you will probably execute a conditional jump after a CMP instruction. The CMP instruction sets the flags so that you can test for less than, greater than, equality, etc.

The conditional JMP instructions take the following form:

```
Jcc label;
```

The "cc" in *Jcc* indicates that you must substitute some character sequence that specifies the type of condition to test. These are the same characters the SETcc instruction uses. For example, "JS" stands for jump if the sign flag is set." A typical JS instruction looks like this

```
js ValueIsNegative;
```

In this example, the JS instruction transfers control to the *ValueIsNegative* statement label if the sign flag is currently set; control falls through to the next instruction following the JS instruction if the sign flag is clear.

Unlike the unconditional JMP instruction, the conditional jump instructions do not provide an indirect form. The only form they allow is a branch to a statement label in your program. Conditional jump instructions have a restriction that the target label must be within 32,768 bytes of the jump instruction. However, since this generally corresponds to somewhere between 8,000 and 32,000 machine instructions, it is unlikely you will ever encounter this restriction.

Note: Intel's documentation defines various synonyms or instruction aliases for many conditional jump instructions. The following tables list all the aliases for a particular instruction. These tables also list out the opposite branches. You'll soon see the purpose of the opposite branches.

Table 1: Jcc Instructions That Test Flags

Instruction	Description	Condition	Aliases	Opposite
JC	Jump if carry	Carry = 1	JB, JNAE	JNC
JNC	Jump if no carry	Carry = 0	JNB, JAE	JC
JZ	Jump if zero	Zero = 1	JE	JNZ
JNZ	Jump if not zero	Zero = 0	JNE	JZ
JS	Jump if sign	Sign = 1		JNS
JNS	Jump if no sign	Sign = 0		JS
JO	Jump if overflow	Ovrflw=1		JNO
JNO	Jump if no Ovrflw	Ovrflw=0		JO
JP	Jump if parity	Parity = 1	JPE	JNP
JPE	Jump if parity even	Parity = 1	JP	JPO
JNP	Jump if no parity	Parity = 0	JPO	JP
JPO	Jump if parity odd	Parity = 0	JNP	JPE

Table 2: Jcc Instructions for Unsigned Comparisons

Instruction	Description	Condition	Aliases	Opposites
JA	Jump if above (>)	Carry=0, Zero=0	JNBE	JNA
JNBE	Jump if not below or equal (not <=)	Carry=0, Zero=0	JA	JBE
JAE	Jump if above or equal (>=)	Carry = 0	JNC, JNB	JNAE
JNB	Jump if not below (not <)	Carry = 0	JNC, JAE	JB
JB	Jump if below (<)	Carry = 1	JC, JNAE	JNB
JNAE	Jump if not above or equal (not >=)	Carry = 1	JC, JB	JAE
JBE	Jump if below or equal (<=)	Carry = 1 or Zero = 1	JNA	JNBE
JNA	Jump if not above (not >)	Carry = 1 or Zero = 1	JBE	JA
JE	Jump if equal (=)	Zero = 1	JZ	JNE
JNE	Jump if not equal (≠)	Zero = 0	JNZ	JE

Table 3: Jcc Instructions for Signed Comparisons

Instruction	Description	Condition	Aliases	Opposite
JG	Jump if greater (>)	Sign = Ovrflw or Zero=0	JNLE	JNG
JNLE	Jump if not less than or equal (not <=)	Sign = Ovrflw or Zero=0	JG	JLE
JGE	Jump if greater than or equal (>=)	Sign = Ovrflw	JNL	JNGE
JNL	Jump if not less than (not <)	Sign = Ovrflw	JGE	JL
JL	Jump if less than (<)	Sign ≠ Ovrflw	JNGE	JNL
JNGE	Jump if not greater or equal (not >=)	Sign ≠ Ovrflw	JL	JGE
JLE	Jump if less than or equal (<=)	Sign ≠ Ovrflw or Zero = 1	JNG	JNLE
JNG	Jump if not greater than (not >)	Sign ≠ Ovrflw or Zero = 1	JLE	JG
JE	Jump if equal (=)	Zero = 1	JZ	JNE
JNE	Jump if not equal (≠)	Zero = 0	JNZ	JE

One brief comment about the “opposites” column is in order. In many instances you will need to be able to generate the opposite of a specific branch instructions (lots of examples of this appear throughout the remainder of this chapter). With only two exceptions, a very simple rule completely describes how to generate an opposite branch:

- If the second letter of the *Jcc* instruction is *not* an “n”, insert an “n” after the “j”. E.g., JE becomes JNE and JL becomes JNL.
- If the second letter of the *Jcc* instruction *is* an “n”, then remove that “n” from the instruction. E.g., JNG becomes JG and JNE becomes JE.

The two exceptions to this rule are JPE (jump if parity is even) and JPO (jump if parity is odd). These exceptions cause few problems because (a) you’ll hardly ever need to test the parity flag, and (b) you can use the aliases JP and JNP synonyms for JPE and JPO. The “N/No N” rule applies to JP and JNP.

Though you *know* that JGE is the opposite of JL, get in the habit of using JNL rather than JGE as the opposite jump instruction for JL. It’s too easy in an important situation to start thinking “greater is the opposite of less” and substitute JG instead. You can avoid this confusion by always using the “N/No N” rule.

The 80x86 conditional jump instruction give you the ability to split program flow into one of two paths depending upon some logical condition. Suppose you want to increment the AX register if BX is equal to CX. You can accomplish this with the following code:

```
cmp( bx, cx );
jne SkipStmts;
```

```
    inc( ax );
SkipStmts:
```

The trick is to use the *opposite* branch to skip over the instructions you want to execute if the condition is true. Always use the “opposite branch (N/no N)” rule given earlier to select the opposite branch.

You can also use the conditional jump instructions to synthesize loops. For example, the following code sequence reads a sequence of characters from the user and stores each character in successive elements of an array until the user presses the Enter key (carriage return):

```
    mov( 0, edi );
RdLnLoop:
    stdin.getc();           // Read a character into the AL register.
    mov( al, Input[ edi ] ); // Store away the character
    inc( edi );            // Move on to the next character
    cmp( al, stdio.cr );   // See if the user pressed Enter
    jne RdLnLoop;
```

For more information concerning the use of the conditional jumps to synthesize IF statements, loops, and other control structures, see “Implementing Common Control Structures in Assembly Language” on page 759.

Like the SETcc instructions, the conditional jump instructions come in two basic categories – those that test specific processor flags (e.g., JZ, JC, JNO) and those that test some condition (less than, greater than, etc.). When testing a condition, the conditional jump instructions almost always follow a CMP instruction. The CMP instruction sets the flags so you can use a JA, JAE, JB, JBE, JE, or JNE instruction to test for unsigned less than, less than or equal, equality, inequality, greater than, or greater than or equal. Simultaneously, the CMP instruction sets the flags so you can also do a signed comparison using the JL, JLE, JE, JNE, JG, and JGE instructions.

The conditional jump instructions only test flags, they do not affect any of the 80x86 flags.

2.6 “Medium-Level” Control Structures: JT and JF

HLA provides two special conditional jump instructions: JT (jump if true) and JF (jump if false). These instructions take the following syntax:

```
    jt( boolean_expression ) target_label;
    jf( boolean_expression ) target_label;
```

The *boolean_expression* is the standard HLA boolean expression allowed by IF.ENDIF and other HLA high level language statements. These instructions evaluate the boolean expression and jump to the specified label if the expression evaluates true (JT) or false (JF).

These are not real 80x86 instructions. HLA compiles them into a sequence of one or more 80x86 machine instructions that achieve the same result. In general, you should not use these two instructions in your main code; they offer few benefits over using an IF.ENDIF statement and they are no more readable than the pure assembly language sequences they compile into. HLA provides these “medium-level” instructions so that you may create your own high level control structures using macros (see the chapters on Macros, the HLA Run-Time Language, and Domain Specific Languages for more details).

2.7 Implementing Common Control Structures in Assembly Language

Since a primary goal of this chapter is to teach you how to use the low-level machine instructions to implement decisions, loops, and other control constructs, it would be wise to show you how to simulate

these high level statements using “pure” assembly language. The following sections provide this information.

2.8 Introduction to Decisions

In its most basic form, a decision is some sort of branch within the code that switches between two possible execution paths based on some condition. Normally (though not always), conditional instruction sequences are implemented with the conditional jump instructions. Conditional instructions correspond to the IF..THEN..ENDIF statement in HLA:

```
if( expression ) then
    << statements >>
endif;
```

Assembly language, as usual, offers much more flexibility when dealing with conditional statements. Consider the following C/C++ statement:

```
if( ( ( x < y ) && ( z > t ) ) || ( a != b ) )
    stmt1;
```

A “brute force” approach to converting this statement into assembly language might produce:

```
mov( x, eax );
cmp( eax, y );
setl( bl );          // Store X<Y in bl.
mov( z, eax );
cmp( eax, t );
setg( bh );          // Store Z > T in bh.
and( bh, bl );      // Put (X<Y) && (Z>T) into bl.
mov( a, eax );
cmp( eax, b );
setne( bh );        // Store A != B into bh.
or( bh, bl );       // Put (X<Y) && (Z>T) || (A!=B) into bl
je SkipStmt1;       // Branch if result is false (OR sets Z-Flag if false).
```

<Code for stmt1 goes here>

SkipStmt1:

As you can see, it takes a considerable number of conditional statements just to process the expression in the example above. This roughly corresponds to the (equivalent) C/C++ statements:

```
bl = x < y;
bh = z > t;
bl = bl && bh;
bh = a != b;
bl = bl || bh;
if( bl )
    stmt1;
```

Now compare this with the following “improved” code:

```
mov( a, eax );
cmp( eax, b );
jne DoStmt;
mov( x, eax );
cmp( eax, y );
jnl SkipStmt;
mov( z, eax );
cmp( eax, t );
jng SkipStmt;
```



```
DoStmt:
    << Place code for Stmt1 here >>
SkipStmt:
```

Two things should be apparent from the code sequences above: first, a single conditional statement in C/C++ (or some other HLL) may require several conditional jumps in assembly language; second, organization of complex expressions in a conditional sequence can affect the efficiency of the code. Therefore, care should be exercised when dealing with conditional sequences in assembly language.

Conditional statements may be broken down into three basic categories: IF statements, SWITCH/CASE statements, and indirect jumps. The following sections will describe these program structures, how to use them, and how to write them in assembly language.

2.8.1 IF..THEN..ELSE Sequences

The most common conditional statement is the IF..THEN or IF..THEN..ELSE statement. These two statements take the form shown in Figure 2.1:

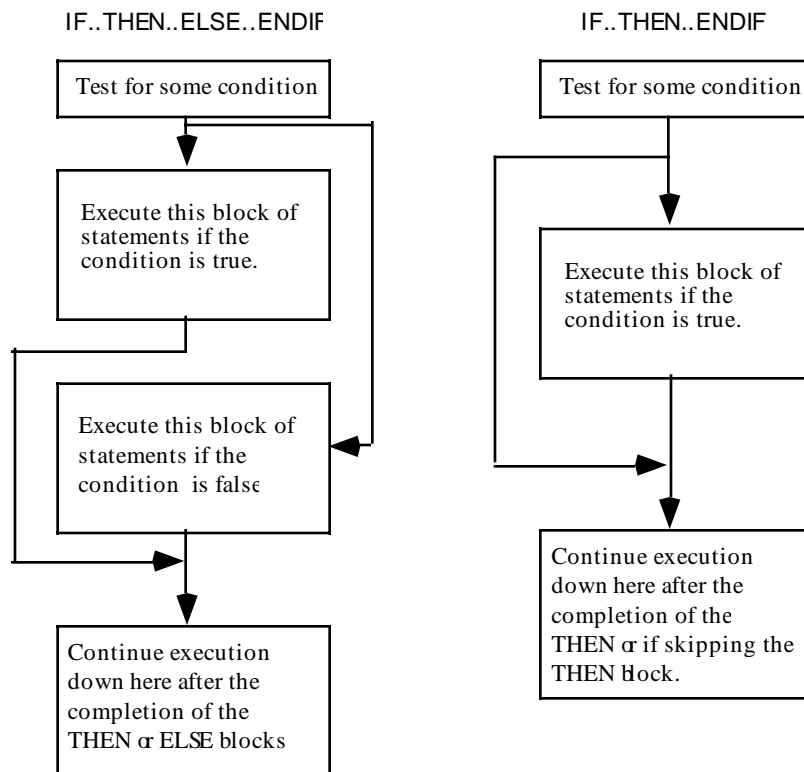


Figure 2.1 IF..THEN..ELSE..ENDIF and IF..ENDIF Statement Flow

The IF..ENDIF statement is just a special case of the IF..ELSE..ENDIF statement (with an empty ELSE block). Therefore, we'll only consider the more general IF..ELSE..ENDIF form. The basic implementation of an IF..THEN..ELSE statement in 80x86 assembly language looks something like this:

```

{Sequence of statements to test some condition}
    Jcc ElseCode
{Sequence of statements corresponding to the THEN block}

    jmp EndOfIF

```

```

ElseCode:
    {Sequence of statements corresponding to the ELSE block}
EndOfIF:

```

Note: *Jcc* represents some conditional jump instruction.

For example, to convert the C/C++ statement:

```

if( a == b )
    c = d;
else
    b = b + 1;

```

to assembly language, you could use the following 80x86 code:

```

    mov( a, eax );
    cmp( eax, b );
    jne ElsePart;
    mov( d, c );
    jmp EndOfIf;

ElseBlk:
    inc( b );

EndOfIf:

```

For simple expressions like “(a == b)” generating the proper code for an IF..ELSE..ENDIF statement is almost trivial. Should the expression become more complex, the associated assembly language code complexity increases as well. Consider the following C/C++ IF statement presented earlier:

```

if( (( x > y ) && ( z < t )) || ( a != b ) )
    c = d;

```

When processing complex IF statements such as this one, you’ll find the conversion task easier if you break this IF statement into a sequence of three different IF statements as follows:

```

if( a != b ) C = D;
else if( x > y )
    if( z < t )
        C = D;

```

This conversion comes from the following C/C++ equivalences:

```

if( expr1 && expr2 ) stmt;

```

is equivalent to

```

if( expr1 ) if( expr2 ) stmt;

```

and

```

if( expr1 || expr2 ) stmt;

```

is equivalent to

```

if( expr1 ) stmt;
else if( expr2 ) stmt;

```

In assembly language, the former IF statement becomes:

```

// if( (( x > y ) && ( z < t )) || ( a != b ) )
//     c = d;

```

```

        mov( a, eax );
        cmp( eax, b );
        jne DoIF;
        mov( x, eax );
        cmp( eax, y );
        jng EndOfIF;
        mov( z, eax );
        cmp( eax, t );
        jnl EndOfIf;
DoIf:
        mov( d, c );
EndOfIF:

```

As you can probably tell, the code necessary to test a condition can easily become more complex than the statements appearing in the ELSE and THEN blocks. Although it seems somewhat paradoxical that it may take more effort to test a condition than to act upon the results of that condition, it happens all the time. Therefore, you should be prepared for this situation.

Probably the biggest problem with the implementation of complex conditional statements in assembly language is trying to figure out what you've done after you've written the code. Probably the biggest advantage high level languages offer over assembly language is that expressions are much easier to read and comprehend in a high level language. This is one of the primary reasons HLA supports high level language control structures. The high level language version is self-documenting whereas assembly language tends to hide the true nature of the code. Therefore, well-written comments are an essential ingredient to assembly language implementations of `if..then..else` statements. An elegant implementation of the example above is:

```

// IF ((X > Y) && (Z < T)) OR (A != B) C = D;
// Implemented as:
// IF (A != B) THEN GOTO DoIF;

        mov( a, eax );
        cmp( eax, b );
        jne DoIF;

// if NOT (X > Y) THEN GOTO EndOfIF;

        mov( x, eax );
        cmp( eax, y );
        jng EndOfIF;

// IF NOT (Z < T) THEN GOTO EndOfIF ;

        mov( z, eax );
        cmp( eax, t );
        jnl EndOfIf;

// THEN Block:
DoIf:
        mov( d, c );

// End of IF statement
EndOfIF:

// if( (( x > y ) && ( z < t )) || ( a != b ) ) c = d;
// Test the boolean expression:

        mov( a, eax );
        cmp( eax, b );
        jne DoIF;
        mov( x, eax );

```

```

    cmp( eax, y );
    jng EndOfIF;
    mov( z, eax );
    cmp( eax, t );
    jnl EndOfIf;

; THEN Block:

DoIf:
    mov( d, c );

; End of IF statement

EndOfIF:

```

However, as your IF statements become complex, the density (and quality) of your comments become more and more important.

2.8.2 Translating HLA IF Statements into Pure Assembly Language

Translating HLA IF statements into pure assembly language is very easy. The boolean expressions that the HLA IF supports were specifically chosen to expand into a few simple machine instructions. The following paragraphs discuss the conversion of each supported boolean expression into pure machine code.

if(*flag_specification*) then <<stmts>> endif;

This form is, perhaps, the easiest HLA IF statement to convert. To execute the code immediately following the THEN keyword if a particular flag is set (or clear), all you need do is skip over the code if the flag is clear (set). This requires only a single conditional jump instruction for implementation as the following examples demonstrate:

```

// if( @c ) then inc( eax ); endif;

    jnc SkipTheInc;

    inc( eax );

SkipTheInc:

// if( @ns ) then neg( eax ); endif;

    js SkipTheNeg;

    neg( eax );

SkipTheNeg:

```

if(*register*) then <<stmts>> endif;

This form of the IF statement uses the TEST instruction to check the specified register for zero. If the register contains zero (false), then the program jumps around the statements after the THEN clause with a JZ instruction. Converting this statement to assembly language requires a TEST instruction and a JZ instruction as the following examples demonstrate:

```

// if( eax ) then mov( false, eax ); endif;

    test( eax, eax );
    jz DontSetFalse;

    mov( false, eax );

```

```

DontSetFalse:

// if( al ) then mov( bl, cl ); endif;

test( al, al );
jz noMove;

    mov( bl, cl );

noMove:

```

if(!register) then <<stmts>> endif;

This form of the IF statement uses the TEST instruction to check the specified register to see if it is zero. If the register is not zero (true), then the program jumps around the statements after the THEN clause with a JNZ instruction. Converting this statement to assembly language requires a TEST instruction and a JNZ instruction in a manner identical to the previous examples.

if(boolean_variable) then <<stmts>> endif;

This form of the IF statement compares the boolean variable against zero (false) and branches around the statements if the variable does contain false. HLA implements this statement by using the CMP instruction to compare the boolean variable to zero and then it uses a JZ (JE) instruction to jump around the statements if the variable is false. The following example demonstrates the conversion:

```

// if( bool ) then mov( 0, al ); endif;

cmp( bool, false );
je SkipZeroAL;

    mov( 0, al );

SkipZeroAL:

```

if(!boolean_variable) then <<stmts>> endif;

This form of the IF statement compares the boolean variable against zero (false) and branches around the statements if the variable contains true (i.e., the opposite condition of the previous example). HLA implements this statement by using the CMP instruction to compare the boolean variable to zero and then it uses a JNZ (JNE) instruction to jump around the statements if the variable contains true. The following example demonstrates the conversion:

```

// if( !bool ) then mov( 0, al ); endif;

cmp( bool, false );
jne SkipZeroAL;

    mov( 0, al );

SkipZeroAL:

```

if(mem_reg relop mem_reg_const) then <<stmts>> endif;

HLA translates this form of the IF statement into a CMP instruction and a conditional jump that skips over the statements on the opposite condition specified by the *relop* operator. The following table lists the correspondence between operators and conditional jump instructions:

Table 4: IF Statement Conditional Jump Instructions

Relop	Conditional jump instruction if both operands are unsigned	Conditional jump instruction if either operand is signed
= or ==	JNE	JNE
<> or !=	JE	JE
<	JNB	JNL
<=	JNBE	JNLE
>	JNA	JNG
>=	JNAE	JNGE

Here are a few examples of IF statements translated into pure assembly language that use expressions involving relational operators:

```
// if( al == ch ) then inc( cl ); endif;

    cmp( al, ch );
    jne SkipIncCL;

    inc( cl );

SkipIncCL:

// if( ch >= 'a' ) then and( $5f, ch ); endif;

    cmp( ch, 'a' );
    jnae NotLowerCase

    and( $5f, ch );

NotLowerCase:

// if( (type int32 eax) < -5 ) then mov( -5, eax ); endif;

    cmp( eax, -5 );
    jnl DontClipEAX;

    mov( -5, eax );

DontClipEAX:

// if( si <> di ) then inc( si ); endif;

    cmp( si, di );
    je DontIncSI;

    inc( si );

DontIncSI:
```

if(*reg/mem* in *LowConst..HiConst*) then <<stmts>> endif;

HLA translates this IF statement into a pair of CMP instructions and a pair of conditional jump instructions. It compares the register or memory location against the lower valued constant and jumps if less than (below) past the statements after the THEN clause. If the register or memory location's value is greater than or equal to *LowConst*, the code falls through to the second CMP/conditional jump pair that compares the register or memory location against the higher constant. If the value is greater than (above) this constant, a conditional jump instruction skips the statements in the THEN clause. Example:

```
// if( eax in 1000..125_000 ) then sub( 1000, eax ); endif;

    cmp( eax, 1000 );
    jb DontSub1000;
    cmp( eax, 125_000 );
    ja DontSub1000;

    sub( 1000, eax );

DontSub1000:

// if( i32 in -5..5 ) then add( 5, i32 ); endif;

    cmp( i32, -5 );
    jl NoAdd5;
    cmp( i32, 5 );
    jg NoAdd5;

    add( 5, i32 );

NoAdd5:
```

if(*reg/mem* not in *LowConst..HiConst*) then <<stmts>> endif;

This form of the HLA IF statement tests a register or memory location to see if its value is outside a specified range. The implementation is very similar to the code above exception you branch to the THEN clause if the value is less than the *LowConst* value or greater than the *HiConst* value and you branch over the code in the THEN clause if the value is within the range specified by the two constants. The following examples demonstrate how to do this conversion:

```
// if( eax not in 1000..125_000 ) then add( 1000, eax ); endif;

    cmp( eax, 1000 );
    jb Add1000;
    cmp( eax, 125_000 );
    jbe SkipAdd1000;

    Add1000:
    add( 1000, eax );

SkipAdd1000:

// if( i32 not in -5..5 ) then mov( 0, i32 ); endif;

    cmp( i32, -5 );
    jl Zeroi32;
    cmp( i32, 5 );
    jle SkipZero;
```

```

Zeroi32:
mov( 0, i32 );

SkipZero:

```

if(*reg₈* in *CSetVar/CSetConst*) then <<stmts>> endif;

This statement checks to see if the character in the specified eight-bit register is a member of the specified character set. HLA emits code that is similar to the following for instructions of this form:

```

movzx( reg8, eax );
bt( eax, CsetVar/CsetConst );
jnc SkipPastStmts;

<< stmts >>

SkipPastStmts:

```

This example modifies the EAX register (the code HLA generates does not, because it pushes and pops the register it uses). You can easily swap another register for EAX if you've got a value in EAX you need to preserve. In the worst case, if no registers are available, you can push EAX, execute the MOVZX and BT instructions, and then pop EAX's value from the stack. The following are some actual examples:

```

// if( al in { 'a'..'z' } ) then or( $20, al ); endif;

movzx( al, eax );
bt( eax, { 'a'..'z' } ); // See if we've got a lower case char.
jnc DontConvertCase;

or( $20, al ); // Convert to uppercase.

DontConvertCase:

// if( ch in { '0'..'9' } ) then and( $f, ch ); endif;

push( eax );
movzx( ch, eax );
bt( eax, { 'a'..'z' } ); // See if we've got a lower case char.
pop( eax );
jnc DontConvertNum;

and( $f, ch ); // Convert to binary form.

DontConvertNum:

```

2.8.3 Implementing Complex IF Statements Using Complete Boolean Evaluation

The previous section did not discuss how to translate boolean expressions involving conjunction (AND) or disjunction (OR) into assembly language. This section will begin that discussion. There are two different ways to convert complex boolean expressions involving conjunction and disjunction into assembly language: using complete boolean evaluation or short circuit evaluation. This section discusses complete boolean evaluation. The next section discusses short circuit boolean evaluation, which is the scheme that HLA uses when converting complex boolean expressions to assembly language.

Using complete boolean evaluation to evaluate a boolean expression for an IF statement is almost identical to converting arithmetic expressions into assembly language. Indeed, the previous volume covers this conversion process (see "Logical (Boolean) Expressions" on page 604). About the only thing worth noting

about that process is that you do not need to store the ultimate boolean result in some variable; once the evaluation of the expression is complete you check to see if you have a false (zero) or true (one, or non-zero) result to determine whether to branch around the THEN portion of the IF statement. As you can see in the examples in the preceding sections, you can often use the fact that the last boolean instruction (AND/OR) sets the zero flag if the result is false and clears the zero flag if the result is true. This lets you avoid explicitly testing the result. Consider the following IF statement and its conversion to assembly language using complete boolean evaluation:

```

    if( (( x < y ) && ( z > t )) || ( a != b ) )
        Stmt1;

        mov( x, eax );
        cmp( eax, y );
        setl( bl );           // Store x<y in bl.
        mov( z, eax );
        cmp( eax, t );
        setg( bh );           // Store z > t in bh.
        and( bh, bl );       // Put (x<y) && (z>t) into bl.
        mov( a, eax );
        cmp( eax, b );
        setne( bh );         // Store a != b into bh.
        or( bh, bl );        // Put (x<y) && (z>t) || (a != b) into bl
        je SkipStmt1;       // Branch if result is false (OR sets Z-Flag if false).

    << Code for Stmt1 goes here >>

SkipStmt1:

```

This code computes a boolean value in the BL register and then, at the end of the computation, tests this resulting value to see if it contains true or false. If the result is false, this sequence skips over the code associated with *Stmt1*. The important thing to note in this example is that the program will execute each and every instruction that computes this boolean result (up to the JE instruction).

For more details on complete boolean evaluation, see “Logical (Boolean) Expressions” on page 604.

2.8.4 Short Circuit Boolean Evaluation

If you are willing to spend a little more effort studying a complex boolean expression, you can usually convert it to a much shorter and faster sequence of assembly language instructions using *short-circuit boolean evaluation*. Short-circuit boolean evaluation attempts to determine whether an expression is true or false by executing only a portion of the instructions that compute the complete expression. By executing only a portion of the instructions, the evaluation is often much faster. For this reason, plus the fact that short circuit boolean evaluation doesn’t require the use of any temporary registers, HLA uses short circuit evaluation when translating complex boolean expressions into assembly language.

To understand how short-circuit boolean evaluation works, consider the expression “A && B”. Once we determine that A is false, there is no need to evaluate B since there is no way the expression can be true. If A and B represent sub-expressions rather than simple variables, you can begin to see the savings that are possible with short-circuit boolean evaluation. As a concrete example, consider the sub-expression “((x<y) && (z>t))” from the previous section. Once you determine that x is not less than y, there is no need to check to see if z is greater than t since the expression will be false regardless of z and t’s values. The following code fragment shows how you can implement short-circuit boolean evaluation for this expression:

```

// if( (x<y) && (z>t) ) then ...

    mov( x, eax );
    cmp( eax, y );
    jnl TestFails;
    mov( z, eax );

```

```

cmp( eax, t );
jng TestFails;

    << Code for THEN clause of IF statement >>

TestFails:

```

Notice how the code skips any further testing once it determines that x is not less than y . Of course, if x is less than y , then the program has to test z to see if it is greater than t ; if not, the program skips over the THEN clause. Only if the program satisfies both conditions does the code fall through to the THEN clause.

For the logical OR operation the technique is similar. If the first sub-expression evaluates to true, then there is no need to test the second operand. Whatever the second operand's value is at that point, the full expression still evaluates to true. The following example demonstrates the use of short-circuit evaluation with disjunction (OR):

```

// if( ch < 'A' || ch > 'Z' ) then stdout.put( "Not an upper case char" ); endif;

cmp( ch, 'A' );
jb ItsNotUC
cmp( ch, 'Z' );
jna ItWasUC;

    ItsNotUC:
        stdout.put( "Not an upper case char" );

    ItWasUC:

```

Since the conjunction and disjunction operators are commutative, you can evaluate the left or right operand first if it is more convenient to do so. As one last example in this section, consider the full boolean expression from the previous section:

```

// if( ( ( x < y ) && ( z > t ) ) || ( a != b ) ) Stmt1;

mov( a, eax );
cmp( eax, b );
jne DoStmt1;
mov( x, eax );
cmp( eax, y );
jnl SkipStmt1;
mov( z, eax );
cmp( eax, t );
jng SkipStmt1;

    DoStmt1:
        << Code for Stmt1 goes here >>

    SkipStmt1:

```

Notice how the code in this example chose to evaluate “ $a \neq b$ ” first and the remaining sub-expression last. This is a common technique assembly language programmers use to write better code.

2.8.5 Short Circuit vs. Complete Boolean Evaluation

One fact about complete boolean evaluation is that every statement in the sequence will execute when evaluating the expression. Short-circuit boolean evaluation may not require the execution of every statement associated with the boolean expression. As you've seen in the previous two sections above, code based on short-circuit evaluation is usually shorter and faster². So it would seem that short-circuit evaluation is the technique of choice when converting complex boolean expressions to assembly language.

Sometimes, unfortunately, short-circuit boolean evaluation may not produce the correct result. In the presence of *side-effects* in an expression, short-circuit boolean evaluation will produce a different result than complete boolean evaluation. Consider the following C/C++ example:

```
if( ( x == y ) && ( ++z != 0 ) ) stmt;
```

Using complete boolean evaluation, you might generate the following code:

```
mov( x, eax );      // See if x == y
cmp( eax, y );
sete( bl );
inc( z );           // ++z
cmp( z, 0 );        // See if incremented z is zero.
setne( bh );
and( bh, bl );     // Test x == y && ++z != 0
jz SkipStmt;

<< code for stmt goes here >>
```

SkipStmt:

Using short-circuit boolean evaluation, you might generate the following code:

```
mov( x, eax );      // See if x == y
cmp( eax, y );
jne SkipStmt;
inc( z );           // ++z
cmp( z, 0 );        // See if incremented z is zero.
je SkipStmt;

<< code for stmt goes here >>
```

SkipStmt:

Notice a very subtle, but important difference between these two conversions: if it turns out that x is equal to y , then the first version above *still increments* z and compares it to zero before it executes the code associated with *stmt*; the short-circuit version, on the other hand skips over the code that increments z if it turns out that x is equal to y . Therefore, the behavior of these two code fragments is different with respect to what happens to z if x is equal to y . Neither implementation is particularly wrong, depending on the circumstances you may or may not want the code to increment z if x is equal to y . However, it is important that you realize that these two schemes produce different results so you can choose an appropriate implementation if the effect of this code on z matters to your program.

Many programs take advantage of short-circuit boolean evaluation and rely upon the fact that the program may not evaluate certain components of the expression. The following C/C++ code fragment demonstrates what is probably the most common example that requires short-circuit boolean evaluation:

```
if( Ptr != NULL && *Ptr == 'a' ) stmt;
```

If it turns out that *Ptr* is NULL in this IF statement, then the expression is false and there is no need to evaluate the remainder of the expression (and, therefore, code that uses short-circuit boolean evaluation will not evaluate the remainder of this expression). This statement relies upon the semantics of short-circuit boolean evaluation for correct operation. Were C/C++ to use complete boolean evaluation, and the variable *Ptr* contained NULL, then the second half of the expression would attempt to dereference a NULL pointer (which tends to crash most programs). Consider the translation of this statement using complete and short-circuit boolean evaluation:

```
// Complete boolean evaluation:
```

2. Note that this does not always mean that the program will run faster. Jumps (conditional or otherwise) are often very slow executing instructions. Sometimes it's faster to execute several instructions in a row rather than execute a few instructions that include a conditional jump.

```

mov( Ptr, eax );
test( eax, eax );    // Check to see if EAX is zero (NULL is zero).
setne( bl );
mov( [eax], al );    // Get *Ptr into AL.
cmp( al, 'a' );
sete( bh );
and( bh, bl );
jz SkipStmt;

<< code for stmt goes here >>

SkipStmt:

```

Notice in this example that if *Ptr* contains NULL (zero), then this program will attempt to access the data at location zero in memory via the “mov([eax], al);” instruction. Under most operating systems this will cause a memory access fault (general protection fault). Now consider the short-circuit boolean conversion:

```

// Short-circuit boolean evaluation

mov( Ptr, eax );    // See if Ptr contains NULL (zero) and
test( eax, eax );    // immediately skip past Stmt if this
jz SkipStmt;        // is the case

mov( [eax], al );    // If we get to this point, Ptr contains
cmp( al, 'a' );    // a non-NULL value, so see if it points
jne SkipStmt;        // at the character 'a'.

<< code for stmt goes here >>

SkipStmt:

```

As you can see in this example, the problem with dereferencing the NULL pointer doesn't exist. If *Ptr* contains NULL, this code skips over the statements that attempt to access the memory address *Ptr* contains.

2.8.6 Efficient Implementation of IF Statements in Assembly Language

Encoding IF statements efficiently in assembly language takes a bit more thought than simply choosing short-circuit evaluation over complete boolean evaluation. To write code that executes as quickly as possible in assembly language you must carefully analyze the situation and generate the code appropriately. The following paragraphs provide some suggestions you can apply to your programs to improve their performance.

Know your data!

A mistake programmers often make is the assumption that data is random. In reality, data is rarely random and if you know the types of values that your program commonly uses, you can use this knowledge to write better code. To see how, consider the following C/C++ statement:

```
if(( a == b ) && ( c < d )) ++i;
```

Since C/C++ uses short-circuit evaluation, this code will test to see if *a* is equal to *b*. If so, then it will test to see if *c* is less than *d*. If you expect *a* to be equal to *b* most of the time but don't expect *c* to be less than *d* most of the time, this statement will execute slower than it should. Consider the following HLA implementation of this code:

```

mov( a, eax );
cmp( eax, b );
jne DontIncI;

mov( c, eax );

```

```

cmp( eax, d );
jnl DontIncI;

    inc( i );

DontIncI:

```

As you can see in this code, if a is equal to b most of the time and c is not less than d most of the time, you will have to execute the first three instructions nearly every time in order to determine that the expression is false. Now consider the following implementation of the above C/C++ statement that takes advantage of this knowledge and the fact that the “&&” operator is commutative:

```

mov( c, eax );
cmp( eax, d );
jnl DontIncI;

mov( a, eax );
cmp( eax, b );
jne DontIncI;

    inc( i );

DontIncI:

```

In this example the code first checks to see if c is less than d . If most of the time c is less than d , then this code determines that it has to skip to the label *DontIncI* after executing only three instructions in the typical case (compared with six instructions in the previous example). This fact is much more obvious in assembly language than in a high level language; this is one of the main reasons that assembly programs are often faster than their high level language counterparts: optimizations are more obvious in assembly language than in a high level language. Of course, the key here is to understand the behavior of your data so you can make intelligent decisions such as the one above.

Rearranging Expressions

Even if your data is random (or you can't determine how the input values will affect your decisions), there may still be some benefit to rearranging the terms in your expressions. Some calculations take far longer to compute than others. For example, the DIV instruction is much slower than a simple CMP instruction. Therefore, if you have a statement like the following you may want to rearrange the expression so that the CMP comes first:

```
if( (x % 10 = 0 ) && (x != y ) ++x;
```

Converted to assembly code, this IF statement becomes:

```

mov( x, eax );           // Compute X % 10
cdq();                  // Must sign extend EAX -> EDX:EAX
imod( 10, edx:eax );    // Remember, remainder goes into EDX
test( edx, edx );      // See if EDX is zero.
jnz SkipIF

mov( x, eax );
cmp( eax, y );
je SkipIF

    inc( x );

SkipIF:

```

The IMOD instruction is very expensive (often 50-100 times slower than most of the other instructions in this example). Unless it is 50-100 times more likely than the remainder is zero rather than x is equal to y , it would be better to do the comparison first and the remainder calculation afterwards:

```

mov( x, eax );
cmp( eax, y );
je SkipIF

mov( x, eax );           // Compute X % 10
cdq();                 // Must sign extend EAX -> EDX:EAX
imod( 10, edx:eax );   // Remember, remainder goes into EDX
test( edx, edx );      // See if EDX is zero.
jnz SkipIF

    inc( x );

SkipIF:

```

Of course, in order to rearrange the expression in this manner, the code must not assume the use of short-circuit evaluation semantics (since the `&&` and `||` operators are not commutative if the code must compute one subexpression before another).

Destructuring Your Code

Although there is a lot of good things to be said about structured programming techniques, there are some drawbacks to writing structured code. Specifically, structured code is sometimes less efficient than unstructured code. Most of the time this is tolerable because unstructured code is difficult to read and maintain; it is often acceptable to sacrifice some performance in exchange for maintainable code. In certain instances, however, you may need all the performance you can get. In those rare instances you might choose to compromise the readability of your code in order to gain some additional performance.

One classic way to do this is to use code movement to move code your program rarely uses out of the way of code that executes most of the time. For example, consider the following pseudo C/C++ statement:

```

if( See_If_an_Error_Has_Ocurred )
{
    << Statements to execute if no error >>
}
else
{
    << Error handling statements >>
}

```

In normal code, one does not expect errors to be frequent. Therefore, you would normally expect the THEN section of the above IF to execute far more often than the ELSE clause. The code above could translate into the following assembly code:

```

cmp( See_If_an_Error_Has_Ocurred, true );
je HandleTheError

    << Statements to execute if no error >>
    jmp EndOfIF;

HandleTheError:
    << Error handling statements >>

EndOfIf:

```

Notice that if the expression is false this code falls through to the normal statements and then jumps over the error handling statements. Instructions that transfer control from one point in your program to

another (e.g., JMP instructions) tend to be slow. It is much faster to execute a sequential set of instructions rather than jump all over the place in your program. Unfortunately, the code above doesn't allow this. One way to rectify this problem is to move the ELSE clause of the code somewhere else in your program. That is, you could rewrite the code as follows:

```

cmp( See_If_an_Error_Has_Occurred, true );
je HandleTheError

    << Statements to execute if no error >>

EndOfIf:

```

At some other point in your program (typically after a JMP instruction) you would insert the following code:

```

HandleTheError:
    << Error handling statements >>
    jmp EndOfIf;

```

Note that the program isn't any shorter. The JMP you removed from the original sequence winds up at the end of the ELSE clause. However, since the ELSE clause rarely executes, moving the JMP instruction from the THEN clause (which executes frequently) to the ELSE clause is a big performance win because the THEN clause executes using only straight-line code. This technique is surprisingly effective in many time-critical code segments.

There is a difference between writing *destructured* code and writing *unstructured* code. Unstructured code is written in an unstructured way to begin with. It is generally hard to read, difficult to maintain, and it often contains defects. Destructured code, on the other hand, starts out as structured code and you make a conscious decision to eliminate the structure in order to gain a small performance boost. Generally, you've already tested the code in its structured form before destructuring it. Therefore, destructured code is often easier to work with than unstructured code.

Calculation Rather than Branching

On many processors in the 80x86 family, branches are very expensive compared to many other instructions (perhaps not as bad as IMOD or IDIV, but typically an order of magnitude worse than instructions like ADD and SUB). For this reason it is sometimes better to execute more instructions in a sequence rather than fewer instructions that involve branching. For example, consider the simple assignment "EAX = abs(EAX);" Unfortunately, there is no 80x86 instruction that computes the absolute value of an integer value. The obvious way to handle this is with an instruction sequence like the following:

```

test( eax, eax );
jns ItsPositive;

    neg( eax );

ItsPositive:

```

However, as you can plainly see in this example, it uses a conditional jump to skip over the NEG instruction (that creates a positive value in EAX if EAX was negative). Now consider the following sequence that will also do the job:

```

// Set EDX to $FFFF_FFFF if EAX is negative, $0000_0000 if EAX is
// zero or positive:

    cdq();

// If EAX was negative, the following code inverts all the bits in EAX,
// otherwise it has no effect on EAX.

```

```

xor( edx, edx);

// If EAX was negative, the following code adds one to EAX, otherwise
// it doesn't modify EAX's value.

and( 1, edx );      // EDX = 0 or 1 (1 if EAX was negative).
add( edx, eax );

```

This code will invert all the bits in EAX and then add one to EAX if EAX was negative prior to the sequence (i.e., it takes the two's complement [negates] the value in EAX). If EAX was zero or positive, then this code does not change the value in EAX.

Note that this sequence takes four instructions rather than the three the previous example requires. However, since there are no transfer of control instructions in this sequence, it may execute faster on many CPUs in the 80x86 family.

2.8.7 SWITCH/CASE Statements

The HLA (Standard Library) SWITCH statement takes the following form :

```

switch( reg32 )
  case( const1 )
    <<stmts1>>

  case( const2 )
    <<stmts2>>
  .
  .
  .
  case( constn )
    <<stmtsn >>

  default      // Note that the default section is optional
    <<stmtsdefault >>

endswitch;

```

When this statement executes, it checks the value of register against the constants $const_1 \dots const_n$. If a match is found then the corresponding statements execute. HLA places a few restrictions on the SWITCH statement. First, the HLA SWITCH statement only allows a 32-bit register as the SWITCH expression. Second, all the constants appearing as CASE clauses must be unique. The reason for these restrictions will become clear in a moment.

Most introductory programming texts introduce the SWITCH/CASE statement by explaining it as a sequence of IF..THEN..ELSEIF statements. They might claim that the following two pieces of HLA code are equivalent:

```

switch( eax )
  case(0) stdout.put("I=0");
  case(1) stdout.put("I=1");
  case(2) stdout.put("I=2");
endswitch;

if( eax = 0 ) then
  stdout.put("I=0")
elseif( eax = 1 ) then
  stdout.put("I=1")
elseif( eax = 2 ) then
  stdout.put("I=2");
endif;

```


While semantically these two code segments may be the same, their implementation is usually different. Whereas the IF.THEN.ELSEIF chain does a comparison for each conditional statement in the sequence, the SWITCH statement normally uses an indirect jump to transfer control to any one of several statements with a single computation. Consider the two examples presented above, they could be written in assembly language with the following code:

```
// IF..THEN..ELSE form:

    mov( i, eax );
    test( eax, eax ); // Check for zero.
    jnz Not0;
        stdout.put( "I=0" );
        jmp EndCase;

Not0:
    cmp( eax, 1 );
    jne Not1;
        stdou.put( "I=1" );
        jmp EndCase;

Not1:
    cmp( eax, 2 );
    jne EndCase;
        stdout.put( "I=2" );
EndCase:

// Indirect Jump Version

readonly
    jmpTbl:dword[3] := [ &Stmt0, &Stmt1, &Stmt2 ];
    .
    .
    .
    mov( i, eax );
    jmp( jmpTbl[ eax*4 ] );

    Stmt0:
        stdout.put( "I=0" );
        jmp EndCase;

    Stmt1:
        stdout.put( "I=1" );
        jmp EndCase;

    Stmt2:
        stdout.put( "I=2" );

EndCase:
```

The implementation of the IF.THEN.ELSEIF version is fairly obvious and doesn't need much in the way of explanation. The indirect jump version, however, is probably quite mysterious to you; so let's consider how this particular implementation of the SWITCH statement works.

Remember that there are three common forms of the JMP instruction (see "Unconditional Transfer of Control (JMP)" on page 753). The standard unconditional JMP instruction, like the "jmp EndCase;" instructions in the previous examples, transfer control directly to the statement label specified as the JMP operand. The second form of the JMP instruction (i.e., "jmp Reg₃₂;") transfers control to the memory location specified by the address found in a 32-bit register. The third form of the JMP instruction, the one the example above uses, transfers control to the instruction specified by the contents of a dword memory location. As this example clearly illustrates, that memory location can use any addressing mode. You are not limited to the

displacement-only addressing mode. Now let's consider exactly how this second implementation of the SWITCH statement works.

To begin with, a SWITCH statement requires that you create an array of pointers with each element containing the address of a statement label in your code (those labels must be attached to the sequence of instructions to execute for each case in the SWITCH statement). In the example above, the *JmpTbl* array serves this purpose. Note that this code initializes *JmpTbl* with the address of the statement labels *Stmt0*, *Stmt1*, and *Stmt2*. The program places this array in the READONLY section because the program should never change these values during execution.

Warning: whenever you initialize an array with a set of address of statement labels as in this example, the declaration section in which you declare the array (e.g., READONLY in this case) must be in the same procedure that contains the statement labels³.

During the execution of this code sequence, the program loads the EAX register with *I*'s value. Then the program uses this value as an index into the *JmpTbl* array and transfers control to the four-byte address found at the specified location. For example, if EAX contains zero, the "jmp(JmpTbl[eax*4]);" instruction will fetch the dword at address *JmpTbl+0* (*eax*4=0*). Since the first double word in the table contains the address of *Stmt0*, the JMP instruction will transfer control to the first instruction following the *Stmt0* label. Likewise, if *I* (and therefore, EAX) contains one, then the indirect JMP instruction fetches the double word at offset four from the table and transfers control to the first instruction following the *Stmt1* label (since the address of *Stmt1* appears at offset four in the table). Finally, if *I/EAX* contains two, then this code fragment transfers control to the statements following the *Stmt2* label since it appears at offset eight in the *JmpTbl* table.

Two things should become readily apparent: the more (consecutive) cases you have, the more efficient the jump table implementation becomes (both in terms of space and speed) over the IF/ELSEIF form. Except for trivial cases, the SWITCH statement is almost always faster and usually by a large margin. As long as the CASE values are consecutive, the SWITCH statement version is usually smaller as well.

What happens if you need to include non-consecutive CASE labels or you cannot be sure that the SWITCH value doesn't go out of range? With the HLA SWITCH statement, such an occurrence will transfer control to the first statement after the ENDSWITCH clause. However, this doesn't happen in the example above. If variable *I* does not contain zero, one, or two, the result of executing the code above is undefined. For example, if *I* contains five when you execute the code in the previous example, the indirect JMP instruction will fetch the dword at offset 20 ($5*4$) in *JmpTbl* and transfer control to that address. Unfortunately, *JmpTbl* doesn't have six entries; so the program will wind up fetching the value of the third double word following *JmpTbl* and using that as the target address. This will often crash your program or transfer control to an unexpected location. Clearly this code does not behave like the HLA SWITCH statement, nor does it have desirable behavior.

The solution is to place a few instructions before the indirect JMP to verify that the SWITCH selection value is within some reasonable range. In the previous example, we'd probably want to verify that *I*'s value is in the range 0..2 before executing the JMP instruction. If *I*'s value is outside this range, the program should simply jump to the *EndCase* label (this corresponds to dropping down to the first statement after the ENDSWITCH clause). The following code provides this modification:

```
readonly
  JmpTbl:dword[3] := [ &Stmt0, &Stmt1, &Stmt2 ];
  .
  .
  .
  mov( i, eax );
  cmp( eax, 2 );           // Verify that I is in the range
  ja EndCase;             // 0..2 before the indirect JMP.
  jmp( JmpTbl[ eax*4 ] );
```

3. If the SWITCH statement appears in your main program, you must declare the array in the declaration section of your main program.

```

Stmt0:
    stdout.put( "I=0" );
    jmp EndCase;

Stmt1:
    stdout.put( "I=1" );
    jmp EndCase;

Stmt2:
    stdout.put( "I=2" );

EndCase:

```

Although the example above handles the problem of selection values being outside the range zero through two, it still suffers from a couple of severe restrictions:

- The cases must start with the value zero. That is, the minimum CASE constant has to be zero in this example.
- The case values must be contiguous; there cannot be any gaps between any two case values.

Solving the first problem is easy and you deal with it in two steps. First, you must compare the case selection value against a lower and upper bounds before determining if the case value is legal, e.g.,

```

// SWITCH statement specifying cases 5, 6, and 7:
// WARNING: this code does *NOT* work. Keep reading to find out why.

```

```

mov( i, eax );
cmp( eax, 5 );
jb EndCase
cmp( eax, 7 );           // Verify that I is in the range
ja EndCase;             // 5..7 before the indirect JMP.
jmp( JmpTbl[ eax*4 ] );

Stmt5:
    stdout.put( "I=5" );
    jmp EndCase;

Stmt6:
    stdout.put( "I=6" );
    jmp EndCase;

Stmt7:
    stdout.put( "I=7" );

EndCase:

```

As you can see, this code adds a pair of extra instructions, `CMP` and `JB`, to test the selection value to ensure it is in the range five through seven. If not, control drops down to the `EndCase` label, otherwise control transfers via the indirect `JMP` instruction. Unfortunately, as the comments point out, this code is broken. Consider what happens if variable `i` contains the value five: The code will verify that five is in the range five through seven and then it will fetch the dword at offset 20 (`5*@size(dword)`) and jump to that address. As before, however, this loads four bytes outside the bounds of the table and does not transfer control to a defined location. One solution is to subtract the smallest case selection value from `EAX` before executing the `JMP` instruction. E.g.,

```

// SWITCH statement specifying cases 5, 6, and 7:
// WARNING: there is a better way to do this. Keep reading.

```

```

readonly
JmpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];

```

```

    .
    .
    .
mov( i, eax );
cmp( eax, 5 );
jb EndCase
cmp( eax, 7 );           // Verify that I is in the range
ja EndCase;             // 5..7 before the indirect JMP.
sub( 5, eax );           // 5->0, 6->1, 7->2.
jmp( JumpTbl[ eax*4 ] );

    Stmt5:
        stdout.put( "I=5" );
        jmp EndCase;

    Stmt6:
        stdout.put( "I=6" );
        jmp EndCase;

    Stmt7:
        stdout.put( "I=7" );

EndCase:

```

By subtracting five from the value in EAX this code forces EAX to take on the values zero, one, or two prior to the JMP instruction. Therefore, case selection value five jumps to *Stmt5*, case selection value six transfers control to *Stmt6*, and case selection value seven jumps to *Stmt7*.

There is a sneaky way to slightly improve the code above. You can eliminate the SUB instruction by merging this subtraction into the JMP instruction's address expression. Consider the following code that does this:

```

// SWITCH statement specifying cases 5, 6, and 7:

readonly
    JumpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];
    .
    .
    .
mov( i, eax );
cmp( eax, 5 );
jb EndCase
cmp( eax, 7 );           // Verify that I is in the range
ja EndCase;             // 5..7 before the indirect JMP.
jmp( JumpTbl[ eax*4 - 5*@size(dword)] );

    Stmt5:
        stdout.put( "I=5" );
        jmp EndCase;

    Stmt6:
        stdout.put( "I=6" );
        jmp EndCase;

    Stmt7:
        stdout.put( "I=7" );

EndCase:

```

The HLA SWITCH statement provides a DEFAULT clause that executes if the case selection value doesn't match any of the case values. E.g.,

```
switch( ebx )

    case( 5 ) stdout.put( "ebx=5" );
    case( 6 ) stdout.put( "ebx=6" );
    case( 7 ) stdout.put( "ebx=7" );
    default
        stdout.put( "ebx does not equal 5, 6, or 7" );

endswitch;
```

Implementing the equivalent of the DEFAULT clause in pure assembly language is very easy. Just use a different target label in the JB and JA instructions at the beginning of the code. The following example implements an HLA SWITCH statement similar to the one immediately above:

// SWITCH statement specifying cases 5, 6, and 7 with a DEFAULT clause:

```
readonly
    jmpTbl:dword[3] := [ &stmt5, &stmt6, &stmt7 ];
    .
    .
    .
mov( i, eax );
cmp( eax, 5 );
jb DefaultCase;
cmp( eax, 7 );           // Verify that I is in the range
ja DefaultCase;         // 5..7 before the indirect JMP.
jmp( jmpTbl[ eax*4 - 5*@size(dword)] );

    stmt5:
        stdout.put( "I=5" );
        jmp EndCase;

    stmt6:
        stdout.put( "I=6" );
        jmp EndCase;

    stmt7:
        stdout.put( "I=7" );
        jmp EndCase;

    DefaultCase:
        stdout.put( "I does not equal 5, 6, or 7" );
EndCase:
```

The second restriction noted earlier, that the case values need to be contiguous, is easy to handle by inserting extra entries into the jump table. Consider the following HLA SWITCH statement:

```
switch( ebx )

    case( 1 ) stdout.put( "ebx = 1" );
    case( 2 ) stdout.put( "ebx = 2" );
    case( 4 ) stdout.put( "ebx = 4" );
    case( 8 ) stdout.put( "ebx = 8" );
    default
        stdout.put( "ebx is not 1, 2, 4, or 8" );

endswitch;
```

The minimum switch value is one and the maximum value is eight. Therefore, the code before the indirect JMP instruction needs to compare the value in EBX against one and eight. If the value is between one and eight, it's still possible that EBX might not contain a legal case selection value. However, since the JMP instruction indexes into a table of dwords using the case selection table, the table must have eight dword entries. To handle the values between one and eight that are not case selection values, simply put the statement label of the default clause (or the label specifying the first instruction after the ENDSWITCH if there is no DEFAULT clause) in each of the jump table entries that don't have a corresponding CASE clause. The following code demonstrates this technique:

```
readonly
    JumpTbl2: dword :=
        [
            &Case1, &Case2, &dfltCase, &Case4,
            &dfltCase, &dfltCase, &dfltCase, &Case8
        ];
    .
    .
    .
    cmp( ebx, 1 );
    jb dfltCase;
    cmp( ebx, 8 );
    ja dfltCase;
    jmp( JumpTbl2[ ebx*4 - 1*@size(dword)] );

    Case1:
        stdout.put( "ebx = 1" );
        jmp EndOfSwitch;

    Case2:
        stdout.put( "ebx = 2" );
        jmp EndOfSwitch;

    Case4:
        stdout.put( "ebx = 4" );
        jmp EndOfSwitch;

    Case8:
        stdout.put( "ebx = 8" );
        jmp EndOfSwitch;

    dfltCase:
        stdout.put( "ebx is not 1, 2, 4, or 8" );

    EndOfSwitch:
```

There is a problem with this implementation of the SWITCH statement. If the CASE values contain non-consecutive entries that are widely spaced the jump table could become exceedingly large. The following SWITCH statement would generate an extremely large code file:

```
switch( ebx )

    case( 1      ) stmt1;
    case( 100    ) stmt2;
    case( 1_000  ) stmt3;
    case( 10_000 ) stmt4;
    default stmt5;

endswitch;
```

In this situation, your program will be much smaller if you implement the SWITCH statement with a sequence of IF statements rather than using an indirect jump statement. However, keep one thing in mind-

the size of the jump table does not normally affect the execution speed of the program. If the jump table contains two entries or two thousand, the SWITCH statement will execute the multi-way branch in a constant amount of time. The IF statement implementation requires a linearly increasing amount of time for each case label appearing in the case statement.

Probably the biggest advantage to using assembly language over a HLL like Pascal or C/C++ is that you get to choose the actual implementation. In some instances you can implement a SWITCH statement as a sequence of IF..THEN..ELSEIF statements, or you can implement it as a jump table, or you can use a hybrid of the two:

```
switch( eax )

    case( 0 ) stmt0;
    case( 1 ) stmt1;
    case( 2 ) stmt2;
    case( 100 ) stmt3;
    default stmt4;

endswitch;
```

could become:

```
cmp( eax, 100 );
je DoStmt3;
cmp( eax, 2 );
ja TheDefaultCase;
jmp( JumpTbl[ eax*4 ]);
etc.
```

Of course, you could do this in HLA using the following code high-level control structures:

```
if( ebx = 100 ) then stmt3
else
    switch( eax )
        case(0) stmt0;
        case(1) stmt1;
        case(2) stmt2;
        Otherwise stmt4
    endswitch;
endif;
```

But this tends to destroy the readability of the program. On the other hand, the extra code to test for 100 in the assembly language code doesn't adversely affect the readability of the program (perhaps because it's so hard to read already). Therefore, most people will add the extra code to make their program more efficient.

The C/C++ SWITCH statement is very similar to the HLA SWITCH statement⁴. There is only one major semantic difference: the programmer must explicitly place a BREAK statement in each CASE clause to transfer control to the first statement beyond the SWITCH. This BREAK corresponds to the JMP instruction at the end of each CASE sequence in the assembly code above. If the corresponding BREAK is not present, C/C++ transfers control into the code of the following CASE. This is equivalent to leaving off the JMP at the end of the CASE'S sequence:

```
switch (i)
{
    case 0: stmt1;
    case 1: stmt2;
    case 2: stmt3;
        break;
    case 3: stmt4;
```

4. The HLA Standard Library SWITCH statement actually provides an option to support C semantics. See the HLA Standard Library documentation for details.

```

        break;
    default: stmt5;
}

```

This translates into the following 80x86 code:

```

readonly
    JumpTbl: dword[4] := [ &case0, &case1, &case2, &case3 ];
    .
    .
    .
    mov( i, ebx );
    cmp( ebx, 3 );
    ja DefaultCase;
    jmp( JumpTbl[ ebx*4 ]);

    case0:
        stmt1;

    case1:
        stmt2;

    case2:
        stmt3;
        jmp EndCase;    // Emitted for the break stmt.

    case3:
        stmt4;
        jmp EndCase;    // Emitted for the break stmt.

    DefaultCase:
        stmt5;

EndCase:

```

2.9 State Machines and Indirect Jumps

Another control structure commonly found in assembly language programs is the *state machine*. A state machine uses a *state variable* to control program flow. The FORTRAN programming language provides this capability with the assigned GOTO statement. Certain variants of C (e.g., GNU's GCC from the Free Software Foundation) provide similar features. In assembly language, the indirect jump provides a mechanism to easily implement state machines.

So what is a state machine? In very basic terms, it is a piece of code⁵ that keeps track of its execution history by entering and leaving certain "states". For the purposes of this chapter, we'll not use a very formal definition of a state machine. We'll just assume that a state machine is a piece of code which (somehow) remembers the history of its execution (its *state*) and executes sections of code based upon that history.

In a very real sense, all programs are state machines. The CPU registers and values in memory constitute the "state" of that machine. However, we'll use a much more constrained view. Indeed, for most purposes only a single variable (or the value in the EIP register) will denote the current state.

Now let's consider a concrete example. Suppose you have a procedure which you want to perform one operation the first time you call it, a different operation the second time you call it, yet something else the third time you call it, and then something new again on the fourth call. After the fourth call it repeats these four different operations in order. For example, suppose you want the procedure to ADD EAX and EBX the

5. Note that state machines need not be software based. Many state machines' implementation are hardware based.

first time, subtract them on the second call, multiply them on the third, and divide them on the fourth. You could implement this procedure as follows:

```

procedure StateMachine;
static
  State:byte := 0;
begin StateMachine;

  cmp( State, 0 );
  jne TryState1;

  // State 0: Add EBX to EAX and switch to state 1:

  add( ebx, eax );
  inc( State );
  exit StateMachine;

TryState1:
  cmp( State, 1 );
  jne TryState2;

  // State 1: subtract ebx from EAX and switch to state 2:

  sub( ebx, eax );
  inc( State );      // State 1 becomes State 2.
  exit StateMachine;

TryState2:
  cmp( State, 2 );
  jne MustBeState3;

  // If this is state 2, multiply EAX by EAX and switch to state 3:

  intmul( ebx, eax );
  inc( State );      // State 2 becomes State 3.
  exit StateMachine;

  // If it isn't one of the above states, we must be in state 3
  // So divide eax by ebx and switch back to state zero.

MustBeState3:
  push( edx );      // Preserve this 'cause it gets whacked by DIV.
  xor( edx, edx );  // Zero extend EAX into EDX.
  div( ebx, edx:eax);
  pop( edx );       // Restore EDX's value preserved above.
  mov( 0, State );  // Reset the state back to zero.

end StateMachine;

```

Technically, this procedure is not the state machine. Instead, it is the variable *State* and the CMP/JNE instructions which constitute the state machine.

There is nothing particularly special about this code. It's little more than a SWITCH statement implemented via the IF.THEN..ELSEIF construct. The only thing special about this procedure is that it remembers how many times it has been called⁶ and behaves differently depending upon the number of calls. While this is a *correct* implementation of the desired state machine, it is not particularly efficient. The astute reader, of course, would recognize that this code could be made a little faster using an actual SWITCH statement rather than the IF.THEN..ELSEIF implementation. However, there is a better way...

6. Actually, it remembers how many times, *MOD 4*, that it has been called.

The more common implementation of a state machine in assembly language is to use an *indirect jump*. Rather than having a state variable which contains a value like zero, one, two, or three, we could load the state variable with the *address* of the code to execute upon entry into the procedure. By simply jumping to that address, the state machine could save the tests above needed to execute the proper code fragment. Consider the following implementation using the indirect jump:

```

procedure StateMachine;
static
    State:dword := &State0;
begin StateMachine;

    jmp( State );

    // State 0: Add EBX to EAX and switch to state 1:

State0:
    add( ebx, eax );
    mov( &State1, State );
    exit StateMachine;

State1:

    // State 1: subtract ebx from EAX and switch to state 2:

    sub( ebx, eax );
    mov( &State2, State );    // State 1 becomes State 2.
    exit StateMachine;

State2:

    // If this is state 2, multiply EAX by EAX and switch to state 3:

    intmul( ebx, eax );
    mov( &State3, State );    // State 2 becomes State 3.
    exit StateMachine;

    // State 3: divide eax by ebx and switch back to state zero.

State3:
    push( edx );                // Preserve this 'cause it gets whacked by DIV.
    xor( edx, edx );            // Zero extend EAX into EDX.
    div( ebx, edx:eax);
    pop( edx );                 // Restore EDX's value preserved above.
    mov( &State0, State );    // Reset the state back to zero.

end StateMachine;

```

The JMP instruction at the beginning of the *StateMachine* procedure transfers control to the location pointed at by the *State* variable. The first time you call *StateMachine* it points at the *State0* label. Thereafter, each subsection of code sets the *State* variable to point at the appropriate successor code.

2.10 Spaghetti Code

One major problem with assembly language is that it takes several statements to realize a simple idea encapsulated by a single HLL statement. All too often an assembly language programmer will notice that s/he can save a few bytes or cycles by jumping into the middle of some program structure. After a few such observations (and corresponding modifications) the code contains a whole sequence of jumps in and out of portions of the code. If you were to draw a line from each jump to its destination, the resulting listing would end up looking like someone dumped a bowl of spaghetti on your code, hence the term “spaghetti code”.

Spaghetti code suffers from one major drawback- it's difficult (at best) to read such a program and figure out what it does. Most programs start out in a "structured" form only to become spaghetti code at the altar of efficiency. Alas, spaghetti code is rarely efficient. Since it's difficult to figure out exactly what's going on, it's very difficult to determine if you can use a better algorithm to improve the system. Hence, spaghetti code may wind up less efficient than structured code.

While it's true that producing some spaghetti code in your programs may improve its efficiency (e.g., destructuring your code, see "Efficient Implementation of IF Statements in Assembly Language" on page 772), doing so should always be a last resort after you've tried everything else and you still haven't achieved what you need. Always start out writing your programs with straight-forward IFs and SWITCH statements. Start combining sections of code (via JMP instructions) once everything is working and well understood. Of course, you should never obliterate the structure of your code unless the gains are worth it.

A famous saying in structured programming circles is "After GOTOs, pointers are the next most dangerous element in a programming language." A similar saying is "Pointers are to data structures what GOTOs are to control structures." In other words, avoid excessive use of pointers. If pointers and gotos are bad, then the indirect jump must be the worst construct of all since it involves both GOTOs and pointers! Seriously though, the indirect jump instructions should be avoided for casual use. They tend to make a program harder to read. After all, an indirect jump can (theoretically) transfer control to any label within a program. Imagine how hard it would be to follow the flow through a program if you have no idea what a pointer contains and you come across an indirect jump using that pointer. Therefore, you should always exercise care when using jump indirect instructions.

2.11 Loops

Loops represent the final basic control structure (sequences, decisions, and loops) that make up a typical program. Like so many other structures in assembly language, you'll find yourself using loops in places you've never dreamed of using loops. Most HLLs have implied loop structures hidden away. For example, consider the BASIC statement "IF A\$ = B\$ THEN 100". This IF statement compares two strings and jumps to statement 100 if they are equal. In assembly language, you would need to write a loop to compare each character in A\$ to the corresponding character in B\$ and then jump to statement 100 if and only if all the characters matched. In BASIC, there is no loop to be seen in the program. In assembly language, this very simple IF statement requires a loop to compare the individual characters in the string⁷. This is but a small example which shows how loops seem to pop up everywhere.

Program loops consist of three components: an optional initialization component, a loop termination test, and the body of the loop. The order with which these components are assembled can dramatically change the way the loop operates. Three permutations of these components appear over and over again. Because of their frequency, these loop structures are given special names in high-level languages: WHILE loops, REPEAT..UNTIL loops (do..while in C/C++), and infinite loops (e.g., FOREVER..ENDFOR in HLA).

2.11.1 While Loops

The most general loop is the WHILE loop. In HLA it takes the following form:

```
while( expression ) do <<statements>> endwhile;
```

There are two important points to note about the WHILE loop. First, the test for termination appears at the beginning of the loop. Second as a direct consequence of the position of the termination test, the body of the loop may never execute. If the termination condition is always true, the loop body will never execute.

7. Of course, the HLA Standard Library provides the *str.eq* routine that compares the strings for you, effectively hiding the loop even in an assembly language program.

Consider the following HLA WHILE loop:

```
mov( 0, I );
while( I < 100 ) do

    inc( I );

endwhile;
```

The “mov(0, I);” instruction is the initialization code for this loop. *I* is a loop control variable, because it controls the execution of the body of the loop. “I<100” is the loop termination condition. That is, the loop will not terminate as long as *I* is less than 100. The single instruction “inc(I);” is the loop body. This is the code that executes on each pass of the loop.

Note that an HLA WHILE loop can be easily synthesized using IF and JMP statements. For example, the HLA WHILE loop presented above can be replaced by:

```
mov( 0, I );
WhileLp:
if( I < 100 ) then

    inc( i );
    jmp WhileLp;

endif;
```

More generally, any WHILE loop can be built up from the following:

```
<< optional initialization code >>

UniqueLabel:
if( not_termination_condition ) then

    <<loop body>>
    jmp UniqueLabel;

endif;
```

Therefore, you can use the techniques from earlier in this chapter to convert IF statements to assembly language along with a single JMP instruction to produce a WHILE loop. The example we’ve been looking at in this section translates to the following “pure” 80x86 assembly code⁸:

```
mov( 0, i );
WhileLp:
    cmp( i, 100 );
    jnl WhileDone;
    inc( i );
    jmp WhileLp;

WhileDone:
```

2.11.2 Repeat..Until Loops

The REPEAT..UNTIL (do..while) loop tests for the termination condition at the end of the loop rather than at the beginning. In HLA, the REPEAT..UNTIL loop takes the following form:

```
<< optional initialization code >>
repeat
```

8. Note that HLA will actually convert most WHILE statements to different 80x86 code than this section presents. The reason for the difference appears a little later in this text when we explore how to write more efficient loop code.

```
<<loop body>>
```

```
until( termination_condition );
```

This sequence executes the initialization code, the loop body, then tests some condition to see if the loop should repeat. If the boolean expression evaluates to false, the loop repeats; otherwise the loop terminates. The two things to note about the REPEAT..UNTIL loop are that the termination test appears at the end of the loop and, as a direct consequence of this, the loop body always executes at least once.

Like the WHILE loop, the REPEAT..UNTIL loop can be synthesized with an IF statement and a JMP . You could use the following:

```
<< initialization code >>
```

```
SomeUniqueLabel:
```

```
<< loop body >>
```

```
if( not_the_termination_condition ) then jmp SomeUniqueLabel; endif;
```

Based on the material presented in the previous sections, you can easily synthesize REPEAT..UNTIL loops in assembly language. The following is a simple example:

```
repeat
```

```
    stdout.put( "Enter a number greater than 100: " );
```

```
    stdin.get( i );
```

```
until( i > 100 );
```

```
// This translates to the following IF/JMP code:
```

```
RepeatLbl:
```

```
    stdout.put( "Enter a number greater than 100: " );
```

```
    stdin.get( i );
```

```
if( i <= 100 ) then jmp RepeatLbl; endif;
```

```
// It also translates into the following "pure" assembly code:
```

```
RepeatLabel:
```

```
    stdout.put( "Enter a number greater than 100: " );
```

```
    stdin.get( i );
```

```
    cmp( i, 100 );
```

```
    jng RepeatLbl;
```

2.11.3 FOREVER..ENDFOR Loops

If WHILE loops test for termination at the beginning of the loop and REPEAT..UNTIL loops check for termination at the end of the loop, the only place left to test for termination is in the middle of the loop. The HLA FOREVER..ENDFOR loop, combined with the BREAK and BREAKIF statements, provide this capability. The FOREVER..ENDFOR loop takes the following form:

```

forever

    << loop body >>

endfor;

```

Note that there is no explicit termination condition. Unless otherwise provided for, the FOREVER..ENDFOR construct simply forms an infinite loop. Loop termination is typically handled by a BREAKIF statement. Consider the following HLA code that employs a FOREVER..ENDFOR construct:

```

forever

    stdin.get( character );
    breakif( character = '.' );
    stdout.put( character );

endfor;

```

Converting a FOREVER loop to pure assembly language is trivial. All you need is a label and a JMP instruction. The BREAKIF statement in this example is really nothing more than an IF and a JMP instruction. The “pure” assembly language version of the code above looks something like the following:

```

foreverLabel:

    stdin.get( character );
    cmp( character, '.' );
    je ForIsDone;
    stdout.put( character );
    jmp foreverLabel;

ForIsDone:

```

2.11.4 FOR Loops

The FOR loop is a special form of the WHILE loop that repeats the loop body a specific number of times. In HLA, the FOR loop takes the following form:

```

for( <<Initialization Stmt>>; <<Termination Expression>>; <<inc_Stmt>> ) do

    << statements >>

endfor;

```

This is completely equivalent to the following:

```

<< Initialization Stmt>>;
while( <<Termination Expression>> ) do

    << statements >>

    <<inc_Stmt>>

endwhile;

```

Traditionally, the FOR loop has been used to process arrays and other objects accessed in sequential numeric order. One normally initializes a loop control variable with the initialization statement and then uses the loop control variable as an index into the array (or other data type), e.g.,

```

for( mov(0, esi); esi < 7; inc( esi ) ) do

```

```

stdout.put( "Array Element = ", SomeArray[ esi*4], nl );
endfor;

```

To convert this to “pure” assembly language, begin by translating the FOR loop into an equivalent WHILE loop:

```

mov( 0, esi );
while( esi < 7 ) do

    stdout.put( "Array Element = ", SomeArray[ esi*4], nl );

    inc( esi );
endwhile;

```

Now, using the techniques from the section on WHILE loops (see “While Loops” on page 787), translate the code into pure assembly language:

```

mov( 0, esi );
WhileLp:
cmp( esi, 7 );
jnl EndWhileLp;

    stdout.put( "Array Element = ", SomeArray[ esi*4], nl );

    inc( esi );
    jmp WhileLp;

EndWhileLp:

```

2.11.5 The BREAK and CONTINUE Statements

The HLA BREAK and CONTINUE statements both translate into a single JMP instruction. The BREAK instruction exits the loop that immediately contains the BREAK statement; the CONTINUE statement restarts the loop that immediately contains the CONTINUE statement.

Converting a BREAK statement to “pure” assembly language is very easy. Just emit a JMP instruction that transfers control to the first statement following the ENDxxx clause of the loop to exit. This is easily accomplished by placing a label after the associated END clause and jumping to that label. The following code fragments demonstrate this technique for the various loops:

```

// Breaking out of a forever loop:

forever
    <<stmts>>
    //break;
    jmp BreakFromForever;
<<stmts>>
endfor;
BreakFromForever:

// Breaking out of a FOR loop;
for( <<initStmt>>; <<expr>>; <<incStmt>> ) do
    <<stmts>>
    //break;
    jmp BrkFromFor;
<<stmts>>
endfor;

```

```
BrkFromFor:

// Breaking out of a WHILE loop:

while( <<expr>> ) do
  <<stmts>>
  //break;
  jmp BrkFromWhile;
<<stmts>>
endwhile;
BrkFromWhile:

// Breaking out of a REPEAT..UNTIL loop:

repeat
  <<stmts>>
  //break;
  jmp BrkFromRpt;
<<stmts>>
until( <<expr>> );
BrkFromRpt:
```

The CONTINUE statement is slightly more difficult to implement than the BREAK statement. The implementation still consists of a single JMP instruction, however the target label doesn't wind up going in the same spot for each of the different loops. The following figures show where the CONTINUE statement transfers control for each of the HLA loops:

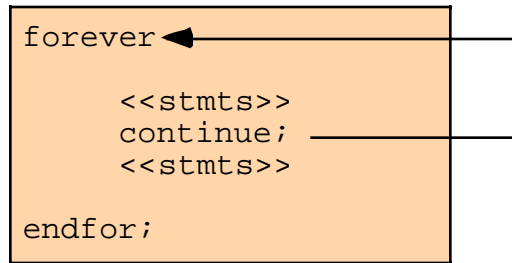


Figure 2.2 CONTINUE Destination for the FOREVER Loop

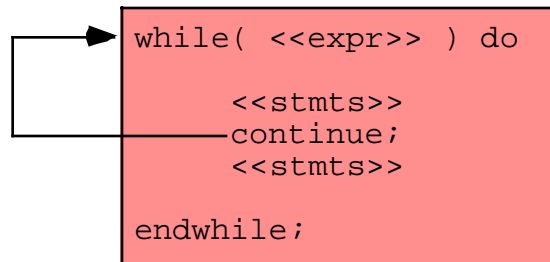
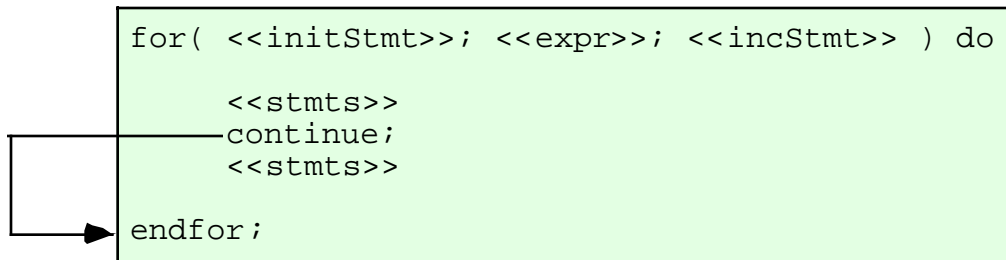


Figure 2.3 CONTINUE Destination and the WHILE Loop



Note: CONTINUE forces the execution of the <<incStmt>> clause and then transfers control to the test for loop termination.

Figure 2.4 CONTINUE Destination and the FOR Loop

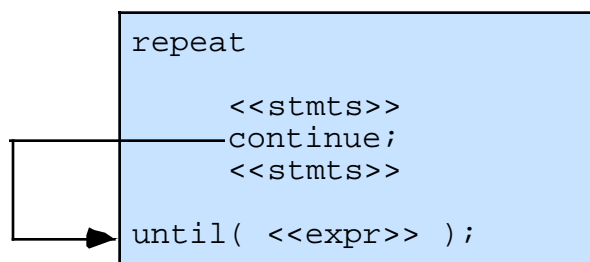


Figure 2.5 CONTINUE Destination and the REPEAT..UNTIL Loop

The following code fragments demonstrate how to convert the CONTINUE statement into an appropriate JMP instruction for each of these loop types.

forever..continue..endfor

```
// Conversion of forever loop w/continue
// to pure assembly:
forever
    <<stmts>>
    continue;
    <<stmts>>
endfor;
```

```
// Converted code:
```

```
foreverLbl:
    <<stmts>>
    jmp foreverLbl;
    <<stmts>>
```

while..continue..endwhile

```
// Conversion of while loop w/continue
// into pure assembly:
```

```
while( <<expr>> ) do
    <<stmts>>
    continue;
    <<stmts>>
endwhile;
```

```
// Converted code:
```

```
whlLabel:
<<Code to evaluate expr>>
Jcc EndOfWhile; // Skip loop on expr failure.
    <<stmts>>
    jmp whlLabel; // Jump to start of loop on continue.
    <<stmts>>
    jmp whlLabel; // Repeat the code.
EndOfwhile:
```

for..continue..endfor

```
// Conversion for a for loop w/continue
// into pure assembly:
```

```
for( <<initStmt>>; <<expr>>; <<incStmt>> ) do
    <<stmts>>
    continue;
    <<stmts>>
endfor;
```

```
// Converted code
```

```
<<initStmt>>
ForLpLbl:
```

```

<<Code to evaluate expr>>
Jcc EndOfFor; // Branch if expression fails.
  <<stmts>>
  jmp ContFor; // Branch to <<incStmt>> on continue.
  <<stmts>>

  ContFor:
  <<incStmt>>
  jmp ForLpLbl;
EndOfFor:

```

repeat..continue..until

```

repeat
  <<stmts>>
  continue;
  <<stmts>>
until( <<expr>> );

// Converted Code:

RptLpLbl:
  <<stmts>>
  jmp ContRpt; // Continue branches to loop termination test.
  <<stmts>>
ContRpt:
  <<code to test expr>>
  Jcc RptLpLbl; // Jumps if expression evaluates false.

```

2.11.6 Register Usage and Loops

Given that the 80x86 accesses registers much faster than memory locations, registers are the ideal spot to place loop control variables (especially for small loops). However, there are some problems associated with using registers within a loop. The primary problem with using registers as loop control variables is that registers are a limited resource. Therefore, the following will not work properly:

```

mov( 8, cx );
loop1:
  mov( 4, cx );
  loop2:
    <<stmts>>
    dec( cx );
    jnz loop2;

  dec( cx );
  jnz loop1;

```

The intent here, of course, was to create a set of nested loops, that is, one loop inside another. The inner loop (*Loop2*) should repeat four times for each of the eight executions of the outer loop (*Loop1*). Unfortunately, both loops use the same register as a loop control variable.. Therefore, this will form an infinite loop since CX will be set to zero at the end of the first loop. Since CX is always zero upon encountering the second DEC instruction, control will always transfer to the LOOP1 label (since decrementing zero produces a non-zero result). The solution here is to save and restore the CX register or to use a different register in place of CX for the outer loop:

```

mov( 8, cx );
loop1:
  push( cx );
  mov( 4, cx );
  loop2:

```

```

<<stmts>>
dec( cx );
jnz loop2;

pop( cx );
dec( cx );
jnz loop1;

```

or:

```

mov( 8, dx );
loop1:
mov( 4, cx );
loop2:
<<stmts>>
dec( cx );
jnz loop2;

dec( dx );
jnz loop1;

```

Register corruption is one of the primary sources of bugs in loops in assembly language programs, always keep an eye out for this problem.

2.12 Performance Improvements

The 80x86 microprocessors execute sequences of instructions at blinding speeds. Therefore, you'll rarely encounter a program that is slow which doesn't contain any loops. Since loops are the primary source of performance problems within a program, they are the place to look when attempting to speed up your software. While a treatise on how to write efficient programs is beyond the scope of this chapter, there are some things you should be aware of when designing loops in your programs. They're all aimed at removing unnecessary instructions from your loops in order to reduce the time it takes to execute one iteration of the loop.

2.12.1 Moving the Termination Condition to the End of a Loop

Consider the following flow graphs for the three types of loops presented earlier:

REPEAT..UNTIL loop:

```

Initialization code
  Loop body
Test for termination
Code following the loop

```

WHILE loop:

```

Initialization code
Loop termination test
  Loop body
  Jump back to test
Code following the loop

```

FOREVER..ENDFOR loop:

```

Initialization code
  Loop body, part one
  Loop termination test

```

```

    Loop body, part two
    Jump back to loop body part 1
Code following the loop

```

As you can see, the REPEAT..UNTIL loop is the simplest of the bunch. This is reflected in the assembly language code required to implement these loops. Consider the following REPEAT..UNTIL and WHILE loops that are identical:

```

// Example involving a WHILE loop:

mov( edi, esi );
sub( 20, esi );
while( esi <= edi ) do

    <<stmts>>
    inc( esi );

endwhile;

// Conversion of the code above into pure assembly language:

mov( edi, esi );
sub( 20, esi );
whlLbl:
cmp( esi, edi );
jnl EndOfWhile;

    <<stmts>>
    inc( esi );
    <<stmts>>
    jmp whlLbl;

EndOfWhile:

//Example involving a REPEAT..UNTIL loop:

mov( edi, esi );
sub( 20, esi );
repeat

    <<stmts>>
    inc( esi );

until( esi > edi );

// Conversion of the REPEAT..UNTIL loop into pure assembly:

rptLabel:
    <<stmts>>
    inc( esi );
    cmp( esi, edi );
    jng rptLabel;

```

As you can see by carefully studying the conversion to pure assembly language, testing for the termination condition at the end of the loop allowed us to remove a JMP instruction from the loop. This can be significant if this loop is nested inside other loops. In the preceding example there wasn't a problem with executing the body at least once. Given the definition of the loop, you can easily see that the loop will be executed exactly 20 times. This suggests that the conversion to a REPEAT..UNTIL loop is trivial and always possible. Unfortunately, it's not always quite this easy. Consider the following HLA code:

```

while( esi <= edi ) do
  <<stmts>>
  inc( esi );
endwhile;

```

In this particular example, we haven't the slightest idea what ESI contains upon entry into the loop. Therefore, we cannot assume that the loop body will execute at least once. So we must test for loop termination before executing the body of the loop. The test can be placed at the end of the loop with the inclusion of a single JMP instruction:

```

jmp WhlTest;
TopOfLoop:
  <<stmts>>
  inc( esi );
  WhlTest:
  cmp( esi, edi );
  jle TopOfLoop;

```

Although the code is as long as the original WHILE loop, the JMP instruction executes only once rather than on each repetition of the loop. Note that this slight gain in efficiency is obtained via a slight loss in readability. The second code sequence above is closer to spaghetti code than the original implementation. Such is often the price of a small performance gain. Therefore, you should carefully analyze your code to ensure that the performance boost is worth the loss of clarity. More often than not, assembly language programmers sacrifice clarity for dubious gains in performance, producing impossible to understand programs.

Note, by the way, that HLA translates its WHILE statement into a sequence of instructions that test the loop termination condition at the bottom of the loop using exactly the technique this section describes. Therefore, you do not have to worry about the HLA WHILE statement introducing slower code into your programs.

2.12.2 Executing the Loop Backwards

Because of the nature of the flags on the 80x86, loops which range from some number down to (or up to) zero are more efficient than any other. Compare the following HLA FOR loop and the code it generates:

```

for( mov( 1, J ); J <= 8; inc(J) ) do
  <<stmts>>
endfor;

```

// Conversion to pure assembly (as well as using a repeat..until form):

```

mov( 1, J );
ForLp:
  <<stmts>>
  inc( J );
  cmp( J, 8 );
  jnge ForLp;

```

Now consider another loop that also has eight iterations, but runs its loop control variable from eight down to one rather than one up to eight:

```

mov( 8, J );
LoopLbl:
  <<stmts>>
  dec( J );
  jnz LoopLbl;

```

Note that by running the loop from eight down to one we saved a comparison on each repetition of the loop.

Unfortunately, you cannot force all loops to run backwards. However, with a little effort and some coercion you should be able to write many FOR loops so they operate backwards. By saving the execution time of the CMP instruction on each iteration of the loop the code may run faster.

The example above worked out well because the loop ran from eight down to one. The loop terminated when the loop control variable became zero. What happens if you need to execute the loop when the loop control variable goes to zero? For example, suppose that the loop above needed to range from seven down to zero. As long as the upper bound is positive, you can substitute the JNS instruction in place of the JNZ instruction above to repeat the loop some specific number of times:

```
mov( 7, J );
LoopLbl:
  <<stmts>>
  dec( J );
  jns LoopLbl;
```

This loop will repeat eight times with *J* taking on the values seven down to zero on each execution of the loop. When it decrements zero to minus one, it sets the sign flag and the loop terminates.

Keep in mind that some values may look positive but they are negative. If the loop control variable is a byte, then values in the range 128..255 are negative. Likewise, 16-bit values in the range 32768..65535 are negative. Therefore, initializing the loop control variable with any value in the range 129..255 or 32769..65535 (or, of course, zero) will cause the loop to terminate after a single execution. This can get you into a lot of trouble if you're not careful.

2.12.3 Loop Invariant Computations

A loop invariant computation is some calculation that appears within a loop that always yields the same result. You needn't do such computations inside the loop. You can compute them outside the loop and reference the value of the computation inside the loop. The following HLA code demonstrates a loop which contains an invariant computation:

```
for( mov( 0, eax ); eax < n; inc( eax ) ) do

  mov( eax, edx );
  add( j, edx );
  sub( 2, edx );
  add( edx, k );

endfor;
```

Since *j* never changes throughout the execution of this loop, the sub-expression “*j*-2” can be computed outside the loop and its value used in the expression inside the loop:

```
mov( j, ecx );
sub( 2, ecx );
for( mov( 0, eax ); eax < n; inc( eax ) ) do

  mov( eax, edx );
  add( ecx, edx );
  add( edx, k );

endfor;
```

Still, the value in ECX never changes inside this loop, so although we've eliminated a single instruction by computing the subexpression “*j*-2” outside the loop, there is still an invariant component to this calculation. Since we note that this invariant component executes *n* times in the loop, we can translate the code above to the following:

```
mov( j, ecx );
sub( 2, ecx );
```

```

intmul( n, ecx ); // Compute n*(j-2) and add this into k outside
add( ecx, k ); // the loop.
for( mov( 0, eax ); eax < n; inc( eax ) ) do

    add( eax, k );

endfor;

```

As you can see, we've shrunk the loop body from four instructions down to one. Of course, if you're really interested in improving the efficiency of this particular loop, you'd be much better off (most of the time) computing k using the formula:

$$k = k + ((n + 1) \times temp) + \frac{(n + 2) \times (n + 2)}{2}$$

This computation for k is based on the formula:

$$\sum_{i=0}^n i = \frac{(n + 1) \times (n)}{2}$$

However, simple computations such as this one aren't always possible. Still, this demonstrates that a better algorithm is almost always better than the trickiest code you can come up with.

Removing invariant computations and unnecessary memory accesses from a loop (particularly an inner loop in a set of nested loops) can produce dramatic performance improvements in a program.

2.12.4 Unraveling Loops

For small loops, that is, those whose body is only a few statements, the overhead required to process a loop may constitute a significant percentage of the total processing time. For example, look at the following Pascal code and its associated 80x86 assembly language code:

```

FOR I := 3 DOWNTO 0 DO A [I] := 0;

mov( 3, I );
LoopLbl:
    mov( I, ebx );
    mov( 0, A[ebx*4] );
    dec( I );
    jns LoopLbl;

```

Each iteration of the loop requires four instructions. Only one instruction is performing the desired operation (moving a zero into an element of A). The remaining three instructions control the repetition of the loop. Therefore, it takes 16 instructions to do the operation logically required by four.

While there are many improvements we could make to this loop based on the information presented thus far, consider carefully exactly what it is that this loop is doing-- it's simply storing four zeros into $A[0]$ through $A[3]$. A more efficient approach is to use four MOV instructions to accomplish the same task. For example, if A is an array of dwords, then the following code initializes A much faster than the code above:

```

mov( 0, A[0] );
mov( 0, A[4] );
mov( 0, A[8] );
mov( 0, A[12] );

```

Although this is a trivial example, it shows the benefit of loop unraveling. If this simple loop appeared buried inside a set of nested loops, the 4:1 instruction reduction could possibly double the performance of that section of your program.

Of course, you cannot unravel all loops. Loops that execute a variable number of times cannot be unraveled because there is rarely a way to determine (at assembly time) the number of times the loop will execute.

Therefore, unraveling a loop is a process best applied to loops that execute a known number of times (and the number of times is known at assembly time).

Even if you repeat a loop some fixed number of iterations, it may not be a good candidate for loop unraveling. Loop unraveling produces impressive performance improvements when the number of instructions required to control the loop (and handle other overhead operations) represent a significant percentage of the total number of instructions in the loop. Had the loop above contained 36 instructions in the body of the loop (exclusive of the four overhead instructions), then the performance improvement would be, at best, only 10% (compared with the 300-400% it now enjoys). Therefore, the costs of unraveling a loop, i.e., all the extra code which must be inserted into your program, quickly reaches a point of diminishing returns as the body of the loop grows larger or as the number of iterations increases. Furthermore, entering that code into your program can become quite a chore. Therefore, loop unraveling is a technique best applied to small loops.

Note that the superscalar x86 chips (Pentium and later) have *branch prediction hardware* and use other techniques to improve performance. Loop unrolling on such systems many actually *slow down* the code since these processors are optimized to execute short loops.

2.12.5 Induction Variables

Consider the following modification of the loop presented in the previous section:

```
FOR I := 0 TO 255 DO csetVar[I] := {};
```

Here the program is initializing each element of an array of character sets to the empty set. The straight-forward code to achieve this is the following:

```
mov( 0, i );
FLp:

// Compute the index into the array (note that each element
// of a CSET array contains 16 bytes).

mov( i, ebx );
shl( 4, ebx );

// Set this element to the empty set (all zero bits).

mov( 0, csetVar[ ebx ] );
mov( 0, csetVar[ ebx+4 ] );
mov( 0, csetVar[ ebx+8 ] );
mov( 0, csetVar[ ebx+12 ] );

inc( i );
cmp( i, 256 );
jb FLp;
```

Although unraveling this code will still produce a performance improvement, it will take 1024 instructions to accomplish this task, too many for all but the most time-critical applications. However, you can reduce the execution time of the body of the loop using *induction variables*. An induction variable is one whose value depends entirely on the value of some other variable. In the example above, the index into the array *csetVar* tracks the loop control variable (it's always equal to the value of the loop control variable times 16). Since *i* doesn't appear anywhere else in the loop, there is no sense in performing all the computations on *i*. Why not operate directly on the array index value? The following code demonstrates this technique:

```
mov( 0, ebx );
FLp:
mov( 0, csetVar[ ebx ] );
mov( 0, csetVar[ ebx+4 ] );
mov( 0, csetVar[ ebx+8 ] );
```

```

mov( 0, csetVar[ ebx+12 ] );

add( 16, ebx );
cmp( ebx, 256*16 );
jb FLp;

```

The induction that takes place in this example occurs when the code increments the loop control variable (moved into EBX for efficiency reasons) by 16 on each iteration of the loop rather than by one. Multiplying the loop control variable by 16 allows the code to eliminate multiplying the loop control variable by 16 on each iteration of the loop (i.e., this allows us to remove the SHL instruction from the previous code). Further, since this code no longer refers to the original loop control variable (*i*), the code can maintain the loop control variable strictly in the EBX register.

2.13 Hybrid Control Structures in HLA

The HLA high level language control structures have a few drawbacks: (1) they're not true assembly language instructions, (2) complex boolean expressions only support short circuit evaluation, and (3) they often introduce inefficient coding practices into a language that most people only use when they need to write high-performance code. On the other hand, while the 80x86 low level control structures let you write efficient code, the resulting code is very difficult to read and maintain. HLA provides a set of hybrid control structures that allow you to use pure assembly language statements to evaluate boolean expressions while using the high level control structures to delineate the statements controlled by the boolean expressions. The result is code that is much more readable than pure assembly language without being a whole lot less efficient.

HLA provides hybrid forms of the IF.ELSEIF.ELSE..ENDIF, WHILE..ENDWHILE, REPEAT..UNTIL, BREAKIF, EXITIF, and CONTINUEIF statements (i.e., those that involve a boolean expression). For example, a hybrid IF statement takes the following form:

```

if( #{ <<statements>> }# ) then <<statements>> endif;

```

Note the use of #{ and }# operators to surround a sequence of instructions within this statement. This is what differentiates the hybrid control structures from the standard high level language control structures. The remaining hybrid control structures take the following forms:

```

while( #{ <<statements>> }# ) <<statements>> endwhile;
repeat <<statements>> until( #{ <<statements>> }# );
breakif( #{ <<statements>> }# );
exitif( #{ <<statements>> }# );
continueif( #{ <<statements>> }# );

```

The statements within the curly braces replace the normal boolean expression in an HLA high level control structure. These particular statements are special insofar as HLA defines two labels, *true* and *false*, within their context. HLA associates the label *true* with the code that would normally execute if a boolean expression were present and that expression's result was true. Similarly, HLA associates the label *false* with the code that would execute if a boolean expression in one of these statements evaluated false. As a simple example, consider the following two (equivalent) IF statements:

```

if( eax < ebx ) then inc( eax ); endif;

if
( #{
  cmp( eax, ebx );
  jnl false;
}# ) then
  inc( eax );

```

```
endif;
```

The JNL that transfers control to the *false* label in this latter example will skip over the INC instruction if EAX is not less than EBX. Note that if EAX is less than EBX then control falls through to the INC instruction. This is roughly equivalent to the following pure assembly code:

```
cmp( eax, ebx );
jnl falseLabel;
    inc( eax );
falseLabel:
```

As a slightly more complex example, consider the statement

```
if( eax >= J && eax <= K ) then sub( J, eax ); endif;
```

The following hybrid IF statement accomplishes the above:

```
if
( #{
    cmp( eax, J );
    jnge false;
    cmp( eax, K );
    jnle false;
}# ) then
    sub( J, eax );
endif;
```

As one final example of the hybrid IF statement, consider the following:

```
// if( ((eax > ebx) && (eax < ecx)) || (eax = edx) ) then mov( ebx, eax ); endif;

if
( #{
    cmp( eax, edx );
    je true;
    cmp( eax, ebx );
    jng false;
    cmp( eax, ecx );
    jnl false;
}# ) then
    mov( ebx, eax );
endif;
```

Since these examples are rather trivial, they don't really demonstrate how much more readable the code can be when using hybrid statements rather than pure assembly code. However, one thing you notice is that the use of hybrid statements eliminate the need to insert labels throughout your code. This is what makes your programs easier to read and understand.

For the IF statement, the *true* label corresponds to the THEN clause of the statement; the *false* label corresponds to the ELSEIF, ELSE, or ENDIF clause (whichever follows the THEN clause). For the WHILE loop, the *true* label corresponds to the body of the loop while the *false* label is attached to the first statement following the corresponding ENDWHILE. For the REPEAT.UNTIL statement, the *true* label is attached to the code following the UNTIL clause while the *false* label is attached to the first statement of the body of the loop. The BREAKIF, EXITIF, and CONTINUEIF statements associate the *false* label with the statement immediately following one of these statements, they associate the *true* label with the code normally associated with a BREAK, EXIT, or CONTINUE statement.

2.14 Putting It All Together

In this chapter we've taken a look at the low-level, or "pure" assembly language, implementation of several common control structures. Although HLA's high level control structures are easy to use and quite a bit more readable than their low-level equivalents, sometimes efficiency demands a low-level implementation. This chapter presents the blueprints for such transformations.

While this chapter covers the principle high level control structures and their translation to assembly language, there are some additional control structures that this chapter does not consider. Iterators and the TRY..ENDTRY statement are two good examples. Fear not, however, the volume on Advanced Assembly Language Programming will tidy up those loose ends.

Intermediate Procedures

Chapter Three

3.1 Chapter Overview

This chapter picks up where the chapter “Introduction to Procedures” in Volume Three leaves off. That chapter presented a high level view of procedures, parameters, and local variables; this chapter takes a look at some of the low-level implementation details. This chapter begins by discussing the CALL instruction and how it affects the stack. Then it discusses activation records and how a program passes parameters to a procedure and how that procedure maintains local (automatic) variables. Next, this chapter presents an in-depth discussion of pass by value and pass by reference parameters. This chapter concludes by discussing procedure variables, procedural parameters, iterators, and the FOREACH..ENDFOR loop.

3.2 Procedures and the CALL Instruction

Most procedural programming languages implement procedures using the call/return mechanism. That is, some code calls a procedure, the procedure does its thing, and then the procedure returns to the caller. The call and return instructions provide the 80x86’s *procedure invocation mechanism*. The calling code calls a procedure with the CALL instruction, the procedure returns to the caller with the RET instruction. For example, the following 80x86 instruction calls the HLA Standard Library *stdout.newln* routine:

```
call stdout.newln;
```

stdout.newln prints a line feed sequence to the video display and returns control to the instruction immediately following the “call stdout.newln;” instruction.

The HLA language lets you call procedures using a high level language syntax. Specifically, you may call a procedure by simply specifying the procedure’s name and (in the case of *stdout.newln*) an empty parameter list. That is, the following is completely equivalent to “call stdout.newln”:

```
stdout.newln();
```

The 80x86 CALL instruction does two things. First, it pushes the address of the instruction immediately following the CALL onto the stack; then it transfers control to the address of the specified procedure. The value that CALL pushes onto the stack is known as the *return address*. When the procedure wants to return to the caller and continue execution with the first statement following the CALL instruction, the procedure simply pops the return address off the stack and jumps (indirectly) to that address. Most procedures return to their caller by executing a RET (return) instruction. The RET instruction pops a return address off the stack and transfers control indirectly to the address it pops off the stack.

By default, the HLA compiler automatically places a RET instruction (along with a few other instructions) at the end of each HLA procedure you write. This is why you haven’t had to explicitly use the RET instruction up to this point. To disable the default code generation in an HLA procedure, specify the following options when declaring your procedures:

```
procedure ProcName; @noframe; @nodisplay;
begin ProcName;
    .
    .
    .
end ProcName;
```

The @NOFRAME and @NODISPLAY clauses are examples of procedure *options*. HLA procedures support several such options, including RETURNS (See “The HLA RETURNS Option in Procedures” on page 560.), the @NOFRAME, @NODISPLAY, and @NOALIGNSTACKK. You’ll see the purpose of @NOALIGNSTACK and a couple of other procedure options a little later in this chapter. These procedure options may appear in any order following the procedure name (and parameters, if any). Note that @NOF-

RAME and @NODISPLAY (as well as @NOALIGNSTACK) may only appear in an actual procedure declaration. You cannot specify these options in an external procedure prototype.

The @NOFRAME option tells HLA that you don't want the compiler to automatically generate entry and exit code for the procedure. This tells HLA not to automatically generate the RET instruction (along with several other instructions).

The @NODISPLAY option tells HLA that it should not allocate storage in procedure's local variable area for a *display*. The display is a mechanism you use to access non-local VAR objects in a procedure. Therefore, a display is only necessary if you nest procedures in your programs. This chapter will not consider the display or nested procedures; for more details on the display and nested procedures see the appropriate chapter in Volume Five. Until then, you can safely specify the @NODISPLAY option on all your procedures. Note that you may specify the @NODISPLAY option independently of the @NOFRAME option. Indeed, for all of the procedures appearing in this text up to this point specifying the @NODISPLAY option makes a lot of sense because none of those procedures have actually used the display. Procedures that have the @NODISPLAY option are a tiny bit faster and a tiny bit shorter than those procedures that do not specify this option.

The following is an example of the minimal procedure:

```
procedure minimal; nodisplay; noframe; noalignstk;
begin minimal;

    ret();

end minimal;
```

If you call this procedure with the CALL instruction, *minimal* will simply pop the return address off the stack and return back to the caller. You should note that a RET instruction is absolutely necessary when you specify the @NOFRAME procedure option¹. If you fail to put the RET instruction in the procedure, the program will not return to the caller upon encountering the "end minimal;" statement. Instead, the program will fall through to whatever code happens to follow the procedure in memory. The following example program demonstrates this problem:

```
program missingRET;
#include( "stdlib.hhf" );

// This first procedure has the NOFRAME
// option but does not have a RET instruction.

procedure firstProc; @noframe; @nodisplay;
begin firstProc;

    stdout.put( "Inside firstProc" nl );

end firstProc;

// Because the procedure above does not have a
// RET instruction, it will "fall through" to
// the following instruction. Note that there
// is no call to this procedure anywhere in
// this program.

procedure secondProc; @noframe; @nodisplay;
begin secondProc;
```

1. Strictly speaking, this isn't true. But some mechanism that pops the return address off the stack and jumps to the return address is necessary in the procedure's body.

```

        stdout.put( "Inside secondProc" nl );
        ret();

    end secondProc;

begin missingRET;

    // Call the procedure that doesn't have
    // a RET instruction.

    call firstProc;

end missingRET;

```

Program 3.1 Effect of Missing RET Instruction in a Procedure

Although this behavior might be desirable in certain rare circumstances, it usually represents a defect in most programs. Therefore, if you specify the @NOFRAME option, always remember to explicitly return from the procedure using the RET instruction.

3.3 Procedures and the Stack

Since procedures use the stack to hold the return address, you must exercise caution when pushing and popping data within a procedure. Consider the following simple (and defective) procedure:

```

procedure MessedUp; noframe; nodisplay;
begin MessedUp;

    push( eax );
    ret();

end MessedUp;

```

At the point the program encounters the RET instruction, the 80x86 stack takes the form shown in Figure 3.1:

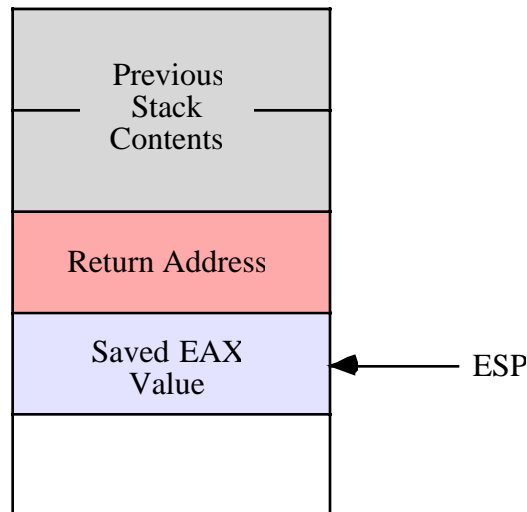


Figure 3.1 Stack Contents Before RET in “MessedUp” Procedure

The RET instruction isn’t aware that the value on the top of stack is not a valid address. It simply pops whatever value is on the top of the stack and jumps to that location. In this example, the top of stack contains the saved EAX value. Since it is very unlikely that EAX contains the proper return address (indeed, there is about a one in four billion chance it is correct), this program will probably crash or exhibit some other undefined behavior. Therefore, you must take care when pushing data onto the stack within a procedure that you properly pop that data prior to returning from the procedure.

Note: if you do not specify the @NOFRAME option when writing a procedure, HLA automatically generates code at the beginning of the procedure that pushes some data onto the stack. Therefore, unless you understand exactly what is going on and you’ve taken care of this data HLA pushes on the stack, you should never execute the bare RET instruction inside a procedure that does not have the @NOFRAME option. Doing so will attempt to return to the location specified by this data (which is not a return address) rather than properly returning to the caller. In procedures that do not have the @NOFRAME option, use the EXIT or EXITIF statements to return from the procedure (See “BEGIN..EXIT..EXITIF..END” on page 740.).

Popping extra data off the stack prior to executing the RET statement can also create havoc in your programs. Consider the following defective procedure:

```
procedure MessedUpToo; noframe; nodisplay;
begin MessedUpToo;

    pop( eax );
    ret();

end MessedUpToo;
```

Upon reaching the RET instruction in this procedure, the 80x86 stack looks something like that shown in Figure 3.2:

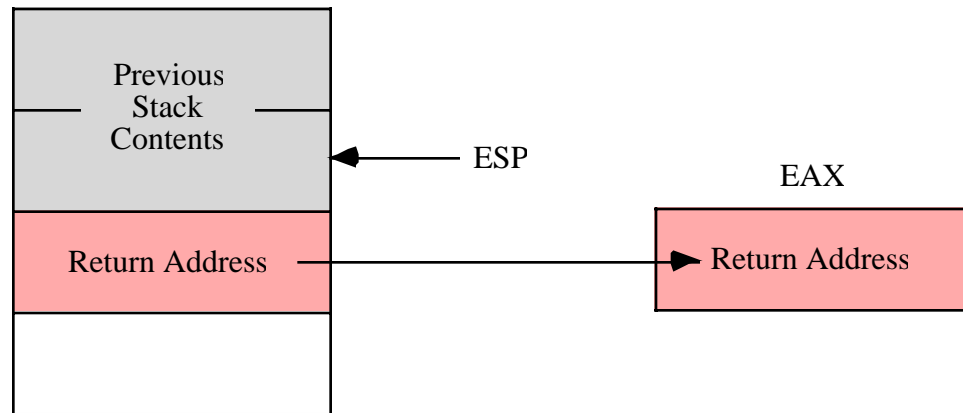


Figure 3.2 Stack Contents Before RET in MessedUpToo

Once again, the RET instruction blindly pops whatever data happens to be on the top of the stack and attempts to return to that address. Unlike the previous example, where it was very unlikely that the top of stack contained a valid return address (since it contained the value in EAX), there is a small possibility that the top of stack in this example actually *does* contain a return address. However, this will not be the proper return address for the *MessedUpToo* procedure; instead, it will be the return address for the procedure that called *MessedUpToo*. To understand the effect of this code, consider the following program:

```

program extraPop;
#include( "stdlib.hhf" );

// Note that the following procedure pops
// excess data off the stack (in this case,
// it pops messedUpToo's return address).

procedure messedUpToo; @noframe; @nodisplay;
begin messedUpToo;

    stdout.put( "Entered messedUpToo" nl );
    pop( eax );
    ret();

end messedUpToo;

procedure callsMU2; @noframe; @nodisplay;
begin callsMU2;

    stdout.put( "calling messedUpToo" nl );
    messedUpToo();

    // Because messedUpToo pops extra data
    // off the stack, the following code
    // never executes (since the data popped
    // off the stack is the return address that
    // points at the following code.

```

```

        stdout.put( "Returned from messedUpToo" nl );
        ret();

    end callsMU2;

begin extraPop;

    stdout.put( "Calling callsMU2" nl );
    callsMU2();
    stdout.put( "Returned from callsMU2" nl );

end extraPop;

```

Program 3.2 Effect of Popping Too Much Data Off the Stack

Since a valid return address is sitting on the top of the stack, you might think that this program will actually work (properly). However, note that when returning from the *MessedUpToo* procedure, this code returns directly to the main program rather than to the proper return address in the *EndSkipped* procedure. Therefore, all code in the *callsMU2* procedure that follows the call to *MessedUpToo* does not execute. When reading the source code, it may be very difficult to figure out why those statements are not executing since they immediately follow the call to the *MessUpToo* procedure. It isn't clear, unless you look very closely, that the program is popping an extra return address off the stack and, therefore, doesn't return back to *callsMU2* but, rather, returns directly to whomever calls *callsMU2*. Of course, in this example it's fairly easy to see what is going on (because this example is a demonstration of this problem). In real programs, however, determining that a procedure has accidentally popped too much data off the stack can be much more difficult. Therefore, you should always be careful about pushing and popping data in a procedure. You should always verify that there is a one-to-one relationship between the pushes in your procedures and the corresponding pops.

3.4 Activation Records

Whenever you call a procedure there is certain information the program associates with that procedure call. The return address is a good example of some information the program maintains for a specific procedure call. Parameters and automatic local variables (i.e., those you declare in the VAR section) are additional examples of information the program maintains for each procedure call. *Activation record* is the term we'll use to describe the information the program associates with a specific call to a procedure².

Activation record is an appropriate name for this data structure. The program creates an activation record when calling (activating) a procedure and the data in the structure is organized in a manner identical to records (see "Records" on page 483). Perhaps the only thing unusual about an activation record (when comparing it to a standard record) is that the base address of the record is in the middle of the data structure, so you must access fields of the record at positive and negative offsets.

Construction of an activation record begins in the code that calls a procedure. The caller pushes the parameter data (if any) onto the stack. Then the execution of the CALL instruction pushes the return address onto the stack. At this point, construction of the activation record continues within the procedure itself. The procedure pushes registers and other important state information and then makes room in the activation record for local variables. The procedure must also update the EBP register so that it points at the base address of the activation record.

2. Stack frame is another term many people use to describe the activation record.

To see what a typical activation record looks like, consider the following HLA procedure declaration:

```
procedure ARDemo( i:uns32; j:int32; k:dword ); nodisplay;
var
  a:int32;
  r:real32;
  c:char;
  b:boolean;
  w:word;
begin ARDemo;
  .
  .
  .
end ARDemo;
```

Whenever an HLA program calls this *ARDemo* procedure, it begins by pushing the data for the parameters onto the stack. The calling code will push the parameters onto the stack in the order they appear in the parameter list, from left to right. Therefore, the calling code first pushes the value for the *i* parameter, then it pushes the value for the *j* parameter, and it finally pushes the data for the *k* parameter. After pushing the parameters, the program calls the *ARDemo* procedure. Immediately upon entry into the *ARDemo* procedure, the stack contains these four items arranged as shown in Figure 3.3

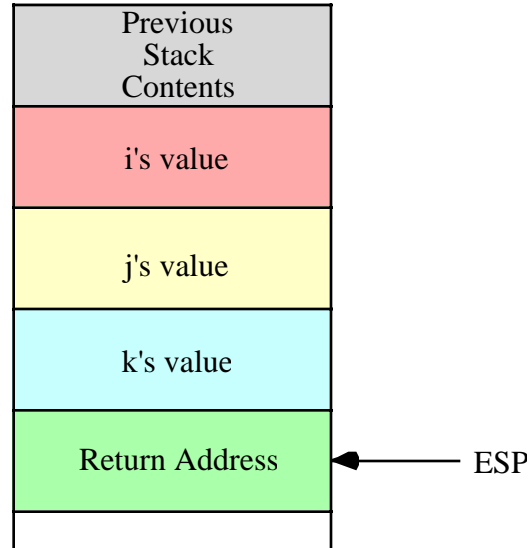


Figure 3.3 Stack Organization Immediately Upon Entry into *ARDemo*

The first few instructions in *ARDemo* (note that it does not have the `@NOFRAME` option) will push the current value of `EBP` onto the stack and then copy the value of `ESP` into `EBP`. Next, the code drops the stack pointer down in memory to make room for the local variables. This produces the stack organization shown in Figure 3.4

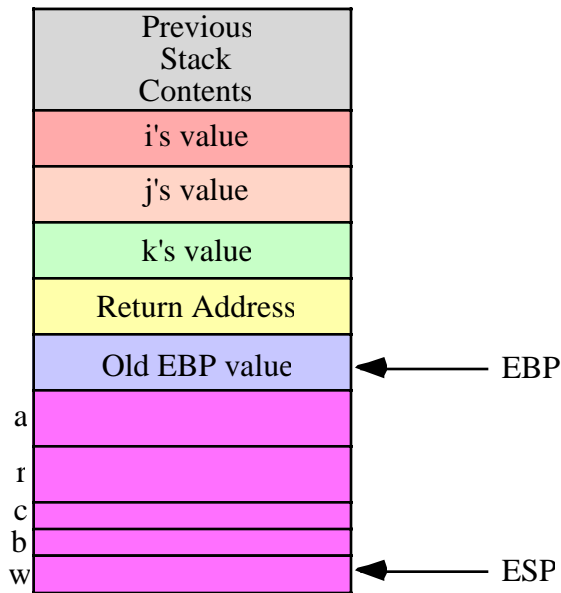


Figure 3.4 Activation Record for ARDemo

To access objects in the activation record you must use offsets from the EBP register to the desired object. The two items of immediate interest to you are the parameters and the local variables. You can access the parameters at positive offsets from the EBP register, you can access the local variables at negative offsets from the EBP register as Figure 3.5 shows:

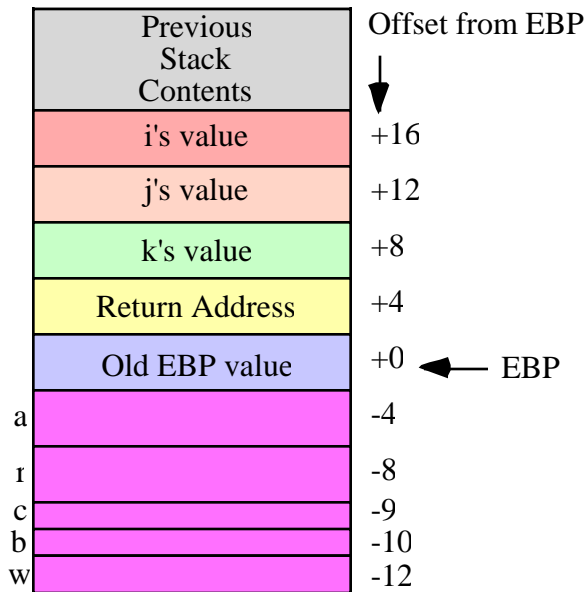


Figure 3.5 Offsets of Objects in the ARDemo Activation Record

Intel specifically reserves the EBP (extended base pointer) for use as a pointer to the base of the activation record. This is why you should never use the EBP register for general calculations. If you arbitrarily

change the value in the EBP register you will lose access to the current procedure's parameters and local variables.

3.5 The Standard Entry Sequence

The caller of a procedure is responsible for pushing the parameters onto the stack. Of course, the CALL instruction pushes the return address onto the stack. It is the procedure's responsibility to construct the rest of the activation record. This is typically accomplished by the following "standard entry sequence" code:

```
push( ebp );           // Save a copy of the old EBP value
mov( esp, ebp );      // Get ptr to base of activation record into EBP
sub( NumVars, esp );  // Allocate storage for local variables.
```

If the procedure doesn't have any local variables, the third instruction above, "sub(NumVars, esp);" isn't needed. *NumVars* represents the number of bytes of local variables needed by the procedure. This is a constant that should be an even multiple of four (so the ESP register remains aligned on a double word boundary). If the number of bytes of local variables in the procedure is not an even multiple of four, you should round the value up to the next higher multiple of four before subtracting this constant from ESP. Doing so will slightly increase the amount of storage the procedure uses for local variables but will not otherwise affect the operation of the procedure.

Warning: if the *NumVars* constant is not an even multiple of four, subtracting this value from ESP (which, presumably, contains a dword-aligned pointer) will virtually guarantee that all future stack accesses are misaligned since the program almost always pushes and pops dword values. This will have a very negative performance impact on the program. Worse still, many OS API calls will fail if the stack is not dword-aligned upon entry into the operating system. Therefore, you must always ensure that your local variable allocation value is an even multiple of four.

Because of the problems with a misaligned stack, by default HLA will also emit a fourth instruction as part of the standard entry sequence. The HLA compiler actually emits the following standard entry sequence for the ARDemo procedure defined earlier:

```
push( ebp );
mov( esp, ebp );
sub( 12, esp );           // Make room for ARDemo's local variables.
and( $FFFF_FFC, esp );  // Force dword stack alignment.
```

The AND instruction at the end of this sequence forces the stack to be aligned on a four-byte boundary (it reduces the value in the stack pointer by one, two, or three if the value in ESP is not an even multiple of four). Although the *ARDemo* entry code correctly subtracts 12 from ESP for the local variables (12 is both an even multiple of four and the number of bytes of local variables), this only leaves ESP double word aligned if it was double word aligned immediately upon entry into the procedure. Had the caller messed with the stack and left ESP containing a value that was not an even multiple of four, subtracting 12 from ESP would leave ESP containing an unaligned value. The AND instruction in the sequence above, however, guarantees that ESP is dword aligned regardless of ESP's value upon entry into the procedure. The few bytes and CPU cycles needed to execute this instruction pay off handsomely if ESP is not double word aligned.

Although it is always safe to execute the AND instruction in the standard entry sequence, it might not be necessary. If you always ensure that ESP contains a double word aligned value, the AND instruction in the standard entry sequence above is unnecessary. Therefore, if you've specified the @NOFRAME procedure option, you don't have to include that instruction as part of the entry sequence.

If you haven't specified the @NOFRAME option (i.e., you're letting HLA emit the instructions to construct the standard entry sequence for you), you can still tell HLA not to emit the extra AND instruction if you're sure the stack will be dword aligned whenever someone calls the procedure. To do this, use the @NOALIGNSTACK procedure option, e.g.,

```
procedure NASDemo( i:uns32; j:int32; k:dword ); @noalignstack;
```

```

var
  LocalVar:int32;
begin NASDemo;
  .
  .
end NASDemo;

```

HLA emits the following entry sequence for the procedure above:

```

push( ebp );
mov( esp, ebp );
sub( 4, esp );

```

3.6 The Standard Exit Sequence

Before a procedure returns to its caller, it needs to clean up the activation record. Although it is possible to share the clean-up duties between the procedure and the procedure's caller, Intel has included some features in the instruction set that allows the procedure to efficiently handle all the clean up chores itself. Standard HLA procedures and procedure calls, therefore, assume that it is the procedure's responsibility to clean up the activation record (including the parameters) when the procedure returns to its caller.

If a procedure does not have any parameters, the calling sequence is very simple. It requires only three instructions:

```

mov( ebp, esp ); // Deallocate locals and clean up stack.
pop( ebp );     // Restore pointer to caller's activation record.
ret();         // Return to the caller.

```

If the procedure has some parameters, then a slight modification to the standard exit sequence is necessary in order to remove the parameter data from the stack. Procedures with parameters use the following standard exit sequence:

```

mov( ebp, esp ); // Deallocate locals and clean up stack.
pop( ebp );     // Restore pointer to caller's activation record.
ret( ParmBytes ); // Return to the caller and pop the parameters.

```

The *ParmBytes* operand of the RET instruction is a constant that specifies the number of *bytes* of parameter data to remove from the stack after the return instruction pops the return address. For example, the ARDemo example code in the previous sections has three double word parameters. Therefore, the standard exit sequence would take the following form:

```

mov( ebp, esp );
pop( ebp );
ret( 12 );

```

If you've declared your parameters using HLA syntax (i.e., a parameter list follows the procedure declaration), then HLA automatically creates a local constant in the procedure, *_parms_*, that is equal to the number of bytes of parameters in that procedure. Therefore, rather than worrying about having to count the number of parameter bytes yourself, you can use the following standard exit sequence for any procedure that has parameters:

```

mov( ebp, esp );
pop( ebp );
ret( _parms_ );

```

Note that if you do not specify a byte constant operand to the RET instruction, the 80x86 will not pop the parameters off the stack upon return. Those parameters will still be sitting on the stack when you execute the first instruction following the CALL to the procedure. Similarly, if you specify a value that is too

small, some of the parameters will be left on the stack upon return from the procedure. If the RET operand you specify is too large, the RET instruction will actually pop some of the caller's data off the stack, usually with disastrous consequences.

Note that if you wish to return early from a procedure that doesn't have the @NOFRAME option, and you don't particularly want to use the EXIT or EXITIF statement, you must execute the standard exit sequence to return to the caller. A simple RET instruction is insufficient since local variables and the old EBP value are probably sitting on the top of the stack.

3.7 HLA Local Variables

Your program accesses local variables in a procedure by using negative offsets from the activation record base address (EBP). For example, consider the following HLA procedure (which admittedly, doesn't do much other than demonstrate the use of local variables):

```
procedure LocalVars; nodisplay;
var
  a:int32;
  b:int32;
begin LocalVars;

  mov( 0, a );
  mov( a, eax );
  mov( eax, b );

end LocalVars;
```

The activation record for LocalVars looks like

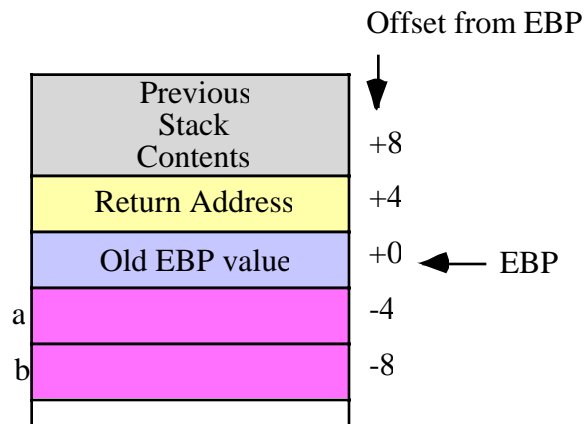


Figure 3.6 Activation Record for LocalVars Procedure

The HLA compiler emits code that is roughly equivalent to the following for the body of this procedure³:

```
mov( 0, (type dword [ebp-4]));
mov( [ebp-4], eax );
mov( eax, [ebp-8] );
```

3. Ignoring the code associated with the standard entry and exit sequences.

You could actually type these statements into the procedure yourself and they would work. Of course, using memory references like “[ebp-4]” and “[ebp-8]” rather than *a* or *b* makes your programs very difficult to read and understand. Therefore, you should always declare and use HLA symbolic names rather than offsets from EBP.

The standard entry sequence for this *LocalVars* procedure will be⁴

```
push( ebp );
mov( esp, ebp );
sub( 8, esp );
```

This code subtracts eight from the stack pointer because there are eight bytes of local variables (two dword objects) in this procedure. Unfortunately, as the number of local variables increases, especially if those variables have different types, computing the number of bytes of local variables becomes rather tedious. Fortunately, for those who wish to write the standard entry sequence themselves, HLA automatically computes this value for you and creates a constant, `_vars_`, that specifies the number of bytes of local variables for you⁵. Therefore, if you intend to write the standard entry sequence yourself, you should use the `_vars_` constant in the SUB instruction when allocating storage for the local variables:

```
push( ebp );
mov( esp, ebp );
sub( _vars_, esp );
```

Now that you’ve seen how assembly language (and, indeed, most languages) allocate and deallocate storage for local variables, it’s easy to understand why automatic (local VAR) variables do not maintain their values between two calls to the same procedure. Since the memory associated with these automatic variables is on the stack, when a procedure returns to its caller the caller can push other data onto the stack obliterating the values of the local variable values previously held on the stack. Furthermore, intervening calls to other procedures (with their own local variables) may wipe out the values on the stack. Also, upon reentry into a procedure, the procedure’s local variables may correspond to different physical memory locations, hence the values of the local variables would not be in their proper locations.

One big advantage to automatic storage is that it efficiently shares a fixed pool of memory among several procedures. For example, if you call three procedures in a row,

```
ProcA();
ProcB();
ProcC();
```

The first procedure (*ProcA* in the code above) allocates its local variables on the stack. Upon return, *ProcA* deallocates that stack storage. Upon entry into *ProcB*, the program allocates storage for *ProcB*’s local variables *using the same memory locations just freed by ProcA*. Likewise, when *ProcB* returns and the program calls *ProcC*, *ProcC* uses the same stack space for its local variables that *ProcB* recently freed up. This memory reuse makes efficient use of the system resources and is probably the greatest advantage to using automatic (VAR) variables.

3.8 Parameters

Although there is a large class of procedures that are totally self-contained, most procedures require some input data and return some data to the caller. Parameters are values that you pass to and from a procedure. There are many facets to parameters. Questions concerning parameters include:

4. This code assumes that ESP is dword aligned upon entry so the “AND(\$FFFF_FFC, ESP);” instruction is unnecessary.

5. HLA even rounds this constant up to the next even multiple of four so you don’t have to worry about stack alignment.

- *where* is the data coming from?
- *what* mechanism do you use to pass and return data?
- *how* much data are you passing?

In this chapter we will take another look at the two most common parameter passing mechanisms: pass by value and pass by reference. We will also discuss three popular places to pass parameters: in the registers, on the stack, and in the code stream. The amount of parameter data has a direct bearing on where and how to pass it. The following sections take up these issues.

3.8.1 Pass by Value

A parameter passed by value is just that – the caller passes a value to the procedure. Pass by value parameters are input only parameters. That is, you can pass them to a procedure but the procedure cannot return values through them. In high level languages the idea of a pass by value parameter being an input only parameter makes a lot of sense. Given the procedure call:

```
CallProc(I);
```

If you pass *I* by value, *CallProc* does not change the value of *I*, regardless of what happens to the parameter inside *CallProc*.

Since you must pass a copy of the data to the procedure, you should only use this method for passing small objects like bytes, words, and double words. Passing arrays and strings by value is very inefficient (since you must create and pass a copy of the structure to the procedure).

3.8.2 Pass by Reference

To pass a parameter by reference you must pass the address of a variable rather than its value. In other words, you must pass a pointer to the data. The procedure must dereference this pointer to access the data. Passing parameters by reference is useful when you must modify the actual parameter or when you pass large data structures between procedures.

Passing parameters by reference can produce some peculiar results. The following Pascal procedure provides an example of one problem you might encounter:

```
program main(input,output);
var m:integer;

  (*
  ** Note: this procedure passes i and j by reference.
  *)

  procedure bletch(var i,j:integer);
  begin
    i := i+2;
    j := j-i;
    writeln(i,' ',j);
  end;

  .
  .
  .

begin {main}
  m := 5;
  bletch(m,m);
end.
```

This particular code sequence will print “00” regardless of m 's value. This is because the parameters i and j are pointers to the actual data and they both point at the same object (that is, they are aliases). Therefore, the statement “ $j:=j-i$;” always produces zero since i and j refer to the same variable.

Pass by reference is usually less efficient than pass by value. You must dereference all pass by reference parameters on each access; this is slower than simply using a value. However, when passing a large data structure, pass by reference is faster because you do not have to copy a large data structure before calling the procedure.

3.8.3 Passing Parameters in Registers

Having touched on how to pass parameters to a procedure, the next thing to discuss is *where* to pass parameters. Where you pass parameters depends on the size and number of those parameters. If you are passing a small number of bytes to a procedure, then the registers are an excellent place to pass parameters to a procedure. If you are passing a single parameter to a procedure you should use the following registers for the accompanying data types:

Data Size	Pass in this Register
Byte:	al
Word:	ax
Double Word:	eax
Quad Word:	edx:eax

This is not a hard and fast rule. If you find it more convenient to pass 16 bit values in the SI or BX register, do so. However, most programmers use the registers above to pass parameters.

If you are passing several parameters to a procedure in the 80x86's registers, you should probably use up the registers in the following order:

First	Last
	eax, edx, esi, edi, ebx, ecx

In general, you should avoid using EBP register. If you need more than six double words, perhaps you should pass your values elsewhere.

As an example, consider the following “strfill(str,c;” that copies the character c (passed by value in AL) to each character position in s (passed by reference in EDI) up to a zero terminating byte:

```
// strfill- Overwrites the data in a string with a character.
//
// EDI- pointer to zero terminated string (e.g., an HLA string)
// AL- character to store into the string.

procedure strfill; nodisplay;
begin strfill;

    push( edi ); // Preserve this because it will be modified.
    while( (type char [edi] <> #0 ) do

        mov( al, [edi] );
        inc( edi );

    endwhile;
    pop( edi );

end strfill;
```

To call the *strfill* procedure you would load the address of the string data into EDI and the character value into AL prior to the call. The following code fragment demonstrates a typical call to *strfill*:

```
mov( s, edi ); // Get ptr to string data into edi (assumes s:string).
mov( '\ ', al );
strfill();
```

Don't forget that HLA string variables are pointers. This example assumes that *s* is a HLA string variable and, therefore, contains a pointer to a zero-terminated string. Therefore, the “mov(*s*, edi);” instruction loads the address of the zero terminated string into the EDI register (hence this code passes the address of the string data to *strfill*, that is, it passes the string by reference).

One way to pass parameters in the registers is to simply load the registers with the appropriate values prior to a call and then reference the values in those registers within the procedure. This is the traditional mechanism for passing parameters in registers in an assembly language program. HLA, being somewhat more high level than traditional assembly language, provides a formal parameter declaration syntax that lets you tell HLA you're passing certain parameters in the general purpose registers. This declaration syntax is the following:

```
parmName: parmType in reg
```

Where *parmName* is the parameter's name, *parmType* is the type of the object, and *reg* is one of the 80x86's general purpose eight, sixteen, or thirty-two bit registers. The size of the parameter's type must be equal to the size of the register or HLA will generate an error. Here is a concrete example:

```
procedure HasRegParms( count: uns32 in ecx; charVal:char in al );
```

One nice feature to this syntax is that you can call a procedure that has register parameters exactly like any other procedure in HLA using the high level syntax, e.g.,

```
HasRegParms( ecx, bl );
```

If you specify the same register as an actual parameter that you've declared for the formal parameter, HLA does not emit any extra code; it assumes that the parameter is already in the appropriate register. For example, in the call above the first actual parameter is the value in ECX; since the procedure's declaration specifies that that first parameter is in ECX HLA will not emit any code. On the other hand, the second actual parameter is in BL while the procedure will expect this parameter value in AL. Therefore, HLA will emit a “mov(bl, al);” instruction prior to calling the procedure so that the value is in the proper register upon entry to the procedure.

You can also pass parameters by reference in a register. Consider the following declaration:

```
procedure HasRefRegParm( var myPtr:uns32 in edi );
```

A call to this procedure always requires some memory operand as the actual parameter. HLA will emit the code to load the address of that memory object into the parameter's register (EDI in this case). Note that when passing reference parameters, the register must be a 32-bit general purpose register since addresses are 32-bits long. Here's an example of a call to *HasRefRegParm*:

```
HasRefRegParm( x );
```

HLA will emit either a “mov(&x, edi);” or “lea(edi, x);” instruction to load the address of *x* into the EDI registers prior to the CALL instruction⁶.

If you pass an anonymous memory object (e.g., “[edi]” or “[ecx]”) as a parameter to *HasRefRegParm*, HLA will not emit any code if the memory reference uses the same register that you declare for the parameter (i.e., “[edi]”). It will use a simple MOV instruction to copy the actual address into EDI if you specify an indirect addressing mode using a register other than EDI (e.g., “[ecx]”). It will use an LEA instruction to compute the effective address of the anonymous memory operand if you use a more complex addressing mode like “[edi+ecx*4+2]”.

6. The choice of instructions is dictated by whether *x* is a static variable (MOV for static objects, LEA for other objects).

Within the procedure's code, HLA creates text equates for these register parameters that map their names to the appropriate register. In the *HasRegParms* example, any time you reference the *count* parameter, HLA substitutes "ecx" for *count*. Likewise, HLA substitutes "al" for *charVal* throughout the procedure's body. Since these names are aliases for the registers, you should take care to always remember that you cannot use ECX and AL independently of these parameters. It would be a good idea to place a comment next to each use of these parameters to remind the reader that *count* is equivalent to ECX and *charVal* is equivalent to AL.

3.8.4 Passing Parameters in the Code Stream

Another place where you can pass parameters is in the code stream immediately after the CALL instruction. Consider the following *print* routine that prints a literal string constant to the standard output device:

```
call print;
byte "This parameter is in the code stream.",0;
```

Normally, a subroutine returns control to the first instruction immediately following the CALL instruction. Were that to happen here, the 80x86 would attempt to interpret the ASCII codes for "This..." as an instruction. This would produce undesirable results. Fortunately, you can skip over this string when returning from the subroutine.

So how do you gain access to these parameters? Easy. The return address on the stack points at them. Consider the following implementation of *print*:

```
program printDemo;
#include( "stdlib.hhf" );

// print-
//
// This procedure writes the literal string
// immediately following the call to the
// standard output device. The literal string
// must be a sequence of characters ending with
// a zero byte (i.e., a C string, not an HLA
// string).

procedure print; @noframe; @nodisplay;
const

    // RtnAdrs is the offset of this procedure's
    // return address in the activation record.

    RtnAdrs:text := "(type dword [ebp+4])";

begin print;

    // Build the activation record (note the
    // "@noframe" option above).

    push( ebp );
    mov( esp, ebp );

    // Preserve the registers this function uses.

    push( eax );
    push( ebx );

    // Copy the return address into the EBX
```

```

// register. Since the return address points
// at the start of the string to print, this
// instruction loads EBX with the address of
// the string to print.

mov( RtnAdrs, ebx );

// Until we encounter a zero byte, print the
// characters in the string.

forever

    mov( [ebx], al ); // Get the next character.
    breakif( !al ); // Quit if it's zero.
    stdout.putc( al ); // Print it.
    inc( ebx ); // Move on to the next char.

endfor;

// Skip past the zero byte and store the resulting
// address over the top of the return address so
// we'll return to the location that is one byte
// beyond the zero terminating byte of the string.

inc( ebx );
mov( ebx, RtnAdrs );

// Restore EAX and EBX.

pop( ebx );
pop( eax );

// Clean up the activation record and return.

pop( ebp );
ret();

end print;

begin printDemo;

// Simple test of the print procedure.

call print;
byte "Hello World!", 13, 10, 0 ;

end printDemo;

```

Program 3.3 Print Procedure Implementation (Using Code Stream Parameters)

Besides showing how to pass parameters in the code stream, the *print* routine also exhibits another concept: *variable length parameters*. The string following the CALL can be any practical length. The zero terminating byte marks the end of the parameter list. There are two easy ways to handle variable length parameters. Either use some special terminating value (like zero) or you can pass a special length value that tells the subroutine how many parameters you are passing. Both methods have their advantages and disadvantages. Using a special value to terminate a parameter list requires that you choose a value that never

appears in the list. For example, *print* uses zero as the terminating value, so it cannot print the NUL character (whose ASCII code is zero). Sometimes this isn't a limitation. Specifying a special length parameter is another mechanism you can use to pass a variable length parameter list. While this doesn't require any special codes or limit the range of possible values that can be passed to a subroutine, setting up the length parameter and maintaining the resulting code can be a real nightmare⁷.

Despite the convenience afforded by passing parameters in the code stream, there are some disadvantages to passing parameters there. First, if you fail to provide the exact number of parameters the procedure requires, the subroutine will get very confused. Consider the *print* example. It prints a string of characters up to a zero terminating byte and then returns control to the first instruction following the zero terminating byte. If you leave off the zero terminating byte, the *print* routine happily prints the following opcode bytes as ASCII characters until it finds a zero byte. Since zero bytes often appear in the middle of an instruction, the *print* routine might return control into the middle of some other instruction. This will probably crash the machine. Inserting an extra zero, which occurs more often than you might think, is another problem programmers have with the *print* routine. In such a case, the *print* routine would return upon encountering the first zero byte and attempt to execute the following ASCII characters as machine code. Once again, this usually crashes the machine. These are the some of the reasons why the HLA *stdout.put* code does *not* pass its parameters in the code stream. Problems notwithstanding, however, the code stream is an efficient place to pass parameters whose values do not change.

3.8.5 Passing Parameters on the Stack

Most high level languages use the stack to pass parameters because this method is fairly efficient. By default, HLA also passes parameters on the stack. Although passing parameters on the stack is slightly less efficient than passing those parameters in registers, the register set is very limited and you can only pass a few value or reference parameters through registers. The stack, on the other hand, allows you to pass a large amount of parameter data without any difficulty. This is the principal reason that most programs pass their parameters on the stack.

HLA passes parameters you specify in a high-level language form on the stack. For example, suppose you define *strfill* from the previous section as follows:

```
procedure strfill( s:string; chr:char );
```

Calls of the form “*strfill*(*s*, ‘ ‘);” will pass the value of *s* (which is an address) and a space character on the 80x86 stack. When you specify a call to *strfill* in this manner, HLA automatically pushes the parameters for you, so you don't have to push them onto the stack yourself. Of course, if you choose to do so, HLA will let you manually push the parameters onto the stack prior to the call.

To manually pass parameters on the stack, push them immediately before calling the subroutine. The subroutine then reads this data from the stack memory and operates on it appropriately. Consider the following HLA procedure call:

```
CallProc( i, j, k );
```

HLA pushes parameters onto the stack in the order that they appear in the parameter list⁸. Therefore, the 80x86 code HLA emits for this subroutine call (assuming you're passing the parameters by value) is

```
push( i );
push( j );
push( k );
call CallProc;
```

Upon entry into *CallProc*, the 80x86's stack looks like that shown in Figure 3.7:

7. Especially if the parameter list changes frequently.

8. Assuming, of course, that you don't instruct HLA otherwise. It is possible to tell HLA to reverse the order of the parameters on the stack. See the chapter on “Mixed Language Programming” for more details.

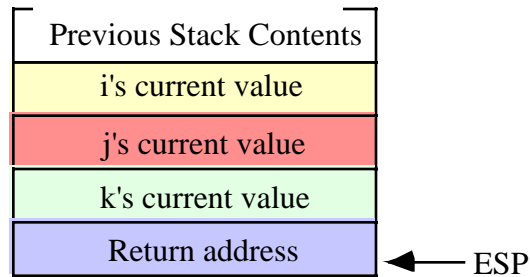


Figure 3.7 Stack Layout Upon Entry into CallProc

You could gain access to the parameters passed on the stack by removing the data from the stack as the following code fragment demonstrates:

```
// Note: to extract parameters off the stack by popping it is very important
// to specify both the @nodisplay and @noframe procedure options.

static
  RtnAdrs: dword;
  p1Parm: dword;
  p2Parm: dword;
  p3Parm: dword;

procedure CallProc( p1:dword; p2:dword; p3:dword ); @nodisplay; @noframe;
begin CallProc;

  pop( RtnAdrs );
  pop( p3Parm );
  pop( p2Parm );
  pop( p1Parm );
  push( RtnAdrs );
  .
  .
  .
  ret();

end CallProc;
```

As you can see from this code, it first pops the return address off the stack and into the *RtnAdrs* variable; then it pops (in reverse order) the values of the *p1*, *p2*, and *p3* parameters; finally, it pushes the return address back onto the stack (so the RET instruction will operate properly). Within the *CallProc* procedure, you may access the *p1Parm*, *p2Parm*, and *p3Parm* variables to use the *p1*, *p2*, and *p3* parameter values.

There is, however, a better way to access procedure parameters. If your procedure includes the standard entry and exit sequences (see “The Standard Entry Sequence” on page 813 and “The Standard Exit Sequence” on page 814), then you may directly access the parameter values in the activation record by indexing off the EBP register. Consider the layout of the activation record for *CallProc* that uses the following declaration:

```
procedure CallProc( p1:dword; p2:dword; p3:dword ); @nodisplay; @noframe;
begin CallProc;

  push( ebp ); // This is the standard entry sequence.
  mov( esp, ebp ); // Get base address of A.R. into EBP.
```

·
·
·

Take a look at the stack immediately after the execution of “`mov(esp, ebp);`” in *CallProc*. Assuming you’ve pushed three double word parameters onto the stack, it should look something like shown in Figure 3.8:

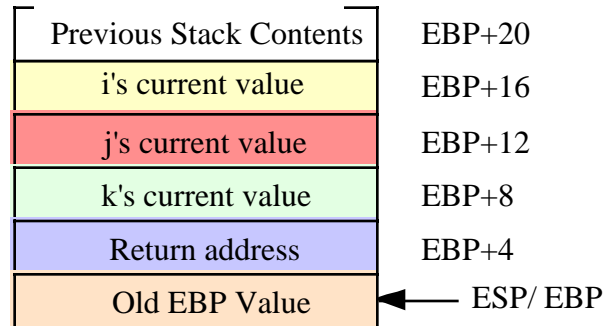


Figure 3.8 Activation Record for CallProc After Standard Entry Sequence Execution

.Now you can access the parameters by indexing off the EBP register:

```
mov( [ebp+16], eax ); // Accesses the first parameter.
mov( [ebp+12], ebx ); // Accesses the second parameter.
mov( [ebp+8], ecx ); // Accesses the third parameter.
```

Of course, like local variables, you’d never really access the parameters in this way. You can use the formal parameter names (*p1*, *p2*, and *p3*) and HLA will substitute a suitable “[*ebp+displacement*]” memory address. Even though you shouldn’t actually access parameters using address expressions like “[*ebp+12*]” it’s important to understand their relationship to the parameters in your procedures.

Other items that often appear in the activation record are register values your procedure preserves. The most rational place to preserve registers in a procedure is in the code immediately following the standard entry sequence. In a standard HLA procedure (one where you do not specify the `NOFRAME` option), this simply means that the code that preserves the registers should appear first in the procedure’s body. Likewise, the code to restore those register values should appear immediately before the `END` clause for the procedure⁹.

3.8.5.1 Accessing Value Parameters on the Stack

Accessing parameters passed by value is no different than accessing a local `VAR` object. As long as you’ve declared the parameter in a formal parameter list and the procedure executes the standard entry sequence upon entry into the program, all you need do is specify the parameter’s name to reference the value of that parameter. The following is an example program whose procedure accesses a parameter the main program passes to it by value:

```
program AccessingValueParameters;
```

9. Note that if you use the `EXIT` statement to exit a procedure, you must duplicate the code to pop the register values and place this code immediately before the `EXIT` clause. This is a good example of a maintenance nightmare and is also a good reason why you should only have one exit point in your program.


```

#include( "stdlib.hhf" )

procedure ValueParm( theParameter: uns32 ); @nodisplay;
begin ValueParm;

    mov( theParameter, eax );
    add( 2, eax );
    stdout.put
    (
        "theParameter + 2 = ",
        (type uns32 eax),
        nl
    );

end ValueParm;

begin AccessingValueParameters;

    ValueParm( 10 );
    ValueParm( 135 );

end AccessingValueParameters;

```

Program 3.4 Demonstration of Value Parameters

Although you may access the value of *theParameter* using the anonymous address “[EBP+8]” within your code, there is absolutely no good reason for doing so. If you declare the parameter list using the HLA high level language syntax, you can access the value parameter by specifying its name within the procedure.

3.8.5.2 Passing Value Parameters on the Stack

As Program 3.4 demonstrates, passing a value parameter to a procedure is very easy. Just specify the value in the actual parameter list as you would for a high level language call. Actually, the situation is a little more complicated than this. Passing value parameters is easy if you’re passing constant, register, or variable values. It gets a little more complex if you need to pass the result of some expression. This section deals with the different ways you can pass a parameter by value to a procedure.

Of course, you do not have to use the HLA high level language syntax to pass value parameters to a procedure. You can push these values on the stack yourself. Since there are many times it is more convenient or more efficient to manually pass the parameters, describing how to do this is a good place to start.

As noted earlier in this chapter, when passing parameters on the stack you push the objects in the order they appear in the formal parameter list (from left to right). When passing parameters by value, you should push the values of the actual parameters onto the stack. The following program demonstrates how to do this:

```

program ManuallyPassingValueParameters;
#include( "stdlib.hhf" )

procedure ThreeValueParms( p1:uns32; p2:uns32; p3:uns32 ); @nodisplay;
begin ThreeValueParms;

    mov( p1, eax );
    add( p2, eax );

```

```

    add( p3, eax );
    stdout.put
    (
        "p1 + p2 + p3 = ",
        (type uns32 eax),
        nl
    );

end ThreeValueParms;

static
    SecondParmValue:uns32 := 25;

begin ManuallyPassingValueParameters;

    pushd( 10 );           // Value associated with p1.
    pushd( SecondParmValue); // Value associated with p2.
    pushd( 15 );           // Value associated with p3.
    call ThreeValueParms;

end ManuallyPassingValueParameters;

```

Program 3.5 Manually Passing Parameters on the Stack

Note that if you manually push the parameters onto the stack as this example does, you must use the CALL instruction to call the procedure. If you attempt to use a procedure invocation of the form “ThreeValueParms();” then HLA will complain about a mismatched parameter list. HLA won’t realize that you’ve manually pushed the parameters (as far as HLA is concerned, those pushes appear to preserve some other data).

Generally, there is little reason to manually push a parameter onto the stack if the actual parameter is a constant, a register value, or a variable. HLA’s high level syntax handles most such parameters for you. There are several instances, however, where HLA’s high level syntax won’t work. The first such example is passing the result of an arithmetic expression as a value parameter. Since arithmetic expressions don’t exist in HLA, you will have to manually compute the result of the expression and pass that value yourself. There are two possible ways to do this: calculate the result of the expression and manually push that result onto the stack, or compute the result of the expression into a register and pass the register as a parameter to the procedure. Program 3.6 demonstrates these two mechanisms.

```

program PassingExpressions;
#include( "stdlib.hhf" )

    procedure ExprParm( exprValue:uns32 ); @nodisplay;
    begin ExprParm;

        stdout.put( "exprValue = ", exprValue, nl );

    end ExprParm;

static
    Operand1: uns32 := 5;
    Operand2: uns32 := 20;

begin PassingExpressions;

```

```

// ExprParm( Operand1 + Operand2 );
//
// Method one: Compute the sum and manually
// push the sum onto the stack.

mov( Operand1, eax );
add( Operand2, eax );
push( eax );
call ExprParm;

// Method two: Compute the sum in a register and
// pass the register using the HLA high level
// language syntax.

mov( Operand1, eax );
add( Operand2, eax );
ExprParm( eax );

end PassingExpressions;

```

Program 3.6 Passing the Result of Some Arithmetic Expression as a Parameter

The examples up to this point in this section have made an important assumption: that the parameter you are passing is a double word value. The calling sequence changes somewhat if you're passing parameters that are not four-byte objects. Because HLA can generate relatively inefficient code when passing objects that are not four-bytes long, manually passing such objects is a good idea if you want to have the fastest possible code.

HLA requires that all value parameters be an even multiple of four bytes long¹⁰. If you pass an object that is less than four bytes long, HLA requires that you *pad* the parameter data with extra bytes so that you always pass an object that is at least four bytes in length. For parameters that are larger than four bytes, you must ensure that you pass an even multiple of four bytes as the parameter value, adding extra bytes at the high-order end of the object to pad it, as necessary.

Consider the following procedure prototype:

```
procedure OneByteParm( b:byte );
```

The activation record for this procedure looks like the following:

10. This only applies if you use the HLA high level language syntax to declare and access parameters in your procedures. Of course, if you manually push the parameters yourself and you access the parameters inside the procedure using an addressing mode like “[ebp+8]” then you can pass any sized object you choose. Of course, keep in mind that most operating systems expect the stack to be dword-aligned, so parameters you push should be a multiple of four bytes long.

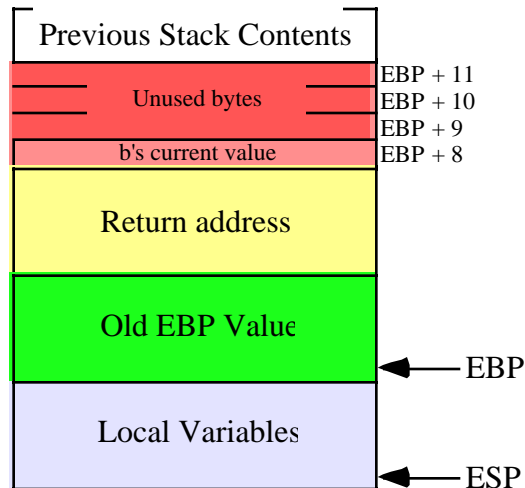


Figure 3.9 OneByteParm Activation Record

As you can see, there are four bytes on the stack associated with the *b* parameter, but only one of the four bytes contains valid data (the L.O. byte). The remaining three bytes are just padding and the procedure should ignore these bytes. In particular, you should never assume that these extra bytes contain zeros or some other consistent value. Depending on the type of parameter you pass, HLA's automatic code generation may or may not push zero bytes as the extra data on the stack.

When passing a byte parameter to a procedure, HLA will automatically emit code that pushes four bytes on the stack. Because HLA's parameter passing mechanism guarantees not to disturb any register or other values, HLA often generates more code than is actually needed to pass a byte parameter. For example, if you decide to pass the AL register as the byte parameter, HLA will emit code that pushes the EAX register onto the stack. This single push instruction is a very efficient way to pass AL as a four-byte parameter object. On the other hand, if you decide to pass the AH register as the byte parameter, pushing EAX won't work because this would leave the value in AH at offset EBP+9 in the activation record shown in Figure 3.9. Unfortunately, the procedure expects this value at offset EBP+8 so simply pushing EAX won't do the job. If you pass AH, BH, CH, or DH as a byte parameter, HLA emits code like the following:

```
sub( 4, esp ); // Make room for the parameter on the stack.
mov( ah, [esp] ); // Store AH into the L.O. byte of the parameter.
```

As you can clearly see, passing one of the "H" registers as a byte parameter is less efficient (two instructions) than passing one of the "L" registers. So you should attempt to use the "L" registers whenever possible if passing an eight-bit register as a parameter¹¹. Note, by the way, that there is very little you can do about the difference in efficiency, even if you manually pass the parameters yourself.

If the byte parameter you decide to pass is a variable rather than a register, HLA generates decidedly worse code. For example, suppose you call OneByteParm as follows:

```
OneByteParm( uns8Var );
```

For this call, HLA will emit code similar to the following to push this single byte parameter:

```
push( eax );
push( eax );
mov( uns8Var, al );
mov( al, [esp+4] );
```

11. Or better yet, pass the parameter directly in the register if you are writing the procedure yourself.

```
pop( eax );
```

As you can plainly see, this is a lot of code to pass a single byte on the stack! HLA emits this much code because (1) it guarantees not to disturb any registers, and (2) it doesn't know whether *uns8Var* is the last variable in allocated memory. You can generate much better code if you don't have to enforce either of these two constraints.

If you've got a spare 32-bit register laying around (especially one of EAX, EBX, ECX or EDX) then you can pass a byte parameter on the stack using only two instructions. Move (or move with zero/sign extension) the byte value into the register and then push the register onto the stack. For the current call to *OneByteParm*, the calling sequence would look like the following in EAX is available:

```
mov( uns8Var, al );
push( eax );
call OneByteParm;
```

If only ESI or EDI were available, you could use code like this:

```
movzx( uns8Var, esi );
push( esi );
call OneByteParm;
```

Another trick you can use to pass the parameter with only a single push instruction is to coerce the byte variable to a double word object, i.e.,

```
push( (type dword uns8Var) );
call OneByteParm;
```

This last example is very efficient. Note that it pushes the first three bytes of whatever value happens to follow *uns8Var* in memory as the padding bytes. HLA doesn't use this technique because there is a (very tiny) chance that using this scheme will cause the program to fail. If it turns out that the *uns8Var* object is the last byte of a given page in memory and the next page of memory is unreadable, the PUSH instruction will cause a memory access exception. To be on the safe side, the HLA compiler does not use this scheme. However, if you always ensure that the actual parameter you pass in this fashion is not the last variable you declare in a static section, then you can get away with code that uses this technique. Since it is nearly impossible for the byte object to appear at the last accessible address on the stack, it is probably safe to use this technique with VAR objects.

When passing word parameters on the stack you must also ensure that you include padding bytes so that each parameter consumes an even multiple of four bytes. You can use the same techniques we use to pass bytes except, of course, there are two valid bytes of data to pass instead of one. For example, you could use either of the following two schemes to pass a word object *w* to a *OneWordParm* procedure:

```
mov( w, ax );
push( eax );
call OneWordParm;

push( (type dword w) );
call OneWordParm;
```

When passing large objects by value on the stack (e.g., records and arrays), you do not have to ensure that each element or field of the object consumes an even multiple of four bytes; all you need to do is ensure that the entire data structure consumes an even multiple of four bytes on the stack. For example, if you have an array of 10 three-byte elements, the entire array will need two bytes of padding (10×3 is 30 bytes which is not evenly divisible by four, but $10 \times 3 + 2$ is 32 which is divisible by four). HLA does a fairly good job of passing large data objects by value to a procedure. For larger objects, you should use the HLA high level language procedure invocation syntax unless you have some special requirements. Of course, if you want efficient operation, you should try to avoid passing large data structures by value.

By default, HLA guarantees that it won't disturb the values of any registers when it emits code to pass parameters to a procedure. Sometimes this guarantee isn't necessary. For example, if you are returning a

function result in EAX and you are not passing a parameter to a procedure in EAX, there really is no reason to preserve EAX upon entry into the procedure. Rather than generating some crazy code like the following to pass a byte parameter:

```
push( eax );
push( eax );
mov( uns8Var, al );
mov( al, [esp+4] );
pop( eax );
```

HLA could generate much better code if it knows that it can use EAX (or some other register):

```
mov( uns8Var, al );
push( eax );
```

You can use the @USE procedure option to tell HLA that it can modify a register's value if doing so would improve the code it generates when passing parameters. The syntax for this option is

```
@use genReg32;
```

The *genReg₃₂* operand can be EAX, EBX, ECX, EDX, ESI, or EDI. You'll obtain the best results if this register is one of EAX, EBX, ECX, or EDX. Particularly, you should note that you cannot specify EBP or ESP here (since the procedure already uses those registers).

The @USE procedure option tells HLA that it's okay to modify the value of the register you specify as an operand. Therefore, if HLA can generate better code by not preserving that register's value, it will do so. For example, when the "@use eax;" option is provided for the *OneByteParm* procedure given earlier, HLA will only emit the two instructions immediately above rather than the five-instruction sequence that preserves EAX.

You must exercise care when specifying the @USE procedure option. In particular, you should not be passing any parameters in the same register you specify in the @USE option (since HLA may inadvertently scramble the parameter's value if you do this). Likewise, you must ensure that it's really okay for the procedure to change the register's value. As noted above, the best choice for an @USE register is EAX when the procedure is returning a function result in EAX (since, clearly, the caller will not expect the procedure to preserve EAX).

If your procedure has a FORWARD or EXTERNAL declaration, the @USE option must only appear in the FORWARD or EXTERNAL definition, not in the actual procedure declaration. If no such procedure prototype appears, then you must attach the @USE option to the procedure declaration.

Example:

```
procedure OneByteParm( b:byte ); @nodisplay; @use EAX;
begin OneByteParm;

    << Do something with b >>

end OneByteParm;
.
.
.
static
    byteVar:byte;
    .
    .
    .
    OneByteParm( byteVar );
```

This call to OneByteParm emits the following instructions:

```
mov( uns8Var, al );
push( eax );
```

3.8.5.3 Accessing Reference Parameters on the Stack

Since HLA passes the address of the actual parameters for reference parameters, accessing the reference parameters within a procedure is slightly more difficult than accessing value parameters because you have to dereference the pointers to the reference parameters. Unfortunately, HLA's high level syntax for procedure declarations and invocations does not (and cannot) abstract this detail away for you. You will have to manually dereference these pointers yourself. This section reviews how you do this.

Consider the following program:

```

program AccessingReferenceParameters;
#include( "stdlib.hhf" )

    procedure RefParm( var theParameter: uns32 ); @nodisplay;
    begin RefParm;

        // Add two directly to the parameter passed by
        // reference to this procedure.

        mov( theParameter, eax );
        add( 2, (type uns32 [eax]) );

        // Fetch the value of the reference parameter
        // and print it's value.

        mov( [eax], eax );
        stdout.put
        (
            "theParameter now equals ",
            (type uns32 eax),
            nl
        );

    end RefParm;

static
    p1: uns32 := 10;
    p2: uns32 := 15;

begin AccessingReferenceParameters;

    RefParm( p1 );
    RefParm( p2 );

    stdout.put( "On return, p1=", p1, " and p2=", p2, nl );

end AccessingReferenceParameters;

```

Program 3.7 Accessing a Reference Parameter

In this example the *RefParm* procedure has a single pass by reference parameter. Pass by reference parameters are always a pointer to the type specified by the parameter's declaration. Therefore, *theParameter* is actual an object of type "pointer to *uns32*" rather than an *uns32* value. In order to access the value associated with *theParameter*, this code has to load that double word address into a 32-bit register and access the data indirectly. The "mov(theParameter, eax);" instruction in the code above fetches this pointer into the EAX register and then the procedure uses the "[eax]" addressing mode to access the actual value of *theParameter*.

Since this procedure accesses the data of the actual parameter, adding two to this data affects the values of the variables passed to the *RefParm* procedure from the main program. Of course, this should come as no surprise since this is the standard semantics for pass by reference parameters.

As you can see, accessing (small) pass by reference parameters is a little less efficient than accessing value parameters because you need an extra instruction to load the address into a 32-bit pointer register (not to mention, you have to reserve a 32-bit register for this purpose). If you access reference parameters frequently, these extra instructions can really begin to add up, reducing the efficiency of your program. Furthermore, it's easy to forget to dereference a reference parameter and use the address of the value instead of the value in your calculations (this is especially true when passing double-word parameters, like the *uns32* parameter in the example above, to your procedures). Therefore, unless you really need to affect the value of the actual parameter, you should use pass by value to pass small objects to a procedure.

Passing large objects, like arrays and records, is where reference parameters become very efficient. When passing these objects by value, the calling code has to make a copy of the actual parameter; if the actual parameter is a large object, the copy process can be very inefficient. Since computing the address of a large object is just as efficient as computing the address of a small scalar object, there is no efficiency loss when passing large objects by reference. Within the procedure you must still dereference the pointer to access the object but the efficiency loss due to indirection is minimal when you contrast this with the cost of copying that large object. The following program demonstrates how to use pass by reference to initialize an array of records:

```

program accessingRefArrayParameters;
#include( "stdlib.hhf" )

const
  NumElements := 64;

type
  Pt: record
    x:uns8;
    y:uns8;
  endrecord;

  Pts: Pt[NumElements];

procedure RefArrayParm( var ptArray: Pts ); @nodisplay;
begin RefArrayParm;

  push( eax );
  push( ecx );
  push( edx );

  mov( ptArray, edx ); // Get address of parameter into EDX.

  for( mov( 0, ecx ); ecx < NumElements; inc( ecx ) ) do

    // For each element of the array, set the "x" field

```



```

    // to (ecx div 8) and set the "y" field to (ecx mod 8).

    mov( cl, al );
    shr( 3, al ); // ECX div 8.
    mov( al, (type Pt [edx+ecx*2]).x );

    mov( cl, al );
    and( %111, al ); // ECX mod 8.
    mov( al, (type Pt [edx+ecx*2]).y );

endfor;
pop( edx );
pop( ecx );
pop( eax );

end RefArrayParm;

static
  MyPts: Pts;

begin accessingRefArrayParameters;

  // Initialize the elements of the array.

  RefArrayParm( MyPts );

  // Display the elements of the array.

  for( mov( 0, ebx ); ebx < NumElements; inc( ebx )) do

    stdout.put
    (
      "RefArrayParm[ ",
      (type uns32 ebx):2,
      "].x=",
      MyPts.x[ ebx*2 ],

      "  RefArrayParm[ ",
      (type uns32 ebx):2,
      "].y=",
      MyPts.y[ ebx*2 ],
      nl
    );

  endfor;

end accessingRefArrayParameters;

```

Program 3.8 Passing an Array of Records by Referencing

As you can see from this example, passing large objects by reference isn't particularly inefficient. Other than tying up the EDX register throughout the *RefArrayParm* procedure plus a single instruction to load EDX with the address of the reference parameter, the *RefArrayParm* procedure doesn't require many more instructions than the same procedure where you would pass the parameter by value.

3.8.5.4 Passing Reference Parameters on the Stack

HLA's high level syntax often makes passing reference parameters a breeze. All you need to do is specify the name of the actual parameter you wish to pass in the procedure's parameter list. HLA will automatically emit some code that will compute the address of the specified actual parameter and push this address onto the stack. However, like the code HLA emits for value parameters, the code HLA generates to pass the address of the actual parameter on the stack may not be the most efficient that is possible. Therefore, if you want to write fast code, you may want to manually write the code to pass reference parameters to a procedure. This section discusses how to do exactly that.

Whenever you pass a static object as a reference parameter, HLA generates very efficient code to pass the address of that parameter to the procedure. As an example, consider the following code fragment:

```
procedure HasRefParm( var d:dword );
.
.
.
static
    FourBytes:dword;

var
    v: dword;
.
.
.
HasRefParm( FourBytes );
.
.
.
```

For the call to the *HasRefParm* procedure, HLA emits the following instruction sequence:

```
pushd( &FourBytes );
call HasRefParm;
```

You really aren't going to be able to do substantially better than this if you are passing your reference parameters on the stack. So if you're passing static objects as reference parameters, HLA generates fairly good code and you should stick with the high level syntax for the procedure call.

Unfortunately, when passing automatic (VAR) objects or indexed variables as reference parameters, HLA needs to compute the address of the object at run-time. This generally requires the use of the LEA instruction. Unfortunately, the LEA instruction requires the use of a 32-bit register and HLA promises not to disturb the values in any registers when it automatically generates code for you¹². Therefore, HLA needs to preserve the value in whatever register it uses when it computes an address via LEA to pass a parameter by reference. The following example shows you the code that HLA actually emits:

```
// Call to the HasRefParm procedure:

    HasRefParm( v );

// HLA actually emits the following code for the above call:

    push( eax );
    push( eax );
    lea( eax, v );
    mov( eax, [esp+4] );
    pop( eax );
```

12. This isn't entirely true. You'll see the exception in the chapter on Classes and Objects. Also, using the @USE procedure option tells HLA that it's okay to modify the value in one of the registers.

```
call HasRefParm;
```

As you can see, this is quite a bit of code, especially if you have a 32-bit register available and you don't need to preserve that register's value. Here's a better code sequence given the availability of EAX:

```
lea( eax, v );
push( eax );
call HasRefParm;
```

Remember, when passing an actual parameter by reference, you must compute the address of that object and push the address onto the stack. For simple static objects you can use the address-of operator (“&”) to easily compute the address of the object and push it onto the stack; however, for indexed and automatic objects, you will probably need to use the LEA instruction to compute the address of the object. Here are some examples that demonstrate this using the *HasRefParm* procedure from the previous examples:

```
static
  i:   int32;
  Ary: int32[16];
  iptr: pointer to int32 := &i;

var
  v:   int32;
  AV:  int32[10];
  vptr: pointer to int32;
  .
  .
  .
  lea( eax, v );
  mov( eax, vptr );
  .
  .
  .
// HasRefParm( i );

  push( &i );           // Simple static object, so just use "&".
  call HasRefParm;

// HasRefParm( Ary[ebx] ); // Pass element of Ary by reference.

  lea( eax, Ary[ ebx*4 ] ); // Must use LEA for indexed addresses.
  push( eax );
  call HasRefParm;

// HasRefParm( *iptr ); -- Pass object pointed at by iptr

  push( iptr );           // Pass address (iptr's value) on stack.
  call HasRefParm;

// HasRefParm( v );

  lea( eax, v );           // Must use LEA to compute the address
  push( eax );             // of automatic vars passed on stack.
  call HasRefParm;

// HasRefParm( AV[ esi ] ); -- Pass element of AV by reference.

  lea( eax, AV[ esi*4 ] ); // Must use LEA to compute address of the
  push( eax );             // desired element.
  call HasRefParm;

// HasRefParm( *vptr ); -- Pass address held by vptr...
```

```

push( vptr );           // Just pass vptr's value as the specified
call HasRefParm;       // address.

```

If you have an extra register to spare, you can tell HLA to use that register when computing the address of reference parameters (without emitting the code to preserve that register's value). The @USE option will tell HLA that it's okay to use the specified register without preserving it's value. As noted in the section on value parameters, the syntax for this procedure option is

```
@use reg32;
```

where *reg₃₂* may be any of EAX, EBX, ECX, EDX, ESI, or EDI. Since reference parameters always pass a 32-bit value, all of these registers are equivalent as far as HLA is concerned (unlike value parameters, that may prefer the EAX, EBX, ECX, or EDX register). Your best choice would be EAX if the procedure is not passing a parameter in the EAX register and the procedure is returning a function result in EAX; otherwise, any currently unused register will work fine.

With the "@USE EAX;" option, HLA emits the shorter code given in the previous examples. It does not emit all the extra instructions needed to preserve EAX's value. This makes your code much more efficient, especially when passing several parameters by reference or when calling procedures with reference parameters several times.

3.8.5.5 Passing Formal Parameters as Actual Parameters

The examples in the previous two sections show how to pass static and automatic variables as parameters to a procedure, either by value or by reference. There is one situation that these sections don't handle properly: the case when you are passing a formal parameter in one procedure as an actual parameter to another procedure. The following simple example demonstrates the different cases that can occur for pass by value and pass by reference parameters:

```

procedure p1( val v:dword; var r:dword );
begin p1;
    .
    .
    .
end p1;

procedure p2( val v2:dword; var r2:dword );
begin p2;

    p1( v2, r2 );    // (1) First call to p1.
    p1( r2, v2 );    // (2) Second call to p1.

end p2;

```

In the statement labelled (1) above, procedure *p2* calls procedure *p1* and passes its two formal parameters as parameters to *p1*. Note that this code passes the first parameter of both procedures by value and it passes the second parameter of both procedures by reference. Therefore, in statement (1), the program passes the *v2* parameter into *p2* by value and passes it on to *p1* by value; likewise, the program passes *r2* in by reference and it passes the value onto *p2* by reference.

Since *p2*'s caller passes *v2* in by value and *p2* passes this parameter to *p1* by value, all the code needs to do is make a copy of *v2*'s value and pass this on to *p1*. The code to do this is nothing more than a single push instruction, e.g.,

```

push( v2 );
<< code to handle r2 >>
call p1;

```

As you can see, this code is identical to passing an automatic variable by value. *Indeed, it turns out that the code you need to write to pass a value parameter to another procedure is identical to the code you would write to pass a local, automatic, variable to that other procedure.*

Passing *r2* in statement (1) above requires a little more thought. You do not take the address of *r2* using the LEA instruction as you would a value parameter or an automatic variable. When passing *r2* on through to *p1*, the author of this code probably expects the *r* formal parameter to contain the address of the variable whose address *p2*'s caller passed into *p2*. In plain English, this means that *p2* must pass the address of *r2*'s actual parameter on through to *p1*. Since the *r2* parameter is actually a double word value containing the address of the corresponding actual parameter, this means that the code must pass the dword value of *r2* on to *p1*. The complete code for statement (1) above looks like the following:

```
push( v2 );    // Pass the value passed in through v2 to p1.
push( r2 );   // Pass the address passed in through r2 to p1.
call p1;
```

The important thing to note in this example is that passing a formal reference parameter (*r2*) as an actual reference parameter (*r*) does not involve taking the address of the formal parameter (*r2*). *P2*'s caller has already done this; *p2* need only pass this address on through to *p1*.

In the second call to *p1* in the example above (2), the code swaps the actual parameters so that the call to *p1* passes *r2* by value and *v2* by reference. Specifically, *p1* expects *p2* to pass it the value of the dword object associated with *r2*; likewise, it expects *p2* to pass it the address of the value associated with *v2*.

To pass the value of the object associated with *r2*, your code must dereference the pointer associated with *r2* and directly pass the value. Here is the code HLA automatically generates to pass *r2* as the first parameter to *p1* in statement (2):

```
sub( 4, esp );    // Make room on stack for parameter.
push( eax );     // Preserve EAX's value.
mov( r2, eax );  // Get address of object passed in to p2.
mov( [eax], eax ); // Dereference to get the value of this object.
mov( eax, [esp+4] ); // Put value of parameter into its location on stack.
pop( eax );     // Restore original EAX value.
```

As usual, HLA generates a little more code than may be necessary because it won't destroy the value in the EAX register (you may use the @USE procedure option to tell HLA that it's okay to use EAX's value, thereby reducing the code it generates). You can write more efficient code if a register is available to use in this sequence. If EAX is unused, you could trim this down to the following:

```
mov( r2, eax );    // Get the pointer to the actual object.
pushd( [eax] );   // Push the value of the object onto the stack.
```

Since you can treat value parameters exactly like local (automatic) variables, you use the same code to pass *v2* by reference to *p1* as you would to pass a local variable in *p2* to *p1*. Specifically, you use the LEA instruction to compute the address of the value in the *v2*. The code HLA automatically emits for statement (2) above preserves all registers and takes the following form (same as passing an automatic variable by reference):

```
push( eax );     // Make room for the parameter.
push( eax );     // Preserve EAX's value.
lea( eax, v2 );  // Compute address of v2's value.
mov( eax, [esp+4] ); // Store away address as parameter value.
pop( eax );     // Restore EAX's value
```

Of course, if you have a register available, you can improve on this code. Here's the complete code that corresponds to statement (2) above:

```
mov( r2, eax );    // Get the pointer to the actual object.
pushd( [eax] );   // Push the value of the object onto the stack.
lea( eax, v2 );   // Push the address of V2 onto the stack.
push( eax );
call p1;
```

3.8.5.6 HLA Hybrid Parameter Passing Facilities

Like control structures, HLA provides a high level language syntax for procedure calls that is convenient to use and easy to read. However, this high level language syntax is sometimes inefficient and may not provide the capabilities you need (for example, you cannot specify an arithmetic expression as a value parameter as you can in high level languages). HLA lets you overcome these limitations by writing low-level (“pure”) assembly language code. Unfortunately, the low-level code is harder to read and maintain than procedure calls that use the high level syntax. Furthermore, it’s quite possible that HLA generates perfectly fine code for certain parameters and only one or two parameters present a problem. Fortunately, HLA provides a hybrid syntax for procedure calls that allows you to use both high-level and low-level syntax as appropriate for a given actual parameter. This lets you use the high level syntax where appropriate and then drop down into pure assembly language to pass those special parameters that HLA’s high level language syntax cannot handle efficiently (if at all).

Within an actual parameter list (using the high level language syntax), if HLA encounters “#{“ followed by a sequence of statements and a closing “}#”, HLA will substitute the instructions between the braces in place of the code it would normally generate for that parameter. For example, consider the following code fragment:

```

procedure HybridCall( i:uns32; j:uns32 );
begin HybridCall;
    .
    .
    .
end HybridCall;

    .
    .
    .

// Equivalent to HybridCall( 5, i+j );

HybridCall
(
    5,
    #{
        mov( i, eax );
        add( j, eax );
        push( eax );
    }#
);

```

The call to *HybridCall* immediately above is equivalent to the following “pure” assembly language code:

```

pushd( 5 );
mov( i, eax );
add( j, eax );
push( eax );
call HybridCall;

```

As a second example, consider the example from the previous section:

```

procedure p2( val v2:dword; var r2:dword );
begin p2;

    p1( v2, r2 );    // (1) First call to p1.
    p1( r2, v2 );   // (2) Second call to p1.

```

```
end p2;
```

HLA generates exceedingly mediocre code for the second call to *p1* in this example. If efficiency is important in the context of this procedure call, and you have a free register available, you might want to rewrite this code as follows¹³:

```
procedure p2( val v2:dword; var r2:dword );
begin p2;

    p1( v2, r2 );    // (1) First call to p1.
    p1              // (2) Second call to p1.
    (               // This code assumes EAX is free.
        #{
            mov( r2, eax );
            pushd( [eax] );
        }#,
        #{
            lea( eax, v2 );
            push( eax );
        }#
    );

end p2;
```

3.8.5.7 Mixing Register and Stack Based Parameters

You can mix register parameters and standard (stack-based) parameters in the same high level procedure declaration, e.g.,

```
procedure HasBothRegAndStack( var dest:dword in edi; count:un32 );
```

When constructing the activation record, HLA ignores the parameters you pass in registers and only processes those parameters you pass on the stack. Therefore, a call to the *HasBothRegAndStack* procedure will push only a single parameter onto the stack (*count*). It will pass the *dest* parameter in the EDI register. When this procedure returns to its caller, it will only remove four bytes of parameter data from the stack.

Note that when you pass a parameter in a register, you should avoid specifying that same register in the @USE procedure option. In the example above, HLA might not generate any code whatsoever at all for the *dest* parameter (because the value is already in EDI). Had you specified “@use edi;” and HLA decided it was okay to disturb EDI’s value, this would destroy the parameter value in EDI; that won’t actually happen in this particular example (since HLA never uses a register to pass a dword value parameter like *count*), but keep this problem in mind.

3.9 Procedure Pointers

The x86 CALL instruction is very similar to the JMP instruction. In particular, it allows the same three basic forms as the JMP instruction: direct calls (to a procedure name), indirect calls through a 32-bit general

13. Of course, you could also use the “@use eax;” procedure option to achieve the same effect in this example.

purpose register, and indirect calls through a double word pointer variable. The CALL instruction allows the following (low-level) syntax supporting these three types of procedure invocations:

```
call Procname;    // Direct call to procedure "Procname" (or stmt label).
call( Reg32 );   // Indirect call to procedure whose address appears
                  //   in the Reg32 general-purpose 32-bit register.
call( dwordVar ); // Indirect call to the procedure whose address appears
                  //   in the dwordVar double word variable.
```

HLA treats procedure names like static objects. Therefore, you can compute the address of a procedure by using the address-of (“&”) operator along with the procedure’s name or by using the LEA instruction. For example, “&Procname” is the address of the very first instruction of the *Procname* procedure. Therefore, all three of the following code sequences wind up calling the *Procname* procedure:

```
call Procname;
.
.
.
mov( &Procname, eax );
call( eax );
.
.
.
lea( eax, Procname );
call( eax );
```

Since the address of a procedure fits in a 32-bit object, you can store such an address into a dword variable; in fact, you can initialize a dword variable with the address of a procedure using code like the following:

```
procedure p;
begin p;
end p;
.
.
.
static
ptrToP: dword := &p;
.
.
.
call( ptrToP ); // Calls the "p" procedure if ptrToP has not changed.
```

Because the use of procedure pointers occurs frequently in assembly language programs, HLA provides a special syntax for declaring procedure pointer variables and for calling procedures indirectly through such pointer variables. To declare a procedure pointer in an HLA program, you can use a variable declaration like the following:

```
static
procPtr: procedure;
```

Note that this syntax uses the keyword PROCEDURE as a data type. It follows the variable name and a colon in one of the variable declaration sections (STATIC, READONLY, STORAGE, or VAR). This sets aside exactly four bytes of storage for the *procPtr* variable. To call the procedure whose address is held by *procPtr*, you can use either of the following two forms:

```
call( procPtr ); // Low-level syntax.
procPtr();       // High-level language syntax.
```

Note that the high level syntax for an indirect procedure call is identical to the high level syntax for a direct procedure call. HLA can figure out whether to use a direct call or an indirect call by the type of the identi-

fier. If you've specified a variable name, HLA assumes it needs to use an indirect call; if you specify a procedure name, HLA uses a direct call.

Like all pointer objects, you should not attempt to indirectly call a procedure through a pointer variable unless you've initialized that variable with the address appropriately. There are two ways to initialize a procedure pointer variable: `STATIC` and `READONLY` objects allow an initializer, or you can compute the address of a routine (as a 32-bit value) and store that 32-bit address directly into the procedure pointer at run-time. The following code fragment demonstrates both ways you can initialize a procedure pointer.

```
static
  ProcPtr: procedure := &p;    // Initialize ProcPtr with the address of p.
  .
  .
  ProcPtr();                  // First invocation calls p.

  mov( &q, ProcPtr );        // Reload ProcPtr with the address of q.
  .
  .
  ProcPtr();                  // This invocation calls the "q" procedure.
```

Procedure pointer variable declarations also allow the declaration of parameters. To declare a procedure pointer with parameters, you must use a declaration like the following:

```
static
  p:procedure( i:int32; c:char );
```

This declaration states that *p* is a 32-bit pointer that contains the address of a procedure having two parameters. If desired, you could also initialize this variable *p* with the address of some procedure by using a static initializer, e.g.,

```
static
  p:procedure( i:int32; c:char ) := &SomeProcedure;
```

Note that *SomeProcedure* must be a procedure whose parameter list exactly matches *p*'s parameter list (i.e., two value parameters, the first is an *int32* parameter and the second is a *char* parameter). To indirectly call this procedure, you could use either of the following sequences:

```
  push( << Value for i >> );
  push( << Value for c >> );
  call( p );
-or-
  p( <<Value for i>>, <<Value for c>> );
```

The high level language syntax has the same features and restrictions as the high level syntax for a direct procedure call. The only difference is the actual `CALL` instruction HLA emits at the end of the calling sequence.

Although all of the examples in this section have used `STATIC` variable declarations, don't get the idea that you can only declare simple procedure pointers in the `STATIC` or other variable declaration sections. You can declare procedure pointer types in the `TYPE` section. You can declare procedure pointers as fields of a `RECORD`. Assuming you create a type name for a procedure pointer in the `TYPE` section, you can even create arrays of procedure pointers. The following code fragments demonstrate some of the possibilities:

```
type
  pptr: procedure;
  prec: record
    p:pptr;
    // other fields...
  endrecord;
static
  pl:pptr;
```

```

p2:pptr[2]
p3:prec;
.
.
.
p1();
p2[ebx*4]();
p3.p();

```

One very important thing to keep in mind when using procedure pointers is that HLA does not (and cannot) enforce strict type checking on the pointer values you assign to a procedure pointer variable. In particular, if the parameter lists do not agree between the declarations of the pointer variable and the procedure whose address you assign to the pointer variable, the program will probably crash if you attempt to call the mismatched procedure indirectly through the pointer using the high level syntax. Like the low-level “pure” procedure calls, it is your responsibility to ensure that the proper number and types of parameters are on the stack prior to the call.

3.10 Procedural Parameters

One place where procedure pointers are quite invaluable is in parameter lists. Selecting one of several procedures to call by passing the address of some procedure, selected from a set of procedures, is not an uncommon operation. Therefore, HLA lets you declare procedure pointers as parameters.

There is nothing special about a procedure parameter declaration. It looks exactly like a procedure variable declaration except it appears within a parameter list rather than within a variable declaration section. The following are some typical procedure prototypes that demonstrate how to declare such parameters:

```

procedure p1( procparm: procedure ); forward;
procedure p2( procparm: procedure( i:int32 ) ); forward;
procedure p3( val procparm: procedure ); forward;

```

The last example above is identical to the first. It does point out, though, that you generally pass procedural parameters by value. This may seem counter-intuitive since procedure pointers are addresses and you will need to pass an address as the actual parameter; however, a pass by reference procedure parameter means something else entirely. consider the following (legal!) declaration:

```

procedure p4( var procPtr:procedure ); forward;

```

This declaration tells HLA that you are passing a procedure *variable* by reference to *p4*. The address HLA expects must be the address of a procedure pointer variable, not a procedure.

When passing a procedure pointer by value, you may specify either a procedure variable (whose value HLA passes to the actual procedure) or a procedure pointer constant. A procedure pointer constant consists of the address-of operator (“&”) immediately followed by a procedure name. Passing procedure constants is probably the most convenient way to pass procedural parameters. For example, the following calls to the *Plot* routine might plot out the function passed as a parameter from -2π to $+2\pi$.

```

Plot( &sineFunc );
Plot( &cosFunc );
Plot( &tanFunc );

```

Note that you cannot pass a procedure as a parameter by simply specifying the procedure’s name. I.e., “Plot(sineFunc);” will not work. Simply specifying the procedure name doesn’t work because HLA will attempt to directly call the procedure whose name you specify (remember, a procedure name inside a parameter list invokes instruction composition). However, since you don’t specify a parameter list, or at least an empty pair of parentheses, after the parameter/procedure’s name, HLA generates a syntax error message. Moral of the story: don’t forget to preface procedure parameter constant names with the address-of operator.

3.11 Untyped Reference Parameters

Sometimes you will want to write a procedure to which you pass a generic memory object by reference without regard to the type of that memory object. A classic example is a procedure that zeros out some data structure. Such a procedure might have the following prototype:

```
procedure ZeroMem( var mem:byte; count:uint32 );
```

This procedure would zero out *count* bytes starting at the address the first parameter specifies. The problem with this procedure prototype is that HLA will complain if you attempt to pass anything other than a byte object as the first parameter. Of course, you can overcome this problem using type coercion like the following, but if you call this procedure several times with lots of different data types, then the following coercion operator is rather tedious to use:

```
ZeroMem( (type byte MyDataObject), @size( MyDataObject ) );
```

Of course, you can always use hybrid parameter passing or manually push the parameters yourself, but these solutions are even more work than using the type coercion operation. Fortunately, HLA provides a far more convenient solution: untyped reference parameters.

Untyped reference parameters are exactly that – pass by reference parameters on which HLA doesn't bother to compare the type of the actual parameter against the type of the formal parameter. With an untyped reference parameter, the call to *ZeroMem* above would take the following form:

```
ZeroMem( MyDataObject, @size( MyDataObject ) );
```

MyDataObject could be any type and multiple calls to *ZeroMem* could pass different typed objects without any objections from HLA.

To declare an untyped reference parameter, you specify the parameter using the normal syntax except that you use the reserved word VAR in place of the parameter's type. This VAR keyword tells HLA that any variable object is legal for that parameter. Note that you must pass untyped reference parameters by reference, so the VAR keyword must precede the parameter's declaration as well. Here's the correct declaration for the *ZeroMem* procedure using an untyped reference parameter:

```
procedure ZeroMem( var mem:var; count:uint32 );
```

With this declaration, HLA will compute the address of whatever memory object you pass as an actual parameter to *ZeroMem* and pass this on the stack.

3.12 Iterators and the FOREACH Loop

One nifty feature HLA provides is support for *true* iterators¹⁴. An iterator is a special type of procedure or function that you use in conjunction with the HLA FOREACH..ENDFOR loop. Combined, these two language features (iterators and the FOREACH..ENDFOR loop) provide a very powerful user-defined looping construct.

The HLA FOREACH..ENDFOR statement uses the following basic syntax:

```
foreach iteratorID( optional_parameters ) do

    << loop body >>
```

14. HLA's iterators are based on the control structure by the same name from the CLU programming language. Those things that C/C++ programmers refer to as iterators are more properly called *cursors*. While it is certainly possible to write cursors in HLA, it is important to note that HLA's iterators are quite a bit more powerful than C/C++'s iterators.

```
endfor;
```

The FOREACH statement calls the specified iterator. If the iterator *succeeds*, then the FOREACH statement executes the loop body; if the iterator *fails*, then control transfers to the first statement following the ENDFOR clause. On each iteration of the loop body, the program re-enters the iterator code and, once again, the iterator returns success or failure to determine whether to repeat the loop body.

At first glance, you might get the impression that the FOREACH loop is nothing more than a WHILE loop and an iterator is a function that returns true (success) or false (failure). However, this is not an accurate picture of how the FOREACH loop operates. First of all, the FOREACH loop does not CALL the iterator on each iteration of the loop; it *re-enters* the iterator. Specifically, control does not (necessarily) begin with the first statement of the iterator whenever control returns to the top of the FOREACH loop. The second big difference between a FOREACH/iterator loop and a WHILE/function loop is that the iterator procedure maintains its activation record in memory for the duration of the FOREACH loop. A function you would call from a WHILE loop, by contrast, builds and destroys the function's activation record on each iteration of the loop. This means that the iterator's local (automatic) variables maintain their values until the FOREACH loop terminates. This has important ramifications, especially for recursive iterator functions.

An iterator declaration looks very similar to a procedure declaration. Indeed, about the only syntactical difference is the use of the reserved word ITERATOR rather than PROCEDURE. The following is an example of a simple iterator:

```
iterator range( start:uns32; last:uns32 ); nodisplay;
begin range;

    mov( start, eax );
    while( eax <= last ) do

        push( eax );
        yield();
        pop( eax );
        inc( eax );

    endwhile;

end range;
```

The only thing special about this iterator declaration, other than the use of the ITERATOR reserved word, is that it calls a special procedure named *yield*. In a few paragraphs you'll see the purpose of the call to the *yield* procedure.

A typical FOREACH loop that calls the *range* iterator might look like the following:

```
foreach range( 1, 10 ) do

    stdout.put( "Iteration =", (type uns32 eax), nl );

endfor;
```

Here's how the iterator and the FOREACH loop work together. Upon first encountering the FOREACH statement, the program makes an *initial call* to the *range* iterator. Except for a few extra parameters HLA pushes on the stack, this call is exactly like a standard procedure call. Upon entry into the iterator, the *start* parameter has the initial value one and the *last* parameter has the initial value ten. The iterator loads *start* into EAX and compares this against the value in *last* (ten). Since EAX's value is less than or equal to ten, the program enters the loop's body. The loop body pushes EAX's value onto the stack and then calls the *yield* procedure. The *yield* procedure transfers control to the body of the FOREACH loop that called the *range* iterator in the first place. Calling *yield* is how the iterator returns success to the FOREACH loop. Within the body of the FOREACH loop, above, the code prints out the value of the EAX register as an unsigned integer. During the first iteration of the loop, EAX contains one so the loop body prints this value.

At the bottom of the FOREACH loop, the program *re-enters* the iterator. When the FOREACH loop re-enters the iterator, it transfers control to the first statement following the call to the *yield* function. Intuitively, you can view the FOREACH loop body as a procedure that the iterator calls whenever you call the *yield* function¹⁵. Whenever the program encounters the ENDFOR clause, it returns to the iterator, executing the first statement beyond the *yield* call. In the current example, this pops the value of EAX off the stack (preserved before the call to *yield*), the loop increments EAX and repeats as long as EAX is less than ten.

When the *range* iterator increments EAX to 11, the WHILE loop in the iterator terminates and control falls off the bottom of the iterator. This is how an iterator returns failure to the calling FOREACH loop. At that point control transfers to the first statement following the ENDFOR in the FOREACH..ENDFOR loop.

By the way, the *range* iterator, combined with the FOREACH loop above, creates a relatively inefficient implementation of the following loop:

```
for( mov( 1, eax ); eax < 10; inc( eax ) ) do

    stdout.put( "Iteration = ", (type uns32 eax), nl );

endfor;
```

However, don't get the impression from this example that iterators are particularly inefficient. Iterators are not a good choice for something like *range*. However, there are many iterators you can write that are just as efficient as other means of loop control and computation.

An important point to remember when using iterators is that the iterator's activation record remains on the stack as long as the iterator returns success. The program only removes the activation record when the iterator fails. The *range* iterator takes advantage of this fact since it refers to the value of its *last* parameter on each re-entry from the FOREACH loop. The fact that parameters and local (automatic) variables maintain their values for the duration of the FOREACH loop is very important to many algorithms that use iterators, especially recursive algorithms.

One side effect of having an iterator maintain its activation record until it fails is that the value of ESP changes considerably between the statement immediately before the FOREACH statement and the first statement in the body of the FOREACH loop. This is because the program "pushes" the activation record onto the stack upon encountering the FOREACH loop and doesn't "pop" this activation record off the stack until the FOREACH loop fails. Therefore, code like the following will not work as expected:

```
pushd( 10 );
foreach range( 1, 25 ) do
    pop( ebx );
    push( ebx );
    stdout.put( "eax=", eax, " ebx=", ebx, nl );
endfor;
pop( ebx );
```

The problem with this code is that the FOREACH loop pushes a whole lot of data onto the stack after the PUSH instruction pushes the value 10 onto the stack. Therefore, the POP instruction inside the loop does not pop the value 10 from the stack. Instead, it pops some data pushed on the stack by the iterator (specifically, it pops the return address that transfers control to the first instruction following the *yield* call). Therefore, you cannot use the stack to transfer data into or out of a FOREACH loop¹⁶.

Another problem with the stack and the FOREACH loop occurs if you try to prematurely exit a FOREACH loop before the iterator returns failure. Whenever an iterator fails, it cleans up the stack and restores ESP to the value it had upon encountering the FOREACH statement. However, statements like BREAK, BREAKIF, EXIT, EXITIF, JMP and any other flow of control transfer instructions will not clean

15. In fact, this is exactly how HLA implements iterators and the FOREACH loop. See the volume on Advanced Procedures for more details.

16. Not that it's a good idea to transfer data into or out of any loop using the stack. Such code tends to have lots of errors due to extra pushes or pops appearing in the program.

up the stack if they transfer control out of a FOREACH loop. For example, the following code will leave the activation record for the *range* iterator sitting on the stack:

```
foreach range( 2, 5 ) do

    jmp ExitFor;

endfor;
ExitFor:
```

Depending on the iterator and the code that calls the iterator, prematurely exiting a FOREACH loop without having the iterator return failure and leaving this junk sitting on the stack may have an adverse effect on the operation of your program. Clearly if you've pushed data onto the stack prior to the FOREACH loop, you will not be able to pop that data off unless you manually clean up the stack yourself (this involves saving the value of ESP prior to the FOREACH statement and restoring this value at the *ExitFor* label, above). Also, don't forget that prematurely exiting a FOREACH loop without letting the iterator finish may wind up grabbing some system resources that the iterator would normally free just before returning failure (e.g., calling *free* and closing files).

The volume on Advanced Procedures will go into the details concerning the low-level implementation of iterators. Until then, keep in mind that iterators build their activation records differently than standard procedures. Until you read that chapter, you should not attempt to call an iterator directly (i.e., outside a FOREACH loop) nor should you use the "noframe" option with an iterator. See the chapter on Advanced Procedures for more details on the implementation of iterators.

3.13 Sample Programs

This section presents two sample programs. The first demonstrates the use of iterators using a fibonacci number iterator. The second demonstrates the use of procedural parameters.

3.13.1 Generating the Fibonacci Sequence Using an Iterator

The following program generates the Fibonacci sequence $f_1, f_2, f_3, \dots, f_{count}$ where *count* is a parameter. This simple example displays all the fibonacci numbers the iterator generates.

```
program iterDemo;
#include( "stdlib.hhf" )

// Basic (recursive version) algorithm for
// the fibonacci sequence.
//
// int fib(int N)
// {
//     if(N<=2)
//         return 1;
//     else
//         return fib(N-1) + fib(N-2)
// }
//
// Iterator (iterative) that computes all the fibonacci
// numbers between fib(1) and fib(count).

iterator fib( count:uns32 ); nodisplay;
var
```

```

    lastVal:      uns32;
    BeforeLastVal: uns32;

begin fib;

    if( count > 0 ) then

        mov( 0, BeforeLastVal );
        mov( 1, eax );
        mov( eax, lastVal );

        // Handle fib(1) as a special case.

        yield();
        dec( count );

        // Okay, handle fib(2)..fib(count) here.

        while( @nz ) do

            // Compute fib(n) = fib(n-1) + fib(n-2).
            // and then copy fib(n-1) {lastVal} to
            // fib(n-2) {BeforeLastVal} and store the
            // current result into lastVal so we'll
            // have the n-1 and n-2 values on the next
            // call.

            mov( lastVal, eax );
            add( BeforeLastVal, eax );
            mov( lastVal, BeforeLastVal );
            mov( eax, lastVal );

            // Yield fib(n) to the FOREACH loop.

            yield();

            // Repeat this iterator the specified number
            // of times.

            dec( count );

        endwhile;

    endif;

end fib;

static
    iteration:uns32;

begin iterDemo;

    // Display the fibonacci sequence for the first
    // ten fibonacci numbers.

    mov( 1, iteration );
    foreach fib( 10 ) do

        stdout.put( "fib(", iteration, ") = ", (type uns32 eax), nl );

```

```

        inc( iteration );
    endfor;
end iterDemo;

```

3.13.2 Outer Product Computation with Procedural Parameters

The following program generates an *addition table*, a *subtraction table*, or a *multiplication table* based on user inputs. These tables are computed using an *outer product* calculation and procedural parameters. An outer product is simply the process of computing all the values for the elements of a matrix by using the row and column indices as inputs to some function (e.g., addition, subtraction, or multiplication).

```

program funcTable;
#include( "stdlib.hhf" )

static
    size: uns32;
    ftbl: array.dArray( uns32, 2 );

    // GenerateTable-
    //
    // This function computes the "Outer Product". That is,
    // take the cartesian product of the indices into
    // the rows and columns of this array [(0,0), (0,1), ... (0,size-1),
    // (1,0), (1,1), ..., (size-1,size-1)], then feed the left and
    // right values of each coordinate to the "func" procedure passed
    // as a parameter. Whatever result the function returns, store that
    // into element (l,r) of the ftbl array.

    procedure GenerateTable( func:procedure( l:uns32; r:uns32 )); nodisplay;
    begin GenerateTable;

        push( eax );
        push( ebx );
        push( ecx );
        push( edi );

        for( mov( 0, ebx ); ebx < size; inc( ebx )) do

            for( mov( 0, ecx ); ecx < size; inc( ecx )) do

                array.index( edi, ftbl, ebx, ecx );
                func( ebx, ecx );
                mov( eax, [edi] );

            endfor;

        endfor;

        pop( edi );
        pop( ecx );
        pop( ebx );
        pop( eax );

    end GenerateTable;

```



```

// The following functions compute the various
// values used to fill the table (obviously,
// "+" = addFunc, "-" = subFunc, and "*" = mulFunc).

procedure addFunc( left:uns32; right:uns32 ); nodisplay;
begin addFunc;

    mov( left, eax );
    add( right, eax );

end addFunc;

procedure subFunc( left:uns32; right:uns32 ); nodisplay;
begin subFunc;

    mov( left, eax );
    sub( right, eax );

end subFunc;

procedure mulFunc( left:uns32; right:uns32 ); nodisplay;
begin mulFunc;

    mov( left, eax );
    intmul( right, eax );

end mulFunc;

begin funcTable;

    stdout.put( "Function table generator: " nl );
    stdout.put( "----- " nl nl );

    // Get the size of the function table from the user:

    forever

        try

            stdout.put( "Enter the size of the matrix: " );
            stdin.getu32();
            bound( eax, 1, 20 );
            unprotected break;

        exception( ex.ConversionError )

            stdout.put( "Illegal character, re-enter" nl );

        exception( ex.ValueOutOfRange )

            stdout.put( "Value out of range (1..20), please re-enter" nl );

        exception( ex.BoundInstr )

            stdout.put( "Value out of range (1..20), please re-enter" nl );

    endtry;

```

```

endfor;

// Allocate storage for the function table:

mov( eax, size );
array.daAlloc( ftbl, size, size );

// Get the function from the user:

stdout.put( "What type of table do you want to generate?" nl nl );
stdout.put( "+" Addition" nl );
stdout.put( "-" Subtraction" nl );
stdout.put( "*" Multiplication" nl );
stdout.newln();
repeat

    stdout.put( "Choice? (+, -, *): " );
    stdin.FlushInput();
    stdin.getc();

until( al in { '+', '-', '*' } );

// Fill in the entries in the table:

if( al = '+' ) then

    GenerateTable( &addFunc );

elseif( al = '-' ) then

    GenerateTable( &subFunc );

elseif( al = '*' ) then

    GenerateTable( &mulFunc );

endif;

// Display the column labels across the top:

stdout.put( nl nl "          " );
for( mov( 0, ebx ); ebx < size; inc( ebx ) ) do

    stdout.put( (type uns32 ebx):5 );

endfor;
stdout.newln();
stdout.put( "          " );
for( mov( 0, ebx ); ebx < size; inc( ebx ) ) do

    stdout.put( "-----" );

endfor;
stdout.newln();

// Display the row labels and fill in the table.
// Note that this code prints the result as int32
// rather than uns32 because the subFunc function
// returns negative values.

```

```

for( mov( 0, ebx); ebx < size; inc( ebx )) do

    stdout.put( (type uns32 ebx):4, ": " );
    for( mov( 0, ecx); ecx < size; inc( ecx )) do

        array.index( edi, ftbl, ebx, ecx );
        stdout.puti32size( [edi], 5, ' ' );

    endfor;
    stdout.newln();

endfor;

end funcTable;

```

3.14 Putting It All Together

In this chapter you saw the low level implementation of procedures and calls to procedures. You learned more about passing parameters by value and reference and you also learned a little more about local variables. This chapter discussed activations records and HLA procedure options. Finally, this chapter wraps up with a discussion of iterators and the FOREACH loop

Your journey through procedures is hardly complete, however. The next volume presents new ways to pass parameters, discusses nested procedures, and explains the low-level implementation of iterators. For more details, see the next volume in this series.

Advanced Arithmetic

Chapter Four

4.1 Chapter Overview

This chapter deals with those arithmetic operations for which assembly language is especially well suited and high level languages are, in general, poorly suited. It covers three main topics: extended precision arithmetic, arithmetic on operands whose sizes are different, and decimal arithmetic.

By far, the most extensive subject this chapter covers is multi-precision arithmetic. By the conclusion of this chapter you will know how to apply arithmetic and logical operations to integer operands of any size. If you need to work with integer values outside the range ± 2 billion (or with unsigned values beyond four billion), no sweat; this chapter will show you how to get the job done.

Operands whose sizes are not the same also present some special problems in arithmetic operations. For example, you may want to add a 128-bit unsigned integer to a 256-bit signed integer value. This chapter discusses how to convert these two operands to a compatible format so the operation may proceed.

Finally, this chapter discusses decimal arithmetic using the BCD (binary coded decimal) features of the 80x86 instruction set and the FPU. This lets you use decimal arithmetic in those few applications that absolutely require base 10 operations (rather than binary).

4.2 Multiprecision Operations

One big advantage of assembly language over high level languages is that assembly language does not limit the size of integer operations. For example, the C programming language defines a maximum of three different integer sizes: short int, int, and long int¹. On the PC, these are often 16 and 32 bit integers. Although the 80x86 machine instructions limit you to processing eight, sixteen, or thirty-two bit integers with a single instruction, you can always use more than one instruction to process integers of any size you desire. If you want to add 256 bit integer values together, no problem, it's relatively easy to accomplish this in assembly language. The following sections describe how extended various arithmetic and logical operations from 16 or 32 bits to as many bits as you please.

4.2.1 Multiprecision Addition Operations

The 80x86 ADD instruction adds two eight, sixteen, or thirty-two bit numbers². After the execution of the add instruction, the 80x86 carry flag is set if there is an overflow out of the H.O. bit of the sum. You can use this information to do multiprecision addition operations. Consider the way you manually perform a multidigit (multiprecision) addition operation:

Step 1: Add the least significant digits together:

289		289
+456	produces	+456
----		----
		5 with carry 1.

1. Newer C standards also provide for a "long long int" which is usually a 64-bit integer.

2. As usual, 32 bit arithmetic is available only on the 80386 and later processors.

Step 2: Add the next significant digits plus the carry:

1 (previous carry)		
289	produces	289
+456		+456
----		----
5		45 with carry 1.

Step 3: Add the most significant digits plus the carry:

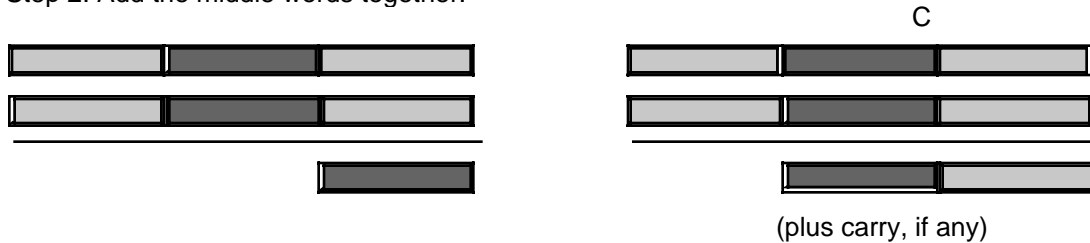
		1 (previous carry)
289	produces	289
+456		+456
----		----
45		745

The 80x86 handles extended precision arithmetic in an identical fashion, except instead of adding the numbers a digit at a time, it adds them together a byte, word, or dword at a time. Consider the three double word (96 bit) addition operation in Figure 4.1.

Step 1: Add the least significant words together:



Step 2: Add the middle words together:



Step 3: Add the most significant words together:

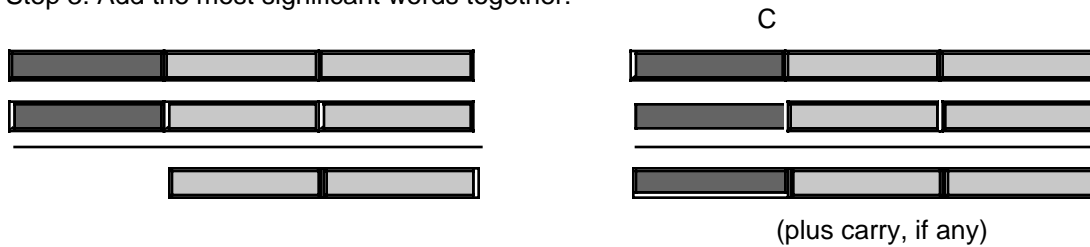


Figure 4.1 Adding Two 96-bit Objects Together

As you can see from this figure, the idea is to break up a larger operation into a sequence of smaller operations. Since the x86 processor family is capable of adding together, at most, 32 bits at a time, the operation must proceed in blocks of 32-bits or less. So the first step is to add the two L.O. double words together

much as we would add the two L.O. digits of a decimal number together in the manual algorithm. There is nothing special about this operation, you can use the ADD instruction to achieve this.

The second step involves adding together the second pair of double words in the two 96-bit values. Note that in step two, the calculation must also add in the carry out of the previous addition (if any). If there was a carry out of the L.O. addition, the ADD instruction sets the carry flag to one; conversely, if there was no carry out of the L.O. addition, the earlier ADD instruction clears the carry flag. Therefore, in this second addition, we really need to compute the sum of the two double words plus the carry out of the first instruction. Fortunately, the x86 CPUs provide an instruction that does exactly this: the ADC (add with carry) instruction. The ADC instruction uses the same syntax as the ADD instruction and performs almost the same operation:

```
adc( source, dest ); // dest := dest + source + C
```

As you can see, the only difference between the ADD and ADC instruction is that the ADC instruction adds in the value of the carry flag along with the source and destination operands. It also sets the flags the same way the ADD instruction does (including setting the carry flag if there is an unsigned overflow). This is exactly what we need to add together the middle two double words of our 96-bit sum.

In step three of Figure 4.1, the algorithm adds together the H.O. double words of the 96-bit value. Once again, this addition operation also requires the addition of the carry out of the sum of the middle two double words; hence the ADC instruction is needed here, as well. To sum it up, the ADD instruction adds the L.O. double words together. The ADC (add with carry) instruction adds all other double word pairs together. At the end of the extended precision addition sequence, the carry flag indicates unsigned overflow (if set), a set overflow flag indicates signed overflow, and the sign flag indicates the sign of the result. The zero flag doesn't have any real meaning at the end of the extended precision addition (it simply means that the sum of the H.O. two double words is zero, this does not indicate that the whole result is zero).

For example, suppose that you have two 64-bit values you wish to add together, defined as follows:

```
static
X: qword;
Y: qword;
```

Suppose, also, that you want to store the sum in a third variable, Z, that is likewise defined with the qword type. The following x86 code will accomplish this task:

```
mov( (type dword X), eax ); // Add together the L.O. 32 bits
add( (type dword Y), eax ); // of the numbers and store the
mov( eax, (type dword Z) ); // result into the L.O. dword of Z.

mov( (type dword X[4]), eax ); // Add together (with carry) the
adc( (type dword Y[4]), eax ); // H.O. 32 bits and store the result
mov( eax, (type dword Z[4]) ); // into the H.O. dword of Z.
```

Remember, these variables are qword objects. Therefore the compiler will not accept an instruction of the form "mov(X, eax);" because this instruction would attempt to load a 64 bit value into a 32 bit register. This code uses the coercion operator to coerce symbols X, Y, and Z to 32 bits. The first three instructions add the L.O. double words of X and Y together and store the result at the L.O. double word of Z. The last three instructions add the H.O. double words of X and Y together, along with the carry out of the L.O. word, and store the result in the H.O. double word of Z. Remember, address expressions of the form "X[4]" access the H.O. double word of a 64 bit entity. This is due to the fact that the x86 address space addresses bytes and it takes four consecutive bytes to form a double word.

You can extend this to any number of bits by using the ADC instruction to add in the higher order words in the values. For example, to add together two 128 bit values, you could use code that looks something like the following:

```
type
tBig: dword[4]; // Storage for four dwords is 128 bits.

static
```

```

BigVal1: tBig;
BigVal2: tBig;
BigVal3: tBig;
.
.
.
mov( BigVal1[0], eax ); // Note there is no need for (type dword BigValx)
add( BigVal2[0], eax ); // because the base type of BitValx is dword.
mov( eax, BigVal3[0] );

mov( BigVal1[4], eax );
adc( BigVal2[4], eax );
mov( eax, BigVal3[4] );

mov( BigVal1[8], eax );
adc( BigVal2[8], eax );
mov( eax, BigVal3[8] );

mov( BigVal1[12], eax );
adc( BigVal2[12], eax );
mov( eax, BigVal3[12] );

```

4.2.2 Multiprecision Subtraction Operations

Like addition, the 80x86 performs multi-byte subtraction the same way you would manually, except it subtracts whole bytes, words, or double words at a time rather than decimal digits. The mechanism is similar to that for the ADD operation, You use the SUB instruction on the L.O. byte/word/double word and the SBB (subtract with borrow) instruction on the high order values. The following example demonstrates a 64 bit subtraction using the 32 bit registers on the x86:

```

static
Left:  qword;
Right: qword;
Diff:  qword;
.
.
.
mov( (type dword Left),  eax );
sub( (type dword Right), eax );
mov( eax, (type dword Diff) );

mov( (type dword Left[4]),  eax );
sbb( (type dword Right[4]), eax );
mov( (type dword Diff[4]),  eax );

```

The following example demonstrates a 128-bit subtraction:

```

type
tBig: dword[4]; // Storage for four dwords is 128 bits.

static
BigVal1: tBig;
BigVal2: tBig;
BigVal3: tBig;
.
.
.

// Compute BigVal3 := BigVal1 - BigVal2

```



```

mov( BigVal1[0], eax ); // Note there is no need for (type dword BigValx)
sub( BigVal2[0], eax ); // because the base type of BitValx is dword.
mov( eax, BigVal3[0] );

mov( BigVal1[4], eax );
sbb( BigVal2[4], eax );
mov( eax, BigVal3[4] );

mov( BigVal1[8], eax );
sbb( BigVal2[8], eax );
mov( eax, BigVal3[8] );

mov( BigVal1[12], eax );
sbb( BigVal2[12], eax );
mov( eax, BigVal3[12] );

```

4.2.3 Extended Precision Comparisons

Unfortunately, there isn't a "compare with borrow" instruction that you can use to perform extended precision comparisons. Since the CMP and SUB instructions perform the same operation, at least as far as the flags are concerned, you'd probably guess that you could use the SBB instruction to synthesize an extended precision comparison; however, you'd only be partly right. There is, however, a better way.

Consider the two unsigned values \$2157 and \$1293. The L.O. bytes of these two values do not affect the outcome of the comparison. Simply comparing \$21 with \$12 tells us that the first value is greater than the second. In fact, the only time you ever need to look at both bytes of these values is if the H.O. bytes are equal. In all other cases comparing the H.O. bytes tells you everything you need to know about the values. Of course, this is true for any number of bytes, not just two. The following code compares two unsigned 64 bit integers:

```

// This sequence transfers control to location "IsGreater" if
// QwordValue > QwordValue2. It transfers control to "IsLess" if
// QwordValue < QwordValue2. It falls through to the instruction
// following this sequence if QwordValue = QwordValue2. To test for
// inequality, change the "IsGreater" and "IsLess" operands to "NotEqual"
// in this code.

mov( (type dword QwordValue[4]), eax ); // Get H.O. dword
cmp( eax, (type dword QwordValue2[4]));
jg IsGreater;
jl IsLess;

mov( (type dword QwordValue[0]), eax ); // If H.O. dwords were equal,
cmp( eax, (type dword QwordValue2[0])); // then we must compare the
ja IsGreater; // L.O. dwords.
jb IsLess;

// Fall through to this point if the two values were equal.

```

To compare signed values, simply use the JG and JL instructions in place of JA and JB for the H.O. words (only). You must continue to use unsigned comparisons for all but the H.O. double words you're comparing.

You can easily synthesize any possible comparison from the sequence above, the following examples show how to do this. These examples demonstrate signed comparisons, substitute JA, JAE, JB, and JBE for JG, JGE, JL, and JLE (respectively) for the H.O. comparisons if you want unsigned comparisons.

```
static
```

```

    QW1: qword;
    QW2: qword;

const
    QW1d: text := "(type dword QW1)";
    QW2d: text := "(type dword QW2)";

// 64 bit test to see if QW1 < QW2 (signed).
// Control transfers to "IsLess" label if QW1 < QW2. Control falls
// through to the next statement (at "NotLess") if this is not true.

    mov( QW1d[4], eax ); // Get H.O. dword
    cmp( eax, QW2d[4] );
    jg NotLess; // Substitute ja here for unsigned comparison.
    jl IsLess; // Substitute jb here for unsigned comparison.

    mov( QW1d[0], eax ); // Fall through to here if the H.O. dwords are equal.
    cmp( eax, QW2d[0] );
    jb IsLess;
NotLess:

// 64 bit test to see if QW1 <= QW2 (signed). Jumps to "IsLessEq" if the
// condition is true.

    mov( QW1d[4], eax ); // Get H.O. dword
    cmp( eax, QW2d[4] );
    jg NotLessEQ; // Substitute ja here for unsigned comparison.
    jl IsLessEQ; // Substitute jb here for unsigned comparison.

    mov( QW1d[0], eax ); // Fall through to here if the H.O. dwords are equal.
    cmp( eax, QW2d[0] );
    jbe IsLessEQ;
NotLessEQ:

// 64 bit test to see if QW1 > QW2 (signed). Jumps to "IsGtr" if this condition
// is true.

    mov( QW1d[4], eax ); // Get H.O. dword
    cmp( eax, QW2d[4] );
    jg IsGtr; // Substitute ja here for unsigned comparison.
    jl NotGtr; // Substitute jb here for unsigned comparison.

    mov( QW1d[0], eax ); // Fall through to here if the H.O. dwords are equal.
    cmp( eax, QW2d[0] );
    ja IsGtr;
NotGtr:

// 64 bit test to see if QW1 >= QW2 (signed). Jumps to "IsGtrEQ" if this
// is the case.

    mov( QW1d[4], eax ); // Get H.O. dword
    cmp( eax, QW2d[4] );
    jg IsGtrEQ; // Substitute ja here for unsigned comparison.
    jl NotGtrEQ; // Substitute jb here for unsigned comparison.

    mov( QW1d[0], eax ); // Fall through to here if the H.O. dwords are equal.
    cmp( eax, QW2d[0] );
    jae IsGtrEQ;
NotGtrEQ:

```

```

// 64 bit test to see if QW1 = QW2 (signed or unsigned). This code branches
// to the label "IsEqual" if QW1 = QW2. It falls through to the next instruction
// if they are not equal.

    mov( QW1d[4], eax ); // Get H.O. dword
    cmp( eax, QW2d[4] );
    jne NotEqual;

    mov( QW1d[0], eax ); // Fall through to here if the H.O. dwords are equal.
    cmp( eax, QW2d[0] );
    je IsEqual;
NotEqual:

// 64 bit test to see if QW1 <> QW2 (signed or unsigned). This code branches
// to the label "NotEqual" if QW1 <> QW2. It falls through to the next
// instruction if they are equal.

    mov( QW1d[4], eax ); // Get H.O. dword
    cmp( eax, QW2d[4] );
    jne NotEqual;

    mov( QW1d[0], eax ); // Fall through to here if the H.O. dwords are equal.
    cmp( eax, QW2d[0] );
    jne NotEqual;

// Fall through to this point if they are equal.

```

You cannot directly use the HLA high level control structures if you need to perform an extended precision comparison. However, you may use the HLA hybrid control structures and bury the appropriate comparison into this statements. Doing so will probably make your code easier to read. For example, the following *if..then..else..endif* statement checks to see if *QW1 > QW2* using a 64-bit extended precision signed comparison:

```

if
( #{
    mov( QW1d[4], eax );
    cmp( eax, QW2d[4] );
    jg true;

    mov( QW1d[0], eax );
    cmp( eax, QW2d[0] );
    jna false;
}# ) then

    << code to execute if QW1 > QW2 >>

else

    << code to execute if QW1 <= QW2 >>

endif;

```

If you need to compare objects that are larger than 64 bits, it is very easy to generalize the code above. Always start the comparison with the H.O. double words of the objects and work you way down towards the L.O. double words of the objects as long as the corresponding double words are equal. The following example compares two 128-bit values to see if the first is less than or equal (unsigned) to the second:

```
type
```

```

t128: dword[4];

static
Big1: t128;
Big2: t128;
.
.
.
if
( #{
  mov( Big1[12], eax );
  cmp( eax, Big2[12] );
  jb true;
  mov( Big1[8], eax );
  cmp( eax, Big2[8] );
  jb true;
  mov( Big1[4], eax );
  cmp( eax, Big2[4] );
  jb true;
  mov( Big1[0], eax );
  cmp( eax, Big2[0] );
  jnbe false;
}# ) then

  << Code to execute if Big1 <= Big2 >>

else

  << Code to execute if Big1 > Big2 >>

endif;

```

4.2.4 Extended Precision Multiplication

Although an 8x8, 16x16, or 32x32 multiply is usually sufficient, there are times when you may want to multiply larger values together. You will use the x86 single operand MUL and IMUL instructions for extended precision multiplication.

Not surprisingly (in view of how we achieved extended precision addition using ADC and SBB), you use the same techniques to perform extended precision multiplication on the x86 that you employ when manually multiplying two values. Consider a simplified form of the way you perform multi-digit multiplication by hand:

1) Multiply the first two digits together (5*3):

```

  123
   45
  ---
   15

```

2) Multiply 5*2:

```

  123
   45
  ---
   15
  10

```

3) Multiply 5*1:

```

123
 45
---
 15
 10
  5

```

4) Multiply 4*3:

```

123
 45
---
 15
 10
  5
 12

```

5) Multiply 4*2:

```

123
 45
---
 15
 10
  5
 12
  8

```

6) Multiply 4*1:

```

123
 45
---
 15
 10
  5
 12
  8
  4

```

7) Add all the partial products together:

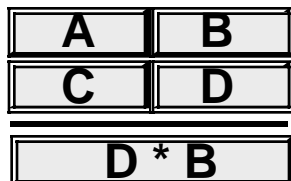
```

123
 45
---
 15
 10
  5
 12
  8
  4
-----
5535

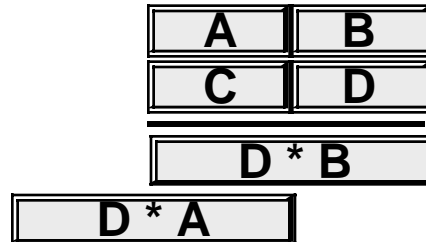
```

The 80x86 does extended precision multiplication in the same manner except that it works with bytes, words, and double words rather than digits. Figure 4.2 shows how this works

1) Multiply the L.O. words



2) Multiply D * A



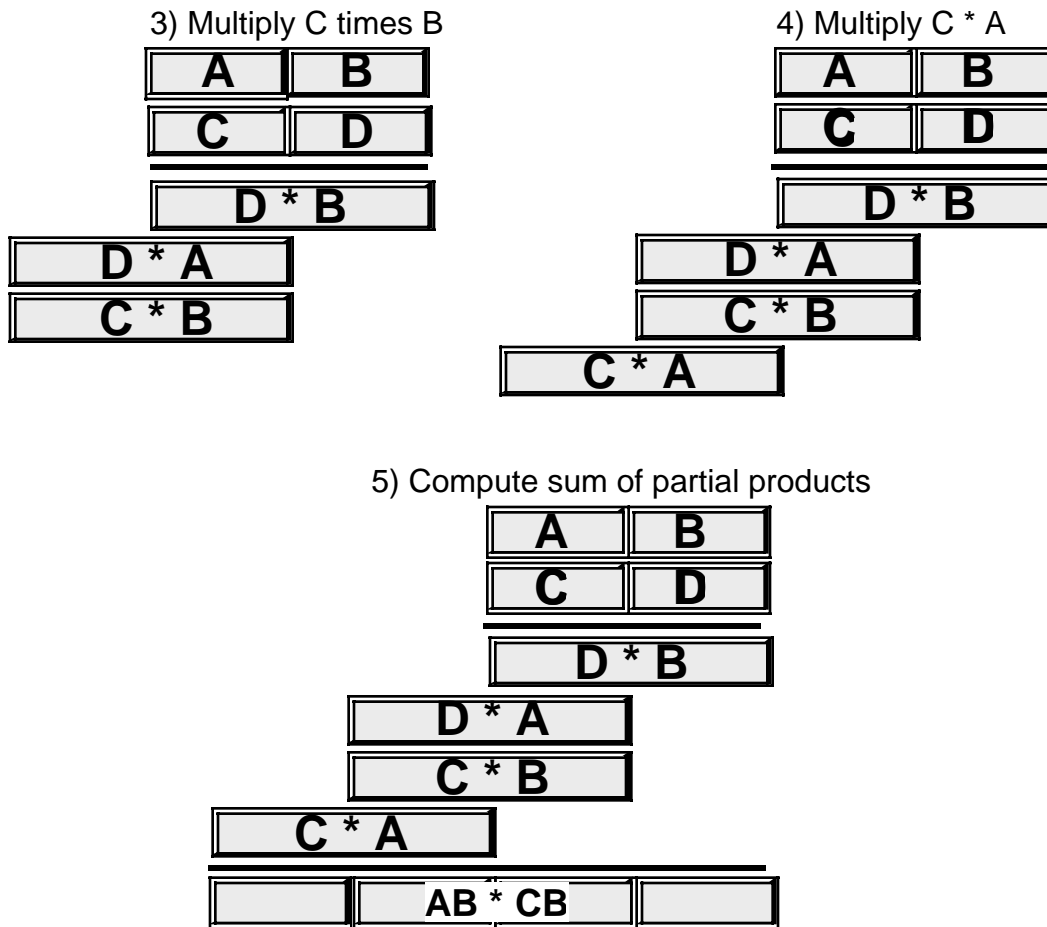


Figure 4.2 Extended Precision Multiplication

Probably the most important thing to remember when performing an extended precision multiplication is that you must also perform a multiple precision addition at the same time. Adding up all the partial products requires several additions that will produce the result. The following listing demonstrates the proper way to multiply two 64 bit values on a 32 bit processor:

Note: *Multiplier* and *Multiplicand* are 64 bit variables declared in the data segment via the `qword` type. *Product* is a 128 bit variable declared in the data segment via the `qword[2]` type.

```

program testMUL64;
#include( "stdlib.hhf" )

type
  t128:dword[4];

procedure MUL64( Multiplier:qword; Multiplicand:qword; var Product:t128 );
const
  mp:text := "(type dword Multiplier)";
  mc:text := "(type dword Multiplicand)";
  prd:text := "(type dword [edi])";

```

```

begin MUL64;

    mov( Product, edi );

    // Multiply the L.O. dword of Multiplier times Multiplicand.

    mov( mp, eax );
    mul( mc, eax );    // Multiply L.O. dwords.
    mov( eax, prd );  // Save L.O. dword of product.
    mov( edx, ecx );  // Save H.O. dword of partial product result.

    mov( mp, eax );
    mul( mc[4], eax ); // Multiply mp(L.O.) * mc(H.O.)
    add( ecx, eax );   // Add to the partial product.
    adc( 0, edx );     // Don't forget the carry!
    mov( eax, ebx );   // Save partial product for now.
    mov( edx, ecx );

    // Multiply the H.O. word of Multiplier with Multiplicand.

    mov( mp[4], eax ); // Get H.O. dword of Multiplier.
    mul( mc, eax );    // Multiply by L.O. word of Multiplicand.
    add( ebx, eax );   // Add to the partial product.
    mov( eax, prd[4] ); // Save the partial product.
    adc( edx, ecx );   // Add in the carry!
    pushfd();          // Save carry out here.

    mov( mp[4], eax ); // Multiply the two H.O. dwords together.
    mul( mc[4], eax );
    popfd();           // Retrieve carry from above
    adc( ecx, eax );   // Add in partial product from above.
    adc( 0, edx );     // Don't forget the carry!
    mov( eax, prd[8] ); // Save the partial product.
    mov( edx, prd[12] );

end MUL64;

static
    op1: qword;
    op2: qword;
    rslt: t128;

begin testMUL64;

    // Initialize the qword values (note that static objects
    // are initialized with zero bits).

    mov( 1234, (type dword op1) );
    mov( 5678, (type dword op2) );
    MUL64( op1, op2, rslt );

    // The following only prints the L.O. qword, but
    // we know the H.O. qword is zero so this is okay.

    stdout.put( "rslt=" );
    stdout.putu64( (type qword rslt) );

end testMUL64;

```

Program 4.1 Extended Precision Multiplication

One thing you must keep in mind concerning this code, it only works for unsigned operands. To multiply two signed values you must note the signs of the operands before the multiplication, take the absolute value of the two operands, do an unsigned multiplication, and then adjust the sign of the resulting product based on the signs of the original operands. Multiplication of signed operands appears in the exercises.

This example was fairly straight-forward since it was possible to keep the partial products in various registers. If you need to multiply larger values together, you will need to maintain the partial products in temporary (memory) variables. Other than that, the algorithm that Program 4.1 uses generalizes to any number of double words.

4.2.5 Extended Precision Division

You cannot synthesize a general n-bit/m-bit division operation using the DIV and IDIV instructions. Such an operation must be performed using a sequence of shift and subtract instructions and is extremely messy. However, a less general operation, dividing an n-bit quantity by a 32 bit quantity is easily synthesized using the DIV instruction. This section presents both methods for extended precision division.

Before describing how to perform a multi-precision division operation, you should note that some operations require an extended precision division even though they may look calculable with a single DIV or IDIV instruction. Dividing a 64-bit quantity by a 32-bit quantity is easy, as long as the resulting quotient fits into 32 bits. The DIV and IDIV instructions will handle this directly. However, if the quotient does not fit into 32 bits then you have to handle this problem as an extended precision division. The trick here is to divide the (zero or sign extended) H.O dword of the dividend by the divisor, and then repeat the process with the remainder and the L.O. dword of the dividend. The following sequence demonstrates this:

```
static
dividend: dword[2] := [$1234, 4]; // = $4_0000_1234.
divisor:  dword := 2;           // dividend/divisor = $2_0000_091A
quotient: dword[2];
remainder:dword;
.
.
.
mov( divisor, ebx );
mov( dividend[4], eax );
xor( edx, edx );           // Zero extend for unsigned division.
div( ebx, edx:eax );
mov( eax, quotient[4] );   // Save H.O. dword of the quotient (2).
mov( dividend[0], eax );  // Note that this code does *NOT* zero extend
div( ebx, edx:eax );      // EAX into EDX before this DIV instr.
mov( eax, quotient[0] );  // Save L.O. dword of the quotient ($91a).
mov( edx, remainder );    // Save away the remainder.
```

Since it is perfectly legal to divide a value by one, it is certainly possible that the resulting quotient after a division could require as many bits as the dividend. That is why the *quotient* variable in this example is the same size (64 bits) as the *dividend* variable. Regardless of the size of the dividend and divisor operands, the remainder is always no larger than the size of the division operation (32 bits in this case). Hence the *remainder* variable in this example is just a double word.

Before analyzing this code to see how it works, let's take a brief look at why a single 64/32 division will *not* work for this particular example even though the DIV instruction does indeed calculate the result for a 64/32 division. The naive approach, assuming that the x86 were capable of this operation, would look something like the following:

```
// This code does *NOT* work!
```



```

mov( dividend[0], eax );    // Get dividend into edx:eax
mov( dividend[4], edx );
div( divisor, edx:eax );    // Divide edx:eax by divisor.

```

Although this code is syntactically correct and will compile, if you attempt to run this code it will raise an *ex.DivideError* exception. The reason, if you'll remember how the DIV instruction works, is that the quotient must fit into 32 bits; since the quotient turns out to be \$2_0000_091A, it will not fit into the EAX register, hence the resulting exception.

Now let's take another look at the former code that correctly computes the 64/32 quotient. This code begins by computing the 32/32 quotient of *dividend[4]/divisor*. The quotient from this division (2) becomes the H.O. double word of the final quotient. The remainder from this division (0) becomes the extension in EDX for the second half of the division operation. The second half divides *edx:dividend[0]* by *divisor* to produce the L.O. double word of the quotient and the remainder from the division. Note that the code does not zero extend EAX into EDX prior to the second DIV instruction. EDX already contains valid bits and this code must not disturb them.

The 64/32 division operation above is actually just a special case of the more general division operation that lets you divide an arbitrary sized value by a 32-bit divisor. To achieve this, you begin by moving the H.O. double word of the dividend into EAX and zero extending this into EDX. Next, you divide this value by the divisor. Then, without modifying EDX along the way, you store away the partial quotients, load EAX with the next lower double word in the dividend, and divide it by the divisor. You repeat this operation until you've processed all the double words in the dividend. At that time the EDX register will contain the remainder. The following program demonstrates how to divide a 128 bit quantity by a 32 bit divisor, producing a 128 bit quotient and a 32 bit remainder:

```

program testDiv128;
#include( "stdlib.hhf" )

type
  t128:dword[4];

procedure div128
(
  Dividend:  t128;
  Divisor:   dword;
  var QuotAdrs: t128;
  var Remainder: dword
); @nodisplay;

const
  Quotient: text := "(type dword [edi])";

begin div128;

  push( eax );
  push( edx );
  push( edi );

  mov( QuotAdrs, edi );    // Pointer to quotient storage.

  mov( Dividend[12], eax ); // Begin division with the H.O. dword.
  xor( edx, edx );         // Zero extend into EDX.
  div( Divisor, edx:eax ); // Divide H.O. dword.
  mov( eax, Quotient[12] ); // Store away H.O. dword of quotient.

  mov( Dividend[8], eax ); // Get dword #2 from the dividend
  div( Divisor, edx:eax ); // Continue the division.

```

```

mov( eax, Quotient[8] ); // Store away dword #2 of the quotient.

mov( Dividend[4], eax ); // Get dword #1 from the dividend.
div( Divisor, edx:eax ); // Continue the division.
mov( eax, Quotient[4] ); // Store away dword #1 of the quotient.

mov( Dividend[0], eax ); // Get the L.O. dword of the dividend.
div( Divisor, edx:eax ); // Finish the division.
mov( eax, Quotient[0] ); // Store away the L.O. dword of the quotient.

mov( Remainder, edi ); // Get the pointer to the remainder's value.
mov( edx, [edi] ); // Store away the remainder value.

pop( edi );
pop( edx );
pop( eax );

end div128;

static
op1:  t128    := [$2222_2221, $4444_4444, $6666_6666, $8888_8888];
op2:  dword   := 2;
quo:  t128;
rmndr: dword;

begin testDiv128;

div128( op1, op2, quo, rmndr );

stdout.put
(
  nl
  nl
  "After the division: " nl
  nl
  "Quotient = $",
  quo[12], "_",
  quo[8], "_",
  quo[4], "_",
  quo[0], nl

  "Remainder = ", (type uns32 rmndr )
);

end testDiv128;

```

Program 4.2 Unsigned 128/32 Bit Extended Precision Division

You can extend this code to any number of bits by simply adding additional MOV / DIV / MOV instructions to the sequence. Like the extended multiplication the previous section presents, this extended precision division algorithm works only for unsigned operands. If you need to divide two signed quantities, you must note their signs, take their absolute values, do the unsigned division, and then set the sign of the result based on the signs of the operands.

If you need to use a divisor larger than 32 bits you're going to have to implement the division using a shift and subtract strategy. Unfortunately, such algorithms are very slow. In this section we'll develop two division algorithms that operate on an arbitrary number of bits. The first is slow but easier to understand, the second is quite a bit faster (in general).

As for multiplication, the best way to understand how the computer performs division is to study how you were taught to perform long division by hand. Consider the operation $3456/12$ and the steps you would take to manually perform this operation:

$\begin{array}{r} 12 \overline{)3456} \\ \underline{24} \\ 105 \end{array}$ <p>(1) 12 goes into 34 two times.</p>	$\begin{array}{r} 2 \\ 12 \overline{)3456} \\ \underline{24} \\ 105 \end{array}$ <p>(2) Subtract 24 from 35 and drop down the 105.</p>
$\begin{array}{r} 28 \\ 12 \overline{)3456} \\ \underline{24} \\ 105 \\ \underline{96} \\ 96 \end{array}$ <p>(3) 12 goes into 105 eight times.</p>	$\begin{array}{r} 28 \\ 12 \overline{)3456} \\ \underline{24} \\ 105 \\ \underline{96} \\ 96 \end{array}$ <p>(4) Subtract 96 from 105 and drop down the 96.</p>
$\begin{array}{r} 28 \\ 12 \overline{)3456} \\ \underline{24} \\ 105 \\ \underline{96} \\ 96 \end{array}$ <p>(5) 12 goes into 96 exactly eight times.</p>	$\begin{array}{r} 288 \\ 12 \overline{)3456} \\ \underline{24} \\ 105 \\ \underline{96} \\ 96 \end{array}$ <p>(6) Therefore, 12 goes into 3456 exactly 288 times.</p>

Figure 4.3 Manual Digit-by-digit Division Operation

This algorithm is actually easier in binary since at each step you do not have to guess how many times 12 goes into the remainder nor do you have to multiply 12 by your guess to obtain the amount to subtract. At each step in the binary algorithm the divisor goes into the remainder exactly zero or one times. As an example, consider the division of 27 (11011) by three (11):

$\begin{array}{r} 11 \overline{)11011} \\ \underline{11} \\ 00000 \end{array}$	11 goes into 11 one time.
$\begin{array}{r} 1 \\ 11 \overline{)11011} \\ \underline{11} \\ 00 \end{array}$	Subtract out the 11 and bring down the zero.
$\begin{array}{r} 1 \\ 11 \overline{)11011} \\ \underline{11} \\ 00 \\ \underline{00} \\ 00 \end{array}$	11 goes into 00 zero times.

$$\begin{array}{r}
 10 \\
 11 \overline{) 11011} \\
 \underline{11} \\
 00 \\
 \underline{00} \\
 01
 \end{array}$$

Subtract out the zero and bring down the one.

$$\begin{array}{r}
 10 \\
 11 \overline{) 11011} \\
 \underline{11} \\
 00 \\
 \underline{00} \\
 01 \\
 \underline{00}
 \end{array}$$

11 goes into 01 zero times.

$$\begin{array}{r}
 100 \\
 11 \overline{) 11011} \\
 \underline{11} \\
 00 \\
 \underline{00} \\
 01 \\
 \underline{00} \\
 11
 \end{array}$$

Subtract out the zero and bring down the one.

$$\begin{array}{r}
 100 \\
 11 \overline{) 11011} \\
 \underline{11} \\
 00 \\
 \underline{00} \\
 01 \\
 \underline{00} \\
 11 \\
 \underline{11} \\
 11
 \end{array}$$

11 goes into 11 one time.

$$\begin{array}{r}
 1001 \\
 11 \overline{) 11011} \\
 \underline{11} \\
 00 \\
 \underline{00} \\
 01 \\
 \underline{00} \\
 11 \\
 \underline{11} \\
 00
 \end{array}$$

This produces the final result of 1001.

Figure 4.4 Longhand Division in Binary

There is a novel way to implement this binary division algorithm that computes the quotient and the remainder at the same time. The algorithm is the following:

```

Quotient := Dividend;
Remainder := 0;
for i:= 1 to NumberBits do

```

```

Remainder:Quotient := Remainder:Quotient SHL 1;
if Remainder >= Divisor then

    Remainder := Remainder - Divisor;
    Quotient := Quotient + 1;

endif
endfor

```

NumberBits is the number of bits in the *Remainder*, *Quotient*, *Divisor*, and *Dividend* variables. Note that the "Quotient := Quotient + 1;" statement sets the L.O. bit of *Quotient* to one since this algorithm previously shifts *Quotient* one bit to the left. The following program implements this algorithm

```

program testDiv128b;
#include( "stdlib.hhf" )

type
    t128:dword[4];

// div128-
//
// This procedure does a general 128/128 division operation
// using the following algorithm:
// (all variables are assumed to be 128 bit objects)
//
// Quotient := Dividend;
// Remainder := 0;
// for i:= 1 to NumberBits do
//
//     Remainder:Quotient := Remainder:Quotient SHL 1;
//     if Remainder >= Divisor then
//
//         Remainder := Remainder - Divisor;
//         Quotient := Quotient + 1;
//
//     endif
// endfor
//

procedure div128
(
    Dividend:  t128;
    Divisor:   t128;
    var QuotAdrs:  t128;
    var RmndrAdrs: t128
); @nodisplay;

const
    Quotient: text := "Dividend"; // Use the Dividend as the Quotient.

var
    Remainder: t128;

begin div128;

    push( eax );
    push( ecx );

```

```

push( edi );

mov( 0, eax );           // Set the remainder to zero.
mov( eax, Remainder[0] );
mov( eax, Remainder[4] );
mov( eax, Remainder[8] );
mov( eax, Remainder[12] );

mov( 128, ecx );        // Count off 128 bits in ECX.
repeat

    // Compute Remainder:Quotient := Remainder:Quotient SHL 1:

    shl( 1, Dividend[0] ); // See the section on extended
    rcl( 1, Dividend[4] ); // precision shifts to see how
    rcl( 1, Dividend[8] ); // this code shifts 256 bits to
    rcl( 1, Dividend[12] ); // the left by one bit.
    rcl( 1, Remainder[0] );
    rcl( 1, Remainder[4] );
    rcl( 1, Remainder[8] );
    rcl( 1, Remainder[12] );

    // Do a 128-bit comparison to see if the remainder
    // is greater than or equal to the divisor.

    if
    ( #{
        mov( Remainder[12], eax );
        cmp( eax, Divisor[12] );
        ja true;
        jb false;

        mov( Remainder[8], eax );
        cmp( eax, Divisor[8] );
        ja true;
        jb false;

        mov( Remainder[4], eax );
        cmp( eax, Divisor[4] );
        ja true;
        jb false;

        mov( Remainder[0], eax );
        cmp( eax, Divisor[0] );
        jb false;
    }# ) then

        // Remainder := Remainder - Divisor

        mov( Divisor[0], eax );
        sub( eax, Remainder[0] );

        mov( Divisor[4], eax );
        sbb( eax, Remainder[4] );

        mov( Divisor[8], eax );
        sbb( eax, Remainder[8] );

        mov( Divisor[12], eax );
        sbb( eax, Remainder[12] );

```

```

        // Quotient := Quotient + 1;

        add( 1, Quotient[0] );
        adc( 0, Quotient[4] );
        adc( 0, Quotient[8] );
        adc( 0, Quotient[12] );

        endif;
        dec( ecx );

until( @z );

// Okay, copy the quotient (left in the Dividend variable)
// and the remainder to their return locations.

mov( QuotAdrs, edi );
mov( Quotient[0], eax );
mov( eax, [edi] );
mov( Quotient[4], eax );
mov( eax, [edi+4] );
mov( Quotient[8], eax );
mov( eax, [edi+8] );
mov( Quotient[12], eax );
mov( eax, [edi+12] );

mov( RmndrAdrs, edi );
mov( Remainder[0], eax );
mov( eax, [edi] );
mov( Remainder[4], eax );
mov( eax, [edi+4] );
mov( Remainder[8], eax );
mov( eax, [edi+8] );
mov( Remainder[12], eax );
mov( eax, [edi+12] );

pop( edi );
pop( ecx );
pop( eax );

end div128;

// Some simple code to test out the division operation:

static
op1:   t128   := [$2222_2221, $4444_4444, $6666_6666, $8888_8888];
op2:   t128   := [2, 0, 0, 0];
quo:   t128;
rmndr: t128;

begin testDiv128b;

div128( op1, op2, quo, rmndr );

stdout.put
(
    nl

```

```

    nl
    "After the division: " nl
    nl
    "Quotient = $",
    quo[12], "_",
    quo[8], "_",
    quo[4], "_",
    quo[0], nl

    "Remainder = ", (type uns32 rmdr )
);

end testDiv128b;

```

Program 4.3 Extended Precision Division

This code looks simple but there are a few problems with it. First, it does not check for division by zero (it will produce the value \$FFFF_FFFF_FFFF_FFFF if you attempt to divide by zero), it only handles unsigned values, and it is very slow. Handling division by zero is very simple, just check the divisor against zero prior to running this code and return an appropriate error code if the divisor is zero (or RAISE the `ex.DivisionError` exception). Dealing with signed values is the same as the earlier division algorithm, this problem appears as a programming exercise. The performance of this algorithm, however, leaves a lot to be desired. It's around an order of magnitude or two worse than the DIV/IDIV instructions on the x86 and they are among the slowest instructions on the CPU.

There is a technique you can use to boost the performance of this division by a fair amount: check to see if the divisor variable uses only 32 bits. Often, even though the divisor is a 128 bit variable, the value itself fits just fine into 32 bits (i.e., the H.O. double words of Divisor are zero). In this special case, that occurs frequently, you can use the DIV instruction which is much faster.

4.2.6 Extended Precision NEG Operations

Although there are several ways to negate an extended precision value, the shortest way for smaller values (96 bits or less) is to use a combination of NEG and SBB instructions. This technique uses the fact that NEG subtracts its operand from zero. In particular, it sets the flags the same way the SUB instruction would if you subtracted the destination value from zero. This code takes the following form (assuming you want to negate the 64-bit value in EDX:EAX):

```

neg( edx );
neg( eax );
sbb( 0, edx );

```

The SBB instruction decrements EDX if there is a borrow out of the L.O. word of the negation operation (which always occurs unless EAX is zero).

To extend this operation to additional bytes, words, or double words is easy; all you have to do is start with the H.O. memory location of the object you want to negate and work towards the L.O. byte. The following code computes a 128 bit negation:

```

static
    Value: dword[4];
    .

```



```

.
.
neg( Value[12] );      // Negate the H.O. double word.
neg( Value[8] );      // Neg previous dword in memory.
sbb( 0, Value[12] );  // Adjust H.O. dword.

neg( Value[4] );      // Negate the second dword in the object.
sbb( 0, Value[8] );   // Adjust third dword in object.
sbb( 0, Value[12] );  // Adjust the H.O. dword.

neg( Value );        // Negate the L.O. dword.
sbb( 0, Value[4] );   // Adjust second dword in object.
sbb( 0, Value[8] );   // Adjust third dword in object.
sbb( 0, Value[12] );  // Adjust the H.O. dword.

```

Unfortunately, this code tends to get really large and slow since you need to propagate the carry through all the H.O. words after each negate operation. A simpler way to negate larger values is to simply subtract that value from zero:

```

static
Value: dword[5];     // 160-bit value.
.
.
.
mov( 0, eax );
sub( Value, eax );
mov( eax, Value );

mov( 0, eax );
sbb( Value[4], eax );
mov( eax, Value[4] );

mov( 0, eax );
sbb( Value[8], eax );
mov( eax, Value[8] );

mov( 0, eax );
sbb( Value[12], eax );
mov( eax, Value[12] );

mov( 0, eax );
sbb( Value[16], eax );
mov( eax, Value[16] );

```

4.2.7 Extended Precision AND Operations

Performing an n-byte AND operation is very easy – simply AND the corresponding bytes between the two operands, saving the result. For example, to perform the AND operation where all operands are 64 bits long, you could use the following code:

```

mov( (type dword source1), eax );
and( (type dword source2), eax );
mov( eax, (type dword dest) );

mov( (type dword source1[4]), eax );
and( (type dword source2[4]), eax );
mov( eax, (type dword dest[4]) );

```

This technique easily extends to any number of words, all you need to is logically AND the corresponding bytes, words, or double words together in the operands. Note that this sequence sets the flags according to the value of the last AND operation. If you AND the H.O. double words last, this sets all but the zero flag correctly. If you need to test the zero flag after this sequence, you will need to logically OR the two resulting double words together (or otherwise compare them both against zero).

4.2.8 Extended Precision OR Operations

Multi-byte logical OR operations are performed in the same way as multi-byte AND operations. You simply OR the corresponding bytes in the two operand together. For example, to logically OR two 96 bit values, use the following code:

```
mov( (type dword source1), eax );
or( (type dword source2), eax );
mov( eax, (type dword dest) );

mov( (type dword source1[4]), eax );
or( (type dword source2[4]), eax );
mov( eax, (type dword dest[4]) );

mov( (type dword source1[8]), eax );
or( (type dword source2[8]), eax );
mov( eax, (type dword dest[8]) );
```

As for the previous example, this does not set the zero flag properly for the entire operation. If you need to test the zero flag after a multiprecision OR, you must logically OR the resulting double words together.

4.2.9 Extended Precision XOR Operations

Extended precision XOR operations are performed in a manner identical to AND/OR – simply XOR the corresponding bytes in the two operands to obtain the extended precision result. The following code sequence operates on two 64 bit operands, computes their exclusive-or, and stores the result into a 64 bit variable.

```
mov( (type dword source1), eax );
xor( (type dword source2), eax );
mov( eax, (type dword dest) );

mov( (type dword source1[4]), eax );
xor( (type dword source2[4]), eax );
mov( eax, (type dword dest[4]) );
```

The comment about the zero flag in the previous two sections applies here.

4.2.10 Extended Precision NOT Operations

The NOT instruction inverts all the bits in the specified operand. An extended precision NOT is performed by simply executing the NOT instruction on all the affected operands. For example, to perform a 64 bit NOT operation on the value in (edx:eax), all you need to do is execute the instructions:

```
not( eax );
not( edx );
```

Keep in mind that if you execute the NOT instruction twice, you wind up with the original value. Also note that exclusive-ORing a value with all ones (\$FF, \$FFFF, or \$FFFF_FFFF) performs the same operation as the NOT instruction.

4.2.11 Extended Precision Shift Operations

Extended precision shift operations require a shift and a rotate instruction. Consider what must happen to implement a 64 bit SHL using 32 bit operations:

- 1) A zero must be shifted into bit zero.
- 2) Bits zero through 30 are shifted into the next higher bit.
- 3) Bit 31 is shifted into bit 32.
- 4) Bits 32 through 62 must be shifted into the next higher bit.
- 5) Bit 63 is shifted into the carry flag.

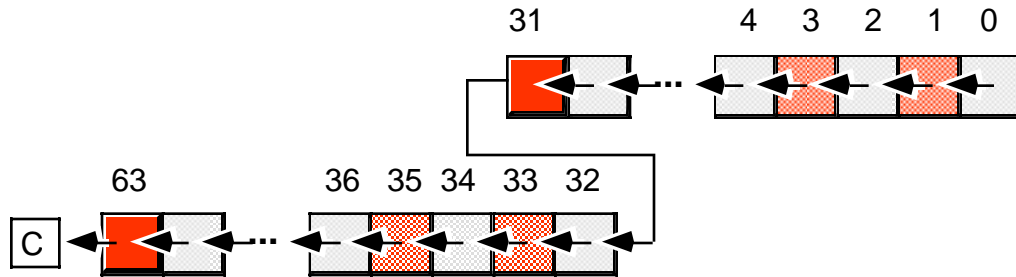


Figure 4.5 64-bit Shift Left Operation

The two instructions you can use to implement this 32 bit shift are SHL and RCL. For example, to shift the 64 bit quantity in (EDX:EAX) one position to the left, you'd use the instructions:

```
shl( 1, eax );
rcl( 1, eax );
```

Note that you can only shift an extended precision value one bit at a time. You cannot shift an extended precision operand several bits using the CL register. Nor can you specify a constant value greater than one using this technique.

To understand how this instruction sequence works, consider the operation of these instructions on an individual basis. The SHL instruction shifts a zero into bit zero of the 64 bit operand and shifts bit 31 into the carry flag. The RCL instruction then shifts the carry flag into bit 32 and then shifts bit 63 into the carry flag. The result is exactly what we want.

To perform a shift left on an operand larger than 64 bits you simply add additional RCL instructions. An extended precision shift left operation always starts with the least significant word and each succeeding RCL instruction operates on the next most significant word. For example, to perform a 96 bit shift left operation on a memory location you could use the following instructions:

```
shl( 1, (type dword Operand[0]) );
rcl( 1, (type dword Operand[4]) );
rcl( 1, (type dword Operand[8]) );
```

If you need to shift your data by two or more bits, you can either repeat the above sequence the desired number of times (for a constant number of shifts) or you can place the instructions in a loop to repeat them some number of times. For example, the following code shifts the 96 bit value *Operand* to the left the number of bits specified in ECX:

```
ShiftLoop:
  shl( 1, (type dword Operand[0]) );
  rcl( 1, (type dword Operand[4]) );
  rcl( 1, (type dword Operand[8]) );
```

```
dec( ecx );
jnz ShiftLoop;
```

You implement SHR and SAR in a similar way, except you must start at the H.O. word of the operand and work your way down to the L.O. word:

```
// Double precision SAR:

sar( 1, (type dword Operand[8]) );
rcr( 1, (type dword Operand[4]) );
rcr( 1, (type dword Operand[0]) );

// Double precision SHR:

shr( 1, (type dword Operand[8]) );
rcr( 1, (type dword Operand[4]) );
rcr( 1, (type dword Operand[0]) );
```

There is one major difference between the extended precision shifts described here and their 8/16/32 bit counterparts – the extended precision shifts set the flags differently than the single precision operations. This is because the rotate instructions affect the flags differently than the shift instructions. Fortunately, the carry is the flag most often tested after a shift operation and the extended precision shift operations (i.e., rotate instructions) properly set this flag.

The SHLD and SHRD instructions let you efficiently implement multiprecision shifts of several bits. These instructions have the following syntax:

```
shld( constant, Operand1, Operand2 );
shld( cl, Operand1, Operand2 );
shrd( constant, Operand1, Operand2 );
shrd( cl, Operand1, Operand2 );
```

The SHLD instruction does the following:

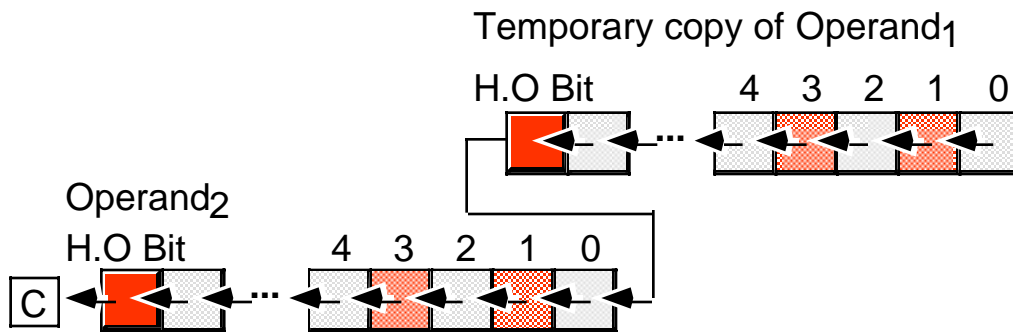


Figure 4.6 SHLD Operation

Operand₁ must be a 16 or 32 bit register. *Operand₂* can be a register or a memory location. Both operands must be the same size. The immediate operand can be a value in the range zero through n-1, where n is the number of bits in the two operands; it specifies the number of bits to shift.

The SHLD instruction shifts bits in *Operand₂* to the left. The H.O. bits shift into the carry flag and the H.O. bits of *Operand₁* shift into the L.O. bits of *Operand₂*. Note that this instruction does not modify the value of *Operand₁*, it uses a temporary copy of *Operand₁* during the shift. The immediate operand specifies the number of bits to shift. If the count is n, then SHLD shifts bit n-1 into the carry flag. It also shifts the H.O. n bits of *Operand₁* into the L.O. n bits of *Operand₂*. The SHLD instruction sets the flag bits as follows:

- If the shift count is zero, the SHLD instruction doesn't affect any flags.
- The carry flag contains the last bit shifted out of the H.O. bit of the *Operand₂*.
- If the shift count is one, the overflow flag will contain one if the sign bit of *Operand₂* changes during the shift. If the count is not one, the overflow flag is undefined.
- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result.

The SHRD instruction is similar to SHLD except, of course, it shifts its bits right rather than left. To get a clear picture of the SHRD instruction, consider Figure 4.7

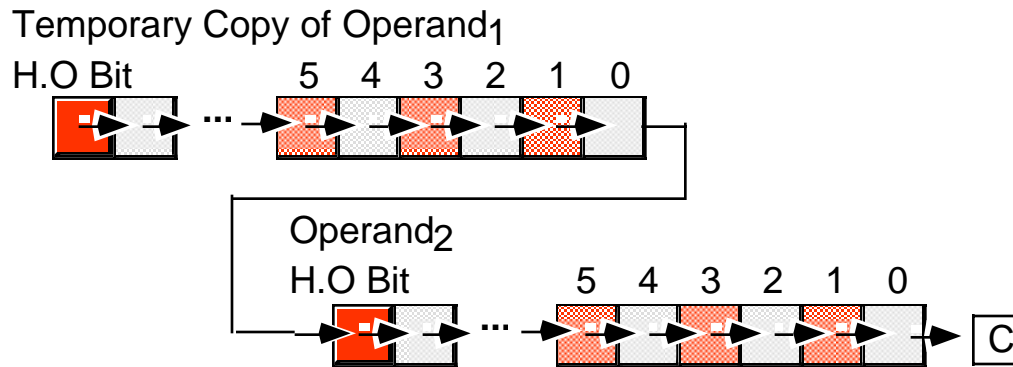


Figure 4.7 SHRD Operation

The SHRD instruction sets the flag bits as follows:

- If the shift count is zero, the SHRD instruction doesn't affect any flags.
- The carry flag contains the last bit shifted out of the L.O. bit of the *Operand₂*.
- If the shift count is one, the overflow flag will contain one if the H.O. bit of *Operand₂* changes. If the count is not one, the overflow flag is undefined.
- The zero flag will be one if the shift produces a zero result.
- The sign flag will contain the H.O. bit of the result.

Consider the following code sequence:

```
static
ShiftMe: dword[3] := [ $1234, $5678, $9012 ];
.
.
.
mov( ShiftMe[4], eax )
shld( 6, eax, ShiftMe[8] );
mov( ShiftMe[0], eax );
shld( 6, eax, ShiftMe[4] );
shl( 6, ShiftMe[0] );
```

The first SHLD instruction above shifts the bits from *ShiftMe+4* into *ShiftMe+8* without affecting the value in *ShiftMe+4*. The second SHLD instruction shifts the bits from SHIFTE into SHIFTE+4. Finally, the SHL instruction shifts the L.O. double word the appropriate amount. There are two important things to note about this code. First, unlike the other extended precision shift left operations, this sequence works from the H.O. double word down to the L.O. double word. Second, the carry flag does not contain the carry out of the H.O. shift operation. If you need to preserve the carry flag at that point, you will need to push the flags after the first SHLD instruction and pop the flags after the SHL instruction.

You can do an extended precision shift right operation using the SHRD instruction. It works almost the same way as the code sequence above except you work from the L.O. double word to the H.O. double word. The solution is left as an exercise.

4.2.12 Extended Precision Rotate Operations

The RCL and RCR operations extend in a manner almost identical to that for SHL and SHR . For example, to perform 96 bit RCL and RCR operations, use the following instructions:

```
rcl( 1, (type dword Operand[0]) );
rcl( 1, (type dword Operand[4]) );
rcl( 1, (type dword Operand[8]) );

rcr( 1, (type dword Operand[8]) );
rcr( 1, (type dword Operand[4]) );
rcr( 1, (type dword Operand[0]) );
```

The only difference between this code and the code for the extended precision shift operations is that the first instruction is a RCL or RCR rather than a SHL or SHR instruction.

Performing an extended precision ROL or ROR instruction isn't quite as simple an operation. You can use the BT, SHLD, and SHRD instructions to implement an extended precision ROL or ROR instruction. The following code shows how to use the SHLD instruction to do an extended precision ROL:

```
// Compute ROL( 4, EDX:EAX );

    mov( edx, ebx );
    shld, 4, eax, edx );
    shld( 4, ebx, eax );
    bt( 0, eax );           // Set carry flag, if desired.
```

An extended precision ROR instruction is similar; just keep in mind that you work on the L.O. end of the object first and the H.O. end last.

4.2.13 Extended Precision I/O

Once you have the ability to compute using extended precision arithmetic, the next problem is how do you get those extended precision values into your program and how do you display those extended precision values to the user? HLA's Standard Library provides routines for unsigned decimal, signed decimal, and hexadecimal I/O for values that are eight, 16, 32, or 64 bits in length. So as long as you're working with values whose size is less than or equal to 64 bits in length, you can use the Standard Library code. If you need to input or output values that are greater than 64 bits in length, you will need to write your own procedures to handle the operation. This section discusses the strategies you will need to write such routines.

The examples in this section work specifically with 128-bit values. The algorithms are perfectly general and extend to any number of bits (indeed, the 128-bit algorithms in this section are really nothing more than an extension of the algorithms the HLA Standard Library uses for 64-bit values). If you need a set of 128-bit unsigned I/O routines, you will probably be able to use the following code as-is. If you need to handle larger values, simple modifications to the following code is all that should be necessary.

The following examples all assume a common data type for 128-bit values. The HLA type declaration for this data type is one of the following depending on the type of value

```
type
    bits128: dword[4];
    uns128: bits128;
```

```
int128: bits128;
```

4.2.13.1 Extended Precision Hexadecimal Output

Extended precision hexadecimal output is very easy. All you have to do is output each double word component of the extended precision value from the H.O. double word to the L.O. double word using a call to the *stdout.putd* routine. The following procedure does exactly this to output a *bits128* value:

```
procedure putb128( b128: bits128 ); nodisplay;
begin putb128;

    stdout.putd( b128[12] );
    stdout.putd( b128[8] );
    stdout.putd( b128[4] );
    stdout.putd( b128[0] );

end putb128;
```

Since HLA provides the *stdout.putq* procedure, you can shorten the code above by calling *stdout.putq* just twice:

```
procedure putb128( b128: bits128 ); nodisplay;
begin putb128;

    stdout.putq( (type qword b128[8]) );
    stdout.putq( (type qword b128[0]) );

end putb128;
```

Note that this code outputs the two quad words with the H.O. quad word output first and L.O. quad word output second.

4.2.13.2 Extended Precision Unsigned Decimal Output

Decimal output is a little more complicated than hexadecimal output because the H.O. bits of a binary number affect the L.O. digits of the decimal representation (this was not true for hexadecimal values which is why hexadecimal output is so easy). Therefore, we will have to create the decimal representation for a binary number by extracting one decimal digit at a time from the number.

The most common solution for unsigned decimal output is to successively divide the value by ten until the result becomes zero. The remainder after the first division is a value in the range 0..9 and this value corresponds to the L.O. digit of the decimal number. Successive divisions by ten (and their corresponding remainder) extract successive digits in the number.

Iterative solutions to this problem generally allocate storage for a string of characters large enough to hold the entire number. Then the code extracts the decimal digits in a loop and places them in the string one by one. At the end of the conversion process, the routine prints the characters in the string in reverse order (remember, the divide algorithm extracts the L.O. digits first and the H.O. digits last, the opposite of the way you need to print them).

In this section, we will employ a recursive solution because it is a little more elegant. The recursive solution begins by dividing the value by 10 and saving the remainder in a local variable. If the quotient was not zero, the routine recursively calls itself to print any leading digits first. On return from the recursive call (which prints all the leading digits), the recursive algorithm prints the digit associated with the remainder to complete the operation. Here's how the operation works when printing the decimal value "123":

- (1) Divide 123 by 10. Quotient is 12, remainder is 3.
- (2) Save the remainder (3) in a local variable and recursively call the routine with the quotient.
- (3) [Recursive Entry 1] Divide 12 by 10. Quotient is 1, remainder is 2.
- (4) Save the remainder (2) in a local variable and recursively call the routine with the quotient.
- (5) [Recursive Entry 2] Divide 1 by 10. Quotient is 0, remainder is 1.
- (6) Save the remainder (1) in a local variable. Since the Quotient is zero, don't call the routine recursively.
- (7) Output the remainder value saved in the local variable (1). Return to the caller (Recursive Entry 1).
- (8) [Return to Recursive Entry 1] Output the remainder value saved in the local variable in recursive entry 1 (2). Return to the caller (original invocation of the procedure).
- (9) [Original invocation] Output the remainder value saved in the local variable in the original call (3). Return to the original caller of the output routine.

The only operation that requires extended precision calculation through this entire algorithm is the "divide by 10" requirement. Everything else is simple and straight-forward. We are in luck with this algorithm, since we are dividing an extended precision value by a value that easily fits into a double word, we can use the fast (and easy) extended precision division algorithm that uses the DIV instruction (see "Extended Precision Division" on page 864). The following program implements a 128-bit decimal output routine utilizing this technique.

```

program out128;

#include( "stdlib.hhf" );

// 128-bit unsigned integer data type:

type
    uns128: dword[4];

// DivideBy10-
//
// Divides "divisor" by 10 using fast
// extended precision division algorithm
// that employs the DIV instruction.
//
// Returns quotient in "quotient"
// Returns remainder in eax.
// Trashes EBX, EDX, and EDI.

procedure DivideBy10( dividend:uns128; var quotient:uns128 ); @nodisplay;
begin DivideBy10;

    mov( quotient, edi );
    xor( edx, edx );
    mov( dividend[12], eax );
    mov( 10, ebx );
    div( ebx, edx:eax );
    mov( eax, [edi+12] );

    mov( dividend[8], eax );
    div( ebx, edx:eax );
    mov( eax, [edi+8] );

    mov( dividend[4], eax );
    div( ebx, edx:eax );
    mov( eax, [edi+4] );

```



```

    mov( dividend[0], eax );
    div( ebx, edx:eax );
    mov( eax, [edi+0] );
    mov( edx, eax );

end DivideBy10;

// Recursive version of putul28.
// A separate "shell" procedure calls this so that
// this code does not have to preserve all the registers
// it uses (and DivideBy10 uses) on each recursive call.

procedure recursivePutul28( b128:uns128 ); @nodisplay;
var
    remainder: byte;

begin recursivePutul28;

    // Divide by ten and get the remainder (the char to print).

    DivideBy10( b128, b128 );
    mov( al, remainder );      // Save away the remainder (0..9).

    // If the quotient (left in b128) is not zero, recursively
    // call this routine to print the H.O. digits.

    mov( b128[0], eax );      // If we logically OR all the dwords
    or( b128[4], eax );       // together, the result is zero if and
    or( b128[8], eax );       // only if the entire number is zero.
    or( b128[12], eax );
    if( @nz ) then

        recursivePutul28( b128 );

    endif;

    // Okay, now print the current digit.

    mov( remainder, al );
    or( '0', al );           // Converts 0..9 -> '0..'9'.
    stdout.putc( al );

end recursivePutul28;

// Non-recursive shell to the above routine so we don't bother
// saving all the registers on each recursive call.

procedure putul28( b128:uns128 ); @nodisplay;
begin putul28;

    push( eax );
    push( ebx );
    push( edx );
    push( edi );

    recursivePutul28( b128 );

```

```

    pop( edi );
    pop( edx );
    pop( ebx );
    pop( eax );

end putul28;

// Code to test the routines above:

static
b0: uns128 := [0, 0, 0, 0];           // decimal = 0
b1: uns128 := [1234567890, 0, 0, 0]; // decimal = 1234567890
b2: uns128 := [$8000_0000, 0, 0, 0]; // decimal = 2147483648
b3: uns128 := [0, 1, 0, 0 ];        // decimal = 4294967296

// Largest uns128 value
// (decimal=340,282,366,920,938,463,463,374,607,431,768,211,455):

b4: uns128 := [$FFFF_FFFF, $FFFF_FFFF, $FFFF_FFFF, $FFFF_FFFF ];

begin out128;

    stdout.put( "b0 = " );
    putul28( b0 );
    stdout.newln();

    stdout.put( "b1 = " );
    putul28( b1 );
    stdout.newln();

    stdout.put( "b2 = " );
    putul28( b2 );
    stdout.newln();

    stdout.put( "b3 = " );
    putul28( b3 );
    stdout.newln();

    stdout.put( "b4 = " );
    putul28( b4 );
    stdout.newln();

end out128;

```

Program 4.4 128-bit Extended Precision Decimal Output Routine

4.2.13.3 Extended Precision Signed Decimal Output

Once you have an extended precision unsigned decimal output routine, writing an extended precision signed decimal output routine is very easy. The basic algorithm takes the following form:

- Check the sign of the number. If it is positive, call the unsigned output routine to print it.

- If the number is negative, print a minus sign. Then negate the number and call the unsigned output routine to print it.

To check the sign of an extended precision integer, of course, you simply test the H.O. bit of the number. To negate a large value, the best solution is to probably subtract that value from zero. Here's a quick version of *puti128* that uses the *putu128* routine from the previous section.

```

procedure puti128( i128: int128 ); nodisplay;
begin puti128;

    if( (type int32 i128[12]) < 0 ) then

        stdout.put( '-' );

        // Extended Precision Negation:

        push( eax );
        mov( 0, eax );
        sub( i128[0], eax );
        mov( eax, i128[0] );

        mov( 0, eax );
        sbb( i128[4], eax );
        mov( eax, i128[4] );

        mov( 0, eax );
        sbb( i128[8], eax );
        mov( eax, i128[8] );

        mov( 0, eax );
        sbb( i128[12], eax );
        mov( eax, i128[12] );
        pop( eax );

    endif;
    putu128( (type uns128 i128));

end puti128;

```

4.2.13.4 Extended Precision Formatted I/O

The code in the previous two sections prints signed and unsigned integers using the minimum number of necessary print positions. To create nicely formatted tables of values you will need the equivalent of a *puti128Size* or *putu128Size* routine. Once you have the "unformatted" versions of these routines, implementing the formatted versions is very easy.

The first step is to write an "i128Size" and a "u128Size" routine that computes the minimum number of digits needed to display the value. The algorithm to accomplish this is very similar to the numeric output routines. In fact, the only difference is that you initialize a counter to zero upon entry into the routine (e.g., the non-recursive shell routine) and you increment this counter rather than outputting a digit on each recursive call. (Don't forget to increment the counter inside "i128Size" if the number is negative; you must allow for the output of the minus sign.) After the calculation is complete, these routines should return the size of the operand in the EAX register.

Once you have the "i128Size" and "u128Size" routines, writing the formatted output routines is very easy. Upon initial entry into *puti128Size* or *putu128Size*, these routines call the corresponding "size" routine to determine the number of print positions for the number to display. If the value that the "size" routine returns is greater than the absolute value of the minimum size parameter (passed into *puti128Size* or

putu128Size) all you need to do is call the *put* routine to print the value, no other formatting is necessary. If the absolute value of the parameter size is greater than the value *i128Size* or *u128Size* returns, then the program must compute the difference between these two values and print that many spaces (or other filler character) before printing the number (if the parameter size value is positive) or after printing the number (if the parameter size value is negative). The actual implementation of these two routines is left as an exercise at the end of the volume. If you have any further questions about how to do this, you can take a look at the HLA Standard Library code for routines like *stdout.putu32Size*.

4.2.13.5 Extended Precision Input Routines

There are a couple of fundamental differences between the extended precision output routines and the extended precision input routines. First of all, numeric output generally occurs without possibility of error³; numeric input, on the other hand, must handle the very real possibility of an input error such as illegal characters and numeric overflow. Also, HLA's Standard Library and run-time system encourages a slightly different approach to input conversion. This section discusses those issues that differentiate input conversion from output conversion.

Perhaps the biggest difference between input and output conversion is the fact that output conversion is *unbracketed*. That is, when converting a numeric value to a string of characters for output, the output routine does not concern itself with characters preceding the output string nor does it concern itself with the characters following the numeric value in the output stream. Numeric output routines convert their data to a string and print that string without considering the context (i.e., the characters before and after the string representation of the numeric value). Numeric input routines cannot be so cavalier; the contextual information surrounding the numeric string is very important.

A typical numeric input operation consists of reading a string of characters from the user and then translating this string of characters into an internal numeric representation. For example, a statement like `"stdin.get(i32);"` typically reads a line of text from the user and converts a sequence of digits appearing at the beginning of that line of text into a 32-bit signed integer (assuming *i32* is an *int32* object). Note, however, that the *stdin.get* routine skips over certain characters in the string that may appear before the actual numeric characters. For example, *stdin.get* automatically skips any leading spaces in the string. Likewise, the input string may contain additional data beyond the end of the numeric input (for example, it is possible to read two integer values from the same input line), therefore the input conversion routine must somehow determine where the numeric data ends in the input stream. Fortunately, HLA provides a simple mechanism that lets you easily determine the start and end of the input data: the *Delimiters* character set.

The *Delimiters* character set is a variable, internal to HLA, that contains the set of legal characters that may precede or follow a legal numeric value. By default, this character set includes the end of string marker (a zero byte), a tab character, a line feed character, a carriage return character, a space, a comma, a colon, and a semicolon. Therefore, HLA's numeric input routines will automatically ignore any characters in this set that occur on input before a numeric string. Likewise, characters from this set may legally follow a numeric string on input (conversely, if any non-delimiter character follows the numeric string, HLA will raise an *ex.ConversionError* exception).

The *Delimiters* character set is a private variable inside the HLA Standard Library. Although you do not have direct access to this object, the HLA Standard Library does provide two accessor functions, *conv.setDelimiters* and *conv.getDelimiters* that let you access and modify the value of this character set. These two functions have the following prototypes (found in the "conv.hhf" header file):

```
procedure conv.setDelimiters( Delims:cset );
procedure conv.getDelimiters( var Delims:cset );
```

The *conv.SetDelimiters* procedure will copy the value of the *Delims* parameter into the internal *Delimiters* character set. Therefore, you can use this procedure to change the character set if you want to use a different set of delimiters for numeric input. The *conv.getDelimiters* call returns a copy of the internal

3. Technically speaking, this isn't entirely true. It is possible for a device error (e.g., disk full) to occur. The likelihood of this is so low that we can effectively ignore this possibility.

Delimiters character set in the variable you pass as a parameter to the *conv.getDelimiters* procedure. We will use the value returned by *conv.getDelimiters* to determine the end of numeric input when writing our own extended precision numeric input routines.

When reading a numeric value from the user, the first step will be to get a copy of the *Delimiters* character set. The second step is to read and discard input characters from the user as long as those characters are members of the *Delimiters* character set. Once a character is found that is not in the *Delimiters* set, the input routine must check this character and verify that it is a legal numeric character. If not, the program should raise an *ex.IllegalChar* exception if the character's value is outside the range \$00..\$7f or it should raise the *ex.ConversionError* exception if the character is not a legal numeric character. Once the routine encounters a numeric character, it should continue reading characters as long as they valid numeric characters; while reading the characters the conversion routine should be translating them to the internal representation of the numeric data. If, during conversion, an overflow occurs, the procedure should raise the *ex.ValueOutOfRange* exception.

Conversion to numeric representation should end when the procedure encounters the first delimiter character at the end of the string of digits. However, it is very important that the procedure does not consume the delimiter character that ends the string. That is, the following is incorrect:

```
static
  Delimiters: cset;
  .
  .
  .
conv.getDelimiters( Delimiters );

// Skip over leading delimiters in the string:

while( stdin.getc() in Delimiters ) do /* getc did the work */ endwhile;
while( al in {'0'..'9'}) do

  // Convert character in AL to numeric representation and
  // accumulate result...

  stdin.getc();

endwhile;
if( al not in Delimiters ) then

  raise( ex.ConversionError );

endif;
```

The first WHILE loop reads a sequence of delimiter characters. When this first WHILE loop ends, the character in AL is not a delimiter character. So far, so good. The second WHILE loop processes a sequence of decimal digits. First, it checks the character read in the previous WHILE loop to see if it is a decimal digit; if so, it processes that digit and reads the next character. This process continues until the call to *stdin.getc* (at the bottom of the loop) reads a non-digit character. After the second WHILE loop, the program checks the last character read to ensure that it is a legal delimiter character for a numeric input value.

The problem with this algorithm is that it consumes the delimiter character after the numeric string. For example, the colon symbol is a legal delimiter in the default *Delimiters* character set. If the user types the input "123:456" and executes the code above, this code will properly convert "123" to the numeric value one hundred twenty-three. However, the very next character read from the input stream will be the character "4" not the colon character (":"). While this may be acceptable in certain circumstances, Most programmers expect numeric input routines to consume only leading delimiter characters and the numeric digit characters. They do not expect the input routine to consume any trailing delimiter characters (e.g., many programs will read the next character and expect a colon as input if presented with the string "123:456"). Since *stdin.getc* consumes an input character, and there is no way to "put the character back" onto the input stream, some other way of reading input characters from the user, that doesn't consume those characters, is needed⁴.

The HLA Standard Library comes to the rescue by providing the *stdin.peekc* function. Like *stdin.getc*, the *stdin.peekc* routine reads the next input character from HLA's internal buffer. There are two major differences between *stdin.peekc* and *stdin.getc*. First, *stdin.peekc* will not force the input of a new line of text from the user if the current input line is empty (or you've already read all the text from the input line). Instead, *stdin.peekc* simply returns zero in the AL register to indicate that there are no more characters on the input line. Since #0 is (by default) a legal delimiter character for numeric values, and the end of line is certainly a legal way to terminate numeric input, this works out rather well. The second difference between *stdin.getc* and *stdin.peekc* is that *stdin.peekc* does not consume the character read from the input buffer. If you call *stdin.peekc* several times in a row, it will always return the same character; likewise, if you call *stdin.getc* immediately after *stdin.peekc*, the call to *stdin.getc* will generally return the same character as returned by *stdin.peekc* (the only exception being the end of line condition). So although we cannot put characters back onto the input stream after we've read them with *stdin.getc*, we can peek ahead at the next character on the input stream and base our logic on that character's value. A corrected version of the previous algorithm might be the following:

```
static
  Delimiters: cset;
  .
  .
  .
conv.getDelimiters( Delimiters );

// Skip over leading delimiters in the string:

while( stdin.peekc() in Delimiters ) do

  // If at the end of the input buffer, we must explicitly read a
  // new line of text from the user.  stdin.peekc does not do this
  // for us.

  if( al = #0 ) then

    stdin.ReadLn();

  else

    stdin.getc(); // Remove delimiter from the input stream.

  endif;

endwhile;
while( stdin.peekc in {'0'..'9'}) do

  stdin.getc(); // Remove the input character from the input stream.

  // Convert character in AL to numeric representation and
  // accumulate result...

endwhile;
if( al not in Delimiters ) then

  raise( ex.ConversionError );

endif;
```

4. The HLA Standard Library routines actually buffer up input lines in a string and process characters out of the string. This makes it easy to "peek" ahead one character when looking for a delimiter to end the input value. Your code can also do this, however, the code in this chapter will use a different approach.

Note that the call to `stdin.peekc` in the second WHILE does not consume the delimiter character when the expression evaluates false. Hence, the delimiter character will be the next character read after this algorithm finishes.

The only remaining comment to make about numeric input is to point out that the HLA Standard Library input routines allow arbitrary underscores to appear within a numeric string. The input routines ignore these underscore characters. This allows the user to input strings like "FFFF_F012" and "1_023_596" which are a little more readable than "FFFFFF012" or "1023596". To allow underscores (or any other symbol you choose) within a numeric input routine is quite simple; just modify the second WHILE loop above as follows:

```
while( stdin.peekc in {'0'..'9', '_'}) do

    stdin.getc(); // Read the character from the input stream.

    // Ignore underscores while processing numeric input.

    if( al <> '_' ) then

        // Convert character in AL to numeric representation and
        // accumulate result...

    endif;

endwhile;
```

4.2.13.6 Extended Precision Hexadecimal Input

As was the case for numeric output, hexadecimal input is the easiest numeric input routine to write. The basic algorithm for hexadecimal string to numeric conversion is the following:

- Initialize the extended precision value to zero.
- For each input character that is a valid hexadecimal digit, do the following:
 - Convert the hexadecimal character to a value in the range 0..15 (\$0..\$F).
 - If the H.O. four bits of the extended precision value are non-zero, raise an exception.
 - Multiply the current extended precision value by 16 (i.e., shift left four bits).
 - Add the converted hexadecimal digit value to the accumulator.
- Check the last input character to ensure it is a valid delimiter. Raise an exception if it is not.

The following program implements this extended precision hexadecimal input routine for 128-bit values.

```
program Xin128;

#include( "stdlib.hhf" );

// 128-bit unsigned integer data type:

type
    b128: dword[4];

procedure getb128( var inValue:b128 ); @nodisplay;
const
    HexChars := {'0'..'9', 'a'..'f', 'A'..'F', '_'};
var
```

```

Delimiters: cset;
LocalValue: b128;

begin getb128;

  push( eax );
  push( ebx );

  // Get a copy of the HLA standard numeric input delimiters:

  conv.getDelimiters( Delimiters );

  // Initialize the numeric input value to zero:

  xor( eax, eax );
  mov( eax, LocalValue[0] );
  mov( eax, LocalValue[4] );
  mov( eax, LocalValue[8] );
  mov( eax, LocalValue[12] );

  // By default, #0 is a member of the HLA Delimiters
  // character set. However, someone may have called
  // conv.setDelimiters and removed this character
  // from the internal Delimiters character set. This
  // algorithm depends upon #0 being in the Delimiters
  // character set, so let's add that character in
  // at this point just to be sure.

  cs.unionChar( #0, Delimiters );

  // If we're at the end of the current input
  // line (or the program has yet to read any input),
  // for the input of an actual character.

  if( stdin.peekc() = #0 ) then

    stdin.readLn();

  endif;

  // Skip the delimiters found on input. This code is
  // somewhat convoluted because stdin.peekc does not
  // force the input of a new line of text if the current
  // input buffer is empty. We have to force that input
  // ourselves in the event the input buffer is empty.

  while( stdin.peekc() in Delimiters ) do

    // If we're at the end of the line, read a new line
    // of text from the user; otherwise, remove the
    // delimiter character from the input stream.

    if( al = #0 ) then

      stdin.readLn(); // Force a new input line.

    else

```



```

        stdin.getc(); // Remove the delimiter from the input buffer.

    endif;

endwhile;

// Read the hexadecimal input characters and convert
// them to the internal representation:

while( stdin.peekc() in HexChars ) do

    // Actually read the character to remove it from the
    // input buffer.

    stdin.getc();

    // Ignore underscores, process everything else.

    if( al <> '_' ) then

        if( al in '0'..'9' ) then

            and( $f, al ); // '0'..'9' -> 0..9

        else

            and( $f, al ); // 'a'/'A'..'f'/'F' -> 1..6
            add( 9, al ); // 1..6 -> 10..15

        endif;

        // Conversion algorithm is the following:
        //
        // (1) LocalValue := LocalValue * 16.
        // (2) LocalValue := LocalValue + al
        //
        // Note that "* 16" is easily accomplished by
        // shifting LocalValue to the left four bits.
        //
        // Overflow occurs if the H.O. four bits of LocalValue
        // contain a non-zero value prior to this operation.

        // First, check for overflow:

        test( $F0, (type byte LocalValue[15]));
        if( @nz ) then

            raise( ex.ValueOutOfRange );

        endif;

        // Now multiply LocalValue by 16 and add in
        // the current hexadecimal digit (in EAX).

        mov( LocalValue[8], ebx );
        shld( 4, ebx, LocalValue[12] );
        mov( LocalValue[4], ebx );
        shld( 4, ebx, LocalValue[8] );
        mov( LocalValue[0], ebx );
        shld( 4, ebx, LocalValue[4] );
        shl( 4, ebx );

```

```

        add( eax, ebx );
        mov( ebx, LocalValue[0] );

    endif;

endwhile;

// Okay, we've encountered a non-hexadecimal character.
// Let's make sure it's a valid delimiter character.
// Raise the ex.ConversionError exception if it's invalid.

if( al not in Delimiters ) then

    raise( ex.ConversionError );

endif;

// Okay, this conversion has been a success. Let's store
// away the converted value into the output parameter.

mov( inValue, ebx );
mov( LocalValue[0], eax );
mov( eax, [ebx] );

mov( LocalValue[4], eax );
mov( eax, [ebx+4] );

mov( LocalValue[8], eax );
mov( eax, [ebx+8] );

mov( LocalValue[12], eax );
mov( eax, [ebx+12] );

pop( ebx );
pop( eax );

end getbl28;

// Code to test the routines above:

static
    bl:bl28;

begin Xin128;

    stdout.put( "Input a 128-bit hexadecimal value: " );
    getbl28( bl );
    stdout.put
    (
        "The value is: $",
        bl[12], '_',
        bl[8],  '_',
        bl[4],  '_',
        bl[0],
        nl
    );

end Xin128;

```

Program 4.5 Extended Precision Hexadecimal Input

Extending this code to handle objects that are not 128 bits long is very easy. There are only three changes necessary: you must zero out the whole object at the beginning of the `getb128` routine; when checking for overflow (the `"test($F, (type byte LocalValue[15]);"` instruction) you must test the H.O. four bits of the new object you're processing; and you must modify the code that multiplies `LocalValue` by 16 (via `SHLD`) so that it multiplies your object by 16 (i.e., shifts it to the left four bits).

4.2.13.7 Extended Precision Unsigned Decimal Input

The algorithm for extended precision unsigned decimal input is nearly identical to that for hexadecimal input. In fact, the only difference (beyond only accepting decimal digits) is that you multiply the extended precision value by 10 rather than 16 for each input character (in general, the algorithm is the same for any base; just multiply the accumulating value by the input base). The following code demonstrates how to write a 128-bit unsigned decimal input routine.

```

program Uin128;

#include( "stdlib.hhf" );

// 128-bit unsigned integer data type:

type
    ul28: dword[4];

procedure getul28( var inValue:ul28 ); @nodisplay;
var
    Delimiters: cset;
    LocalValue: ul28;
    PartialSum: ul28;

begin getul28;

    push( eax );
    push( ebx );
    push( ecx );
    push( edx );

    // Get a copy of the HLA standard numeric input delimiters:

    conv.getDelimiters( Delimiters );

    // Initialize the numeric input value to zero:

    xor( eax, eax );
    mov( eax, LocalValue[0] );
    mov( eax, LocalValue[4] );
    mov( eax, LocalValue[8] );
    mov( eax, LocalValue[12] );

```

```

// By default, #0 is a member of the HLA Delimiters
// character set. However, someone may have called
// conv.setDelimiters and removed this character
// from the internal Delimiters character set. This
// algorithm depends upon #0 being in the Delimiters
// character set, so let's add that character in
// at this point just to be sure.

cs.unionChar( #0, Delimiters );

// If we're at the end of the current input
// line (or the program has yet to read any input),
// for the input of an actual character.

if( stdin.peekc() = #0 ) then

    stdin.readLine();

endif;

// Skip the delimiters found on input. This code is
// somewhat convoluted because stdin.peekc does not
// force the input of a new line of text if the current
// input buffer is empty. We have to force that input
// ourselves in the event the input buffer is empty.

while( stdin.peekc() in Delimiters ) do

    // If we're at the end of the line, read a new line
    // of text from the user; otherwise, remove the
    // delimiter character from the input stream.

    if( al = #0 ) then

        stdin.readLine(); // Force a new input line.

    else

        stdin.getc(); // Remove the delimiter from the input buffer.

    endif;

endwhile;

// Read the decimal input characters and convert
// them to the internal representation:

while( stdin.peekc() in '0'..'9' ) do

    // Actually read the character to remove it from the
    // input buffer.

    stdin.getc();

    // Ignore underscores, process everything else.

    if( al <> '_' ) then

```

```

and( $f, al );          // '0'..'9' -> 0..9
mov( eax, PartialSum[0] ); // Save to add in later.

// Conversion algorithm is the following:
//
// (1) LocalValue := LocalValue * 10.
// (2) LocalValue := LocalValue + al
//
// First, multiply LocalValue by 10:

mov( 10, eax );
mul( LocalValue[0], eax );
mov( eax, LocalValue[0] );
mov( edx, PartialSum[4] );

mov( 10, eax );
mul( LocalValue[4], eax );
mov( eax, LocalValue[4] );
mov( edx, PartialSum[8] );

mov( 10, eax );
mul( LocalValue[8], eax );
mov( eax, LocalValue[8] );
mov( edx, PartialSum[12] );

mov( 10, eax );
mul( LocalValue[12], eax );
mov( eax, LocalValue[12] );

// Check for overflow. This occurs if EDX
// contains a none zero value.

if( edx /* <> 0 */ ) then

    raise( ex.ValueOutOfRange );

endif;

// Add in the partial sums (including the
// most recently converted character).

mov( PartialSum[0], eax );
add( eax, LocalValue[0] );

mov( PartialSum[4], eax );
adc( eax, LocalValue[4] );

mov( PartialSum[8], eax );
adc( eax, LocalValue[8] );

mov( PartialSum[12], eax );
adc( eax, LocalValue[12] );

// Another check for overflow. If there
// was a carry out of the extended precision
// addition above, we've got overflow.

if( @c ) then

    raise( ex.ValueOutOfRange );

```

```

        endif;

    endif;

endwhile;

// Okay, we've encountered a non-decimal character.
// Let's make sure it's a valid delimiter character.
// Raise the ex.ConversionError exception if it's invalid.

if( al not in Delimiters ) then

    raise( ex.ConversionError );

endif;

// Okay, this conversion has been a success. Let's store
// away the converted value into the output parameter.

mov( inValue, ebx );
mov( LocalValue[0], eax );
mov( eax, [ebx] );

mov( LocalValue[4], eax );
mov( eax, [ebx+4] );

mov( LocalValue[8], eax );
mov( eax, [ebx+8] );

mov( LocalValue[12], eax );
mov( eax, [ebx+12] );

pop( edx );
pop( ecx );
pop( ebx );
pop( eax );

end getul28;

// Code to test the routines above:

static
    bl:ul28;

begin Uin128;

    stdout.put( "Input a 128-bit decimal value: " );
    getul28( bl );
    stdout.put
    (
        "The value is: $",
        bl[12], '_',
        bl[8],  '_',
        bl[4],  '_',
        bl[0],
        nl
    );

end Uin128;

```

Program 4.6 Extended Precision Unsigned Decimal Input

As for hexadecimal input, extending this decimal input to some number of bits beyond 128 is fairly easy. All you need do is modify the code that zeros out the *LocalValue* variable and the code that multiplies *LocalValue* by ten (overflow checking is done in this same code, so there are only two spots in this code that require modification).

4.2.13.8 Extended Precision Signed Decimal Input

Once you have an unsigned decimal input routine, writing a signed decimal input routine is easy. The following algorithm describes how to accomplish this:

- Consume any delimiter characters at the beginning of the input stream.
- If the next input character is a minus sign, consume this character and set a flag noting that the number is negative.
- Call the unsigned decimal input routine to convert the rest of the string to an integer.
- Check the return result to make sure it's H.O. bit is clear. Raise the *ex.ValueOutOfRange* exception if the H.O. bit of the result is set.
- If the sign flag was set in step two above, negate the result.

The actual code is left as a programming exercise at the end of this volume.

4.3 Operating on Different Sized Operands

Occasionally you may need to compute some value on a pair of operands that are not the same size. For example, you may need to add a word and a double word together or subtract a byte value from a word value. The solution is simple: just extend the smaller operand to the size of the larger operand and then do the operation on two similarly sized operands. For signed operands, you would sign extend the smaller operand to the same size as the larger operand; for unsigned values, you zero extend the smaller operand. This works for any operation, although the following examples demonstrate this for the addition operation.

To extend the smaller operand to the size of the larger operand, use a sign extension or zero extension operation (depending upon whether you're adding signed or unsigned values). Once you've extended the smaller value to the size of the larger, the addition can proceed. Consider the following code that adds a byte value to a word value:

```
static
    var1: byte;
    var2: word;
    .
    .
    .
// Unsigned addition:

    movzx( var1, ax );
    add( var2, ax );

// Signed addition:
```

```
movsx( var1, ax );
add( var2, ax );
```

In both cases, the byte variable was loaded into the AL register, extended to 16 bits, and then added to the word operand. This code works out really well if you can choose the order of the operations (e.g., adding the eight bit value to the sixteen bit value). Sometimes, you cannot specify the order of the operations. Perhaps the sixteen bit value is already in the AX register and you want to add an eight bit value to it. For unsigned addition, you could use the following code:

```
mov( var2, ax );      // Load 16 bit value into AX
.                    // Do some other operations leaving
.                    // a 16-bit quantity in AX.
add( var1, al );     // Add in the eight-bit value
adc( 0, ah );       // Add carry into the H.O. word.
```

The first ADD instruction in this example adds the byte at *var1* to the L.O. byte of the value in the accumulator. The ADC instruction above adds the carry out of the L.O. byte into the H.O. byte of the accumulator. Care must be taken to ensure that this ADC instruction is present. If you leave it out, you may not get the correct result.

Adding an eight bit signed operand to a sixteen bit signed value is a little more difficult. Unfortunately, you cannot add an immediate value (as above) to the H.O. word of AX. This is because the H.O. extension byte can be either \$00 or \$FF. If a register is available, the best thing to do is the following:

```
mov( ax, bx );      // BX is the available register.
movsx( var1, ax );
add( bx, ax );
```

If an extra register is not available, you might try the following code:

```
push( ax );        // Save word value.
movsx( var1, ax ); // Sign extend 8-bit operand to 16 bits.
add( [esp], ax );  // Add in previous word value
add( 2, esp );     // Pop junk from stack
```

Another alternative is to store the 16 bit value in the accumulator into a memory location and then proceed as before:

```
mov( ax, temp );
movsx( var1, ax );
add( temp, ax );
```

All the examples above added a byte value to a word value. By zero or sign extending the smaller operand to the size of the larger operand, you can easily add any two different sized variables together.

As a last example, consider adding an eight bit signed value to a quadword (64 bit) value:

```
static
QVal:qword;
BVal:int8;
.
.
.
movsx( BVal, eax );
cdq();
add( (type dword QVal), eax );
adc( (type dword QVal[4]), edx );
```

4.4 Decimal Arithmetic

The 80x86 CPUs use the binary numbering system for their native internal representation. The binary numbering system is, by far, the most common numbering system in use in computer systems today. In days long since past, however, there were computer systems that were based on the decimal (base 10) numbering system rather than the binary numbering system. Consequently, their arithmetic system was decimal based rather than binary. Such computer systems were very popular in systems targeted for business/commercial systems⁵. Although systems designers have discovered that binary arithmetic is almost always better than decimal arithmetic for general calculations, the myth still persists that decimal arithmetic is better for money calculations than binary arithmetic. Therefore, many software systems still specify the use of decimal arithmetic in their calculations (not to mention that there is lots of legacy code out there whose algorithms are only stable if they use decimal arithmetic). Therefore, despite the fact that decimal arithmetic is generally inferior to binary arithmetic, the need for decimal arithmetic still persists.

Of course, the 80x86 is not a decimal computer; therefore we have to play tricks in order to represent decimal numbers using the native binary format. The most common technique, even employed by most so-called decimal computers, is to use the *binary coded decimal*, or BCD representation. The BCD representation (see “Nibbles” on page 56) uses four bits to represent the 10 possible decimal digits. The binary value of those four bits is equal to the corresponding decimal value in the range 0..9. Of course, with four bits we can actually represent 16 different values. The BCD format ignores the remaining six bit combinations.

Table 1: Binary Code Decimal (BCD) Representation

BCD Representation	Decimal Equivalent
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	Illegal
1011	Illegal

5. In fact, until the release of the IBM 360 in the middle 1960's, most scientific computer systems were binary based while most commercial/business systems were decimal based. IBM pushed their system\360 as a single purpose solution for both business and scientific applications. Indeed, the model designation (360) was derived from the 360 degrees on a compass so as to suggest that the system\360 was suitable for computations "at all points of the compass" (i.e., business and scientific).

Table 1: Binary Code Decimal (BCD) Representation

BCD Representation	Decimal Equivalent
1100	Illegal
1101	Illegal
1110	Illegal
1111	Illegal

Since each BCD digit requires four bits, we can represent a two-digit BCD value with a single byte. This means that we can represent the decimal values in the range 0..99 using a single byte (versus 0..255 if we treat the value as an unsigned binary number). Clearly it takes a bit more memory to represent the same value in BCD as it does to represent the same value in binary. For example, with a 32-bit value you can represent BCD values in the range 0..99,999,999 (eight significant digits) but you can represent values in the range 0..4,294,967,295 (better than nine significant digits) using the binary representation.

Not only does the BCD format waste memory on a binary computer (since it uses more bits to represent a given integer value), but decimal arithmetic is slower. For these reasons, you should avoid the use of decimal arithmetic unless it is absolutely mandated for a given application.

Binary coded decimal representation does offer one big advantage over binary representation: it is fairly trivial to convert between the string representation of a decimal number and the BCD representation. This feature is particularly beneficial when working with fractional values since fixed and floating point binary representations cannot exactly represent many commonly used values between zero and one (e.g., $1/10$). Therefore, BCD operations can be efficient when reading from a BCD device, doing a simple arithmetic operation (e.g., a single addition) and then writing the BCD value to some other device.

4.4.1 Literal BCD Constants

HLA does not provide, nor do you need, a special literal BCD constant. Since BCD is just a special form of hexadecimal notation that does not allow the values \$A..\$F, you can easily create BCD constants using HLA's hexadecimal notation. Of course, you must take care not to include the symbols 'A'..'F' in a BCD constant since they are illegal BCD values. As an example, consider the following MOV instruction that copies the BCD value '99' into the AL register:

```
mov( $99, al );
```

The important thing to keep in mind is that you must not use HLA literal decimal constants for BCD values. That is, "mov(95, al);" does not load the BCD representation for ninety-five into the AL register. Instead, it loads \$5F into AL and that's an illegal BCD value. Any computations you attempt with illegal BCD values will produce garbage results. Always remember that, even though it seems counter-intuitive, you use hexadecimal literal constants to represent literal BCD values.

4.4.2 The 80x86 DAA and DAS Instructions

The integer unit on the 80x86 does not directly support BCD arithmetic. Instead, the 80x86 requires that you perform the computation using binary arithmetic and use some auxiliary instructions to convert the binary result to BCD. To support packed BCD addition and subtraction with two digits per byte, the 80x86 provides two instructions: decimal adjust after addition (DAA) and decimal adjust after subtraction (DAS). You would execute these two instructions immediately after an ADD/ADC or SUB/SBB instruction to correct the binary result in the AL register.

Two add a pair of two-digit (i.e., single-byte) BCD values together, you would use the following sequence:

```
mov( bcd_1, al );    // Assume that bcd1 and bcd2 both contain
add( bcd_2, al );    // value BCD values.
daa();
```

The first two instructions above add the two byte values together using standard binary arithmetic. This may not produce a correct BCD result. For example, if *bcd_1* contains \$9 and *bcd_2* contains \$1, then the first two instructions above will produce the binary sum \$A instead of the correct BCD result \$10. The DAA instruction corrects this invalid result. It checks to see if there was a carry out of the low order BCD digit and adjusts the value (by adding six to it) if there was an overflow. After adjusting for overflow out of the L.O. digit, the DAA instruction repeats this process for the H.O. digit. DAA sets the carry flag if there was a (decimal) carry out of the H.O. digit of the operation.

The DAA instruction only operates on the AL register. It will not adjust (properly) for a decimal addition if you attempt to add a value to AX, EAX, or any other register. Specifically note that DAA limits you to adding two decimal digits (a single byte) at a time. This means that for the purposes of computing decimal sums, you have to treat the 80x86 as though it were an eight-bit processor, capable of adding only eight bits at a time. If you wish to add more than two digits together, you must treat this as a multiprecision operation. For example, to add four decimal digits together (using DAA), you must execute a sequence like the following:

```
// Assume "bcd_1:byte[2];", "bcd_2:byte[2];", and "bcd_3:byte[2];"

mov( bcd_1[0], al );
add( bcd_2[0], al );
daa();
mov( al, bcd_3[0] );
mov( bcd_1[1], al );
adc( bcd_2[1], al );
daa();
mov( al, bcd_3[1], al );

// Carry is set at this point if there was unsigned overflow.
```

Since a binary addition of a word requires only three instructions, you can see why decimal arithmetic is so expensive⁶.

The DAS (decimal adjust after subtraction) adjusts the decimal result after a binary SUB or SBB instruction. You use it the same way you use the DAA instruction. Examples:

```
// Two-digit (one byte) decimal subtraction:

mov( bcd_1, al );    // Assume that bcd1 and bcd2 both contain
sub( bcd_2, al );    // value BCD values.
das();

// Four-digit (two-byte) decimal subtraction.
// Assume "bcd_1:byte[2];", "bcd_2:byte[2];", and "bcd_3:byte[2];"

mov( bcd_1[0], al );
sub( bcd_2[0], al );
das();
mov( al, bcd_3[0] );
mov( bcd_1[1], al );
sbb( bcd_2[1], al );
das();
mov( al, bcd_3[1], al );
```

6. You'll also soon see that it's rare to find decimal arithmetic done this way. So it hardly matters.

```
// Carry is set at this point if there was unsigned overflow.
```

Unfortunately, the 80x86 only provides support for addition and subtraction of packed BCD values using the DAA and DAS instructions. It does not support multiplication, division, or any other arithmetic operations. Because decimal arithmetic using these instructions is so limited, you'll rarely see any programs use these instructions.

4.4.3 The 80x86 AAA, AAS, AAM, and AAD Instructions

In addition to the *packed decimal* instructions (DAA and DAS), the 80x86 CPUs support four *unpacked decimal* adjustment instructions. Unpacked decimal numbers store only one digit per eight-bit byte. As you can imagine, this data representation scheme wastes a considerable amount of memory. However, the unpacked decimal adjustment instructions support the multiplication and division operations, so they are marginally more useful.

The instruction mnemonics AAA, AAS, AAM, and AAD stand for "ASCII adjust for Addition, Subtraction, Multiplication, and Division" (respectively). Despite their name, these instructions do not process ASCII characters. Instead, they support an unpacked decimal value in AL whose L.O. four bits contain the decimal digit and the H.O. four bits contain zero. Note, though, that you can easily convert an ASCII decimal digit character to an unpacked decimal number by simply ANDing AL with the value \$0F.

The AAA instruction adjusts the result of a binary addition of two unpacked decimal numbers. If the addition of those two values exceeds 10, then AAA will subtract 10 from AL and increment AH by one (as well as set the carry flag). AAA assumes that the two values you add together were legal unpacked decimal values. Other than the fact that AAA works with only one decimal digit at a time (rather than two), you use it the same way you use the DAA instruction. Of course, if you need to add together a string of decimal digits, using unpacked decimal arithmetic will require twice as many operations and, therefore, twice the execution time.

You use the AAS instruction the same way you use the DAS instruction except, of course, it operates on unpacked decimal values rather than packed decimal values. As for AAA, AAS will require twice the number of operations to add the same number of decimal digits as the DAS instruction. If you're wondering why anyone would want to use the AAA or AAS instructions, keep in mind that the unpacked format supports multiplication and division, while the packed format does not. Since packing and unpacking the data is usually more expensive than working on the data a digit at a time, the AAA and AAS instruction are more efficient if you have to work with unpacked data (because of the need for multiplication and division).

The AAM instruction modifies the result in the AX register to produce a correct unpacked decimal result after multiplying two unpacked decimal digits using the MUL instruction. Because the largest product you may obtain is 81 (9*9 produces the largest possible product of two single digit values), the result will fit in the AL register. AAM unpacks the binary result by dividing it by 10, leaving the quotient (H.O. digit) in AH and the remainder (L.O. digit) in AL. Note that AAM leaves the quotient and remainder in different registers than a standard eight-bit DIV operation.

Technically, you do not have to use the AAM instruction immediately after a multiply. AAM simply divides AL by ten and leaves the quotient and remainder in AH and AL (respectively). If you have need of this particular operation, you may use the AAM instruction for this purpose (indeed, that's about the only use for AAM in most programs these days).

If you need to multiply more than two unpacked decimal digits together using MUL and AAM, you will need to devise a multiprecision multiplication that uses the manual algorithm from earlier in this chapter. Since that is a lot of work, this section will not present that algorithm. If you need a multiprecision decimal multiplication, see the next section; it presents a better solution.

The AAD instruction, as you might expect, adjusts a value for unpacked decimal division. The unusual thing about this instruction is that you must execute it *before* a DIV operation. It assumes that AL contains the least significant digit of a two-digit value and AH contains the most significant digit of a two-digit unpacked decimal value. It converts these two numbers to binary so that a standard DIV instruction will produce the correct unpacked decimal result. Like AAM, this instruction is nearly useless for its intended pur-

pose as extended precision operations (e.g., division of more than one or two digits) are extremely inefficient. However, this instruction is actually quite useful in its own right. It computes $AX = AH \cdot 10 + AL$ (assuming that AH and AL contain single digit decimal values). You can use this instruction to easily convert a two-character string containing the ASCII representation of a value in the range 0..99 to a binary value. E.g.,

```
mov( '9', al );
mov( '9', ah ); // "99" is in AH:AL.
and( $0F0F, ax ); // Convert from ASCII to unpacked decimal.
aad(); // After this, AX contains 99.
```

The decimal and ASCII adjust instructions provide an extremely poor implementation of decimal arithmetic. To better support decimal arithmetic on 80x86 systems, Intel incorporated decimal operations into the FPU. The next section discusses how to use the FPU for this purpose. However, even with FPU support, decimal arithmetic is inefficient and less precise than binary arithmetic. Therefore, you should carefully consider whether you really need to use decimal arithmetic before incorporating it into your programs.

4.4.4 Packed Decimal Arithmetic Using the FPU

To improve the performance of applications that rely on decimal arithmetic, Intel incorporated support for decimal arithmetic directly into the FPU. Unlike the packed and unpacked decimal formats of the previous sections, the FPU easily supports values with up to 18 decimal digits of precision, all at FPU speeds. Furthermore, all the arithmetic capabilities of the FPU (e.g., transcendental operations) are available in addition to addition, subtraction, multiplication, and division. Assuming you can live with *only* 18 digits of precision and a few other restrictions, decimal arithmetic on the FPU is the right way to go if you must use decimal arithmetic in your programs.

The first fact you must note when using the FPU is that it doesn't really support decimal arithmetic. Instead, the FPU provides two instruction, FBLD and FBSTP, that convert between packed decimal and binary floating point formats when moving data to and from the FPU. The FBLD (float/BCD load) instruction loads an 80-bit packed BCD value unto the top of the FPU stack after converting that BCD value to the IEEE binary floating point format. Likewise, the FBSTP (float/BCD store and pop) instruction pops the floating point value off the top of stack, converts it to a packed BCD value, and stores the BCD value into the destination memory location.

Once you load a packed BCD value into the FPU, it is no longer BCD. It's just a floating point value. This presents the first restriction on the use of the FPU as a decimal integer processor: calculations are done using binary arithmetic. If you have an algorithm that absolutely positively depends upon the use of decimal arithmetic, it may fail if you use the FPU to implement it⁷.

The second limitation is that the FPU supports only one BCD data type: a ten-byte 18-digit packed decimal value. It will not support smaller values nor will it support larger values. Since 18 digits is usually sufficient and memory is cheap, this isn't a big restriction.

A third consideration is that the conversion between packed BCD and the floating point format is not a cheap operation. The FBLD and FBSTP instructions can be quite slow (more than two orders of magnitude slower than FLD and FSTP, for example). Therefore, these instructions can be costly if you're doing simple additions or subtractions; the cost of conversion far outweighs the time spent adding the values a byte at a time using the DAA and DAS instructions (multiplication and division, however, are going to be faster on the FPU).

You may be wondering why the FPU's packed decimal format only supports 18 digits. After all, with ten bytes it should be possible to represent 20 BCD digits. As it turns out, the FPU's packed decimal format uses the first nine bytes to hold the packed BCD value in a standard packed decimal format (the first byte contains the two L.O. digits and the ninth byte holds the H.O. two digits). The H.O. bit of the tenth byte

7. An example of such an algorithm might be a multiplication by ten by shifting the number one digit to the left. However, such operations are not possible within the FPU itself, so algorithms that misbehave inside the FPU are actually quite rare.

holds the sign bit and the FPU ignores the remaining bits in the tenth byte. If you're wondering why Intel didn't squeeze in one more digit (i.e., use the L.O. four bits of the tenth byte to allow for 19 digits of precision), just keep in mind that doing so would create some possible BCD values that the FPU could not exactly represent in the native floating point format. Hence the limitation to 18 digits.

The FPU uses a one's complement notation for negative BCD values. That is, the sign bit contains a one if the number is negative or zero and it contains a zero if the number is positive or zero (like the binary one's complement format, there are two distinct representations for zero).

HLA's *tbyte* type is the standard data type you would use to define packed BCD variables. The FBLD and FBSTP instructions require a *tbyte* operand. Unfortunately, the current version of HLA does not let you (directly) provide an initializer for a *tbyte* variable. One solution is to use the @NOSTORAGE option and initialize the data following the variable declaration. For example, consider the following code fragment:

```
static
  tbyteObject: tbyte; @nostorage
                byte $21, $43, $65, 0, 0, 0, 0, 0, 0, 0;
```

This *tbyteObject* declaration tells HLA that this is a *tbyte* object but does not explicitly set aside any space for the variable (see "The Static Sections" on page 167). The following BYTE directive sets aside ten bytes of storage and initializes these ten bytes with the value \$654321 (remember that the 80x86 organizes data from the L.O. byte to the H.O. byte in memory). While this scheme is inelegant, it will get the job done. The chapters on Macros and the Compile-Time Language will discuss a better way to initialize *tbyte* and *qword* data.

Because the FPU converts packed decimal values to the internal floating point format, you can mix packed decimal, floating point, and (binary) integer formats in the same calculation. The following program demonstrate how you might achieve this:

```
program MixedArithmetic;
#include( "stdlib.hhf" )

static
  tb: tbyte; @nostorage;
      byte $21,$43,$65,0,0,0,0,0,0,0;

begin MixedArithmetic;

  fbld( tb );
  fmul( 2.0 );
  fiadd( 1 );
  fbstp( tb );
  stdout.put( "bcd value is " );
  stdout.puttb( tb );
  stdout.newln();

end MixedArithmetic;
```

Program 4.7 Mixed Mode FPU Arithmetic

The FPU treats packed decimal values as integer values. Therefore, if your calculations produce fractional results, the FBSTP instruction will round the result according to the current FPU rounding mode. If you need to work with fractional values, you need to stick with floating point results.

4.5 Sample Program

The following sample program demonstrates BCD I/O. The following program provides two procedures, BCDin and BCDout. These two procedures read an 18-digit BCD value from the user (with possible leading minus sign) and write a BCD value to the standard output device.

```

program bcdIO;
#include( "stdlib.hhf" )

// The following is equivalent to TBYTE except it
// lets us easily gain access to the individual
// components of a BCD value.

type
  bcd:record

      LO8:    dword;
      MID8:   dword;
      HO2:    byte;
      Sign:   byte;

  endrecord;

// BCDin-
//
// This function reads a BCD value from the standard input
// device. The number can be up to 18 decimal digits long
// and may contain a leading minus sign.
//
// This procedure stores the BCD value in the variable passed
// by reference as a parameter to this routine.

procedure BCDin( var input:tbyte ); @nodisplay;
var
  bcdVal:    bcd;
  delimiters: cset;

begin BCDin;

  push( eax );
  push( ebx );

  // Get a copy of the input delimiter characters and
  // make sure that #0 is a member of this set.

  conv.getDelimiters( delimiters );
  cs.unionChar( #0, delimiters );

  // Skip over any leading delimiter characters in the text:

  while( stdin.peekc() in delimiters ) do

```

```

// If we're at the end of an input line, read a new
// line of text from the user, otherwise remove the
// delimiter character from the input stream.

if( stdin.peekc() = #0 ) then

    stdin.readLine(); // Get a new line of input text.

else

    stdin.getc(); // Remove the delimiter.

endif;

endwhile;

// Initialize our input accumulator to zero:

xor( eax, eax );
mov( eax, bcdVal.LO8 );
mov( eax, bcdVal.MID8 );
mov( al, bcdVal.HO2 );
mov( al, bcdVal.Sign );

// If the first character is a minus sign, then eat it and
// set the sign bit to one.

if( stdin.peekc() = '-' ) then

    stdin.getc(); // Eat the sign character.
    mov( $80, bcdVal.Sign ); // Make this number negative.

endif;

// We must have at least one decimal digit in this number:

if( stdin.peekc() not in '0'..'9' ) then

    raise( ex.ConversionError );

endif;

// Okay, read in up to 18 decimal digits:

while( stdin.peekc() in '0'..'9' ) do

    stdin.getc(); // Read this decimal digit.
    shl( 4, al ); // Move digit to H.O. bits of AL

    mov( 4, ebx );
    repeat

        // Cheesy way to SHL bcdVal by four bits and
        // merge in the new character.

        shl( 1, al );
        rcl( 1, bcdVal.LO8 );
        rcl( 1, bcdVal.MID8 );
        rcl( 1, bcdVal.HO2 );
    
```



```

        // If the user has entered more than 18
        // decimal digits, the carry will be set
        // after the RCL above.  Test that here.

        if( @c ) then

            raise( ex.ValueOutOfRange );

        endif;
        dec( ebx );

    until( @z );

endwhile;

// Be sure that the number ends with a proper delimiter:

if( stdin.peekc() not in delimiters ) then

    raise( ex.ConversionError );

endif;

// Okay, store the ten-byte input result into
// the location specified by the parameter.

mov( input, ebx );
mov( bcdVal.LO8, eax );
mov( eax, [ebx] );
mov( bcdVal.MID8, eax );
mov( eax, [ebx+4] );
mov( (type word bcdVal.HO2), ax ); // Grabs "Sign" too.
mov( ax, [ebx+8] );

pop( ebx );
pop( eax );

end BCDin;

// BCDout-
//
// The converse of the above.  Prints the string representation
// of the packed BCD value to the standard output device.

procedure BCDout( output:tbyte ); @nodisplay;
var
    q:qword;

begin BCDout;

    // This code cheats *big time*.
    // It converts the BCD value to a 64-bit integer
    // and then calls the stdout.puti64 routine to
    // actually print the number.  In theory, this is
    // a whole lot slower than converting the BCD value
    // to ASCII and printing the ASCII chars, however,

```

```

// I/O is so much slower than the conversion that
// no one will notice the extra time.

fbld( output );
fistp( q );
stdout.puti64( q );

end BCDout;

static
    tb1:    tbyte;
    tb2:    tbyte;
    tbRslt: tbyte;

begin bcdIO;

    stdout.put( "Enter a BCD value: " );
    BCDin( tb1 );
    stdout.put( "Enter a second BCD value: " );
    BCDin( tb2 );

    fbld( tb1 );
    fbld( tb2 );
    fadd();
    fbstp( tbRslt );

    stdout.put( "The sum of " );
    BCDout( tb1 );
    stdout.put( " + " );
    BCDout( tb2 );
    stdout.put( " is " );
    BCDout( tbRslt );
    stdout.newln();

end bcdIO;

```

Program 4.8 BCD I/O Sample Program

4.6 Putting It All Together

Extended precision arithmetic is one of those activities where assembly language truly shines. It's much easier to perform extended precision arithmetic in assembly language than in most high level languages; it's far more efficient to do it in assembly language, as well. Extended precision arithmetic was, perhaps, the most important subject that this chapter teaches.

Although extended precision arithmetic and logical calculations are important, what good are extended precision calculations if you can't get the extend precision values in and out of the machine? Therefore, this chapter devotes a fair amount of space to describing how to write your own extended precision I/O routines. Between the calculations and the I/O this chapter describes, you're set to perform those really hairy calculations you've always dreamed of!

Although decimal arithmetic is nowhere near as prominent as it once was, the need for decimal arithmetic does arise on occasion. Therefore, this chapter spends some time discussing BCD arithmetic on the 80x86.

5.1 Chapter Overview

Manipulating bits in memory is, perhaps, the thing that assembly language is most famous for. Indeed, one of the reasons people claim that the “C” programming language is a “medium-level” language rather than a high level language is because of the vast array of bit manipulation operators that it provides. Even with this wide array of bit manipulation operations, the C programming language doesn’t provide as complete a set of bit manipulation operations as assembly language.

This chapter will discuss how to manipulate strings of bits in memory and registers using 80x86 assembly language. This chapter begins with a review of the bit manipulation instructions covered thus far and it also introduces a few new instructions. This chapter reviews information on packing and unpacking bit strings in memory since this is the basis for many bit manipulation operations. Finally, this chapter discusses several bit-centric algorithms and their implementation in assembly language.

5.2 What is Bit Data, Anyway?

Before describing how to manipulate bits, it might not be a bad idea to define exactly what this text means by “bit data.” Most readers probably assume that “bit manipulation programs” twiddle individual bits in memory. While programs that do this are definitely “bit manipulation programs,” we’re not going to limit this title to just those programs. For our purposes, bit manipulation refers to working with data types that consist of strings of bits that are non-contiguous or are not an even multiple of eight bits long. Generally, such bit objects will not represent numeric integers, although we will not place this restriction on our bit strings.

A *bit string* is some contiguous sequence of one or more bits (this term even applies if the bit string’s length is an even multiple of eight bits). Note that a bit string does not have to start or end at any special point. For example, a bit string could start in bit seven of one byte in memory and continue through to bit six of the next byte in memory. Likewise, a bit string could begin in bit 30 of EAX, consume the upper two bits of EAX, and then continue from bit zero through bit 17 of EBX. In memory, the bits must be physically contiguous (i.e., the bit numbers are always increasing except when crossing a byte boundary, and at byte boundaries the byte number increases by one). In registers, if a bit string crosses a register boundary, the application defines the continuation register but the bit string always continues in bit zero of that second register.

A *bit set* is a collection of bits, not necessarily contiguous (though it may be), within some larger data structure. For example, bits 0..3, 7, 12, 24, and 31 from some double word object forms a set of bits. Usually, we will limit bit sets to some reasonably sized *container object* (that is, the data structure that encapsulates the bit set), but the definition doesn’t specifically limit the size. Normally, we will deal with bit sets that are part of an object no more than about 32 or 64 bits in size. Note that bit strings are special cases of bit sets.

A *bit run* is a sequence of bits with all the same value. A *run of zeros* is a bit string containing all zeros, a *run of ones* is a bit string containing all ones. The *first set bit* in a bit string is the bit position of the first bit containing a one in a bit string, i.e., the first ‘1’ bit following a possible run of zeros. A similar definition exists for the *first clear bit*. The *last set bit* is the last bit position in a bit string containing that contains ‘1’; afterwards, the remainder of the string forms an uninterrupted run of zeros. A similar definition exists for the *last clear bit*.

A *bit offset* is the number of bits from some boundary position (usually a byte boundary) to the specified bit. As noted in Volume One, we number the bits starting from zero at the boundary location. If the offset is less than 32, then the bit offset is the same as the bit number in a byte, word, or double word value.

A *mask* is a sequence of bits that we'll use to manipulate certain bits in another value. For example, the bit string %0000_1111_0000, when used with the AND instruction, can mask away (clear) all the bits except bits four through seven. Likewise, if you use this same value with the OR instruction, it can force bits four through seven to ones in the destination operand. The term "mask" comes from the use of these bit strings with the AND instruction; in those situations the one and zero bits behave like masking tape when you're painting something; they pass through certain bits unchanged while masking out the other bits.

Armed with these definitions, we're ready to start manipulating some bits!

5.3 Instructions That Manipulate Bits

Bit manipulation generally consists of six activities: setting bits, clearing bits, inverting bits, testing and comparing bits, extracting bits from a bit string, and inserting bits into a bit string. By now you should be familiar with most of the instructions we'll use to perform these operations; their introduction started way back in the earliest chapters of Volume One. Nevertheless, it's worthwhile to review the old instructions here as well as present the few bit manipulation instructions we've yet to consider.

The most basic bit manipulation instructions are the AND, OR, XOR, NOT, TEST, and shift and rotate instructions. Indeed, on the earliest 80x86 processors, these were the only instructions available for bit manipulation. The following paragraphs review these instructions, concentrating on how you could use them to manipulate bits in memory or registers.

The AND instruction provides the ability to strip away unwanted bits from some bit sequence, replacing the unwanted bits with zeros. This instruction is especially useful for isolating a bit string or a bit set that is merged with other, unrelated data (or, at least, data that is not part of the bit string or bit set). For example, suppose that a bit string consumes bit positions 12 through 24 of the EAX register, we can isolate this bit string by setting all other bits in EAX to zero by using the following instruction:

```
and( %1_1111_1111_1111_0000_0000_0000, eax );
```

Most programs use the AND instruction to clear bits that are not part of the desired bit string. In theory, you could use the OR instruction to mask all unwanted bits to ones rather than zeros, but later comparisons and operations are often easier if the unneeded bit positions contain zero.

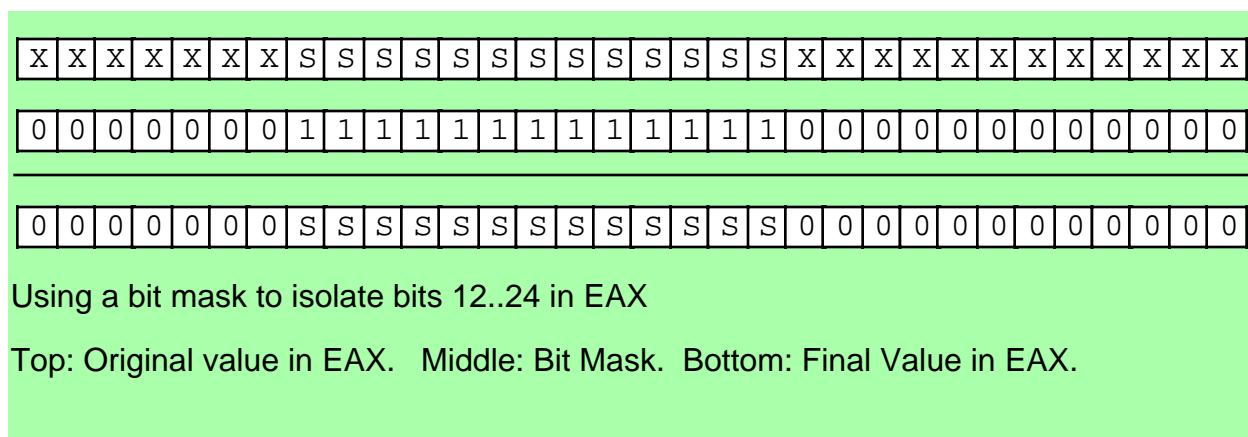


Figure 5.1 Isolating a Bit String Using the AND Instruction

Once you've cleared the unneeded bits in a set of bits, you can often operate on the bit set in place. For example, to see if the string of bits in positions 12 through 24 of EAX contain \$12F3 you could use the following code:

```
and( %1_1111_1111_1111_0000_0000_0000, eax );
cmp( eax, %1_0010_1111_0011_0000_0000_0000 );
```

Here's another solution, using constant expressions, that's a little easier to digest:

```
and( %1_1111_1111_1111_0000_0000_0000, eax );
cmp( eax, $12F3 << 12 ); // "<<12" shifts $12F3 to the left 12 bits.
```

Most of the time, however, you'll want (or need) the bit string aligned with bit zero in EAX prior to any operations you would want to perform. Of course, you can use the SHR instruction to properly align the value after you've masked it:

```
and( %1_1111_1111_1111_0000_0000_0000, eax );
shr( 12, eax );
cmp( eax, $12F3 );
<< Other operations that requires the bit string at bit #0 >>
```

Now that the bit string is aligned to bit zero, the constants and other values you use in conjunction with this value are easier to deal with.

You can also use the OR instruction to mask unwanted bits around a set of bits. However, the OR instruction does not let you clear bits, it allows you to set bits to ones. In some instances setting all the bits around your bit set may be desirable; most software, however, is easier to write if you clear the surrounding bits rather than set them.

The OR instruction is especially useful for inserting a bit set into some other bit string. To do this, there are several steps you must go through:

- Clear all the bits surrounding your bit set in the source operand.
- Clear all the bits in the destination operand where you wish to insert the bit set.
- OR the bit set and destination operand together.

For example, suppose you have a value in bits 0..12 of EAX that you wish to insert into bits 12..24 of EBX without affecting any of the other bits in EBX. You would begin by stripping out bits 13 and above from EAX; then you would strip out bits 12..24 in EBX. Next, you would shift the bits in EAX so the bit string occupies bits 12..24 of EBX. Finally, you would OR the value in EAX into EBX (see Figure 5.2):

```
and( $1FFF, eax ); // Strip all but bits 0..12 from EAX
and( $1FF_F000, ebx ); // Clear bits 12..24 in EBX.
shl( 12, eax ); // Move bits 0..12 to 12..24 in EAX.
or( eax, ebx ); // Merge the bits into EBX.
```

EBX :

X	X	X	X	X	X	X	X	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

EAX :

U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	U	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Step One: Strip the unneeded bits from EAX (the "U" bits)

EBX :

X	X	X	X	X	X	X	X	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

EAX :

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Step Two: Mask out the destination bit field in EBX.

EBX :

X	X	X	X	X	X	X	0	0	0	0	0	0	0	0	0	0	0	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

EAX :

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Step Three: Shift the bits in EAX 12 positions to the left to align them with the destination bit field.

EBX :

X	X	X	X	X	X	X	0	0	0	0	0	0	0	0	0	0	0	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

EAX :

0	0	0	0	0	0	0	A	A	A	A	A	A	A	A	A	A	A	A	A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Step Four: Merge the value in EAX with the value in EBX.

EBX :

X	X	X	X	X	X	X	A	A	A	A	A	A	A	A	A	A	A	A	A	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

EAX :

0	0	0	0	0	0	0	A	A	A	A	A	A	A	A	A	A	A	A	A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Final result is in EBX.

Figure 5.2 Inserting Bits 0..12 of EAX into Bits 12..24 of EBX

In this example the desired bits (AAAAAAAAAAAA) formed a bit string. However, this algorithm still works fine even if you're manipulating a non-contiguous set of bits. All you've got to do is to create an appropriate bit mask you can use for ANDing that has ones in the appropriate places.

When working with bit masks, it is incredibly poor programming style to use literal numeric constants as in the past few examples. You should always create symbolic constants in the HLA CONST (or VAL) section for your bit masks. Combined with some constant expressions, you can produce code that is much easier to read and maintain. The current example code is more properly written as:


```

const
    StartPosn: 12;
    BitMask: dword := $1FFF << StartPosn;    // Mask occupies bits 12..24
    .
    .
    .
    shl( StartPosn, eax );    // Move into position.
    and( BitMask, eax );    // Strip all but bits 12..24 from EAX
    and( !BitMask, ebx );    // Clear bits 12..24 in EBX.
    or( eax, ebx );    // Merge the bits into EBX.

```

Notice the use of the compile-time not operator (“!”) to invert the bit mask in order to clear the bit positions in EBX where the code inserts the bits from EAX. This saves having to create another constant in the program that has to be changed anytime you modify the *BitMask* constant. Having to maintain two separate symbols whose values are dependent on one another is not a good thing in a program.

Of course, in addition to merging one bit set with another, the OR instruction is also useful for forcing bits to one in a bit string. By setting various bits in a source operand to one you can force the corresponding bits in the destination operand to one by using the OR instruction.

The XOR instruction, as you may recall, gives you the ability to invert selected bits belonging to a bit set. Although the need to invert bits isn’t as common as the need to set or clear them, the XOR instruction still sees considerable use in bit manipulation programs. Of course, if you want to invert all the bits in some destination operand, the NOT instruction is probably more appropriate than the XOR instruction; however, to invert selected bits while not affecting others, the XOR is the way to go.

One interesting fact about XOR’s operation is that it lets you manipulate known data in just about any way imaginable. For example, if you know that a field contains %1010 you can force that field to zero by XORing it with %1010. Similarly, you can force it to %1111 by XORing it with %0101. Although this might seem like a waste, since you can easily force this four-bit string to zero or all ones using AND or OR, the XOR instruction has two advantages: (1) you are not limited to forcing the field to all zeros or all ones; you can actually set these bits to any of the 16 valid combinations via XOR; (2) if you need to manipulate other bits in the destination operand at the same time, AND/OR may not be able to accommodate you. For example, suppose that you know that one field contains %1010 that you want to force to zero and another field contains %1000 and you wish to increment that field by one (i.e., set the field to %1001). You cannot accomplish both operations with a single AND or OR instruction, but you can do this with a single XOR instruction; just XOR the first field with %1010 and the second field with %0001. Remember, however, that this trick only works if you know the current value of a bit set within the destination operand. Of course, while you’re adjusting the values of bit fields containing known values, you can invert bits in other fields simultaneously.

In addition to setting, clearing, and inverting bits in some destination operand, the AND, OR, and XOR instructions also affect various condition codes in the FLAGS register. These instructions affect the flags as follows:

- These instructions always clear the carry and overflow flags.
- These instructions set the sign flag if the result has a one in the H.O. bit; they clear it otherwise. I.e., these instructions copy the H.O. bit of the result into the sign flag.
- These instructions set/clear the zero flag depending on whether the result is zero.
- These instructions set the parity flag if there are an even number of set bits in the L.O. byte of the destination operand, they clear the parity flag if there are an odd number of one bits in the L.O. byte of the destination operand.

The first thing to note is that these instructions always clear the carry and overflow flags. This means that you cannot expect the system to preserve the state of these two flags across the execution of these instructions. A very common mistake in many assembly language programs is the assumption that these instructions do not affect the carry flag. Many people will execute an instruction that sets/clears the carry flag, execute an AND/OR/XOR instruction, and then attempt to test the state of the carry from the previous instruction. This simply will not work.

One of the more interesting aspects to these instructions is that they copy the H.O. bit of their result into the sign flag. This means that you can easily test the setting of the H.O. bit of the result by testing the sign flag (using SETS/SETNS, JS/JNS, or by using the @S/@NS flags in a boolean expression). For this reason, many assembly language programmers will often place an important boolean variable in the H.O. bit of some operand so they can easily test the state of that bit using the sign flag after a logical operation.

We haven't talked much about the parity flag in this text. Indeed, earlier volumes have done little more than acknowledge its existence. We're not going to get into a big discussion of this flag and what you use it for since the primary purpose for this flag has been taken over by hardware¹. However, since this is a chapter on bit manipulation and parity computation is a bit manipulation operation, it seems only fitting to provide a brief discussion of the parity flag at this time.

Parity is a very simple error detection scheme originally employed by telegraphs and other serial communication schemes. The idea was to count the number of set bits in a character and include an extra bit in the transmission to indicate whether that character contained an even or odd number of set bits. The receiving end of the transmission would also count the bits and verify that the extra "parity" bit indicated a successful transmission. We're not going to explore the information theory aspects of this error checking scheme at this point other than to point out that the purpose of the parity flag is to help compute the value of this extra bit.

The 80x86 AND, OR, and XOR instructions set the parity bit if the L.O. byte of their operand contains an even number of set bits. An important fact bears repeating here: the parity flag only reflects the number of set bits in the L.O. byte of the destination operand; it does not include the H.O. bytes in a word, double word, or other sized operand. The instruction set only uses the L.O. byte to compute the parity because communication programs that use parity are typically character-oriented transmission systems (there are better error checking schemes if you transmit more than eight bits at a time).

Although the need to know whether the L.O. (or only) byte of some computation has an even or odd number of set bits isn't common in modern programs, it does come in useful once in a great while. Since this is, intrinsically, a bit operation, it's worthwhile to mention the use of this flag and how the AND/OR/XOR instructions affect this flag.

The zero flag setting is one of the more important results the AND/OR/XOR instructions produce. Indeed, programs reference this flag so often after the AND instruction that Intel added a separate instruction, TEST, whose main purpose was to logically AND two results and set the flags without otherwise affecting either instruction operand.

There are three main uses of the zero flag after the execution of an AND or TEST instruction: (1) checking to see if a particular bit in an operand is set; (2) checking to see if at least one of several bits in a bit set is one; and (3) checking to see if an operand is zero. Use (1) is actually a special case of (2) where the bit set contains only a single bit. We'll explore each of these uses in the following paragraphs.

A common use for the AND instruction, and also the original reason for the inclusion of the TEST instruction in the 80x86 instruction set, is to test to see if a particular bit is set in a given operand. To perform this type of test, you would normally AND/TEST a constant value containing a single set bit with the operand you wish to test. These clears all the other bits in the second operand leaving a zero in the bit position under test (the bit position with the single set bit in the constant operand) if the operand contains a zero in that position and leaving a one if the operand contains a one in that position. Since all of the other bits in the result are zero, the entire result will be zero if that particular bit is zero, the entire result will be non-zero if that bit position contains a one. The 80x86 reflects this status in the zero flag (Z=1 indicates a zero bit, Z=0 indicates a one bit). The following instruction sequence demonstrates how to test to see if bit four is set in EAX:

```
test( %1_000, eax ); // Check bit #4 to see if it is 0/1
if( @nz ) then
```

```
<< Do this if the bit is set >>
```

1. Serial communications chips and other communication hardware that uses parity for error checking normally computes the parity in hardware, you don't have to use software for this purpose.

```

else

    << Do this if the bit is clear >>

endif;

```

You can also use the AND/TEST instructions to see if any one of several bits is set. Simply supply a constant that has one bits in all the positions you want to test (and zeros everywhere else). ANDing such a value with an unknown quantity will produce a non-zero value if one or more of the bits in the operand under test contain a one. The following example tests to see if the value in EAX contains a one in bit positions one, two, four, and seven:

```

test( %1001_0010, eax );
if( @nz ) then // at least one of the bits is set.

    << do whatever needs to be done if one of the bits is set >>

endif;

```

Note that you cannot use a single AND or TEST instruction to see if all the corresponding bits in the bit set are equal to one. To accomplish this, you must first mask out the bits that are not in the set and then compare the result against the mask itself. If the result is equal to the mask, then all the bits in the bit set contain ones. You must use the AND instruction for this operation as the TEST instruction does not mask out any bits. The following example checks to see if all the bits corresponding to a value this code calls *bitMask* are equal to one:

```

and( bitMask, eax );
cmp( eax, bitMask );
if( @e ) then

    << All the bit positions in EAX corresponding to the set >>
    << bits in bitMask are equal to one if we get here. >>

endif;

```

Of course, once we stick the CMP instruction in there, we don't really have to check to see if all the bits in the bit set contain ones. We can check for any combination of values by specifying the appropriate value as the operand to the CMP instruction.

Note that the TEST/AND instructions will only set the zero flag in the above code sequences if all the bits in EAX (or other destination operand) have zeros in the positions where ones appear in the constant operand. This suggests another way to check for all ones in the bit set: invert the value in EAX prior to using the AND or TEST instruction. Then if the zero flag is set, you know that there were all ones in the (original) bit set, e.g.,

```

not( eax );
test( bitMask, eax );
if( @z ) then

    << At this point, EAX contained all ones in the bit positions >>
    << occupied by ones in the bitMask constant. >>

endif;

```

The paragraphs above all suggest that the *bitMask* (i.e., source operand) is a constant. This was for purposes of example only. In fact, you can use a variable or other register here, if you prefer. Simply load that variable or register with the appropriate bit mask before you execute the TEST, AND, or CMP instructions in the examples above.

Another set of instructions we've already seen that we can use to manipulate bits are the bit test instructions. These instructions include BT (bit test), BTS (bit test and set), BTC (bit test and complement), and

BTR (bit test and reset). We've used these instructions to manipulate bits in HLA character set variables, we can also use them to manipulate bits in general. The BT_x instructions allow the following syntactical forms:

```

btx( BitNumber, BitsToTest );

btx( reg16, reg16 );
btx( reg32, reg32 );
btx( constant, reg16 );
btx( constant, reg32 );

btx( reg16, mem16 );
btx( reg32, mem32 );
btx( constant, mem16 );
btx( constant, mem32 );

```

The BT instruction's first operand is a bit number that specifies which bit to check in the second operand. If the second operand is a register, then the first operand must contain a value between zero and the size of the register (in bits) minus one; since the 80x86's largest registers are 32 bits, this value has the maximum value 31 (for 32-bit registers). If the second operand is a memory location, then the bit count is not limited to values in the range 0..31. If the first operand is a constant, it can be any eight-bit value in the range 0..255. If the first operand is a register, it has no limitation.

The BT instruction copies the specified bit from the second operand into the carry flag. For example, the "bt(8, eax);" instruction copies bit number eight of the EAX register into the carry flag. You can test the carry flag after this instruction to determine whether bit eight was set or clear in EAX.

In general, the BT instruction is, perhaps, not the best instruction for testing individual bits in a register. The TEST (or AND) instruction is a bit more efficient. These latter two instructions are Intel "RISC Core" instructions while the BT instruction is a "Complex" instruction. Therefore, you will often get better performance using TEST or AND rather than BT. If you want to test bits in memory operands (especially in bit arrays), then the BT instruction is probably a reasonable way to go.

The BTS, BTC, and BTR instructions manipulate the bit they test while they are testing it. These instructions are rather slow and you should avoid them if performance is your primary concern. In a later volume when we discuss semaphores you will see the true purpose for these instructions. Until then, if performance (versus convenience) is an issue, you should always try two different algorithms, one that uses these instructions, one that uses AND/OR instructions, and measure the performance difference; then choose the best of the two different approaches.

The shift and rotate instructions are another group of instructions you can use to manipulate and test bits. Of course, all of these instructions move the H.O. (left shift/rotate) or L.O. (right shift/rotate) bits into the carry flag. Therefore, you can test the carry flag after you execute one of these instructions to determine the original setting of the operand's H.O. or L.O. bit (depending on the direction). Of course, the shift and rotate instructions are invaluable for aligning bit strings, packing, and unpacking data. Volume One has several examples of this and, of course, some earlier examples in the section also use the shift instructions for this purpose.

5.4 The Carry Flag as a Bit Accumulator

The BT_x, shift, and rotate instructions all set or clear the carry flag depending on the operation and/or selected bit. Since these instructions place their "bit result" in the carry flag, it is often convenient to think of the carry flag as a one-bit register or accumulator for bit operations. In this section we will explore some of the operations possible with this bit result in the carry flag.

Instructions that will be useful for manipulating bit results in the carry flag are those that use the carry flag as some sort of input value. The following is a sampling of such instructions:

- ADC, SBB
- RCL, RCR

- CMC (we'll throw in CLC and STC even though they don't use the carry as input)
- JC, JNC
- SETC, SETNC

The ADC and SBB instructions add or subtract their operands along with the carry flag. So if you've computed some bit result into the carry flag, you can figure that result into an addition or subtraction using these instructions. This isn't a common operation, but it is available if it's useful to you.

To merge a bit result in the carry flag, you most often use the rotate through carry instructions (RCL and RCR). These instructions, of course, move the carry flag into the L.O. or H.O. bits of their destination operand. These instructions are very useful for packing a set of bit results into a byte, word, or double word value.

The CMC (complement carry) instruction lets you easily invert the result of some bit operation. You can also use the CLC and STC instructions to initialize the carry flag prior to some string of bit operations involving the carry flag.

Of course, instructions that test the carry flag are going to be very popular after a calculation that leaves a bit result in the carry flag. The JC, JNC, SETC, and SETNC instructions are quite useful here. You can also use the HLA @C and @NC operands in a boolean expression to test the result in the carry flag.

If you have a sequence of bit calculations and you would like to test to see if the calculations produce a specific sequence of one-bit results, the easiest way to do this is to clear a register or memory location and use the RCL or RCR instructions to shift each result into that location. Once the bit operations are complete, then you can compare the register or memory location against a constant value to see if you've obtained a particular result sequence. If you want to test a sequence of results involving conjunction and disjunction (i.e., strings of results involving ANDs and ORs) then you could use the SETC and SETNC instruction to set a register to zero or one and then use the AND/OR instructions to merge the results.

5.5 Packing and Unpacking Bit Strings

A common bit operation is inserting a bit string into an operand or extracting a bit string from an operand. Previous chapters in this text have provided simple examples of packing and unpacking such data, now it is time to formally describe how to do this.

For the purposes of the current discussion, we will assume that we're dealing with bit strings; that is, a contiguous sequence of bits. A little later in this chapter we'll take a look at how to extract and insert bit sets in an operand. Another simplification we'll make is that the bit string completely fits within a byte, word, or double word operand. Large bit strings that cross object boundaries require additional processing; a discussion of bit strings that cross double word boundaries appears later in this section.

A bit string has two attributes that we must consider when packing and unpacking that bit string: a starting bit position and a length. The starting bit position is the bit number of the L.O. bit of the string in the larger operand. The length, of course, is the number of bits in the operand. To insert (pack) data into a destination operand we will assume that we start with a bit string of the appropriate length that is right-justified (i.e., starts in bit position zero) in an operand and is zero extended to eight, sixteen, or thirty-two bits. The task is to insert this data at the appropriate starting position in some other operand that is eight, sixteen, or thirty-bits wide. There is no guarantee that the destination bit positions contain any particular value.

The first two steps (which can occur in any order) is to clear out the corresponding bits in the destination operand and shift (a copy of) the bit string so that the L.O. bit begins at the appropriate bit position. After completing these two steps, the third step is to OR the shifted result with the destination operand. This inserts the bit string into the destination operand.

Destination:

X	X	X	X	X	X	X	D	D	D	D	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Source:

0	0	0	0	0	0	0	0	0	0	0	0	Y	Y	Y	Y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Step One: Insert YYYY into the positions occupied by DDDD in the destination operand.
Begin by shifting the source operand to the left five bits.

Destination:

X	X	X	X	X	X	X	D	D	D	D	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Source:

0	0	0	0	0	0	0	0	Y	Y	Y	Y	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Step Two: Clear out the destination bits using the AND instruction.

Destination:

X	X	X	X	X	X	X	0	0	0	0	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Source:

0	0	0	0	0	0	0	0	Y	Y	Y	Y	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Step Three: OR the two values together

Destination:

X	X	X	X	X	X	X	Y	Y	Y	Y	X	X	X	X	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Source:

0	0	0	0	0	0	0	0	Y	Y	Y	Y	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Final result appears in the destination operand.

Figure 5.3 Inserting a Bit String Into a Destination Operand

It only takes three instructions to insert a bit string of known length into a destination operand. The following three instructions demonstrate how to handle the insertion operation in Figure 5.3; These instructions assume that the source operand is in BX and the destination operand is AX:

```
shl( 5, bx );
and( %111111000011111, ax );
or( bx, ax );
```

If the length and the starting position aren't known when you're writing the program (that is, you have to calculate them at run time), then bit string insertion is a little more difficult. However, with the use of a lookup table it's still an easy operation to accomplish. Let's assume that we have two eight-bit values: a starting bit position for the field we're inserting and a non-zero eight-bit length value. Also assume that the source operand is in EBX and the destination operand is EAX. The code to insert one operand into another could take the following form:

```
readonly

// The index into the following table specifies the length of the bit string
// at each position:

MaskByLen: dword[ 32 ] :=
[
    0, $1, $3, $7, $f, $1f, $3f, $7f,
    $ff, $1ff, $3ff, $7ff, $fff, $1fff, $3fff, $7fff, $ffff,
    $1_ffff, $3_ffff, $7_ffff, $f_ffff,
    $1f_ffff, $3f_ffff, $7f_ffff, $ff_ffff,
    $1ff_ffff, $3ff_ffff, $7ff_ffff, $fff_ffff,
    $1fff_ffff, $3fff_ffff, $7fff_ffff, $ffff_ffff
];
.
.
.
movzx( Length, edx );
mov( MaskByLen[ edx*4 ], edx );
mov( StartingPosition, cl );
shl( cl, edx );
not( edx );
shl( cl, ebx );
and( edx, eax );
or( ebx, eax );
```

Each entry in the *MaskByLen* table contains the number of one bits specified by the index into the table. Using the *Length* value as an index into this table fetches a value that has as many one bits as the *Length* value. The code above fetches an appropriate mask, shifts it to the left so that the L.O. bit of this run of ones matches the starting position of the field into which we want to insert the data, then it inverts the mask and uses the inverted value to clear the appropriate bits in the destination operand.

To extract a bit string from a larger operand is just as easy as inserting a bit string into some larger operand. All you've got to do is mask out the unwanted bits and then shift the result until the L.O. bit of the bit string is in bit zero of the destination operand. For example, to extract the four-bit field starting at bit position five in EBX and leave the result in EAX, you could use the following code:

```
mov( ebx, eax );           // Copy data to destination.
and( %1_1110_0000, ebx ); // Strip unwanted bits.
shr( 5, eax );           // Right justify to bit position zero.
```

If you do not know the bit string's length and starting position when you're writing the program, you can still extract the desired bit string. The code is very similar to insertion (though a tiny bit simpler). Assuming you have the *Length* and *StartingPosition* values we used when inserting a bit string, you can extract the corresponding bit string using the following code (assuming source=EBX and dest=EAX):

```
movzx( Length, edx );
mov( MaskByLen[ edx*4 ], edx );
mov( StartingPosition, cl );
mov( ebx, eax );
shr( cl, eax );
and( edx, eax );
```

The examples up to this point all assume that the bit string appears completely within a double word (or smaller) object. This will always be the case if the bit string is less than or equal to 24 bits in length. However, if the length of the bit string plus its starting position (mod eight) within an object is greater than 32, then the bit string will cross a double word boundary within the object. To extract such bit strings requires up to three operations: one operation to extract the start of the bit string (up to the first double word boundary), an operation that copies whole double words (assuming the bit string is so long that it consumes several double words), and a final operation that copies left-over bits in the last double word at the end of the bit string. The actual implementation of this operation is left as an exercise at the end of this volume.

5.6 Coalescing Bit Sets and Distributing Bit Strings

Inserting and extracting bit sets is little different than inserting and extracting bit strings if the “shape” of the bit set you’re inserting (or resulting bit set you’re extracting) is the same as the bit set in the main object. The “shape” of a bit set is the distribution of the bits in the set, ignoring the starting bit position of the set. So a bit set that includes bits zero, four, five, six, and seven has the same shape as a bit set that includes bits 12, 16, 17, 18, and 19 since the distribution of the bits is the same. The code to insert or extract this bit set is nearly identical to that of the previous section; the only difference is the mask value you use. For example, to insert this bit set starting at bit number zero in EAX into the corresponding bit set starting at position 12 in EBX, you could use the following code:

```
and( %1111_0001_0000_0000_0000, ebx ); // Mask out destination bits.
shl( 12, eax ); // Move source bits into posn.
or( eax, ebx ); // Merge the bit set into EBX.
```

However, suppose you have five bits in bit positions zero through four in EAX and you want to merge them into bits 12, 16, 17, 18, and 19 in EBX. Somehow you’ve got to distribute the bits in EAX prior to logically ORing the values into EBX. Given the fact that this particular bit set has only two runs of one bits, the process is somewhat simplified, the following code achieves this in a somewhat sneaky fashion:

```
and( %1111_0001_0000_0000_0000, ebx );
shl( 3, eax ); // Spread out the bits: 1-4 goes to 4-7 and 0 to 3.
btr( 3, eax ); // Bit 3->carry and then clear bit 3
rcl( 12, eax ); // Shift in carry and put bits into final position
or( eax, ebx ); // Merge the bit set into EBX.
```

This trick with the BTR (bit test and reset) instruction worked well because we only had one bit out of place in the original source operand. Alas, had the bits all been in the wrong location relative to one another, this scheme might not have worked quite as well. We’ll see a more general solution in just a moment.

Extracting this bit set and collecting (“coalescing”) the bits into a bit string is not quite as easy. However, there are still some sneaky tricks we can pull. Consider the following code that extracts the bit set from EBX and places the result into bits 0..4 of EAX:

```
mov( ebx, eax );
and( %1111_0001_0000_0000_0000, eax ); // Strip unwanted bits.
shr( 5, eax ); // Put bit 12 into bit 7, etc.
shr( 3, ah ); // Move bits 11..14 to 8..11.
shr( 7, eax ); // Move down to bit zero.
```

This code moves (original) bit 12 into bit position seven, the H.O. bit of AL. At the same time it moves bits 16..19 down to bits 11..14 (bits 3..6 of AH). Then the code shifts the bits 3..6 in AH down to bit zero. This positions the H.O. bits of the bit set so that they are adjacent to the bit left in AL. Finally, the code shifts all the bits down to bit zero. Again, this is not a general solution, but it shows a clever way to attack this problem if you think about it carefully.

The problem with the coalescing and distribution algorithms above is that they are not general. They apply only to their specific bit sets. In general, specific solutions are going to provide the most efficient solution. A generalized solution (perhaps that lets you specify a mask and the code distributes or coalesces the

bits accordingly) is going to be a bit more difficult. The following code demonstrates how to distribute the bits in a bit string according to the values in a bit mask:

```
// EAX- Originally contains some value into which we insert bits from EBX.
// EBX- L.O. bits contain the values to insert into EAX.
// EDX- bitmap with ones indicating the bit positions in EAX to insert.
// CL- Scratchpad register.

        mov( 32, cl );    // Count # of bits we rotate.
        jmp DistLoop;

CopyToEAX: rcr( 1, ebx ); // Don't use SHR here, must preserve Z-flag.
          rcr( 1, eax );
          jz Done;
DistLoop: dec( cl );
          shr( 1, edx );
          jc CopyToEAX;
          ror( 1, eax ); // Keep current bit in EAX.
          jnz DistLoop;

Done:    ror( cl, eax ); // Reposition remaining bits.
```

In the code above, if we load EDX with %1100_1001 then this code will copy bits 0..3 to bits 0, 3, 6, and 7 in EAX. Notice the short circuit test that checks to see if we've exhausted the values in EDX (by checking for a zero in EDX). Note that the rotate instructions do not affect the zero flag while the shift instructions do. Hence the SHR instruction above will set the zero flag when there are no more bits to distribute (i.e., when EDX becomes zero).

The general algorithm for coalescing bits is a tad more efficient than distribution. Here's the code that will extract bits from EBX via the bit mask in EDX and leave the result in EAX:

```
// EAX- Destination register.
// EBX- Source register.
// EDX- Bitmap with ones representing bits to copy to EAX.
// EBX and EDX are not preserved.

        sub( eax, eax ); // Clear destination register.
        jmp ShiftLoop;

ShiftInEAX: rcl( 1, ebx ); // Up here we need to copy a bit from
          rcl( 1, eax ); // EBX to EAX.
ShiftLoop: shl( 1, edx ); // Check mask to see if we need to copy a bit.
          jc ShiftInEAX; // If carry set, go copy the bit.
          rcl( 1, ebx ); // Current bit is uninteresting, skip it.
          jnz ShiftLoop; // Repeat as long as there are bits in EDX.
```

This sequence takes advantage of one sneaky trait of the shift and rotate instructions: the shift instructions affect the zero flag while the rotate instructions do not. Therefore, the "shl(1, edx);" instruction sets the zero flag when EDX becomes zero (after the shift). If the carry flag was also set, the code will make one additional pass through the loop in order to shift a bit into EAX, but the next time the code shifts EDX one bit to the left, EDX is still zero and so the carry will be clear. On this iteration, the code falls out of the loop.

Another way to coalesce bits is via table lookup. By grabbing a byte of data at a time (so your tables don't get too large) you can use that byte's value as an index into a lookup table that coalesces all the bits down to bit zero. Finally, you can merge the bits at the low end of each byte together. This might produce a more efficient coalescing algorithm in certain cases. The implementation is left to the reader..

5.7 Packed Arrays of Bit Strings

Although it is far more efficient to create arrays whose elements' have an integral number of bytes, it is quite possible to create arrays of elements whose size is not a multiple of eight bits. The drawback is that calculating the "address" of an array element and manipulating that array element involves a lot of extra work. In this section we'll take a look at a few examples of packing and unpacking array elements in an array whose elements are an arbitrary number of bits long.

Before proceeding, it's probably worthwhile to discuss why you would want to bother with arrays of bit objects. The answer is simple: space. If an object only consumes three bits, you can get 2.67 times as many elements into the same space if you pack the data rather than allocating a whole byte for each object. For very large arrays, this can be a substantial savings. Of course, the cost of this space savings is speed: you've got to execute extra instructions to pack and unpack the data, thus slowing down access to the data.

The calculation for locating the bit offset of an array element in a large block of bits is almost identical to the standard array access; it is

$$\text{Element_Address_in_bits} = \text{Base_address_in_bits} + \text{index} * \text{element_size_in_bits}$$

Once you calculate the element's address in bits, you need to convert it to a byte address (since we have to use byte addresses when accessing memory) and extract the specified element. Because the base address of an array element (almost) always starts on a byte boundary, we can use the following equations to simplify this task:

$$\begin{aligned} \text{Byte_of_1st_bit} &= \text{Base_Address} + (\text{index} * \text{element_size_in_bits}) / 8 \\ \text{Offset_to_1st_bit} &= (\text{index} * \text{element_size_in_bits}) \% 8 \quad (\text{note } \% = \text{MOD}) \end{aligned}$$

For example, suppose we have an array of 200 three-bit objects that we declare as follows:

```
static
AO3Bobjects: byte[ (200*3)/8 + 1 ]; // "+1" handles truncation.
```

The constant expression in the dimension above reserves space for enough bytes to hold 600 bits (200 elements, each three bits long). As the comment notes, the expression adds an extra byte at the end to ensure we don't lose any odd bits (that won't happen in this example since 600 is evenly divisible by 8, but in general you can't count on this; one extra byte usually won't hurt things).

Now suppose you want to access the i^{th} three-bit element of this array. You can extract these bits by using the following code:

```
// Extract the ith group of three bits in AO3Bobjects and leave this value
// in EAX.

sub( ecx, ecx ); // Put i/8 remainder here.
mov( i, eax ); // Get the index into the array.
shrd( 3, eax, ecx ); // EAX/8 -> EAX and EAX mod 8 -> ECX (H.O. bits)
shr( 3, eax ); // Remember, shrd above doesn't modify eax.
rol( 3, ecx ); // Put remainder into L.O. three bits of ECX.

// Okay, fetch the word containing the three bits we want to extract.
// We have to fetch a word because the last bit or two could wind up
// crossing the byte boundary (i.e., bit offset six and seven in the
// byte).

mov( AO3Bobjects[ecx], eax );
shr( cl, eax ); // Move bits down to bit zero.
and( %111, eax ); // Remove the other bits.
```

Inserting an element into the array is a bit more difficult. In addition to computing the base address and bit offset of the array element, you've also got to create a mask to clear out the bits in the destination where you're going to insert the new data. The following code inserts the L.O. three bits of EAX into the i^{th} element of the *AO3Bobjects* array.

```

// Insert the L.O. three bits of AX into the ith element of AO3Bobjects:

readonly
    Masks: word[8] :=
        [
            !%0000_0111, !%0000_1110, !%0001_1100, !%0011_1000,
            !%0111_0000, !%1110_0000, !%1_1100_0000, !%11_1000_0000
        ];
        .
        .
        .
    sub( ecx, ecx ); // Put remainder here.
    mov( i, ebx ); // Get the index into the array.
    shrd( 3, ebx, ecx ); // i/8 -> EBX, i % 8 -> ECX.
    shr( 3, ebx );
    rol( 3, ecx );

    and( %111, ax ); // Clear unneeded bits from AX.
    mov( Masks[ecx], dx ); // Mask to clear out our array element.
    and( AO3Bobjects[ ebx ], dx ); // Grab the bits and clear those
    // we're inserting.
    shl( cl, ax ); // Put our three bits in their proper location.
    or( ax, dx ); // Merge bits into destination.
    mov( dx, AO3Bobjects[ ebx ] ); // Store back into memory.

```

Notice the use of a lookup table to generate the masks needed to clear out the appropriate position in the array. Each element of this array contains all ones except for three zeros in the position we need to clear for a given bit offset (note the use of the “!” operator to invert the constants in the table).

5.8 Searching for a Bit

A very common bit operation is to locate the end of some run of bits. A very common special case of this operation is to locate the first (or last) set or clear bit in a 16- or 32-bit value. In this section we’ll explore ways to accomplish this.

Before describing how to search for the first or last bit of a given value, perhaps it’s wise to discuss exactly what the terms “first” and “last” mean in this context. The term “first set bit” means the first bit in a value, scanning from bit zero towards the high order bit, that contains a one. A similar definition exists for the “first clear bit.” The “last set bit” is the first bit in a value, scanning from the high order bit towards bit zero, that contains a one. A similar definition exists for the last clear bit.

One obvious way to scan for the first or last bit is to use a shift instruction in a loop and count the number of iterations before you shift out a one (or zero) into the carry flag. The number of iterations specifies the position. Here’s some sample code that checks for the first set bit in EAX and returns that bit position in ECX:

```

    mov( -32, ecx ); // Count off the bit positions in ECX.
TstLp: shr( 1, eax ); // Check to see if current bit position contains
    jc Done // a one; exit loop if it does.
    inc( ecx ); // Bump up our bit counter by one.
    jnz TstLp; // Exit if we execute this loop 32 times.

Done: add( 32, cl ); // Adjust loop counter so it holds the bit posn.

// At this point, ECX contains the bit position of the first set bit.
// ECX contains 32 if EAX originally contained zero (no set bits).

```

The only thing tricky about this code is the fact that it runs the loop counter from -32 to zero rather than 32 down to zero. This makes it slightly easier to calculate the bit position once the loop terminates.

The drawback to this particular loop is that it's expensive. This loop repeats as many as 32 times depending on the original value in EAX. If the values you're checking often have lots of zeros in the L.O. bits of EAX, this code runs rather slow.

Searching for the first (or last) set bit is such a common operation that Intel added a couple of instructions on the 80386 specifically to accelerate this process. These instructions are BSF (bit scan forward) and BSR (bit scan reverse). Their syntax is as follows:

```
bsr( source, destReg );
bsf( source, destReg );
```

The source and destination operands must be the same size and they must both be 16- or 32-bit objects. The destination operand has to be a register, the source operand can be a register or a memory location.

The BSF instruction scans for the first set bit (starting from bit position zero) in the source operand. The BSR instruction scans for the last set bit in the source operand by scanning from the H.O. bit towards the L.O. bit. If these instructions find a bit that is set in the source operand then they clear the zero flag and put the bit position into the destination register. If the source register contains zero (i.e., there are no set bits) then these instructions set the zero flag and leave an indeterminate value in the destination register. Note that you should test the zero flag immediately after the execution of these instructions to validate the destination register's value. Examples:

```
mov( SomeValue, ebx );      // Value whose bits we want to check.
bsf( ebx, eax );           // Put position of first set bit in EAX.
jz  NoBitsSet;             // Branch if SomeValue contains zero.
mov( eax, FirstBit );     // Save location of first set bit.
.
.
.
```

You use the BSR instruction in an identical fashion except that it computes the bit position of the last set bit in an operand (that is, the first set bit it finds when scanning from the H.O. bit towards the L.O. bit).

The 80x86 CPUs do not provide instructions to locate the first bit containing a zero. However, you can easily scan for a zero bit by first inverting the source operand (or a copy of the source operand if you must preserve the source operand's value). If you invert the source operand, then the first "1" bit you find corresponds to the first zero bit in the original operand value.

The BSF and BSR instructions are complex instructions (i.e., they are not a part of the 80x86 "RISC core" instruction set). Therefore, these instructions are necessarily as fast as other instructions. Indeed, in some circumstances it may be faster to locate the first set bit using discrete instructions. However, since the execution time of these instructions varies widely from CPU to CPU, you should first test the performance of these instructions prior to using them in time critical code.

Note that the BSF and BSR instructions do not affect the source operand. A common operation is to extract the first (or last) set bit you find in some operand. That is, you might want to clear the bit once you find it. If the source operand is a register (or you can easily move it into a register) then you can use the BTR (or BTC) instruction to clear the bit once you've found it. Here's some code that achieves this result:

```
bsf( eax, ecx );          // Locate first set bit in EAX.
if( @nz ) then           // If we found a bit, clear it.

    btr( ecx, eax );     // Clear the bit we just found.

endif;
```

At the end of this sequence, the zero flag indicates whether we found a bit (note that BTR does not affect the zero flag). Alternately, you could add an ELSE section to the IF statement above that handles the case when the source operand (EAX) contains zero at the beginning of this instruction sequence.

Since the BSF and BSR instructions only support 16- and 32-bit operands, you will have to compute the first bit position of an eight-bit operand a little differently. There are a couple of reasonable approaches. First, of course, you can usually zero extend an eight-bit operand to 16 or 32 bits and then use the BSF or

BSR instructions on this operand. Another alternative is to create a lookup table where each entry in the table contains the number of bits in the value you use as an index into the table; then you can use the XLAT instruction to “compute” the first bit position in the value (note that you will have to handle the value zero as a special case). Another solution is to use the shift algorithm appearing at the beginning of this section; for an eight-bit operand, this is not an entirely inefficient solution.

One interesting use of the BSF and BSR instructions is to “fill in” a character set with all the values from the lowest-valued character in the set through the highest-valued character. For example, suppose a character set contains the values {‘A’, ‘M’, ‘a’..‘n’, ‘z’}; if we filled in the gaps in this character set we would have the values {‘A’..‘z’}. To compute this new set we can use BSF to determine the ASCII code of the first character in the set and BSR to determine the ASCII code of the last character in the set. After doing this, we can feed those two ASCII codes to the *cs.rangeChar* function to compute the new set.

You can also use the BSF and BSR instructions to determine the size of a run of bits, assuming that you have a single run of bits in your operand. Simply locate the first and last bits in the run (as above) and the compute the difference (plus one) of the two values. Of course, this scheme is only valid if there are no intervening zeros between the first and last set bits in the value.

5.9 Counting Bits

The last example in the previous section demonstrates a specific case of a very general problem: counting bits. Unfortunately, that example has a severe limitation: it only counts a single run of one bits appearing in the source operand. This section discusses a more general solution to this problem.

Hardly a week goes by that someone doesn’t ask how to count the number of bits in a register operand on one of the Internet news groups. This is a common request, undoubtedly, because many assembly language course instructors assign this task a project to their students as a way to teach them about the shift and rotate instructions. Undoubtedly, the solution these instructor expect is something like the following:

```
// BitCount1:
//
// Counts the bits in the EAX register, returning the count in EBX.

        mov( 32, cl );    // Count the 32 bits in EAX.
        sub( ebx, ebx ); // Accumulate the count here.
CntLoop: shr( 1, eax );   // Shift next bit out of EAX and into Carry.
        adc( 0, bl );    // Add the carry into the EBX register.
        dec( cl );      // Repeat 32 times.
        jnz CntLoop
```

The “trick” worth noting here is that this code uses the ADC instruction to add the value of the carry flag into the BL register. Since the count is going to be less than 32, the result will fit comfortably into BL. This code uses “adc(0, bl);” rather than “adc(0, ebx);” because the former instruction is smaller.

Tricky code or not, this instruction sequence is not particularly fast. As you can tell with just a small amount of analysis, the loop above always executes 32 times, so this code sequence executes 130 instructions (four instructions per iteration plus two extra instructions). One might ask if there is a more efficient solution, the answer is yes. The following code, taken from the AMD Athlon optimization guide, provides a faster solution (see the comments for a description of the algorithm):

```
// bitCount-
//
// Counts the number of "1" bits in a dword value.
// This function returns the dword count value in EAX.

procedure bits.cnt( BitsToCnt:dword ); nodisplay;

const
    EveryOtherBit      := $5555_5555;
```

```

EveryAlternatePair := $3333_3333;
EvenNibbles       := $0f0f_0f0f;

begin cnt;

    push( edx );
    mov( BitsToCnt, eax );
    mov( eax, edx );

    // Compute sum of each pair of bits
    // in EAX. The algorithm treats
    // each pair of bits in EAX as a two
    // bit number and calculates the
    // number of bits as follows (description
    // is for bits zero and one, it generalizes
    // to each pair):
    //
    // EDX =  BIT1 BIT0
    // EAX =    0 BIT1
    //
    // EDX-EAX =  00 if both bits were zero.
    //             01 if Bit0=1 and Bit1=0.
    //             01 if Bit0=0 and Bit1=1.
    //             10 if Bit0=1 and Bit1=1.
    //
    // Note that the result is left in EDX.

    shr( 1, eax );
    and( EveryOtherBit, eax );
    sub( eax, edx );

    // Now sum up the groups of two bits to
    // produces sums of four bits. This works
    // as follows:
    //
    // EDX = bits 2,3, 6,7, 10,11, 14,15, ..., 30,31
    //       in bit positions 0,1, 4,5, ..., 28,29 with
    //       zeros in the other positions.
    //
    // EAX = bits 0,1, 4,5, 8,9, ... 28,29 with zeros
    //       in the other positions.
    //
    // EDX+EAX produces the sums of these pairs of bits.
    // The sums consume bits 0,1,2, 4,5,6, 8,9,10, ... 28,29,30
    // in EAX with the remaining bits all containing zero.

    mov( edx, eax );
    shr( 2, edx );
    and( EveryAlternatePair, eax );
    and( EveryAlternatePair, edx );
    add( edx, eax );

    // Now compute the sums of the even and odd nibbles in the
    // number. Since bits 3, 7, 11, etc. in EAX all contain
    // zero from the above calculation, we don't need to AND
    // anything first, just shift and add the two values.
    // This computes the sum of the bits in the four bytes
    // as four separate value in EAX (AL contains number of
    // bits in original AL, AH contains number of bits in
    // original AH, etc.)

```

```

mov( eax, edx );
shr( 4, eax );
add( edx, eax );
and( EvenNibbles, eax );

// Now for the tricky part.
// We want to compute the sum of the four bytes
// and return the result in EAX. The following
// multiplication achieves this. It works
// as follows:
// (1) the $01 component leaves bits 24..31
//     in bits 24..31.
//
// (2) the $100 component adds bits 17..23
//     into bits 24..31.
//
// (3) the $1_0000 component adds bits 8..15
//     into bits 24..31.
//
// (4) the $1000_0000 component adds bits 0..7
//     into bits 24..31.
//
// Bits 0..23 are filled with garbage, but bits
// 24..31 contain the actual sum of the bits
// in EAX's original value. The SHR instruction
// moves this value into bits 0..7 and zeroes
// out the H.O. bits of EAX.

intmul( $0101_0101, eax );
shr( 24, eax );

pop( edx );

end cnt;

```

5.10 Reversing a Bit String

Another common programming project instructions assign, and a useful function in its own right, is a program that reverses the bits in an operand. That is, it swaps the L.O. bit with the H.O. bit, bit #1 with the next-to-H.O. bit, etc. The typical solution an instructor probably expects for this assignment is the following:

```

// Reverse the 32-bits in EAX, leaving the result in EBX:

mov( 32, cl );
RvsLoop: shr( 1, eax ); // Move current bit in EAX to the carry flag.
          rcl( 1, ebx ); // Shift the bit back into EBX, backwards.
          dec( cl );
          jnz RvsLoop

```

As with the previous examples, this code suffers from the fact that it repeats the loop 32 times for a grand total of 129 instructions. By unrolling the loop you can get it down to 64 instructions, but this is still somewhat expensive.

As usual, the best solution to an optimization problem is often a better algorithm rather than attempting to tweak your code by trying to choose faster instructions to speed up some code. However, a little intelligence goes a long way when manipulating bits. In the last section, for example, we were able to speed up

counting the bits in a string by substituting a more complex algorithm for the simplistic “shift and count” algorithm. In the example above, we are once again faced with a very simple algorithm with a loop that repeats for one bit in each number. The question is: “Can we discover an algorithm that doesn’t execute 129 instructions to reverse the bits in a 32-bit register?” The answer is “yes” and the trick is to do as much work as possible in parallel.

Suppose that all we wanted to do was swap the even and odd bits in a 32-bit value. We can easily swap the even and odd bits in EAX using the following code:

```
mov( eax, edx );           // Make a copy of the odd bits in the data.
shr( 1, eax );            // Move the even bits to the odd positions.
and( $5555_5555, edx );   // Isolate the odd bits by clearing even bits.
and( $5555_5555, eax );   // Isolate the even bits (in odd posn now).
shl( 1, edx );            // Move the odd bits to the even positions.
or( edx, eax );           // Merge the bits and complete the swap.
```

Of course, swapping the even and odd bits, while somewhat interesting, does not solve our larger problem of reversing all the bits in the number. But it does take us part of the way there. For example, if after executing the code sequence above, we swap adjacent pairs of bits, then we’ve managed to swap the bits in all the nibbles in the 32-bit value. Swapping adjacent pairs of bits is done in a manner very similar to the above, the code is

```
mov( eax, edx );           // Make a copy of the odd numbered bit pairs.
shr( 2, eax );            // Move the even bit pairs to the odd posn.
and( $3333_3333, edx );   // Isolate the odd pairs by clearing even pairs.
and( $3333_3333, eax );   // Isolate the even pairs (in odd posn now).
shl( 2, edx );            // Move the odd pairs to the even positions.
or( edx, eax );           // Merge the bits and complete the swap.
```

After completing the sequence above we swap the adjacent nibbles in the 32-bit register. Again, the only difference is the bit mask and the length of the shifts. Here’s the code:

```
mov( eax, edx );           // Make a copy of the odd numbered nibbles.
shr( 4, eax );            // Move the even nibbles to the odd position.
and( $0f0f_0f0f, edx );   // Isolate the odd nibbles.
and( $0f0f_0f0f, eax );   // Isolate the even nibbles (in odd posn now).
shl( 4, edx );            // Move the odd pairs to the even positions.
or( edx, eax );           // Merge the bits and complete the swap.
```

You can probably see the pattern developing and can figure out that in the next two steps we’ve got to swap the bytes and then the words in this object. You can use code like the above, but there is a better way – use the BSWAP instruction. The BSWAP (byte swap) instruction uses the following syntax:

```
bswap( reg32 );
```

This instruction swaps bytes zero and three and it swaps bytes one and two in the specified 32-bit register. The principle use of this instruction is to convert data between the so-called “little endian” and “big-endian” data formats². Although we don’t specifically need this instruction for this purpose here, the BSWAP instruction does swap the bytes and words in a 32-bit object exactly the way we want them when reversing bits, so rather than sticking in another 12 instructions to swap the bytes and then the words, we can simply use a “bswap(eax);” instruction to complete the job after the instructions above. The final code sequence is

```
mov( eax, edx );           // Make a copy of the odd bits in the data.
shr( 1, eax );            // Move the even bits to the odd positions.
and( $5555_5555, edx );   // Isolate the odd bits by clearing even bits.
and( $5555_5555, eax );   // Isolate the even bits (in odd posn now).
shl( 1, edx );            // Move the odd bits to the even positions.
or( edx, eax );           // Merge the bits and complete the swap.
```

2. In the little endian system, which the native 80x86 format, the L.O. byte of an object appears at the lowest address in memory. In the big endian system, which various RISC processors use, the H.O. byte of an object appears at the lowest address in memory. The BSWAP instruction converts between these two data formats.


```

mov( eax, edx );           // Make a copy of the odd numbered bit pairs.
shr( 2, eax );            // Move the even bit pairs to the odd posn.
and( $3333_3333, edx );   // Isolate the odd pairs by clearing even pairs.
and( $3333_3333, eax );   // Isolate the even pairs (in odd posn now).
shl( 2, edx );            // Move the odd pairs to the even positions.
or( edx, eax );           // Merge the bits and complete the swap.

mov( eax, edx );           // Make a copy of the odd numbered nibbles.
shr( 4, eax );            // Move the even nibbles to the odd position.
and( $0f0f_0f0f, edx );   // Isolate the odd nibbles.
and( $0f0f_0f0f, eax );   // Isolate the even nibbles (in odd posn now).
shl( 4, edx );            // Move the odd pairs to the even positions.
or( edx, eax );           // Merge the bits and complete the swap.

bswap( eax );             // Swap the bytes and words.

```

This algorithm only requires 19 instructions and it executes much faster than the bit shifting loop appearing earlier. Of course, this sequence does consume a bit more memory, so if you're trying to save memory rather than clock cycles, the loop is probably a better solution.

5.11 Merging Bit Strings

Another common bit string operation is producing a single bit string by merging, or interleaving, bits from two different sources. The following example code sequence creates a 32-bit string by merging alternate bits from two 16-bit strings:

```

// Merge two 16-bit strings into a single 32-bit string.
// AX - Source for even numbered bits.
// BX - Source for odd numbered bits.
// CL - Scratch register.
// EDX- Destination register.

mov( 16, cl );
MergeLp: shrd( 1, eax, edx ); // Shift a bit from EAX into EDX.
          shrd( 1, ebx, edx ); // Shift a bit from EBX into EDX.
          dec( cl );
          jne MergeLp;

```

This particular example merged two 16-bit values together, alternating their bits in the result value. For a faster implementation of this code, unrolling the loop is probably your best bet since this eliminates half the instructions that execute on each iteration of the loop above.

With a few slight modifications, we could also have merged four eight-bit values together, or we could have generated the result using other bit sequences; for example, the following code copies bits 0..5 from EAX, then bits 0..4 from EBX, then bits 6..11 from EAX, then bits 5..15 from EBX, and finally bits 12..15 from EAX:

```

shrd( 6, eax, edx );
shrd( 5, ebx, edx );
shrd( 6, eax, edx );
shrd( 11, ebx, edx );
shrd( 4, eax, edx );

```

5.12 Extracting Bit Strings

Of course, we can easily accomplish the converse of merging two bit streams; i.e., we can extract and distribute bits in a bit string among multiple destinations. The following code takes the 32-bit value in EAX and distributes alternate bits among the BX and DX registers:

```

        mov( 16, cl );           // Count off the number of loop iterations.
ExtractLp: ` shr( 1, eax );       // Extract even bits to (E)BX.
              rcr( 1, ebx );
              shr( 1, eax );       // Extract odd bits to (E)DX.
              rcr( 1, edx );
              dec( cl );           // Repeat 16 times.
              jnz ExtractLp;
              shr( 16, ebx );      // Need to move the results from the H.O.
              shr( 16, edx );      // bytes of EBX/EDX to the L.O. bytes.

```

This sequence executes 99 instructions. This isn't terrible, but we can probably do a little better by using a better algorithm that extracts bits in parallel. Employing the technique we used to reverse bits in a register, we can come up with the following algorithm that relocates all the even bits to the L.O. word of EAX and all the odd bits to the H.O. word of EAX.

```

// Swap bits at positions (1,2), (5,6), (9,10), (13,14), (17,18),
// (21,22), (25,26), and (29, 30).

        mov( eax, edx );
        and( $9999_9999, eax );    // Mask out the bits we'll keep for now.
        mov( edx, ecx );
        shr( 1, edx );             // Move 1st bits in tuple above to the
        and( $2222_2222, ecx );    // correct position and mask out the
        and( $2222_2222, edx );    // unneeded bits.
        shl( 1, ecx );             // Move 2nd bits in tuples above.
        or( edx, ecx );            // Merge all the bits back together.
        or( ecx, eax );

// Swap bit pairs at positions ((2,3), (4,5)), ((10,11), (12,13)), etc.

        mov( eax, edx );
        and( $c3c3_c3c3, eax );    // The bits we'll leave alone.
        mov( edx, ecx );
        shr( 2, edx );
        and( $0c0c_0c0c, ecx );
        and( $0c0c_0c0c, edx );
        shl( 2, ecx );
        or( edx, ecx );
        or( ecx, eax );

// Swap nibbles at nibble positions (1,2), (5,6), (9,10), etc.

        mov( eax, edx );
        and( $f00f_f00f, eax );
        mov( edx, ecx );
        shr(4, edx );
        and( $0f0f_0f0f, ecx );
        and( $0f0f_0f0f, edx );
        shl( 4, ecx );
        or( edx, ecx );
        or( ecx, eax );

// Swap bits at positions 1 and 2.

```

```

ror( 8, eax );
xchg( al, ah );
rol( 8, eax );

```

This sequence require 30 instructions. At first blush it looks like a winner since the original loop executes 64 instructions. However, this code isn't quite as good as it looks. After all, if we're willing to write this much code, why not unroll the loop above 16 times? That sequence only requires 64 instructions. So the complexity of the previous algorithm may not gain much on instruction count. As to which sequence is faster, well, you'll have to time them to figure this out. However, the SHRD instructions are not particularly fast, neither are the instructions in the other sequence. This example does not appear here to show you a better algorithm, but rather to demonstrate that writing really tricky code doesn't always provide a big performance boost.

Extracting other bit combinations is left as an exercise for the reader.

5.13 Searching for a Bit Pattern

Another bit-related operation you may need is the ability to search for a particular bit pattern in a string of bits. For example, you might want to locate the bit index of the first occurrence of %1011 starting at some particular position in a bit string. In this section we'll explore some simple algorithms to accomplish this task.

To search for a particular bit pattern we're going to need to know four things: (1) the pattern to search for (the *pattern*), (2) the length of the pattern we're searching for, (3) the bit string that we're going to search through (the *source*), and (4) the length of the bit string to search through. The basic idea behind the search is to create a mask based on the length of the pattern and mask a copy of the source with this value. Then we can directly compare the pattern with the masked source for equality. If they are equal, we're done; if they're not equal, then increment a bit position counter, shift the source one position to the right, and try again. We repeat this operation $length(source) - length(pattern)$ times. The algorithm fails if it does not detect the bit pattern after this many attempts (since we will have exhausted all the bits in the source operand that could match the pattern's length). Here's a simple algorithm that searches for a four-bit pattern through-out the EBX register:

```

mov( 28, cl ); // 28 attempts since 32-4 = 28 (len(src)-len(pat)).
mov( %1111, ch ); // Mask for the comparison.
mov( pattern, al ); // Pattern to search for.
and( ch, al ); // Mask unnecessary bits in AL.
mov( source, ebx ); // Get the source value.
ScanLp: mov( bl, dl ); // Make a copy of the L.O. four bits of EBX
and( ch, dl ); // Mask unwanted bits.
cmp( dl, al ); // See if we match the pattern.
jz Matched;
dec( cl ); // Repeat the specified number of times.
jnz ScanLp;

```

<< If we get to this point, we failed to match the bit string >>

```

jmp Done;

```

Matched:

<< If we get to this point, we matched the bit string. We can >>
 << compute the position in the original source as 28-cl. >>

Done:

Bit string scanning is a special case of string matching. String matching is a well studied problem in Computer Science and many of the algorithms you can use for string matching are applicable to bit string matching as well. Such algorithms are a bit beyond the scope of this chapter, but to give you a preview of how this works, you compute some function (like XOR or SUB) between the pattern and the current source bits and use the result as an index into a lookup table to determine how many bits you can skip. Such algo-

gorithms let you skip several bits rather than only shifting once per each iteration of the scanning loop (as is done by the algorithm above). For more details on string scanning and their possible application to bit string matching, see the appropriate chapter in the volume on Advanced String Handling.

5.14 The HLA Standard Library Bits Module

The HLA Standard Library provides a “bits” module that provides several bit related functions, including built-in functions for many of the algorithms we’ve studied in this chapter. This section will describe these functions available in the HLA Standard Library.

```
procedure bits.cnt( b:dword ); returns( "EAX" );
```

This procedure returns the number of one bits present in the “b” parameter. It returns the count in the EAX register. To count the number of zero bits in the parameter value, invert the value of the parameter before passing it to *bits.cnt*. If you want to count the number of bits in a 16-bit operand, simply zero extend it to 32 bits prior to calling this function. Here are a couple of examples:

```
// Compute the number of bits in a 16-bit register:
```

```
    pushw( 0 );
    push( ax );
    call bits.cnt;
```

```
// If you prefer to use a higher-level syntax, try the following:
```

```
    bits.cnt( #{ pushw(0); push(ax); }# );
```

```
// Compute the number of bits in a 16-bit memory location:
```

```
    pushw( 0 );
    push( mem16 );
    bits.cnt;
```

If you want to compute the number of bits in an eight-bit operand it’s probably faster to write a simple loop that rotates all the bits in the source operand and adds the carry into the accumulating sum. Of course, if performance isn’t an issue, you can zero extend the byte to 32 bits and call the *bits.cnt* procedure.

```
procedure bits.distribute( source:dword; mask:dword; dest:dword );
    returns( "EAX" );
```

This function takes the L.O. *n* bits of *source*, where *n* is the number of “1” bits in *mask*, and inserts these bits into *dest* at the bit positions specified by the “1” bits in *mask* (i.e., the same as the distribute algorithm appearing earlier in this chapter). This function does not change the bits in *dest* that correspond to the zeros in the *mask* value. This function does not affect the value of the actual *dest* parameter, instead, it returns the new value in the EAX register.

```
procedure bits.coalesce( source:dword; mask:dword );
    returns( "EAX" );
```

This function is the converse of *bits.distribute*. It extracts all the bits in *source* whose corresponding positions in *mask* contain a one. This function coalesces (right justifies) these bits in the L.O. bit positions of the result and returns the result in EAX.

```
procedure bits.extract( var d:dword ); returns( "EAX" ); // Really a macro.
```

This function extracts the first set bit in *d* searching from bit #0 and returns the index of this bit in the EAX register; the function will also return the zero flag clear in this case. This function also clears that bit in the operand. If *d* contains zero, then this function returns the zero flag set and EAX will contain -1.

Note that HLA actually implements this function as a macro, not a procedure (see the chapter on Macros for details). This means that you can pass any double word operand as a parameter (i.e., a memory or a register operand). However, the results are undefined if you pass EAX as the parameter (since this function computes the bit number in EAX).

```
procedure bits.reverse32( d:dword ); returns( "EAX" );
procedure bits.reverse16( w:word ); returns( "AX" );
procedure bits.reverse8( b:byte ); returns( "AL" );
```

These three routines return their parameter value with the its bits reversed in the accumulator register (AL/AX/EAX). Call the routine appropriate for your data size.

```
procedure bits.merge32( even:dword; odd:dword ); returns( "EDX:EAX" );
procedure bits.merge16( even:word; odd:word ); returns( "EAX" );
procedure bits.merge8( even:byte; odd:byte ); returns( "AX" );
```

These routines merge two streams of bits to produce a value whose size is the combination of the two parameters. The bits from the “even” parameter occupy the even bits in the result, the bits from the “odd” parameter occupy the odd bits in the result. Notice that these functions return 16, 32, or 64 bits based on byte, word, and double word parameter values.

```
procedure bits.nibbles32( d:dword ); returns( "EDX:EAX" );
procedure bits.nibbles16( w:word ); returns( "EAX" );
procedure bits.nibbles8( b:byte ); returns( "AX" );
```

These routines extract each nibble from the parameter and place those nibbles into individual bytes. The *bits.nibbles8* function extracts the two nibbles from the *b* parameter and places the L.O. nibble in AL and the H.O. nibble in AH. The *bits.nibbles16* function extracts the four nibbles in *w* and places them in each of the four bytes of EAX. You can use the BSWAP or ROx instructions to gain access to the nibbles in the H.O. word of EAX. The *bits.nibbles32* function extracts the eight nibbles in EAX and distributes them through the eight bytes in EDX:EAX. Nibble zero winds up in AL and nibble seven winds up in the H.O. byte of EDX. Again, you can use BSWAP or the rotate instructions to access the upper bytes of EAX and EDX.

5.15 Putting It All Together

Bit manipulation is one area where assembly language really shines. Not only is bit manipulation far more efficient in assembly language than in high level languages, but it's often easier as well. Although the need to manipulate bits is not an everyday requirement, bit manipulation is still a very important problem area. In this chapter we've explored several ways to manipulate data as bits. Although this chapter only begins to cover the possibilities, it should give you some ideas for developing your own bit manipulation algorithms for use in your applications.

The String Instructions

Chapter Six

6.1 Chapter Overview

A string is a collection of objects stored in contiguous memory locations. Strings are usually arrays of bytes, words, or (on 80386 and later processors) double words. The 80x86 microprocessor family supports several instructions specifically designed to cope with strings. This chapter explores some of the uses of these string instructions.

The 80x86 CPUs can process three types of strings: byte strings, word strings, and double word strings. They can move strings, compare strings, search for a specific value within a string, initialize a string to a fixed value, and do other primitive operations on strings. The 80x86's string instructions are also useful for manipulating arrays, tables, and records. You can easily assign or compare such data structures using the string instructions. Using string instructions may speed up your array manipulation code considerably.

6.2 The 80x86 String Instructions

All members of the 80x86 family support five different string instructions: `MOVSw`, `CMPSw`, `SCASw`, `LODSw`, and `STOSw`¹. ($x = B, W, \text{ or } D$ for byte, word, or double word, respectively. This text will generally drop the x suffix when talking about these string instructions in a general sense.) They are the string primitives since you can build most other string operations from these five instructions. How you use these five instructions is the topic of the next several sections.

For `MOVSw`:

```
movsb();
movsw();
movsd();
```

For `CMPSw`:

```
cmpsb(); // Note: repz is a synonym for repe
cmpsw();
cmpsd();

cmpsb(); // Note: repnz is a synonym for repne.
cmpsw();
cmpsd();
```

For `SCASw`:

```
scasb(); // Note: repz is a synonym for repe
scasw();
scasd();

scasb(); // Note: repnz is a synonym for repne.
scasw();
scasd();
```

For `STOSw`:

```
stosb();
stosw();
stosd();
```

1. The 80x86 processor support two additional string instructions, `INS` and `OUTS` which input strings of data from an input port or output strings of data to an output port. We will not consider these instructions since they are privileged instructions and you cannot execute them in a standard 32-bit OS application.

```

For LODS:
    lodsb();
    lodsw();
    lodsd();

```

6.2.1 How the String Instructions Operate

The string instructions operate on blocks (contiguous linear arrays) of memory. For example, the MOVS instruction moves a sequence of bytes from one memory location to another. The CMPS instruction compares two blocks of memory. The SCAS instruction scans a block of memory for a particular value. These string instructions often require three operands, a destination block address, a source block address, and (optionally) an element count. For example, when using the MOVS instruction to copy a string, you need a source address, a destination address, and a count (the number of string elements to move).

Unlike other instructions which operate on memory, the string instructions don't have any explicit operands. The operands for the string instructions include

- the ESI (source index) register,
- the EDI (destination index) register,
- the ECX (count) register,
- the AL/AX/EAX register, and
- the direction flag in the FLAGS register.

For example, one variant of the MOVS (move string) instruction copies a string from the source address specified by ESI to the destination address specified by EDI, of length ECX. Likewise, the CMPS instruction compares the string pointed at by ESI, of length ECX, to the string pointed at by EDI.

Not all instructions have source and destination operands (only MOVS and CMPS support them). For example, the SCAS instruction (scan a string) compares the value in the accumulator (AL, AX, or EAX) to values in memory.

6.2.2 The REP/REPE/REPZ and REPNE/REPNE Prefixes

The string instructions, by themselves, do not operate on strings of data. The MOVS instruction, for example, will move a single byte, word, or double word. When executed by itself, the MOVS instruction ignores the value in the ECX register. The repeat prefixes tell the 80x86 to do a multi-byte string operation. The syntax for the repeat prefix is:

```

For MOVS:
    rep.movsb();
    rep.movsw();
    rep.movsd();

For CMPS:
    repe.cmpsb(); // Note: repz is a synonym for repe.
    repe.cmpsw();
    repe.cmpsd();

    repne.cmpsb(); // Note: repnz is a synonym for repne.
    repne.cmpsw();
    repne.cmpsd();

For SCAS:
    repe.scasb(); // Note: repz is a synonym for repe.
    repe.scasw();
    repe.scasd();

```



```
repne.scasb(); // Note: repnz is a synonym for repne.
repne.scasw();
repne.scasd();
```

```
For STOS:
rep.stosb();
rep.stosw();
rep.stosd();
```

You don't normally use the repeat prefixes with the LODS instruction.

When specifying the repeat prefix before a string instruction, the string instruction repeats ECX times². Without the repeat prefix, the instruction operates only on a single byte, word, or double word.

You can use repeat prefixes to process entire strings with a single instruction. You can use the string instructions, without the repeat prefix, as string primitive operations to synthesize more powerful string operations.

6.2.3 The Direction Flag

Besides the ESI, EDI, ECX, and AL/AX/EAX registers, one other register controls the 80x86's string instructions – the flags register. Specifically, the *direction flag* in the flags register controls how the CPU processes strings.

If the direction flag is clear, the CPU increments ESI and EDI after operating upon each string element. For example, if the direction flag is clear, then executing MOVS will move the byte, word, or double word at ESI to EDI and will increment ESI and EDI by one, two, or four. When specifying the REP prefix before this instruction, the CPU increments ESI and EDI for each element in the string. At completion, the ESI and EDI registers will be pointing at the first item beyond the strings.

If the direction flag is set, then the 80x86 decrements ESI and EDI after processing each string element. After a repeated string operation, the ESI and EDI registers will be pointing at the first byte or word before the strings if the direction flag was set.

The direction flag may be set or cleared using the CLD (clear direction flag) and STD (set direction flag) instructions. When using these instructions inside a procedure, keep in mind that they modify the machine state. Therefore, you may need to save the direction flag during the execution of that procedure. The following example exhibits the kinds of problems you might encounter:

```
procedure Str2; nodisplay;
begin Str2;

    std();
    <Do some string operations>
    .
    .
    .
end Str2;
    .
    .
    .
    cld();
    <do some operations>
    Str2();
    <do some string operations requiring D=0>
```

2. Except for the cmps instruction which repeats *at most* the number of times specified in the cx register.

This code will not work properly. The calling code assumes that the direction flag is clear after *Str2* returns. However, this isn't true. Therefore, the string operations executed after the call to *Str2* will not function properly.

There are a couple of ways to handle this problem. The first, and probably the most obvious, is always to insert the CLD or STD instructions immediately before executing a sequence of one or more string instructions. The other alternative is to save and restore the direction flag using the PUSHFD and POPFD instructions. Using these two techniques, the code above would look like this:

Always issuing CLD or STD before a string instruction:

```
procedure Str2; nodisplay;
begin Str2;

    std();
    <Do some string operations>
    .
    .
    .
end Str2;
.
.
.
    cld();
    <do some operations>
    Str2();
    cld();
    <do some string operations requiring D=0>
```

Saving and restoring the flags register:

```
procedure Str2; nodisplay;
begin Str2;

    pushfd();
    std();
    <Do some string operations>
    .
    .
    .
    popfd();
end Str2;
.
.
.
    cld();
    <do some operations>
    Str2();
    <do some string operations requiring D=0>
```

If you use the PUSHFD and POPFD instructions to save and restore the flags register, keep in mind that you're saving and restoring all the flags. Therefore, such subroutines cannot return any information in the flags. For example, you will not be able to return an error condition in the carry flag if you use PUSHFD and POPFD.

6.2.4 The MOVS Instruction

The MOVS instruction uses the following syntax:

```
movsb()
```

```

movsw()
movsd()
rep.movsb()
rep.movsw()
rep.movsd()

```

The MOVSB (move string, bytes) instruction fetches the byte at address ESI, stores it at address EDI and then increments or decrements the ESI and EDI registers by one. If the REP prefix is present, the CPU checks ECX to see if it contains zero. If not, then it moves the byte from ESI to EDI and decrements the ECX register. This process repeats until ECX becomes zero.

The MOVSW (move string, words) instruction fetches the word at address ESI, stores it at address EDI and then increments or decrements ESI and EDI by two. If there is a REP prefix, then the CPU repeats this procedure as many times as specified in ECX.

The MOVSD instruction operates in a similar fashion on double words. Incrementing or decrementing ESI and EDI by four for each data movement.

When you use the *rep* prefix, the MOVSB instruction moves the number of bytes you specify in the ECX register. The following code segment copies 384 bytes from *CharArray1* to *CharArray2*:

```

CharArray1: byte[ 384 ];
CharArray2: byte[ 384 ];
.
.
.
cld();
lea( esi, CharArray1 );
lea( edi, CharArray2 );
mov( 384, ecx );
rep.movsb();

```

If you substitute MOVSW for MOVSB, then the code above will move 384 words (768 bytes) rather than 384 bytes:

```

WordArray1: word[ 384 ];
WordArray2: word[ 384 ];
.
.
.
cld();
lea( esi, WordArray1 );
lea( edi, WordArray2 );
mov( 384, ecx );
rep.movsw();

```

Remember, the ECX register contains the element count, not the byte count. When using the MOVSW instruction, the CPU moves the number of words specified in the ECX register. Similarly, MOVSD moves the number of double words you specify in the ECX register, not the number of bytes.

If you've set the direction flag before executing a MOVSB/MOVSW/MOVSD instruction, the CPU decrements the ESI and EDI registers after moving each string element. This means that the ESI and EDI registers must point at the end of their respective strings before issuing a MOVSB, MOVSW, or MOVSD instruction. For example,

```

CharArray1: byte[ 384 ];
CharArray2: byte[ 384 ];
.
.
.
cld();
lea( esi, CharArray1[383] );
lea( edi, CharArray2[383] );

```

```

mov( 384, ecx );
rep.movsb();

```

Although there are times when processing a string from tail to head is useful (see the CMPS description in the next section), generally you'll process strings in the forward direction since it's more straightforward to do so. There is one class of string operations where being able to process strings in both directions is absolutely mandatory: processing strings when the source and destination blocks overlap. Consider what happens in the following code:

```

CharArray1: byte;
CharArray2: byte[ 384 ];
.
.
.
cld();
lea( esi, CharArray1 );
lea( edi, CharArray2 );
mov( 384, ecx );
rep.movsb();

```

This sequence of instructions treats *CharArray1* and *CharArray2* as a pair of 384 byte strings. However, the last 383 bytes in the *CharArray1* array overlap the first 383 bytes in the *CharArray2* array. Let's trace the operation of this code byte by byte.

When the CPU executes the MOVSB instruction, it copies the byte at ESI (*CharArray1*) to the byte pointed at by EDI (*CharArray2*). Then it increments ESI and EDI, decrements ECX by one, and repeats this process. Now the ESI register points at *CharArray1+1* (which is the address of *CharArray2*) and the EDI register points at *CharArray2+1*. The MOVSB instruction copies the byte pointed at by ESI to the byte pointed at by EDI. However, this is the byte originally copied from location *CharArray1*. So the MOVSB instruction copies the value originally in location *CharArray1* to both locations *CharArray2* and *CharArray2+1*. Again, the CPU increments ESI and EDI, decrements ECX, and repeats this operation. Now the movsb instruction copies the byte from location *CharArray1+2* (*CharArray2+1*) to location *CharArray2+2*. But once again, this is the value that originally appeared in location *CharArray1*. Each repetition of the loop copies the next element in *CharArray1[0]* to the next available location in the *CharArray2* array. Pictorially, it looks something like that shown in Figure 6.1.

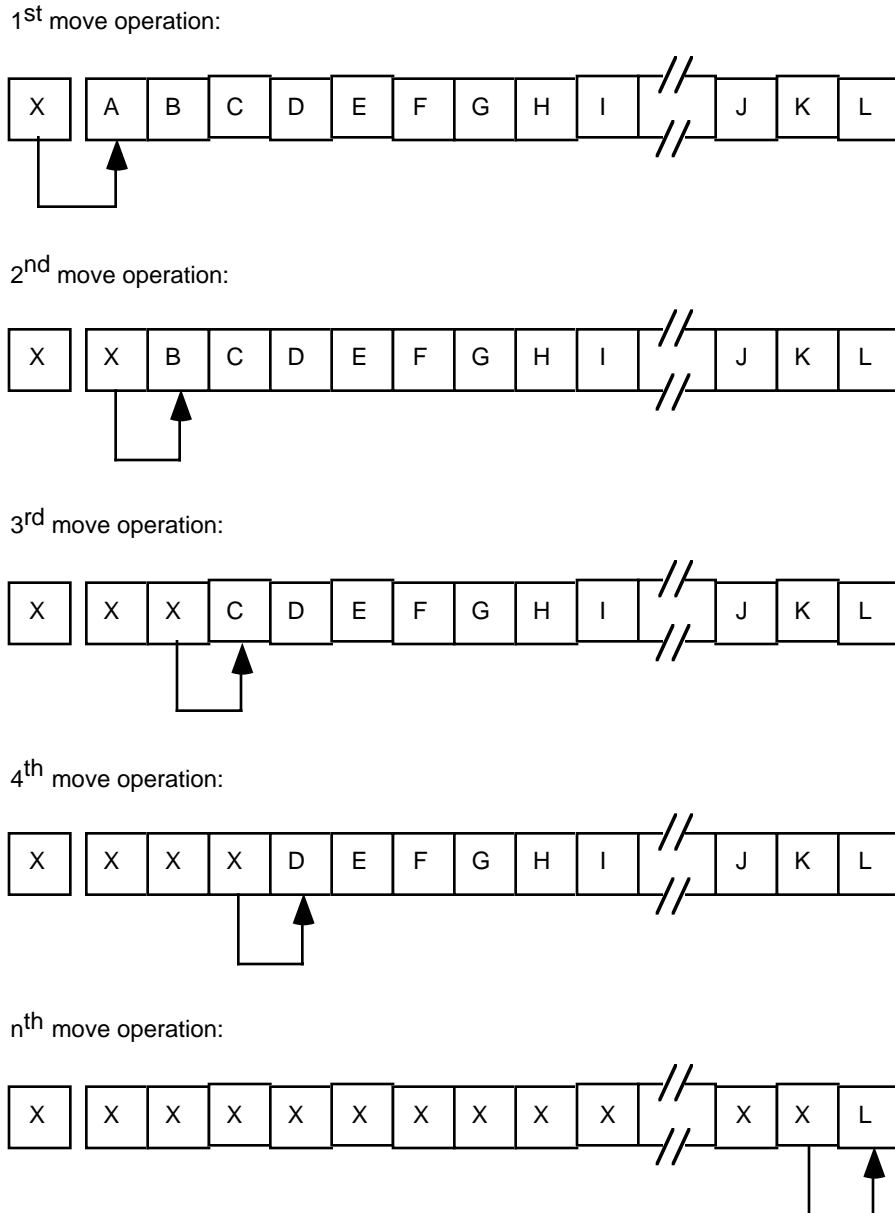


Figure 6.1 Copying Data Between Two Overlapping Arrays (forward direction)

The end result is that the MOVSB instruction replicates X throughout the string. The MOVSB instruction copies the source operand into the memory location which will become the source operand for the very next move operation, which causes the replication.

If you really want to move one array into another when they overlap, you should move each element of the source string to the destination string starting at the end of the two strings as shown in Figure 6.2.

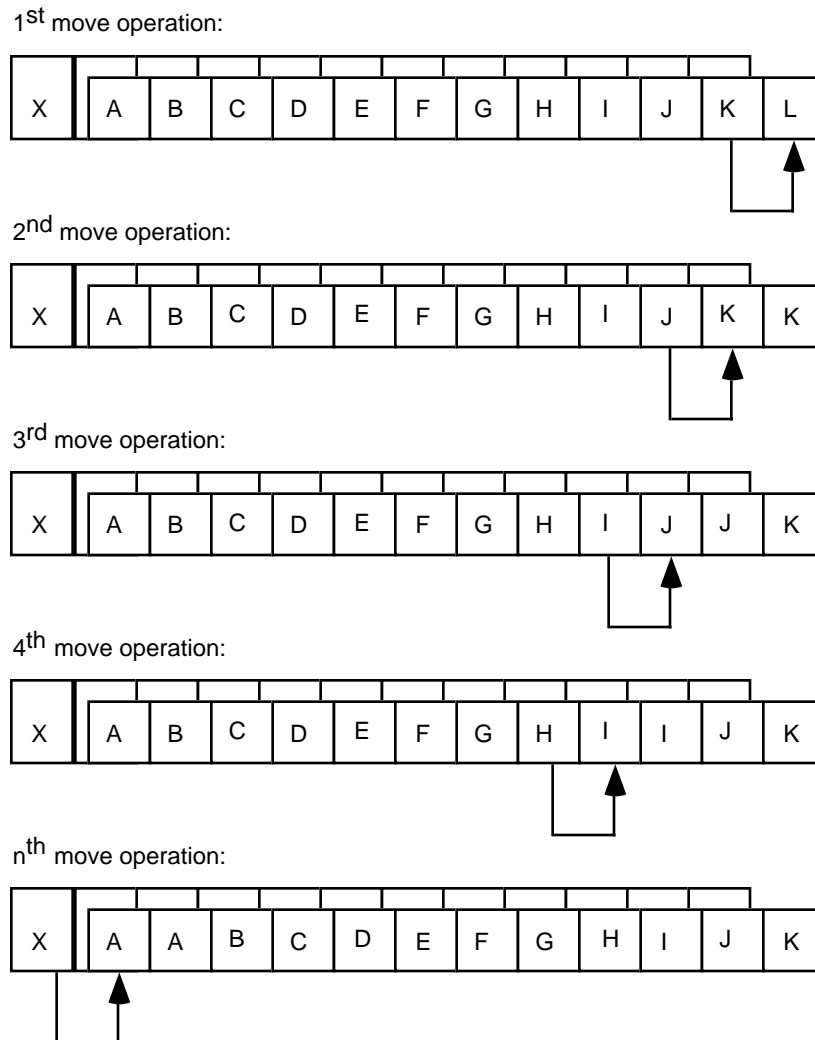


Figure 6.2 Using a Backwards Copy to Copy Data in Overlapping Arrays

Setting the direction flag and pointing ESI and EDI at the end of the strings will allow you to (correctly) move one string to another when the two strings overlap and the source string begins at a lower address than the destination string. If the two strings overlap and the source string begins at a higher address than the destination string, then clear the direction flag and point ESI and EDI at the beginning of the two strings.

If the two strings do not overlap, then you can use either technique to move the strings around in memory. Generally, operating with the direction flag clear is the easiest, so that makes the most sense in this case.

You shouldn't use the `MOVSB` instruction to fill an array with a single byte, word, or double word value. Another string instruction, `STOS`, is much better for this purpose. However, for arrays whose elements are larger than four bytes, you can use the `MOVSD` instruction to initialize the entire array to the content of the first element.

The `MOVSD` instruction is generally more efficient when copying double words than it is copying bytes or words. In fact, it typically takes the same amount of time to copy a byte using `MOVSB` as it does to copy a double word using `MOVSD`³. Therefore, if you are moving a large number of bytes from one array to another, the copy operation will be faster if you can use the `MOVSD` instruction rather than the `MOVSB`

3. This is true for `MOVSW`, as well.

instruction. Of course, if the number of bytes you wish to move is an even multiple of four, this is a trivial change; just divide the number of bytes to copy by four, load this value into ECX, and then use the MOVSB instruction. If the number of bytes is not evenly divisible by four, then you can use the MOVSD instruction to copy all but the last one, two, or three bytes of the array (that is, the remainder after you divide the byte count by four). For example, if you want to efficiently move 4099 bytes, you can do so with the following instruction sequence:

```
lea( esi, Source );
lea( edi, Destination );
mov( 1024, ecx );      // Copy 1024 dwords = 4096 bytes
rep.movsd();
movsw();              // Copy bytes 4097 and 4098.
movsb();              // Copy the last byte.
```

Using this technique to copy data never requires more than three MOVSB instructions since you can copy one, two, or three bytes with no more than two MOVSB and MOVSW instructions. The scheme above is most efficient if the two arrays are aligned on double word boundaries. If not, you might want to move the MOVSB or MOVSW instruction (or both) before the MOVSD so that the MOVSD instruction works with dword-aligned data (see Chapter Three for an explanation of the performance benefits of double word aligned data).

If you do not know the size of the block you are copying until the program executes, you can still use code like the following to improve the performance of a block move of bytes:

```
lea( esi, Source );
lea( edi, Dest );
mov( Length, ecx );
shr( 2, ecx );        // divide by four.
if( @nz ) then       // Only execute MOVSD if four or more bytes.

    rep.movsd();      // Copy the dwords.

endif;
mov( Length, ecx );
and( %11, ecx );     // Compute (Length mod 4).
if( @nz ) then       // Only execute MOVSB if #bytes/4 <> 0.

    rep.movsb();      // Copy the remaining one, two, or three bytes.

endif;
```

On most computer systems, the MOVSD instruction provides about the fastest way to copy bulk data from one location to another. While there are, arguably, faster ways to copy the data on certain CPUs, ultimately the memory bus performance is the limiting factor and the CPUs are generally much faster than the memory bus. Therefore, unless you have a special system, writing fancy code to improve memory to memory transfers is probably a waste of time. Also note that Intel has improved the performance of the MOVSB instructions on later processors so that MOVSB operates almost as efficiently as MOVSW and MOVSD when copying the same number of bytes. Therefore, when working on a later x86 processor, it may be more efficient to simply use MOVSB to copy the specified number of bytes rather than go through all the complexity outlined above.

6.2.5 The CMPS Instruction

The CMPS instruction compares two strings. The CPU compares the string referenced by EDI to the string pointed at by ESI. ECX contains the length of the two strings (when using the REPE or REPNE prefix). Like the MOVS instruction, HLA allows several different forms of this instruction:

```
cmpsb();
cmpsw();
```

```

cmpsd( );

repe.cmpsb( );
repe.cmpsw( );
repe.cmpsd( );

repne.cmpsb( );
repne.cmpsw( );
repne.cmpsd( );

```

Like the MOVSB instruction you specify the actual operand addresses in the ESI and EDI registers.

Without a repeat prefix, the CMPS instruction subtracts the value at location EDI from the value at ESI and updates the flags. Other than updating the flags, the CPU doesn't use the difference produced by this subtraction. After comparing the two locations, CMPS increments or decrements the ESI and EDI registers by one, two, or four (for CMPSB/CMPSW/CMPSD, respectively). CMPS increments the ESI and EDI registers if the direction flag is clear and decrements them otherwise.

Of course, you will not tap the real power of the CMPS instruction using it to compare single bytes, words, or double words in memory. This instruction shines when you use it to compare whole strings. With CMPS, you can compare consecutive elements in a string until you find a match or until consecutive elements do not match.

To compare two strings to see if they are equal or not equal, you must compare corresponding elements in a string until they don't match. Consider the following strings:

```

"String1"
"String1"

```

The only way to determine that these two strings are equal is to compare each character in the first string to the corresponding character in the second. After all, the second string could have been "String2" which definitely is not equal to "String1". Of course, once you encounter a character in the destination string which doesn't equal the corresponding character in the source string, the comparison can stop. You needn't compare any other characters in the two strings.

The REPE prefix accomplishes this operation. It will compare successive elements in a string as long as they are equal and ECX is greater than zero. We could compare the two strings above using the following 80x86 assembly language code:

```

cld( );
mov( AdrsString1, esi );
mov( AdrsString2, edi );
mov( 7, ecx );
repe.cmpsb( );

```

After the execution of the CMPSB instruction, you can test the flags using the standard conditional jump instructions. This lets you check for equality, inequality, less than, greater than, etc.

Character strings are usually compared using *lexicographical ordering*. In lexicographical ordering, the least significant element of a string carries the most weight. This is in direct contrast to standard integer comparisons where the most significant portion of the number carries the most weight. Furthermore, the length of a string affects the comparison only if the two strings are identical up to the length of the shorter string. For example, "Zebra" is less than "Zebras", because it is the shorter of the two strings, however, "Zebra" is greater than "AAAAAAAHAH!" even though it is shorter. Lexicographical comparisons compare corresponding elements until encountering a character which doesn't match, or until encountering the end of the shorter string. If a pair of corresponding characters do not match, then this algorithm compares the two strings based on that single character. If the two strings match up to the length of the shorter string, we must compare their length. The two strings are equal if and only if their lengths are equal and each corresponding pair of characters in the two strings is identical. Lexicographical ordering is the standard alphabetical ordering you've grown up with.

For character strings, use the CMPS instruction in the following manner:

- The direction flag must be cleared before comparing the strings.
- Use the CMPSB instruction to compare the strings on a byte by byte basis. Even if the strings contain an even number of characters, you cannot use the CMPSW or CMPSD instructions. They do not compare strings in lexicographical order.
- You must load the ECX register with the length of the smaller string.
- Use the REPE prefix.
- The ESI and EDI registers must point at the very first character in the two strings you want to compare.

After the execution of the CMPS instruction, if the two strings were equal, their lengths must be compared in order to finish the comparison. The following code compares a couple of character strings:

```

mov( AdrsStr1, esi );
mov( AdrsStr2, edi );
mov( LengthSrc, ecx );
if( ecx > LengthDest ) then // Put the length of the shorter string in ECX.

    mov( LengthDest, ecx );

endif;
repe.cmpsb();
if( @z ) then // If equal to the length of the shorter string, cmp lengths.

    mov( LengthSrc, ecx );
    cmp( ecx, LengthDest );

endif;

```

If you're using bytes to hold the string lengths, you should adjust this code appropriately (i.e., use a MOVZX instruction to load the lengths into ECX). Of course, HLA strings use a double word to hold the current length value, so this isn't an issue when using HLA strings.

You can also use the CMPS instruction to compare multi-word integer values (that is, extended precision integer values). Because of the amount of setup required for a string comparison, this isn't practical for integer values less than six or eight double words in length, but for large integer values, it's an excellent way to compare such values. Unlike character strings, we cannot compare integer strings using a lexicographical ordering. When comparing strings, we compare the characters from the least significant byte to the most significant byte. When comparing integers, we must compare the values from the most significant byte (or word/double word) down to the least significant byte, word or double word. So, to compare two 32-byte (256-bit) integer values, use the following code on the 80x86:

```

std();
lea( esi, SourceInteger[28] );
lea( edi, DestInteger[28] );
mov( 8, ecx );
rep.cmpsd();

```

This code compares the integers from their most significant word down to the least significant word. The CMPSD instruction finishes when the two values are unequal or upon decrementing ECX to zero (implying that the two values are equal). Once again, the flags provide the result of the comparison.

The REPNE prefix will instruct the CMPS instruction to compare successive string elements as long as they do not match. The 80x86 flags are of little use after the execution of this instruction. Either the ECX register is zero (in which case the two strings are totally different), or it contains the number of elements compared in the two strings until a match. While this form of the CMPS instruction isn't particularly useful for comparing strings, it is useful for locating the first pair of matching items in a couple of byte, word, or double word arrays. In general, though, you'll rarely use the REPNE prefix with CMPS.

One last thing to keep in mind with using the CMPS instruction – the value in the ECX register determines the number of elements to process, not the number of bytes. Therefore, when using CMPSW, ECX

specifies the number of words to compare. This, of course, is twice the number of bytes to compare. Likewise, for CMPSD, ECX contains the number of double words to process.

6.2.6 The SCAS Instruction

The CMPS instruction compares two strings against one another. You do not use it to search for a particular element within a string. For example, you could not use the CMPS instruction to quickly scan for a zero throughout some other string. You can use the SCAS (scan string) instruction for this task.

Unlike the MOVS and CMPS instructions, the SCAS instruction only requires a destination string (pointed at by EDI) rather than both a source and destination string. The source operand is the value in the AL (SCASB), AX (SCASW), or EAX (SCASD) register. The SCAS instruction compares the value in the accumulator (AL, AX, or EAX) against the value pointed at by EDI and then increments (or decrements) EDI by one, two, or four. The CPU sets the flags according to the result of the comparison. While this might be useful on occasion, SCAS is a lot more useful when using the REPE and REPNE prefixes.

With the REPE prefix (repeat while equal), SCAS scans the string searching for an element which does not match the value in the accumulator. When using the REPNE prefix (repeat while not equal), SCAS scans the string searching for the first string element which is equal to the value in the accumulator.

You're probably wondering "why do these prefixes do exactly the opposite of what they ought to do?" The paragraphs above haven't quite phrased the operation of the SCAS instruction properly. When using the REPE prefix with SCAS, the 80x86 scans through the string while the value in the accumulator is equal to the string operand. This is equivalent to searching through the string for the first element which does not match the value in the accumulator. The SCAS instruction with REPNE scans through the string while the accumulator is not equal to the string operand. Of course, this form searches for the first value in the string which matches the value in the accumulator register. The SCAS instructions take the following forms:

```
scasb()
scasw()
scasd()

repe.scasb()
repe.scasw()
repe.scasd()

repne.scasb()
repne.scasw()
repne.scasd()
```

Like the CMPS and MOVS instructions, the value in the ECX register specifies the number of elements to process, not bytes, when using a repeat prefix.

6.2.7 The STOS Instruction

The STOS instruction stores the value in the accumulator at the location specified by EDI. After storing the value, the CPU increments or decrements EDI depending upon the state of the direction flag. Although the STOS instruction has many uses, its primary use is to initialize arrays and strings to a constant value. For example, if you have a 256-byte array you want to clear out with zeros, use the following code:

```
cld();
lea( edi, DestArray );
mov( 64, ecx );           // 64 double words = 256 bytes.
xor( eax, eax );         // Zero out EAX.
rep.stosd();
```

This code writes 64 double words rather than 256 bytes because a single STOSD operation is faster than four STOSB operations.

The STOS instructions take four forms. They are

```
stosb();
stosw();
stosd();

rep.stosb();
rep.stosw();
rep.stosd();
```

The STOSB instruction stores the value in the AL register into the specified memory location(s), the STOSW instruction stores the AX register into the specified memory location(s) and the STOSD instruction stores EAX into the specified location(s).

Keep in mind that the STOS instruction is useful only for initializing a byte, word, or double word array to a constant value. If you need to initialize an array to different values, you cannot use the STOS instruction. See the exercises for additional details.

6.2.8 The LODS Instruction

The LODS instruction is unique among the string instructions. You will probably never use a repeat prefix with this instruction. The LODS instruction copies the byte, word, or double word pointed at by ESI into the AL, AX, or EAX register, after which it increments or decrements the ESI register by one, two, or four. Repeating this instruction via the repeat prefix would serve no purpose whatsoever since the accumulator register will be overwritten each time the LODS instruction repeats. At the end of the repeat operation, the accumulator will contain the last value read from memory.

Instead, use the LODS instruction to fetch bytes (LODSB), words (LODSW), or double words (LODSD) from memory for further processing. By using the STOS instruction, you can synthesize powerful string operations.

Like the STOS instruction, the LODS instructions take four forms:

```
lodsb();
lodsw();
lodsd();

rep.lodsb();
rep.lodsw();
rep.lodsd();
```

As mentioned earlier, you'll rarely, if ever, use the REP prefixes with these instructions⁴. The 80x86 increments or decrements ESI by one, two, or four depending on the direction flag and whether you're using the LODSB, LODSW, or LODSD instruction.

6.2.9 Building Complex String Functions from LODS and STOS

The 80x86 supports only five different string instructions: MOVSB, CMPSB, SCASB, LODSB, and STOSB⁵. These certainly aren't the only string operations you'll ever want to use. However, you can use the LODS and STOS instructions to easily generate any particular string operation you like. For example, suppose you wanted a string operation that converts all the upper case characters in a string to lower case. You could use the following code:

```
mov( StringAddress, esi ); // Load string address into ESI.
mov( esi, edi );          // Also point EDI here.
```

4. They appear here simply because they are allowed. They're not very useful, but they are allowed.

5. Not counting INS and OUTS which we're ignoring here.

```
mov( (type str.strrec [esi].length, ecx );

repeat

    lodsb();           // Get the next character in the string.
    if( al in 'A'..'Z' ) then

        or( $20, al ); // Convert upper case character to lower case.

    endif;
    stosb();           // Store converted character back into string.
    dec( ecx );

until( ecx == 0 );
```

Since the LODS and STOS instructions use the accumulator as an intermediary, you can use any accumulator operation to quickly manipulate string elements.

6.3 Putting It All Together

In this chapter we took a quick look at the 80x86's string instructions. We studied their implementation and saw how to use them. These instructions are quite useful for synthesizing character set functions (see the source code for the HLA Standard Library string module for examples). We also saw how to use these instructions for non-character string purpose such as moving large blocks of memory (i.e., assigning one array to another) and comparing large integer values. For more information on the use of these instructions, please see the volume on Advanced String Handling.

The HLA Compile-Time Language

Chapter Seven

7.1 Chapter Overview

Now we come to the fun part. For the past nine chapters this text has been molding and conforming you to deal with the HLA language and assembly language programming in general. In this chapter you get to turn the tables; you'll learn how to force HLA to conform to your desires. This chapter will teach you how to extend the HLA language using HLA's *compile-time language*. By the time you are through with this chapter, you should have a healthy appreciation for the power of the HLA compile-time language. You will be able to write short compile-time programs. You will also be able to add new statements, of your own choosing, to the HLA language.

7.2 Introduction to the Compile-Time Language (CTL)

HLA is actually two languages rolled into a single program. The *run-time language* is the standard 80x86/HLA assembly language you've been reading about in all the past chapters. This is called the run-time language because the programs you write execute when you run the executable file. HLA contains an interpreter for a second language, the HLA Compile-Time Language (or CTL) that executes programs while HLA is compiling a program. The source code for the CTL program is embedded in an HLA assembly language source file; that is, HLA source files contain instructions for both the HLA CTL and the run-time program. HLA executes the CTL program during compilation. Once HLA completes compilation, the CTL program terminates; the CTL application is not a part of the run-time executable that HLA emits, although the CTL application can *write* part of the run-time program for you and, in fact, this is the major purpose of the CTL.

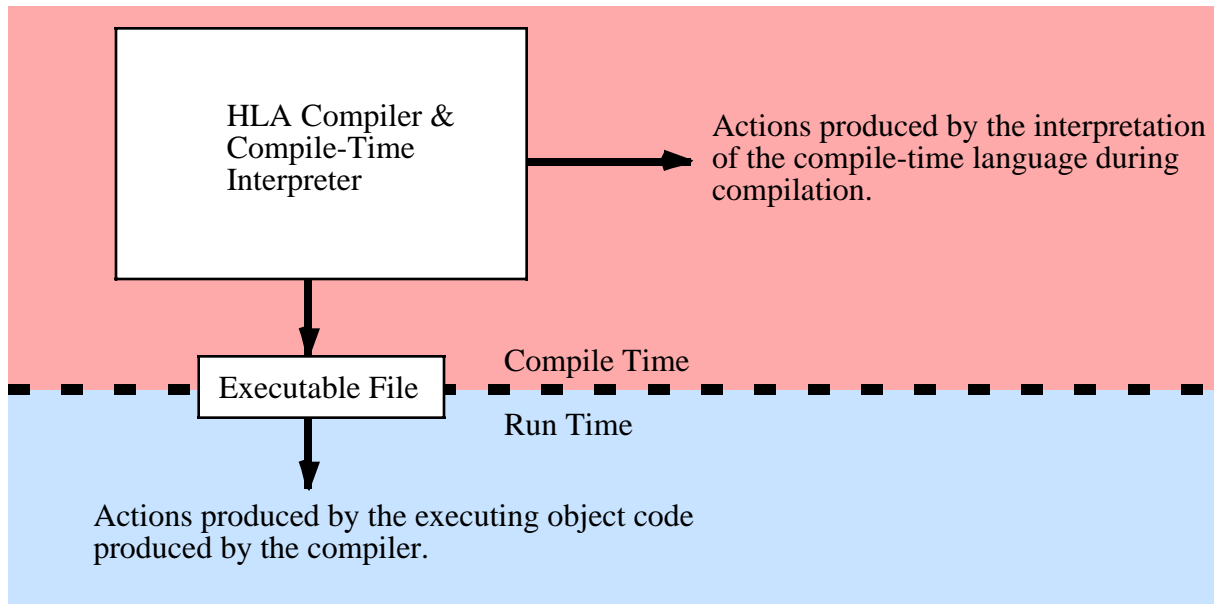


Figure 7.1 Compile-Time vs. Run-Time Execution

It may seem confusing to have two separate languages built into the same compiler. Perhaps you're even questioning why anyone would need a compile time language. To understand the benefits of a compile time language, consider the following statement that you should be very comfortable with at this point:

```
stdout.put( "i32=", i32, " strVar=", strVar, " charVar=", charVar, nl );
```

This statement is neither a statement in the HLA language nor a call to some HLA Standard Library procedure. Instead, *stdout.put* is actually a statement in a CTL application provided by the HLA Standard Library. The *stdout.put* "application" processes a list of objects (the parameter list) and makes calls to various other Standard Library procedures; it chooses the procedure to call based on the type of the object it is currently processing. For example, the *stdout.put* "application" above will emit the following statements to the run-time executable:

```
stdout.puts( "i32=" );
stdout.puti32( i32 );
stdout.puts( " strVar=" );
stdout.puts( strVar );
stdout.puts( " charVar=" );
stdout.putc( charVar );
stdout.newln();
```

Clearly the *stdout.put* statement is much easier to read and write than the sequence of statements that *stdout.put* emits in response to its parameter list. This is one of the more powerful capabilities of the HLA programming language: the ability to modify the language to simplify common programming tasks. Printing lots of different data objects in a sequential fashion is a common task; the *stdout.put* "application" greatly simplifies this process.

The HLA Standard Library is *loaded* with lots of HLA CTL examples. In addition to standard library usage, the HLA CTL is quite adept at handling "one-off" or "one-use" applications. A classic example is filling in the data for a lookup table. An earlier chapter in this text noted that it is possible to construct look-up tables using the HLA CTL (see "Tables" on page 647 and "Generating Tables" on page 651). Not only is this possible, but it is often far less work to use the HLA CTL to construct these look-up tables. This chapter abounds with examples of exactly this application of the CTL.

Although the CTL itself is relatively inefficient and you would not use it to write end-user applications, it does maximize the use of that one precious commodity of which there is so little available: your time. By learning how to use the HLA CTL, and applying it properly, you can develop assembly language applications as rapidly as high level language applications (even faster since HLA's CTL lets you create *very* high level language constructs).

7.3 The #PRINT and #ERROR Statements

Chapter One of this textbook began with the typical first program most people write when learning a new language; the "Hello World" program. It is only fitting for this chapter to present that same program when discussing the second language of this text. So here it is, the basic "Hello World" program written in the HLA compile time language:

```

program ctlHelloWorld;
begin ctlHelloWorld;

    #print( "Hello, World of HLA/CTL" )

end ctlHelloWorld;

```

Program 7.1 The CTL "Hello World" Program

The only CTL statement in this program is the "#print" statement. The remaining lines are needed just to keep the compiler happy (though we could have reduced the overhead to two lines by using a UNIT rather than a PROGRAM declaration).

The #PRINT statement displays the textual representation of its argument list during the compilation of an HLA program. Therefore, if you compile the program above with the command "hla ctlHW.hla" the HLA compiler will immediately print, before returning control to the command line, the text:

```

Hello, World of HLA/CTL

```

Note that there is a big difference between the following two statements in an HLA source file:

```

#print( "Hello World" )
stdout.puts( "Hello World" nl );

```

The first statement prints "Hello World" (and a newline) during the compilation process. This first statement does not have any effect on the executable program. The second line doesn't affect the compilation process (other than the emission of code to the executable file). However, when you run the executable file, the second statement prints the string "Hello World" followed by a new line sequence.

The HLA/CTL #PRINT statement uses the following basic syntax:

```

#print( list_of_comma_separated_constants )

```

Note that a semicolon does not terminate this statement. Semicolons terminate run-time statements, they generally do not terminate compile-time statements (there is one big exception, as you will see a little later).

The #PRINT statement must have at least one operand; if multiple operands appear in the parameter list, you must separate each operand with a comma (just like *stdout.put*). If a particular operand is not a string constant, HLA will translate that constant to its corresponding string representation and print that string. Example:

```

#print( "A string Constant ", 45, ' ', 54.9, ' ', true )

```

You may specify named symbolic constants and constant expressions. However, all #PRINT operands must be constants (either literal constants or constants you define in the CONST or VAL sections) and those constants must be defined before you use them in the #PRINT statement. Example:

```
const
  pi := 3.14159;
  charConst := 'c';

#print( "PI = ", pi, " CharVal=", CharConst )
```

The HLA #PRINT statement is particularly invaluable for debugging CTL programs (since there is no debugger available for CTL code). This statement is also useful for displaying the progress of the compilation and displaying assumptions and default actions that take place during compilation. Other than displaying the text associated with the #PRINT parameter list, the #PRINT statement does not have any affect on the compilation of the program.

The #ERROR statement allows a single string constant operand. Like #PRINT this statement will display the string to the console during compilation. However, the #ERROR statement treats the string as an error message and displays the string as part of an HLA error diagnostic. Further, the #ERROR statement increments the error count and this will cause HLA to stop the compilation (without assembly or linking) at the conclusion of the source file. You would normally use the #ERROR statement to display an error message during compilation if your CTL code discovers something that prevents it from creating valid code. Example:

```
#error( "Statement must have exactly one operand" )
```

Like the #PRINT statement, the #ERROR statement does not end with a semicolon. Although #ERROR only allows a string operand, it's very easy to print other values by using the string (constant) concatenation operator and several of the HLA built-in compile-time functions (see "Compile-Time Constants and Variables" on page 952 and "Compile-Time Functions" on page 956) for more details).

7.4 Compile-Time Constants and Variables

Just as the run-time language supports constants and variables, so does the compile-time language. You declare compile-time constants in the CONST section, the same as for the run-time language. You declare compile-time variables in the VAL section. Objects you declare in the VAL section are constants as far as the run-time language is concerned, but remember that you can change the value of an object you declare in the VAL section throughout the source file. Hence the term "compile-time variable." See "HLA Constant and Value Declarations" on page 397 for more details.

The CTL assignment statement ("?") computes the value of the constant expression to the right of the assignment operator (":=") and stores the result into the VAL object name appearing immediately to the left of the assignment operator¹. The following example is a rework of the example above; this example, however, may appear anywhere in your HLA source file, not just in the VAL section of the program.

```
?ConstToPrint := 25;
#print( "ConstToPrint = ", ConstToPrint )
?ConstToPrint := ConstToPrint + 5;
#print( "Now ConstToPrint = ", ConstToPrint )
```

Note that HLA's CTL ignores the distinction between the different sizes of numeric objects. HLA always reserves storage for the largest possible object of a given type, so HLA merges the following types:

```
byte, word, dword -> dword
uns8, uns16, uns32 -> uns32
int8, int16, int32 -> int32
```

1. If the identifier to the left of the assignment operator is undefined, HLA will automatically declare this object at the current scope level.


```
real32, real64, real80 -> real80
```

For most practical applications of the CTL, this shouldn't make a difference in the operation of the program.

7.5 Compile-Time Expressions and Operators

As the previous section states, the HLA CTL supports constant expressions in the CTL assignment statement. Unlike the run-time language (where you have to translate algebraic notation into a sequence of machine instructions), the HLA CTL allows a full set of arithmetic operations using familiar expression syntax. This gives the HLA CTL considerable power, especially when combined with the built-in compile-time functions the next section discusses.

HLA's CTL supports the following operators in compile-time expressions:

Table 1: Compile-Time Operators

Operator(s)	Operand Types ^a	Description
- (unary)	numeric	Negates the specific numeric value (int, uns, real).
	cset	Returns the complement of the specified character set.
! (unary)	integer	Inverts all the bits in the operand (bitwise not).
	boolean	Boolean NOT of the operand.
*	numericL * numericR	Multiplies the two operands.
	csetL * csetR	Computes the intersection of the two sets
div	integerL div integerR	Computes the integer quotient of the two integer (int/uns/dword) operands.
mod	integerL mod integerR	Computes the remainder of the division of the two integer (int/uns/dword) operands.
/	numericL / numericR	Computes the real quotient of the two numeric operands. Returns a real result even if both operands are integers.
<<	integerL << integerR	Shifts integerL operand to the left the number of bits specified by the integerR operand.
>>	integerL >> integerR	Shifts integerL operand to the right the number of bits specified by the integerR operand.
+	numericL + numericR	Adds the two numeric operands.
	csetL + csetR	Computes the union of the two sets.
	strL + strR	Concatenates the two strings.

Table 1: Compile-Time Operators

Operator(s)	Operand Types ^a	Description
-	numericL - numericR	Computes the difference between numericL and numericR.
	csetL - csetR	Computes the set difference of csetL-csetR.
= or ==	numericL = numericR	Returns true if the two operands have the same value.
	csetL = csetR	Returns true if the two sets are equal.
	strL = strR	Returns true if the two strings/chars are equal.
	typeL = typeR	Returns true if the two values are equal. They must be the same type.
<> or !=	typeL <> typeR (Same as =)	Returns false if the two (compatible) operands are not equal to one another.
<	numericL < numericR	Returns true if numericL is less than numericR.
	csetL < csetR	Returns true if csetL is a proper subset of csetR.
	strL < strR	Returns true if strL is less than strR
	booleanL < booleanR	Returns true if left operand is less than right operand (note: false < true).
	enumL < enumR	Returns true if enumL appears in the same enum list as enumR and enumL appears first.
<=	Same as <	Returns true if the left operand is less than or equal to the right operand. For character sets, this means that the left operand is a subset of the right operand.
>	Same as <	Returns true if the left operand is greater than the right operand. For character sets, this means that the left operand is a proper superset of the right operand.
>=	Same as <=	Returns true if the left operand is greater than or equal to the right operand. For character sets, this means that the left operand is a superset of the right operand.
&	integerL & integerR	Computes the bitwise AND of the two operands.
	booleanL & booleanR	Computes the logical AND of the two operands.

Table 1: Compile-Time Operators

Operator(s)	Operand Types ^a	Description
	integerL integerR	Computes the bitwise OR of the two operands.
	booleanL booleanR	Computes the logical OR of the two operands.
^	integerL ^ integerR	Computes the bitwise XOR of the two operands.
	booleanL ^ booleanR	Computes the logical XOR of the two operands. Note that this is equivalent to "booleanL <> booleanR".
in	charL in csetR	Returns true if charL is a member of csetR.

a. Numeric is {intXX, unsXX, byte, word, dword, and realXX} values. Cset is a character set operand. Type integer is { intXX, unsXX, byte, word, dword }. Type str is any string or character value. "TYPE" indicates an arbitrary HLA type. Other types specify an explicit HLA data type.

Table 2: Operator Precedence and Associativity

Associativity	Precedence (Highest to Lowest)	Operator
Right-to-left	6	! (unary)
		- (unary)
Left to right	5	*
		div
		mod
		/
		>>
		<<
Left to right	4	+
		-
Left to right	3	= or ==
		<> or !=
		<
		<=
		>
		>=

Table 2: Operator Precedence and Associativity

Associativity	Precedence (Highest to Lowest)	Operator
Left to right	2	&
		^
Nonassociative	1	in

Of course, you can always override the default precedence and associativity of an operator by using parentheses in an expression.

7.6 Compile-Time Functions

HLA provides a wide range of compile-time functions you can use. These functions compute values during compilation the same way a high level language function computes values at run-time. The HLA compile-time language includes a wide variety of numeric, string, and symbol table functions that help you write sophisticated compile-time programs.

Most of the names of the built-in compile-time functions begin with the special symbol "@" and have names like `@sin` or `@length`. The use of these special identifiers prevents conflicts with common names you might want to use in your own programs (like `length`). The remaining compile-time functions (those that do not begin with "@") are typically data conversion functions that use type names like `int8` and `real64`. You can even create your own compile-time functions using macros (see "Macros" on page 969).

HLA organizes the compile-time functions into various classes depending on the type of operation. For example, there are functions that convert constants from one form to another (e.g., string to integer conversion), there are many useful string functions, and HLA provides a full set of compile-time numeric functions.

The complete list of HLA compile-time functions is too lengthy to present here. Instead, a complete description of each of the compile-time objects and functions appears in Appendix H (see "HLA Compile-Time Functions" on page 1493); this section will highlight a few of the functions in order to demonstrate their use. Later sections in this chapter, as well as future chapters, will make extensive use of the various compile-time functions.

Perhaps the most important concept to understand about the compile-time functions is that they are equivalent to constants in your assembly language code (i.e., the run-time program). For example, the compile-time function invocation `"@sin(3.1415265358979328)"` is roughly equivalent to specifying "0.0" at that point in your program². A function invocation like `"@sin(x)"` is legal only if `x` is a constant with a previous declaration at the point of the function call in the source file. In particular, `x` cannot be a run-time variable or other object whose value exists at run-time rather than compile-time. Since HLA replaces compile-time function calls with their constant result, you may ask why you should even bother with compile time functions. After all, it's probably more convenient to type "0.0" than it is to type `"@sin(3.1415265358979328)"` in your program. However, compile-time functions are really handy for generating lookup tables (see "Generating Tables" on page 651) and other mathematical results that may

2. Actually, since `@sin`'s parameter in this example is not exactly π , you will get a small positive number instead of zero as the function result, but in theory you should get zero.

change whenever you change a `CONST` value in your program. Later sections in this chapter will explore these ideas farther.

7.6.1 Type Conversion Compile-time Functions

One set of commonly used compile-time functions are the type conversion functions. These functions take a single parameter of one type and convert that information to some specified type. These functions use several of the HLA built-in data type names as the function names. Functions in this category are

- `boolean`
- `int8`, `int16`, and `int32`
- `uns8`, `uns16`, and `uns32`
- `byte`, `word`, `dword` (these are effectively equivalent to `uns8`, `uns16`, and `uns32`)
- `real32`, `real64`, and `real80`
- `char`
- `string`
- `cset`
- `text`

These functions accept a single constant expression parameter and, if at all reasonable, convert that expression's value to the type specified by the type name. For example, the following function call returns the value `-128` since it converts the string constant to the corresponding integer value:

```
int8( "-128" )
```

Certain conversions don't make sense or have restrictions associated with them. For example, the *boolean* function will accept a string parameter, but that string must be "true" or "false" or the function will generate a compile-time error. Likewise, the numeric conversion functions (e.g., *int8*) allow a string operand but the string operand must represent a legal numeric value. Some conversions (e.g., *int8* with a character set parameter) simply don't make sense and are always illegal.

One of the most useful functions in this category is the *string* function. This function accepts nearly all constant expression types and it generates a string that represents the parameter's data. For example, the invocation `"string(128)"` produces the string `"128"` as the return result. This function is real handy when you have a value that you wish to use where HLA requires a string. For example, the `#ERROR` compile-time statement only allows a single string operand. You can use the *string* function and the string concatenation operator (`"+"`) to easily get around this limitation, e.g.,

```
#error( "Value ( " + string( Value ) + " ) is out of range" )
```

7.6.2 Numeric Compile-Time Functions

The functions in this category perform standard mathematical operations at compile time. These functions are real handy for generating lookup tables and "parameterizing" your source code by recalculating functions on constants defined at the beginning of your program. Functions in this category include the following:

- `@abs`
- `@ceil`, `@floor`
- `@sin`, `@cos`, `@tan`
- `@exp`, `@log`, `@log10`
- `@min`, `@max`
- `@random`, `@randomize`
- `@sqrt`

See Appendix H for more details on these functions.

7.6.3 Character Classification Compile-Time Functions

The functions in this group all return a boolean result. They test a character (or all the characters in a string) to see if it belongs to a certain class of characters. The functions in this category include

- @isAlpha, @isAlphanumeric
- @isDigit, @isxDigit
- @isLower, @isUpper
- @isSpace

In addition to these character classification functions, the HLA language provides a set of pattern matching functions that you can also use to classify character and string data. See the appropriate sections a little later for the discussion of these routines.

7.6.4 Compile-Time String Functions

The functions in this category operate on string parameters. Most return a string result although a few (e.g., @length and @index) return integer results. These functions do not directly affect the values of their parameters; instead, they return an appropriate result that you can assign back to the parameter if you wish to do so.

- @delete, @insert
- @index, @rindex
- @length
- @lowercase, @uppercase
- @strbrk, @strspan
- @strset
- @substr, @tokenize, @trim

For specific details concerning these functions and their parameters and their types, see Appendix H. Combined with the pattern matching functions, the string handling functions give you the ability to extend the HLA language by processing textual data that appears in your source files. Later sections appearing in this chapter will discuss ways to do this.

The @length function deserves a special discussion because it is probably the most popular function in this category. It returns an *uns32* constant specifying the number of characters found in its string parameter. The syntax is the following:

```
@length( string_expression )
```

Where *string_expression* represents any compile-time string expression. As noted above, this function returns the length, in characters, of the specified expression.

7.6.5 Compile-Time Pattern Matching Functions

HLA provides a very rich set of string/pattern matching functions that let you test a string to see if it begins with certain types of characters or strings. Along with the string processing functions, the pattern matching functions let you extend the HLA language and provide several other benefits as well. There are far too many pattern matching functions to list here (see Appendix H for complete details). However, a few examples will demonstrate the power and convenience of these routines.

The pattern matching functions all return a boolean true/false result. If a function returns true, we say that the function *succeeds* in matching its operand. If the function returns false, then we say it *fails* to match its operand. An important feature of the pattern matching functions is that they do not have to match the entire string you supply as a parameter, these patterns will (usually) succeed as long as they match a prefix of

the string parameter. The `@matchStr` function is a good example, the following function invocation always returns true:

```
@matchStr( "Hello World", "Hello" )
```

The first parameter of all the pattern matching functions ("Hello World" in this example) is the string to match. The matching functions will attempt to match the characters at the beginning of the string with the other parameters supplied for that particular function. In the `@matchStr` example above, the function succeeds if the first parameter begins with the string specified as the second parameter (which it does). The fact that the "Hello World" string contains additional characters beyond "Hello" is irrelevant; it only needs to begin with the string "Hello" is doesn't require equality with "Hello".

Most of the compile-time pattern matching functions support two optional parameters. The functions store additional data into the VAL objects specified by these two parameters if the function is successful (conversely, if the function fails, it does not modify these objects). The first parameter is where the function stores the *remainder*. The remainder after the execution of a pattern matching function is those characters that follow the matched characters in the string. In the example above, the remainder would be " World". If you wanted to capture this remainder data, you would add a third parameter to the `@matchStr` function invocation:

```
@matchStr( "Hello World", "Hello", World )
```

This function invocation would leave " World" sitting in the *World* VAL object. Note that *World* must be pre-declared as a string in the VAL section (or via the "?" statement) prior to the invocation of this function.

By using the conjunction operator ("&") you can combine several pattern matching functions into a single expression, e.g.,

```
@matchStr( "Hello There World", "Hello ", tw ) & @matchStr( tw, "There ", World )
```

This full expression returns true and leaves "World" sitting in the *World* variable. It also leaves "There World" sitting in *tw*, although *tw* is probably a temporary object whose value has no meaning beyond this expression. Of course, the above could be more efficiently implemented as follows:

```
@matchStr( "Hello There World", "Hello There", World )
```

However, keep in mind that you can combine different pattern matching functions using conjunction, they needn't all be calls to `@matchStr`.

The second optional parameter to most pattern matching functions holds a copy of the text that the function matched. E.g., the following call to `@matchStr` returns "Hello" in the Hello VAL object³

```
@matchStr( "Hello World", "Hello", World, Hello )
```

For more information on these pattern matching functions please see Appendix H. The chapter on Domain Specific Languages (see "Domain Specific Embedded Languages" on page 1003) and several other sections in this chapter will also make further use of these functions.

7.6.6 Compile-Time Symbol Information

During compilation HLA maintains an internal database known as the *symbol table*. The symbol table contains lots of useful information concerning all the identifiers you've defined up to a given point in the program. In order to generate machine code output, HLA needs to query this database to determine how to treat certain symbols. In your compile-time programs, it is often necessary to query the symbol table to determine how to handle an identifier or expression in your code. The HLA compile-time symbol information functions handle this task.

3. Strictly speaking, this example is rather contrived since we generally know the string that `@matchStr` matches. However, for other pattern matching functions this is not the case.

Many of the compile-time symbol information functions are well beyond the scope of this chapter (and, in some cases, beyond the scope of this text). This chapter will present a few of the functions and later chapters will add to this list. For a complete list of the compile-time symbol table functions, see Appendix H. The functions we will consider in this chapter include the following:

- @size
- @defined
- @typeName
- @elements
- @elementSize

Without question, the @size function is probably the most important function in this group. Indeed, previous chapters have made use of this function already. The @size function accepts a single HLA identifier or constant expression as a parameter. It returns the size, in bytes, of the data type of that object (or expression). If you supply an identifier, it can be a constant, type, or variable identifier. HLA returns the size of the type of the object. As you've seen in previous chapters, this function is invaluable for allocating storage via *malloc* and allocating arrays.

Another very useful function in this group is the @defined function. This function accepts a single HLA identifier as a parameter, e.g.,

```
@defined( MyIdentifier )
```

This function returns true if the identifier is defined at that point in the program, it returns false otherwise.

The @typeName function returns a string specifying the type name of the identifier or expression you supply as a parameter. For example, if *i32* is an *int32* object, then "@typeName(i32)" returns the string "int32". This function is useful for testing the types of objects you are processing in your compile-time programs.

The @elements function requires an array identifier or expression. It returns the total number of array elements as the function result. Note that for multi-dimensional arrays this function returns the product of all the array dimensions⁴.

The @elementSize function returns the size, in bytes, of an element of an array whose name you pass as a parameter. This function is extremely valuable for computing indices into an array (i.e., this function computes the *element_size* component of the array index calculation, see "Accessing Elements of a Single Dimension Array" on page 465).

7.6.7 Compile-Time Expression Classification Functions

The HLA compile-time language provides functions that will classify some arbitrary text and determine if that text is a constant expression, a register, a memory operand, a type identifier, and more. Some of the more common functions are

- @isConst
- @isReg, @isReg8, @isReg16, @isReg32, @isFReg
- @isMem
- @isType

Except for @isType, which requires an HLA identifier as a parameter, these functions all take some arbitrary text as their parameter. These functions return true or false depending upon whether that parameter satisfies the function requirements (e.g., @isConst returns true if its parameter is a constant identifier or expression). The @isType function returns true if its parameter is a type identifier.

The HLA compile-time language includes several other classification functions that are beyond the scope of this chapter. See Appendix H for details on those functions.

4. There is an @dim function that returns an array specifying the bounds on each dimension of a multidimensional array. See the appendices for more details if you're interested in this function.

7.6.8 Miscellaneous Compile-Time Functions

The HLA compile-time language contains several additional functions that don't fall into one of the categories above. Some of the more useful miscellaneous functions include

- @odd
- @lineNumber
- @text

The @odd function takes an ordinal value (i.e., non-real numeric or character) as a parameter and returns true if the value is odd, false if it is even. The @lineNumber function requires no parameters, it returns the current line number in the source file. This function is quite useful for debugging compile-time (and run-time!) programs.

The @text function is probably the most useful function in this group. It requires a single string parameter. It expands that string as text in place of the @text function call. This function is quite useful in conjunction with the compile-time string processing functions. You can build an instruction (or a portion of an instruction) using the string manipulation functions and then convert that string to program source code using the @text function. The following is a trivial example of this function in operation:

```
?id1:string := "eax";
?id2:string := "i32";
@text( "mov( " + id1 + ", " + id2 + ");" )
```

The sequence above compiles to

```
mov( eax, i32 );
```

7.6.9 Predefined Compile-Time Variables

In addition to functions, HLA also includes several predefined compile-time variables. The use of most of HLA's compile time variables is beyond the scope of this text. However, the following you've already seen:

- @bound
- @into

Volume Three (see "Some Additional Instructions: INTMUL, BOUND, INTO" on page 393) discusses the use of these objects to control the emission of the INTO and BOUND instructions. These two boolean pseudo-variables determine whether HLA will compile the BOUND (@bound) and INTO (@into) instructions or treat them as comments. By default, these two variables contain true and HLA will compile these instructions to machine code. However, if you set these values to false, using one or both of the following statements then HLA will not compile the associated statement:

```
?@bound := false;
?@into := false;
```

If you set @BOUND to false, then HLA treats BOUND instructions as though they were comments. If you set @INTO to false, then HLA treats INTO instructions as comments. You can control the emission of these statements throughout your program by selectively setting these pseudo-variables to true or false at different points in your code.

7.6.10 Compile-Time Type Conversions of TEXT Objects

Once you create a text constant in your program, it's difficult to manipulate that object. The following example demonstrates a programmer's desire to change the definition of a text symbol within a program:

```
val
  t:text := "stdout.put";
```

```

.
.
.
?t:text := "fileio.put";

```

The basic idea in this example is that *t* expands to "stdout.put" in the first half of the code and it expands to "fileio.put" in the second half of the program. Unfortunately, this simple example will not work. The problem is that HLA will expand a text symbol in place almost anywhere it finds that symbol. This includes occurrences of *t* within the "?" statement above. Therefore, the code above expands to the following (incorrect) text:

```

val
  t:text := "stdout.put";
  .
  .
  .
  ?stdout.put:text := "fileio.put";

```

HLA doesn't know how to deal with the "?" statement above, so it generates a syntax error.

At times you may not want HLA to expand a text object. Your code may want to process the string data held by the text object. HLA provides a couple of operators that deal with these two problems:

- @string:identifier
- @toString:identifier

The @string:identifier operator consists of @string, immediately followed by a colon and a text identifier (with no interleaving spaces or other characters). HLA returns a string constant corresponding to the text data associated with the text object. In other words, this operator lets you treat a text object as though it were a string constant within an expression.

Unfortunately, the @string operator converts a text object to a string constant, not a string identifier. Therefore, you cannot say something like

```
?@string:t := "Hello"
```

This doesn't work because @string:t replaces itself with the string constant associated with the text object *t*. Given the former assignment to *t*, this statement expands to

```
?"stdout.put" := "Hello";
```

This statement is still illegal.

The @toString:identifier operator comes to the rescue in this case. The @toString operator requires a text object as the associated identifier. It converts this text object to a string object (still maintaining the same string data) and then returns the identifier. Since the identifier is now a string object, you can assign a value to it (and change its type to something else, e.g., *text*, if that's what you need). To achieve the original goal, therefore, you'd use code like the following:

```

val
  t:text := "stdout.put";
  .
  .
  .
  ?@tostring:t : text := "fileio.put";

```

7.7 Conditional Compilation (Compile-Time Decisions)

HLA's compile-time language provides an IF statement, #IF, that lets you make various decisions at compile-time. The #IF statement has two main purposes: the traditional use of #IF is to support *conditional*

compilation (or *conditional assembly*) allowing you to include or exclude code during a compilation depending on the status of various symbols or constant values in your program. The second use of this statement is to support the standard IF statement decision making process in the HLA compile-time language. This section will discuss these two uses for the HLA #IF statement.

The simplest form of the HLA compile-time #IF statement uses the following syntax:

```
#if( constant_boolean_expression )

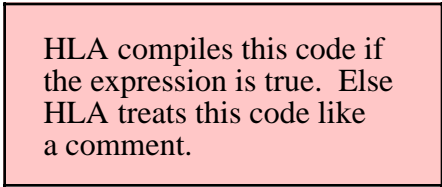
    << text >>

#endif
```

Note that you do not place semicolons after the #ENDIF clause. If you place a semicolon after the #ENDIF, it becomes part of the source code and this would be identical to inserting that semicolon immediately before the next text item in the program.

At compile-time, HLA evaluates the expression in the parentheses after the #IF. This must be a constant expression and its type must be boolean. If the expression evaluates true, then HLA continues processing the text in the source file as though the #IF statement were not present. However, if the expression evaluates false, then HLA treats all the text between the #IF and the corresponding #ENDIF clause as though it were a comment (i.e., it ignores this text).

```
#if( constant_boolean_expression )
```



HLA compiles this code if the expression is true. Else HLA treats this code like a comment.

```
#endif
```

Figure 7.2 Operation of HLA Compile-Time #IF Statement

Keep in mind that HLA's constant expressions support a full expression syntax like you'd find in a high level language like C or Pascal. The #IF expression syntax is not limited as are expressions in the HLA IF statement. Therefore, it is perfectly reasonable to write fancy expressions like the following:

```
#if( @length( someStrConst ) < 10 & ( MaxItems*2 < 100 | MinItems-5 < 10 ) )

    << text >>

#endif
```

Keep in mind that the items in a compile-time expression must all be CONST or VAL identifiers or an HLA compile-time function call (with appropriate parameters). In particular, remember that HLA evaluates these expressions at compile-time so they cannot contain run-time variables⁵. Also note that HLA's compile time language uses complete boolean evaluation, so any side effects that occur in the expression may produce undesired results.

The HLA #IF statement supports optional #ELSEIF and #ELSE clauses that behave in the intuitive fashion. The complete syntax for the #IF statement looks like the following:

```
#if( constant_boolean_expression1 )
```

5. Except, of course, as parameters to certain HLA compile-time functions like @size or @typeName.

```

    << text1 >>

#elseif( constant_boolean_expression2 )

    << text2 >>

#else

    << text3 >>

#endif

```

If the first boolean expression evaluates true then HLA processes the text up to the #ELSEIF clause. It then skips all text (i.e., treats it like a comment) until it encounters the #ENDIF clause. HLA continues processing the text after the #ENDIF clause in the normal fashion.

If the first boolean expression above evaluates false, then HLA skips all the text until it encounters a #ELSEIF, #ELSE, or #ENDIF clause. If it encounters a #ELSEIF clause (as above), then HLA evaluates the boolean expression associated with that clause. If it evaluates true, then HLA processes the text between the #ELSEIF and the #ELSE clauses (or to the #ENDIF clause if the #ELSE clause is not present). If, during the processing of this text, HLA encounters another #ELSEIF or, as above, a #ELSE clause, then HLA ignores all further text until it finds the corresponding #ENDIF.

If both the first and second boolean expressions in the example above evaluate false, then HLA skips their associated text and begins processing the text in the #ELSE clause. As you can see, the #IF statement behaves in a relatively intuitive fashion once you understand how HLA "executes" the body of these statements (that is, it processes the text or treats it as a comment depending on the state of the boolean expression). Of course, you can create a nearly infinite variety of different #IF statement sequences by including zero or more #ELSEIF clauses and optionally supplying the #ELSE clause. Since the construction is identical to the HLA IF..THEN..ELSEIF..ELSE..ENDIF statement, there is no need to elaborate further here.

A very traditional use of conditional compilation is to develop software that you can easily configure for several different environments. For example, the FCOMIP instruction makes floating point comparisons very easy but this instruction is available only on Pentium Pro and later processors. If you want to use this instruction on the processors that support it, and fall back to the standard floating point comparison on the older processors you would normally have to write two versions of the program - one with the FCOMIP instruction and one with the traditional floating point comparison sequence. Unfortunately, maintaining two different source files (one for newer processors and one for older processors) is very difficult. Most engineers prefer to use conditional compilation to embed the separate sequences in the same source file. The following example demonstrates how to do this.

```

const
    PentProOrLater: boolean := false; // Set true to use FCOMIxx instrs.
    .
    .
    .
    #if( PentProOrLater )

        fcomip(); // Compare st1 to st0 and set flags.

    #else

        fcomp(); // Compare st1 to st0.
        fstsw( ax ); // Move the FPU condition code bits
        sahf(); // into the FLAGS register.

    #endif

```

As currently written, this code fragment will compile the three instruction sequence in the #ELSE clause and ignore the code between the #IF and #ELSE clauses (because the constant *PentProOrLater* is

false). By changing the value of *PentProOrLater* to true, you can tell HLA to compile the single FCOMIP instruction rather than the three-instruction sequence. Of course, you can use the *PentProOrLater* constant in other #IF statements throughout your program to control how HLA compiles your code.

Note that conditional compilation does not let you create a single *executable* that runs efficiently on all processors. When using this technique you will still have to create two executable programs (one for Pentium Pro and later processors, one for the earlier processors) by compiling your source file twice; during the first compilation you must set the *PentProOrLater* constant to false, during the second compilation you must set this constant to true. Although you must create two separate executables, you need only maintain a single source file.

If you are familiar with conditional compilation in other languages, such as the C/C++ language, you may be wondering if HLA supports a statement like C's "#ifdef" statement. The answer is no, it does not. However, you can use the HLA compile-time function @DEFINED to easily test to see if a symbol has been defined earlier in the source file. Consider the following modification to the above code that uses this technique:

```
const
    // Note: uncomment the following line if you are compiling this
    // code for a Pentium Pro or later CPU.

    // PentProOrLater :=0; // Value and type are irrelevant
    .
    .
    .
    #if( @defined( PentProOrLater ) )

        fcomip();          // Compare st1 to st0 and set flags.

    #else

        fcomp();          // Compare st1 to st0.
        fstsw( ax );      // Move the FPU condition code bits
        sahf();           // into the FLAGS register.

    #endif
```

Another common use of conditional compilation is to introduce debugging and testing code into your programs. A typical debugging technique that many HLA programmers use is to insert "print" statements at strategic points throughout their code in order to trace through their code and display important values at various checkpoints. A big problem with this technique is that they must remove the debugging code prior to completing the project. The software's customer (or a student's instructor) probably doesn't want to see debugging output in the middle of a report the program produces. Therefore, programmers who use this technique tend to insert code temporarily and then remove the code once they run the program and determine what is wrong. There are at least two problems with this technique:

- Programmers often forget to remove some debugging statements and this creates defects in the final program, and
- After removing a debugging statement, these programmers often discover that they need that same statement to debug some different problem at a later time. Hence they are constantly inserting, removing, and inserting the same statements over and over again.

Conditional compilation can provide a solution to this problem. By defining a symbol (say, *debug*) to control debug output in your program, you can easily activate or deactivate *all* debugging output by simply modifying a single line of source code. The following code fragment demonstrates this:

```
const
    debug: boolean := false; // Set to true to activate debug output.
    .
    .
    .
```

```

#if( debug )

    stdout.put( "At line ", @lineNumber, " i=", i, nl );

#endif

```

As long as you surround all debugging output statements with a #IF statement like the one above, you don't have to worry about debug output accidentally appearing in your final application. By setting the *debug* symbol to false you can automatically disable all such output. Likewise, you don't have to remove all your debugging statements from your programs once they've served their immediate purpose. By using conditional compilation, you can leave these statements in your code because they are so easy to deactivate. Later, if you decide you need to view this same debugging information during a program run, you won't have to reenter the debugging statement - you simply reactivate it by setting the *debug* symbol to true.

We will return to this issue of inserting debugging code into your programs in the chapter on macros.

Although program configuration and debugging control are two of the more common, traditional, uses for conditional compilation, don't forget that the #IF statement provides the basic conditional statement in the HLA compile-time language. You will use the #IF statement in your compile-time programs the same way you would use an IF statement in HLA or some other language. Later sections in this text will present lots of examples of using the #IF statement in this capacity.

7.8 Repetitive Compilation (Compile-Time Loops)

HLA's #WHILE..#ENDWHILE statement provides a compile-time loop construct. The #WHILE statement tells HLA to repetitively process the same sequence of statements during compilation. This is very handy for constructing data tables (see "Constructing Data Tables at Compile Time" on page 996) as well as providing a traditional looping structure for compile-time programs. Although you will not employ the #WHILE statement anywhere near as often as the #IF statement, this compile-time control structure is very important when writing advanced HLA programs.

The #WHILE statement uses the following syntax:

```

#while( constant_boolean_expression )

    << text >>

#endwhile

```

When HLA encounters the #WHILE statement during compilation, it will evaluate the constant boolean expression. If the expression evaluates false, then HLA will skip over the text between the #WHILE and the #ENDWHILE clause (the behavior is similar to the #IF statement if the expression evaluates false). If the expression evaluates true, then HLA will process the statements between the #WHILE and #ENDWHILE clauses and then "jump back" to the start of the #WHILE statement in the source file and repeat this process.

```
#while( constant_boolean_expression )
```

HLA repetitively compiles this code as long as the expression is true. It effectively inserts multiple copies of this statement sequence into your source file (the exact number of copies depends on the value of the loop control expression).

```
#endwhile
```

Figure 7.3 HLA Compile-Time #WHILE Statement Operation

To understand how this process works, consider the following program:

```
program ctWhile;
#include( "stdlib.hhf" )

static
  ary: uns32[5] := [ 2, 3, 5, 8, 13 ];

begin ctWhile;

  ?i := 0;
  #while( i < 5 )

    stdout.put( "array[ ", i, " ] = ", ary[i*4], nl );
    ?i := i + 1;

  #endwhile

end ctWhile;
```

Program 7.2 #WHILE..#ENDWHILE Demonstration

As you can probably surmise, the output from this program is the following:

```
array[ 0 ] = 2
array[ 1 ] = 3
array[ 2 ] = 4
array[ 3 ] = 5
array[ 4 ] = 13
```

What is not quite obvious is how this program generates this output. Remember, the #WHILE..#ENDWHILE construct is a compile-time language feature, not a run-time control construct. Therefore, the #WHILE loop above repeats five times during *compilation*. On each repetition of the loop, the HLA compiler processes the statements between the #WHILE and #ENDWHILE clauses. Therefore, the program above is really equivalent to the following:

```
program ctWhile;
#include( "stdlib.hhf" )

static
    ary: uns32[5] := [ 2, 3, 5, 8, 13 ];

begin ctWhile;

    stdout.put( "array[ ", 0, " ] = ", ary[0*4], nl );
    stdout.put( "array[ ", 1, " ] = ", ary[1*4], nl );
    stdout.put( "array[ ", 2, " ] = ", ary[2*4], nl );
    stdout.put( "array[ ", 3, " ] = ", ary[3*4], nl );
    stdout.put( "array[ ", 4, " ] = ", ary[4*4], nl );

end ctWhile;
```

Program 7.3 Program Equivalent to the Code in Program 7.2

As you can see, the #WHILE statement is very convenient for constructing repetitive code sequences. This is especially invaluable for unrolling loops. Additional uses of the #WHILE loop appear in later sections of this text.

7.9 Putting It All Together

The HLA compile-time language provides considerable power. With the compile-time language you can automate the generation of tables, selectively compile code for different environments, easily unroll loops to improve performance, and check the validity of code you're writing. Combined with macros and other features that HLA provides, the compile-time language is probably the premier feature of the HLA language – no other assembler provides comparable features. For more information about the HLA compile time language, be sure to read the next chapter on macros.

Macros

Chapter Eight

8.1 Chapter Overview

This chapter continues where the previous chapter left off – continuing to discuss the HLA compile time language. This chapter discusses what is, perhaps, the most important component of the HLA compile-time language, macros. Many people judge the power of an assembler by the power of its macro processing capabilities. If you happen to be one of these people, you'll probably agree that HLA is one of the more powerful assemblers on the planet after reading this chapter; because HLA has one of the most powerful macro processing facilities of any computer language processing system.

8.2 Macros (Compile-Time Procedures)

Macros are symbols that a language processor replaces with other text during compilation. Macros are great devices for replacing long repetitive sequences of text with much shorter sequences of text. In addition to the traditional role that macros play (e.g., "#define" in C/C++), HLA's macros also serve as the equivalent of a compile-time language procedure or function. Therefore, macros are very important in HLA's compile-time language; just as important as functions and procedures are in other high level languages.

Although macros are nothing new, HLA's implementation of macros far exceeds the macro processing capabilities of most other programming languages (high level or low level). The following sections explore HLA's macro processing facilities and the relationship between macros and other HLA CTL control constructs.

8.2.1 Standard Macros

HLA supports a straight-forward macro facility that lets you define macros in a manner that is similar to declaring a procedure. A typical, simple, macro declaration takes the following form:

```
#macro macroname ;

    << macro body >>

#endmacro ;
```

Although macro and procedure declarations are similar, there are several immediate differences between the two that are obvious from this example. First, of course, macro declarations use the reserved word #MACRO rather than PROCEDURE. Second, you do not begin the body of the macro with a "BEGIN *macroname*;" clause. This is because macros don't have a declaration section like procedures so there is no need for a keyword that separates the macro declarations from the macro body. Finally, you will note that macros end with the "#ENDMACRO" clause rather than "END *macroname*;" The following is a concrete example of a macro declaration:

```
#macro neg64 ;

    neg( edx );
    neg( eax );
    sbb( 0, edx );

#endmacro ;
```

Execution of this macro's code will compute the two's complement of the 64-bit value in EDX:EAX (see "Extended Precision NEG Operations" on page 872).

To execute the code associated with *neg64*, you simply specify the macro's name at the point you want to execute these instructions, e.g.,

```
mov( (type dword i64), eax );
mov( (type dword i64+4), edx );
neg64;
```

Note that you do *not* follow the macro's name with a pair of empty parentheses as you would a procedure call (the reason for this will become clear a little later).

Other than the lack of parentheses following *neg64*'s invocation¹ this looks just like a procedure call. You could implement this simple macro as a procedure using the following procedure declaration:

```
procedure neg64p;
begin neg64p;

    neg( edx );
    neg( eax );
    sbb( 0, edx );

end neg64p;
```

Note that the following two statements will both negate the value in EDX:EAX:

```
neg64;           neg64p( );
```

The difference between these two (i.e., the macro invocation versus the procedure call) is the fact that macros expand their text in-line whereas a procedure call emits a call to the associate procedure elsewhere in the text. That is, HLA replaces the invocation "neg64;" directly with the following text:

```
neg( edx );
neg( eax );
sbb( 0, edx );
```

On the other hand, HLA replaces the procedure call "neg64p();" with the single call instruction:

```
call neg64p;
```

Presumably, you've defined the *neg64p* procedure earlier in the program.

You should make the choice of macro versus procedure call on the basis of efficiency. Macros are slightly faster than procedure calls because you don't execute the CALL and corresponding RET instructions. On the other hand, the use of macros can make your program larger because a macro invocation expands to the text of the macro's body on each invocation. Procedure calls jump to a single instance of the procedure's body. Therefore, if the macro body is large and you invoke the macro several times throughout your program, it will make your final executable much larger. Also, if the body of your macro executes more than a few simple instructions, the overhead of a CALL/RET sequence has little impact on the overall execution time of the code, so the execution time savings are nearly negligible. On the other hand, if the body of a procedure is very short (like the *neg64* example above), you'll discover that the macro implementation is much faster and doesn't expand the size of your program by much. Therefore, a good rule of thumb is

- ❑ Use macros for short, time-critical program units. Use procedures for longer blocks of code and when execution time is not as critical.

Macros have many other disadvantages over procedures. Macros cannot have local (automatic) variables, macro parameters work differently than procedure parameters, macros don't support (run-time) recur-

1. To differentiate macros and procedures, this text will use the term *invocation* when describing the use of a macro and *call* when describing the use of a procedure.

sion, and macros are a little more difficult to debug than procedures (just to name a few disadvantages). Therefore, you shouldn't really use macros as a substitute for procedures except in some rare situations.

8.2.2 Macro Parameters

Like procedures, macros allow you to define parameters that let you supply different data on each macro invocation. This lets you write generic macros whose behavior can vary depending on the parameters you supply. By processing these macro parameters at compile-time, you can write very sophisticated macros.

Macro parameter declaration syntax is very straight-forward. You simply supply a list of parameter names within parentheses in a macro declaration:

```
#macro neg64( reg32HO, reg32LO );

    neg( reg32HO );
    neg( reg32LO );
    sbb( 0, reg32HO );

#endmacro;
```

Note that you do not associate a data type with a macro parameter like you do procedural parameters. This is because HLA macros are always *text* objects. The next section will explain the exact mechanism HLA uses to substitute an actual parameter for a formal parameter.

When you invoke a macro, you simply supply the actual parameters the same way you would for a procedure call:

```
neg64( edx, eax );
```

Note that a macro invocation that requires parameters expects you to enclose the parameter list within parentheses.

8.2.2.1 Standard Macro Parameter Expansion

As the previous section explains, HLA automatically associates the type *text* with macro parameters. This means that during a macro expansion, HLA substitutes the text you supply as the actual parameter everywhere the formal parameter name appears. The semantics of "pass by textual substitution" are a little different than "pass by value" or "pass by reference" so it is worthwhile exploring those differences here.

Consider the following macro invocations, using the *neg64* macro from the previous section:

```
neg64( edx, eax );
neg64( ebx, ecx );
```

These two invocations expand into the following code:

```
// neg64(edx, eax );

    neg( edx );
    neg( eax );
    sbb( 0, edx );

// neg64( ebx, ecx );

    neg( ebx );
    neg( ecx );
    sbb( 0, ebx );
```

Note that macro invocations do not make a local copy of the parameters (as pass by value does) nor do they pass the address of the actual parameter to the macro. Instead, a macro invocation of the form "neg64(edx, eax);" is equivalent to the following:

```
?reg32HO: text := "edx";
?reg32LO: text := "eax";

neg( reg32HO );
neg( reg32LO );
sbb( 0, reg32HO );
```

Of course, the text objects immediately expand their string values in-line, producing the former expansion for "neg64(edx, eax);".

Note that macro parameters are not limited to memory, register, or constant operands as are instruction or procedure operands. Any text is fine as long as its expansion is legal wherever you use the formal parameter. Similarly, formal parameters may appear anywhere in the macro body, not just where memory, register, or constant operands are legal. Consider the following macro declaration and sample invocations:

```
#macro chkError( instr, jump, target );

    instr;
    jump target;

#endmacro

    chkError( cmp( eax, 0 ), jnl, RangeError );    // Example 1
    ...
    chkError( test( 1, bl ), jnz, ParityError );    // Example 2

// Example 1 expands to

    cmp( eax, 0 );
    jnl RangeError;

// Example 2 expands to

    test( 1, bl );
    jnz ParityError;
```

In general, HLA assumes that all text between commas constitutes a single macro parameter. If HLA encounters any opening "bracketing" symbols (left parentheses, left braces, or left brackets) then it will include all text up to the appropriate closing symbol, ignoring any commas that may appear within the bracketing symbols. This is why the *chkError* invocations above treat "cmp(eax, 0)" and "test(1, bl)" as single parameters rather than as a pair of parameters. Of course, HLA does not consider commas (and bracketing symbols) within a string constant as the end of an actual parameter. So the following macro and invocation is perfectly legal:

```
#macro print( strToPrint );

    stdout.out( strToPrint );

#endmacro

.
.
.
print( "Hello, world!" );
```

HLA treats the string "Hello, world!" as a single parameter since the comma appears inside a literal string constant, just as your intuition suggests.

If you are unfamiliar with textual macro parameter expansion in other languages, you should be aware that there are some problems you can run into when HLA expands your actual macro parameters. Consider the following macro declaration and invocation:

```
#macro Echo2nTimes( n, theStr );

    ?echoCnt: uns32 := 0;
    #while( echoCnt < n*2 )

        #print( theStr )
        ?echoCnt := echoCnt + 1;

    #endwhile

#endmacro;

.
.
.
Echo2nTimes( 3+1, "Hello" );
```

This example displays "Hello" five times during compilation rather than the eight times you might intuitively expect. This is because the #WHILE statement above expands to

```
#while( echoCnt < 3+1*2 )
```

The actual parameter for *n* is "3+1", since HLA expands this text directly in place of *n*, you get the text above. Of course, at compile time HLA computes "3+1*2" as the value five rather than as the value eight (which you would get had HLA passed this parameter by value rather than by textual substitution).

The common solution to this problem, when passing numeric parameters that may contain compile-time expressions, is to surround the formal parameter in the macro with parentheses. E.g., you would rewrite the macro above as follows:

```
#macro Echo2nTimes( n, theStr );

    ?echoCnt: uns32 := 0;
    #while( echoCnt < (n)*2 )

        #print( theStr )
        ?echoCnt := echoCnt + 1;

    #endwhile

#endmacro;
```

The previous invocation would expand to the following code:

```
?echoCnt: uns32 := 0;
#while( echoCnt < (3+1)*2 )

    #print( theStr )
    ?echoCnt := echoCnt + 1;

#endwhile
```

This version of the macro produces the intuitive result.

If the number of actual parameters does not match the number of formal parameters, HLA will generate a diagnostic message during compilation. Like procedures, the number of actual parameters must agree with the number of formal parameters. If you would like to have optional macro parameters, then keep reading...

8.2.2.2 Macros with a Variable Number of Parameters

You may have noticed by now that some HLA macros don't require a fixed number of parameters. For example, the *stdout.put* macro in the HLA Standard Library allows one or more actual parameters. HLA uses a special array syntax to tell the compiler that you wish to allow a variable number of parameters in a macro parameter list. If you follow the last macro parameter in the formal parameter list with "[]" then HLA will allow a variable number of actual parameters (zero or more) in place of that formal parameter. E.g.,

```
#macro varParms( varying[ ] );

    << macro body >>

#endmacro;

.
.
.
varParms( 1 );
varParms( 1, 2 );
varParms( 1, 2, 3 );
varParms();
```

Note, especially, the last example above. If a macro has any formal parameters, you must supply parentheses with the macro list after the macro invocation. This is true even if you supply zero actual parameters to a macro with a varying parameter list. Keep in mind this important difference between a macro with no parameters and a macro with a varying parameter list but no actual parameters.

When HLA encounters a formal macro parameter with the "[]" suffix (which must be the last parameter in the formal parameter list), HLA creates a constant string array and initializes that array with the text associated with the remaining actual parameters in the macro invocation. You can determine the number of actual parameters assigned to this array using the `@ELEMENTS` compile-time function. For example, "`@elements(varying)`" will return some value, zero or greater, that specifies the total number of parameters associated with that parameter. The following declaration for *varParms* demonstrates how you might use this:

```
#macro varParms( varying[ ] );

    ?vpCnt := 0;
    #while( vpCnt < @elements( varying ) )

        #print( varying[ vpCnt ] )
        ?vpCnt := vpCnt + 1;

    #endwhile

#endmacro;

.
.
.
varParms( 1 );           // Prints "1" during compilation.
varParms( 1, 2 );       // Prints "1" and "2" on separate lines.
varParms( 1, 2, 3 );    // Prints "1", "2", and "3" on separate lines.
varParms();             // Doesn't print anything.
```

Since HLA doesn't allow arrays of *text* objects, the varying parameter must be an array of strings. This, unfortunately, means you must treat the varying parameters differently than you handle standard macro parameters. If you want some element of the varying string array to expand as text within the macro body, you can always use the `@TEXT` function to achieve this. Conversely, if you want to use a non-varying for-

mal parameter as a string object, you can always use the @STRING:name operator. The following example demonstrates this:

```
#macro ReqAndOpt( Required, optional[] );

    ?@text( optional[0] ) := @string:ReqAndOpt;
    #print( @text( optional[0] ) )

#endmacro;

.
.
.
ReqAndOpt( i, j );

// The macro invocation above expands to

?@text( "j" ) := @string:i;
#print( "j" )

// The above further expands to

j := "i";
#print( j )

// The above simply prints "i" during compilation.
```

Of course, it would be a good idea, in a macro like the above, to verify that there are at least two parameters before attempting to reference element zero of the *optional* parameter. You can easily do this as follows:

```
#macro ReqAndOpt( Required, optional[] );

    #if( @elements( optional ) > 0 )

        ?@text( optional[0] ) := @string:ReqAndOpt;
        #print( @text( optional[0] ) )

    #else

        #error( "ReqAndOpt must have at least two parameters" )

    #endif

#endmacro;
```

8.2.2.3 Required Versus Optional Macro Parameters

As noted in the previous section, HLA requires exactly one actual parameter for each non-varying formal macro parameter. If there is no varying macro parameter (and there can be at most one) then the number of actual parameters must exactly match the number of formal parameters. If a varying formal parameter is present, then there must be at least as many actual macro parameters as there are non-varying (or required) formal macro parameters. If there is a single, varying, actual parameter, then a macro invocation may have zero or more actual parameters.

There is one big difference between a macro invocation of a macro with no parameters and a macro invocation of a macro with a single, varying, parameter that has no actual parameters: the macro with the

varying parameter list must have an empty set of parentheses after it while the macro invocation of the macro without any parameters does not allow this. You can use this fact to your advantage if you wish to write a macro that doesn't have any parameters but you want to follow the macro invocation with "(") so that it matches the syntax of a procedure call with no parameters. Consider the following macro:

```
#macro neg64( JustForTheParens[ ] );

    #if( @elements( JustForTheParens ) = 0 )

        neg( edx );
        neg( eax );
        sbb( 0, edx );

    #else

        #error( "Unexpected operand(s)" )

    #endif

#endmacro;
```

The macro above allows invocations of the form "neg64();" using the same syntax you would use for a procedure call. This feature is useful if you want the syntax of your parameterless macro invocations to match the syntax of a parameterless procedure call. It's not a bad idea to do this, just in the off chance you need to convert the macro to a procedure at some point (or vice versa, for that matter).

If, for some reason, it is more convenient to operate on your macro parameters as *string* objects rather than *text* objects, you can specify a single varying parameter for the macro and then use #IF and @ELEMENTS to enforce the number of required actual parameters.

8.2.2.4 The "#(" and ")#" Macro Parameter Brackets

Once in a (really) great while, you may want to include arbitrary commas (i.e., outside a bracketing pair) within a macro parameter. Or, perhaps, you might want to include other text as part of a macro expansion that HLA would normally process before storing away the text as the value for the formal parameter². The "#(" and ")#" bracketing symbols tell HLA to collect all text, except for surrounding whitespace, between these two symbols and treat that text as a single parameter. Consider the following macro:

```
#macro PrintName( theParm );

    ?theName := @string:theParm;
    #print( theName )

#endmacro;
```

Normally, this macro will simply print the text of the actual parameter you pass to it. So were you to invoke the macro with "PrintName(j);" HLA would simply print "j" during compilation. This occurs because HLA associates the parameter data ("j") with the string value for the text object *theParm*. The macro converts this text data to a string, puts the string data in *theName*, and then prints this string.

Now consider the following statements:

```
?tt:text := "j";
PrintName( tt );
```

This macro invocation will also print "j". The reason is that HLA expands text constants immediately upon encountering them. So after this expansion, the invocation above is equivalent to

2. For example, HLA will normally expand all *text* objects prior to the creation of the data for the formal parameter. You might not want this expansion to occur.


```
PrintName( j );
```

So this macro invocation prints "j" for the same reason the last example did.

What if you want the macro to print "tt" rather than "j"? Unfortunately, HLA's *eager* evaluation of the text constant gets in the way here. However, if you bracket "tt" with the "#(" and ")#" brackets, you can instruct HLA to *defer* the expansion of this text constant until it actually expands the macro. I.e., the following macro invocation prints "tt" during compilation:

```
PrintName( #( tt )# );
```

Note that HLA allows any amount of arbitrary text within the "#(" and ")#" brackets. This can include commas and other arbitrary text. The following macro invocation prints "Hello, World!" during compilation:

```
PrintName( #( Hello, World! )# );
```

Normally, HLA would complain about the mismatched number of parameters since the comma would suggest that there are two parameters here. However, the deferred evaluation brackets tell HLA to consider all the text between the "#(" and ")#" symbols as a single parameter.

8.2.2.5 Eager vs. Deferred Macro Parameter Evaluation

HLA uses two schemes to process macro parameters. As you saw in the previous section, HLA uses *eager* evaluation when processing text constants appearing in a macro parameter list. You can force *deferred* evaluation of the text constant by surrounding the text with the "#(" and ")#" brackets. For other types of operands, HLA uses deferred macro parameter evaluation. This section discusses the difference between these two forms and describes how to force eager evaluation if necessary.

Eager evaluation occurs while HLA is collecting the text associated with each macro parameter. For example, if "T" is a text constant containing the string "U" and "M" is a macro, then when HLA encounters "M(T)" it will first expand "T" to "U". Then HLA processes the macro invocation "M(U)" as though you had supplied the text "U" as the parameter to begin with.

Deferred evaluation of macro parameters means that HLA does not process the parameter(s), but rather passes the text unchanged to the macro. Any expansion of the text associated with macro parameters occurs within the macro itself. For example, if M and N are both macros accepting a single parameter, then the invocation "M(N(0))" defers the evaluation of "N(0)" until HLA processes the macro body. It does not evaluate "N(0)" first and pass this expansion as a parameter to the macro. The following program demonstrates eager and deferred evaluation:

```
// This program demonstrates the difference
// between deferred and eager macro parameter
// processing.

program EagerVsDeferredEvaluation;

macro ToDefer( tdParm );

    #print( "ToDefer: ", @string:tdParm )
    @string:tdParm

endmacro;

macro testEVD( theParm );

    #print( "testEVD:'", @string:theParm, "' " )
```

```

endmacro;

const

    txt:text := "Hello";
    str:string := "there";

begin EagerVsDeferredEvaluation;

    testEVD( str );           // Deferred evaluation.
    testEVD( txt );         // Eager evaluation.
    testEVD( ToDefer( World ) ); //Deferred evaluation.

end EagerVsDeferredEvaluation;

```

Program 8.1 Eager vs. Deferred Macro Parameter Evaluation

Note that the macro *testEVD* outputs the text associated with the formal parameter as a string during compilation. When you compile Program 8.1 it produces the following output:

```

testEVD: 'Hello'
testEVD: 'Hello'
testEVD: 'ToDefer( World )'

```

The first line prints 'Hello' because this is the text supplied as a parameter for the first call to *testEVD*. Since this is a string constant, not a text constant, HLA uses deferred evaluation. This means that it passes the text appearing between the parentheses unchanged to the *testEVD* macro. That text is "Hello" hence the same output as the parameter text.

The second *testEVD* invocation prints 'Hello'. This is because the macro parameter, *txt*, is a text object. HLA eagerly processes text constants before invoking the macro. Therefore, HLA translates "testEVD(txt)" to "testEVD(Hello)" prior to invoking the macro. Since the macro parameter text is now "Hello", that's what HLA prints during compilation while processing this macro.

The third invocation of *testEVD* above is semantically identical to the first. It is present just to demonstrate that HLA defers processing macros just like it defers the processing of everything else except text constants.

Although the code in Program 8.1 does not actually evaluate the *ToDefer* macro invocation, this is only because the body of *testEVD* does not directly use the parameter. Instead, it converts *theParm* to a string and prints its value. Had this code actually referred to *theParm* in an expression (or as a statement), then HLA would have invoked *ToDefer* and let it do its job. Consider the following modification to the above program:

```

// This program demonstrates the difference
// between deferred and eager macro parameter
// processing.

program DeferredEvaluation;

macro ToDefer( tdParm );

    @string:tdParm

endmacro;

```

```

macro testEVD( theParm );

    #print( "Hello ", theParm )

endmacro;

begin DeferredEvaluation;

    testEVD( ToDefer( World ) );

end DeferredEvaluation;

```

Program 8.2 Deferred Macro Parameter Expansion

The macro invocation "testEVD(ToDefer(World));" defers the evaluation of its parameter. Therefore, the actual parameter *theParm* is a text object containing the string "ToDefer(World)". Inside the testEVD macro, HLA encounters theParm and expands it to this string, i.e.,

```
#print( "Hello ", theParm )
```

expands to

```
#print( "Hello ", ToDefer( World ) )
```

When HLA processes the #PRINT statement, it eagerly processes all parameters. Therefore, HLA expands the statement above to

```
#print( "Hello ", "World" )
```

since "ToDefer(World)" expands to *@string:tdParm* and that expands to "World".

Most of the time, the choice between deferred and eager evaluation produces the same result. In Program 8.2, for example, it doesn't matter whether the *ToDefer* macro expansion is eager (thus passing the string "World" as the parameter to *testEVD*) or deferred. Either mechanism produces the same output.

There are situations where deferred evaluation is not interchangeable with eager evaluation. The following program demonstrates a problem that can occur when you use deferred evaluation rather than eager evaluation. In this example the program attempts to pass the current line number in the source file as a parameter to a macro. This does not work because HLA expands (and evaluates) the @LINENUMBER function call inside the macro on every invocation. Therefore, this program always prints the same line number (eight) regardless of the invocation line number:

```

// This program a situation where deferred
// evaluation fails to work properly.

program DeferredFails;

macro printAt( where );

    #print( "at line ", where )

endmacro;

```

```
begin DeferredFails;

    printAt( @linenumber );
    printAt( @lineNumber );

end DeferredFails;
```

Program 8.3 An Example Where Deferred Evaluation Fails to Work Properly

Intuitively, this program should print:

```
at line 14
at line 15
```

Unfortunately, because of deferred evaluation, the two *printAt* invocations simply pass the text "@linenumber" as the actual parameter value rather than the string representing the line numbers of these two statements in the program. Since the formal parameter always expands to @LINENUMBER on the same line (line eight), this program always prints the same line number regardless of the line number of the macro invocation.

If you need an eager evaluation of a macro parameter there are three ways to achieve this. First of all, of course, you can specify a *text* object as a macro parameter and HLA will immediately expand that object prior to passing it as the macro parameter. The second option is to use the @TEXT function (with a string parameter). HLA will also immediately process this object, expanding it to the appropriate text, prior to processing that text as a macro parameter. The third option is to use the @EVAL pseudo-function. Within a macro invocation's parameter list, the @EVAL function instructs HLA to evaluate the @EVAL parameter prior to passing the text to the macro. Therefore, you can correct the problem in Program 8.3 by using the following code (which properly prints at "at line 14" and "at line 15"):

```
// This program a situation where deferred
// evaluation fails to work properly.

program EvalSucceeds;

macro printAt( where );

    #print( "at line ", where )

endmacro;

begin EvalSucceeds;

    printAt( @eval( @linenumber ));
    printAt( @eval( @lineNumber ));

end EvalSucceeds;
```

Program 8.4 Demonstration of @EVAL Compile-time Function

In addition to immediately processing built-in compiler functions like @LINENUMBER, the @EVAL pseudo-function will also invoke any macros appearing in the @EVAL parameter. @EVAL usually leaves other values unchanged.

8.2.3 Local Symbols in a Macro

Consider the following macro declaration:

```
macro JZC( target );

    jnz NotTarget;
    jc target;
    NotTarget:

endmacro;
```

The purpose of this macro is to simulate an instruction that jumps to the specified target location if the zero flag is set *and* the carry flag is set. Conversely, if either the zero flag is clear or the carry flag is clear this macro transfers control to the instruction immediately following the macro invocation.

There is a serious problem with this macro. Consider what happens if you use this macro more than once in your program:

```
JZC( Dest1 );
.
.
.
JZC( Dest2 );
.
.
.
```

The macro invocations above expand to the following code:

```
    jnz NotTarget;
    jc Dest1;
NotTarget:
.
.
.
    jnz NotTarget;
    jc Dest2;
NotTarget:
.
.
.
```

The problem with the expansion of these two macro invocations is that they both emit the same label, *NotTarget*, during macro expansion. When HLA processes this code it will complain about a duplicate symbol definition. Therefore, you must take care when defining symbols inside a macro because multiple invocations of that macro may lead to multiple definitions of that symbol.

HLA's solution to this problem is to allow the use of *local symbols* within a macro. Local macro symbols are unique to a specific invocation of a macro. For example, had *NotTarget* been a local symbol in the *JZC* macro invocations above, the program would have compiled properly since HLA treats each occurrence of *NotTarget* as a unique symbol.

HLA does not automatically make internal macro symbol definitions local to that macro³. Instead, you must explicitly tell HLA which symbols must be local. You do this in a macro declaration using the following generic syntax:

```
#macro macroname ( optional_parameters ) : optional_list_of_local_names ;
    << macro body >>
#endmacro;
```

3. Sometimes you actually want the symbols to be global.

The list of local names is a sequence of one or more HLA identifiers separated by commas. Whenever HLA encounters this name in a particular macro invocation it automatically substitutes some unique name for that identifier. For each macro invocation, HLA substitutes a different name for the local symbol.

You can correct the problem with the *JZC* macro by using the following macro code:

```
#macro JZC( target ):NotTarget;

    jnz NotTarget;
    jc target;
    NotTarget:

#endmacro
;
```

Now whenever HLA processes this macro it will automatically associate a unique symbol with each occurrence of *NotTarget*. This will prevent the duplicate symbol error that occurs if you do not declare *NotTarget* as a local symbol.

HLA implements local symbols by substituting a symbol of the form "*_nnnn_*" (where *nnnn* is a four-digit hexadecimal number) wherever the local symbol appears in a macro invocation. For example, a macro invocation of the form "JZC(SomeLabel);" might expand to

```
    jnz _010A_;
    jc SomeLabel;
_010A_:
```

For each local symbol appearing within a macro expansion HLA will generate a unique temporary identifier by simply incrementing this numeric value for each new local symbol it needs. As long as you do not explicitly create labels of the form "*_nnnn_*" (where *nnnn* is a hexadecimal value) there will never be a conflict in your program. HLA explicitly reserves all symbols that begin and end with a single underscore for its own private use (and for use by the HLA Standard Library). As long as you honor this restriction, there should be no conflicts between HLA local symbol generation and labels in your own programs since all HLA-generated symbols begin and end with a single underscore.

HLA implements local symbols by effectively converting that local symbol to a text constant that expands to the unique symbol HLA generates for the local label. That is, HLA effectively treats local symbol declarations as indicated by the following example:

```
#macro JZC( target );
    ?NotTarget:text := "_010A_";

    jnz NotTarget;
    jc target;
    NotTarget:

#endmacro;
```

Whenever HLA expands this macro it will substitute "*_010A_*" for each occurrence of *NotTarget* it encounters in the expansion. This analogy isn't perfect because the text symbol *NotTarget* in this example is still accessible after the macro expansion whereas this is not the case when defining local symbols within a macro. But this does give you an idea of how HLA implements local symbols.

One important consequence of HLA's implementation of local symbols within a macro is that HLA will produce some puzzling error messages if an error occurs on a line that uses a local symbol. Consider the following (incorrect) macro declaration:

```
#macro LoopZC( TopOfLoop ): ExitLocation;

    jnz ExitLocation;
    jc TopOfLoop;

#endmacro;
```

Note that in this example the macro does not define the *ExitLocation* symbol even though there is a jump (JNZ) to this label. If you attempt to compile this program, HLA will complain about an undefined statement label and it will state that the symbol is something like "_010A_" rather than *ExitLocation*.

Locating the exact source of this problem can be challenging since HLA cannot report this error until the end of the procedure or program in which *LoopZC* appears (long after you've invoked the macro). If you have lots of macros with lots of local symbols, locating the exact problem is going to be a lot of work; your only option is to carefully analyze the macros you do call (perhaps by commenting them out of your program one by one until the error goes away) to discover the source of the problem. Once you determine the offending macro, the next step is to determine which local symbol is the culprit (if the macro contains more than one local symbol). Because tracking down bugs associated with local symbols can be tough, you should be especially careful when using local symbols within a macro.

Because local symbols are effectively text constants, don't forget that HLA eagerly processes any local symbols you pass as parameters to other macros. To see this effect, consider the following sample program:

```

// LocalDemo.HLA
//
// This program demonstrates the effect
// of passing a local macro symbol as a
// parameter to another macro. Remember,
// local macro symbols are text constants
// so HLA eager evaluates them when they
// appear as macro parameters.

program LocalExpansionDemo;

macro printIt( what );

    #print( @string:what )
    #print( what )

endmacro;

macro LocalDemo:local;

    ?local:string := "localStr";

    printIt( local );          // Eager evaluation, passes "_nnnn".
    printIt( #( local )# )    // Force deferred evaluation, passes "local".

endmacro;

begin LocalExpansionDemo;

    LocalDemo;

end LocalExpansionDemo;

```

Program 8.5 Local Macro Symbols as Macro Parameters

Inside *LocalDemo* HLA associates the unique symbol "_0001_" (or something similar) with the local symbol *local*. Next, HLA defines "_0001_" to be a string constant and associates the text "localStr" with this constant.

The first *printIt* macro invocation expands to "printIt(_0001_)" because HLA eagerly processes text constants in macro parameter lists (remember, local symbols are, effectively, text constants). Therefore, *printIt's* *what* parameter contains the text "_0001_" for this first invocation. Therefore, the first #PRINT statement prints this textual data ("_0001_") and the second print statement prints the value associated with "_0001_" which is "localStr".

The second *printIt* macro invocation inside the *LocalDemo* macro explicitly forces HLA to use deferred evaluation since it surrounds *local* with the "#(" and ")#" bracketing symbols. Therefore, HLA associates the text "local" with *printIt's* formal parameter rather than the expansion "_0001_". Inside *printIt*, the first #PRINT statement displays the text associated with the *what* parameter (which is "local" at this point). The second #PRINT statement expands *what* to produce "local". Since *local* is a currently defined text constant (defined within *LocalDemo* that invokes *printIt*), HLA expands this text constant to produce "_0001_". Since "_0001_" is a string constant, HLA prints the specified string ("localStr") during compilation. The complete output during compilation is

```
_0001_
localStr
local
localStr
```

Discussing the expansion of local symbols may seem like a lot of unnecessary detail. However, as your macros become more complex you may run into difficulties with your code based on the way HLA expands local symbols. Hence it is necessary to have a good grasp on how HLA processes these symbols.

Quick tip: if you ever need to generate a unique label in your program, you can use HLA local symbol facility to achieve this. Normally, you can only reference HLA's local symbols within the macro that defines the symbol. However, you can convert that local symbol to a string and process that string in your program as the following simple program demonstrates:

```
// UniqueSymbols.HLA
//
// This program demonstrates how to generate
// unique symbols in a program.

program UniqueSymsDemo;

macro unique:theSym;

    @string:theSym

endmacro;

begin UniqueSymsDemo;

    ?lbl:text := unique;

    jmp lbl;

lbl:

    ?@tostring:lbl :text := unique;
    jmp lbl;

lbl:

end UniqueSymsDemo;
```

Program 8.6 A Macro That Generates Unique Symbols for a Program

The first instance of *label:* in this program expands to "_0001_:" while the second instance of *label:* in this program expands to "_0003:". Of course, reusing symbols in this manner is horrible programming style (it's very confusing), but there are some cases you'll encounter when writing advanced macros where you will want to generate a unique symbol for use in your program. The *unique* macro in this program demonstrates exactly how to do this.

8.2.4 Macros as Compile-Time Procedures

Although programmers typically use macros to expand to some sequence of machine instructions, there is absolutely no requirement that a macro body contain any executable instructions. Indeed, many macros contain only compile-time language statements (e.g., #IF, #WHILE, "?" assignments, etc.). By placing only compile-time language statements in the body of a macro, you can effectively write compile-time procedures and functions using macros.

The *unique* macro from the previous section is a good example of a compile-time function that returns a string result. Consider, again, the definition of this macro:

```
#macro unique:theSym;

    @string:theSym

#endmacro;
```

Whenever your code references this macro, HLA replaces the macro invocation with the text "@string:theSym" which, of course, expands to some string like "_021F_". Therefore, you can think of this macro as a compile-time function that returns a string result.

Be careful that you don't take the function analogy too far. Remember, macros always expand to their body text at the point of invocation. Some expansions may not be legal at any arbitrary point in your programs. Fortunately, most compile-time statements are legal anywhere whitespace is legal in your programs. Therefore, macros generally behave as you would expect functions or procedures to behave during the execution of your compile-time programs.

Of course, the only difference between a procedure and a function is that a function returns some explicit value while procedures simply do some activity. There is no special syntax for specifying a compile-time function return value. As the example above indicates, simply specifying the value you wish to return as a statement in the macro body suffices. A compile-time procedure, on the other hand, would not contain any non-compile-time language statements that expand into some sort of data during macro invocation.

8.2.5 Multi-part (Context-Free) Macros

HLA's macro facilities, as described up to this point, are not particularly amazing. Indeed, most assemblers provide macro facilities very similar to those this chapter presents up to this point. Earlier, this chapter made the claim that HLA's macro facilities are quite a bit more powerful than those found in other assembly languages (or any programming language for that matter). Part of this power comes from the synergy that exists between the HLA compile-time language and HLA's macros. However, the one feature that sets HLA's macro facilities apart from all others is HLA's ability to handle multi-part, or context-free⁴, macros. This section describes this powerful feature.

4. The term "context-free" is an automata theory term used to describe constructs, like programming language control structures, that allow nesting.

The best way to introduce HLA's context-free macro facilities is via an example. Suppose you wanted to create a macro to define a new high level language statement in HLA (a very common use for macros). Let's say you wanted to create a statement like the following:

```
nLoop( 10 )
    << body >>
endloop;
```

The basic idea is that this code would execute the body of the loop ten times (or however many times the *nLoop* parameter specifies). A typical low-level implementation of this control structure might take the following form:

```
    mov( 10, ecx );
UniqueLabel:
    << body >>
    dec( ecx );
    jne UniqueLabel;
```

Clearly it will require two macros (*nLoop* and *endloop*) to implement this control structure. The first attempt a beginner might try is doomed to failure:

```
#macro nLoop( cnt );
    mov( cnt, ecx );
UniqueLabel:

#endmacro;

#macro endloop;
    dec( ecx );
    jne UniqueLabel;
#endmacro;
```

You've already seen the problem with this pair of macros: they use a global target label. Any attempt to use the *nLoop* macro more than once will result in a duplicate symbol error. Previously, we utilized HLA's local symbol facilities to overcome this problem. However, that approach will not work here because local symbols are local to a specific macro invocation; unfortunately, the *endloop* macro needs to reference *UniqueLabel* inside the *nLoop* invocation, so *UniqueLabel* cannot be a local symbol in this example.

A quick and dirty solution might be to take advantage of the trick employed by the *unique* macro appearing in previous sections. By utilizing a global text constant, you can share the label information across two macros using an implementation like the following:

```
#macro nLoop( cnt ):UniqueLabel;

    ?nLoop_target:string := @string:UniqueLabel;
    mov( cnt, ecx );
    UniqueLabel:

#endmacro;

#macro endloop;

    dec( ecx );
    jnz @text( nLoop_target );

#endmacro;
```

Using this definition, you can have multiple calls to the *nLoop* and *endloop* macros and HLA will not generate a duplicate symbol error:

```
nLoop( 10 )

    stdout.put( "Loop counter = ", ecx, nl );

endloop;

nLoop( 5 )

    stdout.put( "Second Loop Counter = ", ecx, nl );

endloop;
```

The macro invocations above produce something like the following (reasonably correct) expansion:

```
mov( 10, ecx );
_023A_:          //UniqueLabel, first invocation

    stdout.put( "Loop counter = ", ecx, nl );

    dec( ecx );
    jne _023A_;    // Expansion of nLoop_target becomes _023A_.

    mov( 5, ecx );
_023B_:          // UniqueLabel, second invocation.

    stdout.put( "Second Loop Counter = ", ecx, nl );

    dec( ecx );
    jnz _023B_;    // Expansion of nLoop_target becomes _023B_.
```

This scheme looks like it's working properly. However, this implementation suffers from a big drawback- it fails if you attempt to nest the *nLoop..endloop* control structure:

```
nLoop( 10 )

    push( ecx ); // Must preserve outer loop counter.
    nLoop( 5 )

        stdout.put( "ecx=", ecx, " [esp]=", (type dword [esp]), nl );

    endloop;
    pop( ecx ); // Restore outer loop counter.

endloop;
```

You would expect to see this code print its message 50 times. However, the macro invocations above produce code like the following:

```
mov( 10, ecx );
_0321_:          //UniqueLabel, first invocation

    push( ecx );
    mov( 5, ecx );
_0322_:

        stdout.put( "ecx=", ecx, " [esp]=", (type dword [esp]), nl );

    dec( ecx );
    jne _0322_;    // Expansion of nLoop_target becomes _0322_.
```

```

pop( ecx );
dec( ecx );
jne _0322_;      // Expansion of nLoop_target incorrectly becomes _0322_.

```

Note that the last JNE should jump to label "_0321_" rather than "_0322_". Unfortunately, the nested invocation of the *nLoop* macro overwrites the value of the global string constant *nLoop_target* thus the last JNE transfers control to the wrong label.

It is possible to correct this problem using an array of strings and another compile-time constant to create a *stack* of labels. By pushing and popping these labels as you encounter *nLoop* and *endloop* you can emit the correct code. However, this is a lot of work, is very inelegant, and you must repeat this process for every nestable control structure you dream up. In other words, it's a total kludge. Fortunately, HLA provides a better solution: multi-part macros.

Multi-part macros let you define a set of macros that work together. The *nLoop* and the *endloop* macros in this section are a good example of a pair of macros that work intimately together. By defining *nLoop* and *endloop* within a multi-part macro definition, the problems with communicating the target label between the two macros goes away because multi-part macros share parameters and local symbols. This provides a much more elegant solution to this problem than using global constants to hold target label information.

As its name suggests, a multi-part macro consists of a sequence of statements containing two matched macro names (e.g., *nLoop* and *endloop*). Multi-part macro invocations always consist of at least two macro invocations: a *beginning* invocation (e.g., *nLoop*) and a *terminating* invocation (e.g., *endloop*). Some number of unrelated (to the macro bodies) instructions may appear between the two invocations. To declare a multi-part macro, you use the following syntax:

```

#macro beginningMacro (optional_parameters) : optional_local_symbols;

    << beginningMacro body >>

#terminator terminatingMacro (optional_parameters) : optional_local_symbols;

    << terminatingMacro body >>

#endmacro;

```

The presence of the #TERMINATOR section in the macro declaration tells HLA that this is a multi-part macro declaration. It also ends the macro declaration of the beginning macro and begins the declaration of the terminating macro (i.e., the invocation of *beginningMacro* does not emit the code associated with the #TERMINATOR macro). As you would expect, parameters and local symbols are optional in both declarations and the associated glue characters (parentheses and colons) are not present if the parameters and local symbol lists are not present.

Now let's look at the multi-part macro declaration for the *nLoop..endloop* macro pair:

```

#macro nLoop( cnt ):TopOfLoop;

    mov( cnt, ecx );
    TopOfLoop:

#terminator endloop;

    dec( ecx );
    jne TopOfLoop;

#endmacro;

```

As you can see in this example, the definition of the *nLoop..endloop* control structure is much simpler when using multi-part macros; better still, multi-part macro declarations work even if you nest the invocations.

The most notable thing in this particular macro declaration is that the *endloop* macro has access to *nLoop*'s parameters and local symbols (in this example the *endloop* macro does not reference *cnt*, but it could if this was necessary). This makes communication between the two macros trivial.

Multi-part macro invocations must always occur in pairs. If the beginning macro appears in the text, the terminating macro must follow at some point. A terminating macro may never appear in the source file without a previous, matching, instance of the beginning macro. These semantics are identical to many of the HLA high level control structures; i.e., you cannot have an ENDIF without having a corresponding IF clause earlier in the source file.

When you nest multi-part macro invocations, HLA "magically" keeps track of local symbols and always emits the appropriate local label value. The nested macros appearing earlier are no problem for multi-part macros:

```
nLoop( 10 )

    push( ecx ); // Must preserve outer loop counter.
    nLoop( 5 )

        stdout.put( "ecx=", ecx, " [esp]=", (type dword [esp]), nl );

    endloop;
    pop( ecx ); // Restore outer loop counter.

endloop;
```

The above code properly compiles to something like:

```
    mov( 10, ecx );
_01FE_:

    push( ecx );
    mov( 5, ecx );
_01FF_:

    stdout.put( "ecx=", ecx, " [esp]=", (type dword [esp]), nl );

    dec( ecx );
    jne _01FF_;

    pop( ecx );

    dec( ecx );
    jne _01FE_;    // Note the correct label here.
```

In addition to terminating macros, HLA's multi-part macro facilities also provide an option for introducing additional macro declarations associated with the beginning/terminating macro pair: #KEYWORD macros. #KEYWORD macros are macros that are active only between a specific beginning and terminating macro pair. The classic use for #KEYWORD macros is to allow the introduction of context-sensitive keywords into the macro (context-sensitive, in this case, meaning that the terms are only active within the context of the body of statements between the beginning and terminating macros). Classic examples of statements that could employ these types of macros include the BREAK and CONTINUE statements within a loop body and the CASE clause within a SWITCH..ENDSWITCH statement.

The syntax for a multi-part macro declaration that includes one or more #KEYWORD macros is the following:

```
#macro beginningMacro( optional_parameters ): optional_local_labels;

    << beginningMacro Body >>

#keyword keywordMacro( optional_parameters ): optional_local_labels;
```

```

    << keywordMacro Body >>

#terminator terminatingMacro( optional_parameters ): optional_local_labels;

    << terminatingMacro Body >>

#endmacro;

```

If a `#KEYWORD` macro is present in a macro declaration there must also be a terminating macro declaration. You cannot have a `#KEYWORD` macro without a corresponding `#TERMINATOR` macro. The `#TERMINATOR` macro declaration is always last in a multi-part macro declaration.

The syntax example above specifies only a single `#KEYWORD` macro. HLA, however, allows zero or more `#KEYWORD` macro declarations in a multi-part macro. The HLA `SWITCH` statement, for example, defines two `#KEYWORD` macros, *case* and *default*.

`#KEYWORD` and `#TERMINATOR` macros may refer to the parameters and local symbols defined in the beginning macro, but they may not refer to locals and parameters in other `#KEYWORD` macros. Parameters and local symbols in `#KEYWORD` macro declarations are local to that specific macro. If you really need to communicate information between `#KEYWORD` and `#TERMINATOR` macros, define some local symbols in the beginning macro and assign these local symbols the parameter (or local symbol) values in the affected `#KEYWORD` macro. Then refer to this beginning macro local symbol in other parts of the macro. The following is a trivial example of this:

```

#macro ShareParameter:parmValue;

    << beginning macro body >>

#keyword ParmToShare( p );

    ?parmValue:text := @string:p;

    << keyword macro body >>

#terminator UsesSharedParm;

    mov( parmValue, ecx );

    << terminator macro body >>

#endmacro;

```

By assigning *ParmToShare*'s parameter value to the beginning macro's *parmValue* local symbol, this code makes the value of *p* accessible by the *UsesSharedParm* terminating macro.

This section only touches on the capabilities of HLA's multi-part macro facilities. Additional examples appear later in this chapter in the section on Domain Specific Embedded Languages (see "Domain Specific Embedded Languages" on page 1003). This text will make use of HLA's multi-part macros in later chapters as well. For more information on multi-part macros, see these sections in this text or check out the HLA documentation.

8.2.6 Simulating Function Overloading with Macros

The C++ language supports a nifty feature known as *function overloading*. Function overloading lets you write several different functions or procedures that all have the same name. The difference between these functions is the types of their parameters or the number of parameters. A procedure declaration is said to be unique if it has a different number of parameters than other functions with the same name or if the types of its parameters differs from another function with the same name. HLA does not directly support

procedure overloading but you can use macros to achieve the same result. This section explains how to use HLA's macros and the compile-time language to achieve function/procedure overloading.

One good use for procedure overloading is to reduce the number of standard library routines you must remember how to use. For example, the HLA Standard Library provides four different "puti" routines that output an integer value: *stdout.puti64*, *stdout.puti32*, *stdout.puti16*, and *stdout.puti8*. The different routines, as their name suggests, output integer values according to the size of their integer parameter. In the C++ language (or another other language supporting procedure/function overloading) the engineer designing the input routines would probably have chosen to name them all *stdout.puti* and leave it up to the compiler to select the appropriate one based on the operand size⁵. The following macro demonstrates how to do this in HLA using the compile-time language to figure out the size of the parameter operand:

```

// Puti.hla
//
// This program demonstrates procedure overloading via macros.
//
// It defines a "puti" macro that calls stdout.puti8, stdout.puti16,
// stdout.puti32, or stdout.puti64 depending on the size of the operand.

program putiDemo;
#include( "stdlib.hhf" )

// puti-
//
// Automatically decides whether we have a 64, 32, 16, or 8-bit
// operand and calls the appropriate stdout.putiX routine to
// output this value.

macro puti( operand );

    // If we have an eight-byte operand, call puti64:

    #if( @size( operand ) = 8 )

        stdout.puti64( operand );

    // If we have a four-byte operand, call puti32:

    #elseif( @size( operand ) = 4 )

        stdout.puti32( operand );

    // If we have a two-byte operand, call puti16:

    #elseif( @size( operand ) = 2 )

        stdout.puti16( operand );

    // If we have a one-byte operand, call puti8:

    #elseif( @size( operand ) = 1 )

```

5. By the way, the HLA Standard Library does this as well. Although it doesn't provide *stdout.puti*, it does provide *stdout.put* that will choose an appropriate output routine based upon the parameter's type. This is a bit more flexible than a *puti* routine.

```

        stdout.puti8( operand );

// If it's not an eight, four, two, or one-byte operand,
// then print an error message:

    #else

        #error( "Expected a 64, 32, 16, or 8-bit operand" )

    #endif

endmacro;

// Some sample variable declarations so we can test the macro above.

static
    i8:      int8      := -8;
    i16:     int16     := -16;
    i32:     int32     := -32;
    i64:     qword;

begin putiDemo;

    // Initialize i64 since we can't do this in the static section.

    mov( -64, (type dword i64 ) );
    mov( $FFFF_FFFF, (type dword i64[4]) );

    // Demo the puti macro:

    puti( i8 );  stdout.newln();
    puti( i16 ); stdout.newln();
    puti( i32 ); stdout.newln();
    puti( i64 ); stdout.newln();

end putiDemo;

```

Program 8.7 Simple Procedure Overloading Based on Operand Size

The example above simply tests the size of the operand to determine which output routine to use. You can use other HLA compile-time functions, like @TYPENAME, to do more sophisticated processing. Consider the following program that demonstrates a macro that overloads `stdout.puti32`, `stdout.putu32`, and `stdout.putd` depending on the type of the operand:

```

// put32.hla
//
// This program demonstrates procedure overloading via macros.
//
// It defines a "put32" macro that calls stdout.puti32, stdout.putu32,
// or stdout.putdw depending on the type of the operand.

```



```

program put32Demo;
#include( "stdlib.hhf" )

// put32-
//
// Automatically decides whether we have an int32, uns32, or dword
// operand and calls the appropriate stdout.putX routine to
// output this value.

macro put32( operand );

    // If we have an int32 operand, call puti32:

    #if( @typename( operand ) = "int32" )

        stdout.puti32( operand );

    // If we have an uns32 operand, call putu32:

    #elseif( @typename( operand ) = "uns32" )

        stdout.putu32( operand );

    // If we have a dword operand, call putidw:

    #elseif( @typename( operand ) = "dword" )

        stdout.putd( operand );

    // If it's not a 32-bit integer value, report an error:

    #else

        #error( "Expected an int32, uns32, or dword operand" )

    #endif

endmacro;

// Some sample variable declarations so we can test the macro above.

static
    i32:    int32    := -32;
    u32:    uns32    := 32;
    d32:    dword    := $32;

begin put32Demo;

    // Demo the put32 macro:

    put32( d32 ); stdout.newln();
    put32( u32 ); stdout.newln();
    put32( i32 ); stdout.newln();

```

```
end put32Demo;
```

Program 8.8 Procedure Overloading Based on Operand Type

You can easily extend the macro above to output eight and sixteen-bit operands as well as 32-bit operands. That is left as an exercise.

The number of actual parameters is another way to resolve which overloaded procedure to call. If you specify a variable number of macro parameters (using the "[]" syntax, see "Macros with a Variable Number of Parameters" on page 974) you can use the @ELEMENTS compile-time function to determine exactly how many parameters are present and call the appropriate routine. The following sample program uses this trick to determine whether it should call *stdout.puti32* or *stdout.puti32Size*:

```
// puti32.hla
//
// This program demonstrates procedure overloading via macros.
//
// It defines a "puti32" macro that calls stdout.puti32 or stdout.puti32size
// depending on the number of parameters present.

program puti32Demo;
#include( "stdlib.hhf" )

// puti32-
//
// Automatically decides whether we have an int32, uns32, or dword
// operand and calls the appropriate stdout.putX routine to
// output this value.

macro puti32( operand[] );

    // If we have a single operand, call stdout.puti32:

    #if( @elements( operand ) = 1 )

        stdout.puti32( @text(operand[0]) );

    // If we have two operands, call stdout.puti32size and
    // supply a default value of ' ' for the padding character:

    #elseif( @elements( operand ) = 2 )

        stdout.puti32Size( @text(operand[0]), @text(operand[1]), ' ' );

    // If we have three parameters, then pass all three of them
    // along to puti32size:

    #elseif( @elements( operand ) = 3 )

        stdout.puti32Size
        (
            @text(operand[0]),
```

```

        @text(operand[1]),
        @text(operand[2])
    );

    // If we don't have one, two, or three operands, report an error:

    #else

        #error( "Expected one, two, or three operands" )

    #endif

endmacro;

// A sample variable declaration so we can test the macro above.

static
    i32:    int32    := -32;

begin puti32Demo;

    // Demo the put32 macro:

    puti32( i32 ); stdout.newln();
    puti32( i32, 5 ); stdout.newln();
    puti32( i32, 5, '*' ); stdout.newln();

end puti32Demo;

```

Program 8.9 Using the Number of Parameters to Resolve Overloaded Procedures

All the examples up to this point provide procedure overloading for Standard Library routines (specifically, the integer output routines). Of course, you are not limited to overloading procedures in the HLA Standard Library. You can create your own overloaded procedures as well. All you've got to do is write a set of procedures, all with unique names, and then use a single macro to decide which routine to actually call based on the macro's parameters. Rather than call the individual routines, invoke the common macro and let it decide which procedure to actually call.

8.3 Writing Compile-Time "Programs"

The HLA compile-time language provides a powerful facility with which to write "programs" that execute while HLA is compiling your assembly language programs. Although it is possible to write some general purpose programs using the HLA compile-time language, the real purpose of the HLA compile-time language is to allow you to write short programs *that write other programs*. In particular, the primary purpose of the HLA compile-time language is to automate the creation of large or complex assembly language sequences. The following subsections provide some simple examples of such compile-time programs.

8.3.1 Constructing Data Tables at Compile Time

Earlier, this text suggested that you could write programs to generate large, complex, lookup tables for your assembly language programs (see “Generating Tables” on page 651). That chapter provided examples in HLA but suggested that writing a separate program was unnecessary. This is true, you can generate most look-up tables you’ll need using nothing more than the HLA compile-time language facilities. Indeed, filling in table entries is one of the principle uses of the HLA compile-time language. In this section we will take a look at using the HLA compile-time language to construct data tables during compilation.

In the section on generating tables, this text gave an example of an HLA program that writes a text file containing a lookup table for the trigonometric *sine* function. The table contains 360 entries with the index into the table specifying an angle in degrees. Each *int32* entry in the table contained the value $\sin(\text{angle}) * 1000$ where *angle* is equal to the index into the table. The section on generating tables suggested running this program and then including the text output from that program into the actual program that used the resulting table. You can avoid much of this work by using the compile-time language. The following HLA program includes a short compile-time code fragment that constructs this table of sines directly.

```

// demoSines.hla
//
// This program demonstrates how to create a lookup table
// of sine values using the HLA compile-time language.

program demoSines;
#include( "stdlib.hhf" )

const
    pi :real80 := 3.1415926535897;

readonly
    sines: int32[ 360 ] :=
        [
            // The following compile-time program generates
            // 359 entries (out of 360). For each entry
            // it computes the sine of the index into the
            // table and multiplies this result by 1000
            // in order to get a reasonable integer value.

            ?angle := 0;
            #while( angle < 359 )

                // Note: HLA's @sin function expects angles
                // in radians. radians = degrees*pi/180.
                // the "int32" function truncates its result,
                // so this function adds 1/2 as a weak attempt
                // to round the value up.

                int32( @sin( angle * pi / 180.0 ) * 1000 + 0.5 ),
                ?angle := angle + 1;

            #endwhile

            // Here's the 360th entry in the table. This code
            // handles the last entry specially because a comma
            // does not follow this entry in the table.

            int32( @sin( 359 * pi / 180.0 ) * 1000 + 0.5 )
        ]
];

```

```

begin demoSines;

    // Simple demo program that displays all the values in the table.

    for( mov( 0, ebx); ebx<360; inc( ebx )) do

        mov( sines[ ebx*4 ], eax );
        stdout.put
        (
            "sin( ",
            (type uns32 ebx ),
            " )*1000 = ",
            (type int32 eax ),
            nl
        );

    endfor;

end demoSines;

```

Program 8.10 Generating a SINE Lookup Table with the Compile-time Language

Another common use for the compile-time language is to build ASCII character lookup tables for use by the XLAT instruction at run-time. Common examples include lookup tables for alphabetic case manipulation. The following program demonstrates how to construct an upper case conversion table and a lower case conversion table⁶. Note the use of a macro as a compile-time procedure to reduce the complexity of the table generating code:

```

// demoCase.hla
//
// This program demonstrates how to create a lookup table
// of alphabetic case conversion values using the HLA
// compile-time language.

program demoCase;
#include( "stdlib.hhf" )

const
    pi :real80 := 3.1415926535897;

// emitCharRange-
//
// This macro emits a set of character entries
// for an array of characters. It emits a list
// of values (with a comma suffix on each value)
// from the starting value up to, but not including,
// the ending value.

```

6. Note that on modern processors, using a lookup table is probably not the most efficient way to convert between alphabetic cases. However, this is just an example of filling in the table using the compile-time language. The principles are correct even if the code is not exactly the best it could be.

```

macro emitCharRange( start, last ): index;

    ?index:uns8 := start;
    #while( index < last )

        char( index ),
        ?index := index + 1;

    #endwhile

endmacro;

readonly

// toUC:
// The entries in this table contain the value of the index
// into the table except for indicies #61..#7A (those entries
// whose indicies are the ASCII codes for the lower case
// characters). Those particular table entries contain the
// codes for the corresponding upper case alphabetic characters.
// If you use an ASCII character as an index into this table and
// fetch the specified byte at that location, you will effectively
// translate lower case characters to upper case characters and
// leave all other characters unaffected.

toUC: char[ 256 ] :=
    [
        // The following compile-time program generates
        // 255 entries (out of 256). For each entry
        // it computes toupper( index ) where index is
        // the character whose ASCII code is an index
        // into the table.

        emitCharRange( 0, uns8('a') )

        // Okay, we've generated all the entries up to
        // the start of the lower case characters. Output
        // Upper Case characters in place of the lower
        // case characters here.

        emitCharRange( uns8('A'), uns8('Z') + 1 )

        // Okay, emit the non-alphabetic characters
        // through to byte code #FE:

        emitCharRange( uns8('z') + 1, $FF )

        // Here's the last entry in the table. This code
        // handles the last entry specially because a comma
        // does not follow this entry in the table.

        #$FF
    ];

// The following table is very similar to the one above.
// You would use this one, however, to translate upper case
// characters to lower case while leaving everything else alone.
// See the comments in the previous table for more details.

```

```

Tolc:  char[ 256 ] :=
      [
        emitCharRange( 0, uns8('A') )
        emitCharRange( uns8('a'), uns8('z') + 1 )
        emitCharRange( uns8('Z') + 1, $FF )

        # $FF
      ];

begin demoCase;

  for( mov( uns32( ' ' ), eax ); eax <= $FF; inc( eax ) ) do

    mov( toUC[ eax ], bl );
    mov( Tolc[ eax ], bh );
    stdout.put
    (
      "toupper( '",
      (type char al),
      "' ) = '",
      (type char bl),
      "          tolower( '",
      (type char al),
      "' ) = '",
      (type char bh),
      "'",
      nl
    );

  endfor;

end demoCase;

```

Program 8.11 Generating Case Conversion Tables with the Compile-Time Language

One important thing to note about this sample is the fact that a semicolon does not follow the *emitCharRange* macro invocations. Macro invocations do not require a closing semicolon. Often, it is legal to go ahead and add one to the end of the macro invocation because HLA is normally very forgiving about having extra semicolons inserted into the code. In this case, however, the extra semicolons are illegal because they would appear between adjacent entries in the *Tolc* and *toUC* tables. Keep in mind that macro invocations don't require a semicolon, especially when using macro invocations as compile-time procedures.

8.3.2 Unrolling Loops

In the chapter on Low-Level Control Structures (see “Unraveling Loops” on page 800) this text points out that you can unravel loops to improve the performance of certain assembly language programs. One problem with unravelling, or unrolling, loops is that you may need to do a lot of extra typing, especially if many iterations are necessary. Fortunately, HLA's compile-time language facilities, especially the #WHILE loop, comes to the rescue. With a small amount of extra typing plus one copy of the loop body, you can unroll a loop as many times as you please.

If you simply want to repeat the same exact code sequence some number of times, unrolling the code is especially trivial. All you've got to do is wrap an HLA #WHILE..#ENDWHILE loop around the sequence

and count down a VAL object the specified number of times. For example, if you wanted to print "Hello World" ten times, you could encode this as follows:

```
?count := 0;
#while( count < 10 )

    stdout.put( "Hello World", nl );
    ?count := count + 1;

#endwhile
```

Although the code above looks very similar to a WHILE (or FOR) loop you could write in your program, remember the fundamental difference: the code above simply consists of ten straight *stdout.put* calls in the program. Were you to encode this using a FOR loop, there would be only one call to *stdout.put* and lots of additional logic to loop back and execute that single call ten times.

Unrolling loops becomes slightly more complicated if any instructions in that loop refer to the value of a loop control variable or other value that changes with each iteration of the loop. A typical example is a loop that zeros the elements of an integer array:

```
mov( 0, eax );
for( mov( 0, ebx ); ebx < 20; inc( ebx ) ) do

    mov( eax, array[ ebx*4 ] );

endfor;
```

In this code fragment the loop uses the value of the loop control variable (in EBX) to index into *array*. Simply copying "mov(eax, array[ebx*4]);" twenty times is not the proper way to unroll this loop. You must substitute an appropriate constant index in the range 0..76 (the corresponding loop indices, times four) in place of "EBX*4" in this example. Correctly unrolling this loop should produce the following code sequence:

```
mov( eax, array[ 0*4 ] );
mov( eax, array[ 1*4 ] );
mov( eax, array[ 2*4 ] );
mov( eax, array[ 3*4 ] );
mov( eax, array[ 4*4 ] );
mov( eax, array[ 5*4 ] );
mov( eax, array[ 6*4 ] );
mov( eax, array[ 7*4 ] );
mov( eax, array[ 8*4 ] );
mov( eax, array[ 9*4 ] );
mov( eax, array[ 10*4 ] );
mov( eax, array[ 11*4 ] );
mov( eax, array[ 12*4 ] );
mov( eax, array[ 13*4 ] );
mov( eax, array[ 14*4 ] );
mov( eax, array[ 15*4 ] );
mov( eax, array[ 16*4 ] );
mov( eax, array[ 17*4 ] );
mov( eax, array[ 18*4 ] );
mov( eax, array[ 19*4 ] );
```

You can do this more efficiently using the following compile-time code sequence:

```
?iteration := 0;
#while( iteration < 20 )

    mov( eax, array[ iteration*4 ] );
    ?iteration := iteration+1;
```



```
#endwhile
```

If the statements in a loop make use of the loop control variable's value, it is only possible to unroll such loops if those values are known at compile time. You cannot unroll loops when user input (or other run-time information) controls the number of iterations.

8.4 Using Macros in Different Source Files

Unlike procedures, macros do not have a fixed piece of code at some address in memory. Therefore, you cannot create "external" macros and link them with other modules in your program. However, it is very easy to share macros with different source files – just put the macros you wish to reuse in a header file and include that file using the `#include` directive. You can make the macro will be available to any source file you choose using this simple trick.

8.5 Putting It All Together

This chapter has barely touched on the capabilities of the HLA macro processor and compile-time language. The HLA language has one of the most powerful macro processors around. None of the other 80x86 assemblers even come close to HLA's capabilities with regard to macros. Indeed, if you could say just one thing about HLA in relation to other assemblers, it would have to be that HLA's macro facilities are, by far, the best.

The combination of the HLA compile-time language and the macro processor give HLA users the ability to extend the HLA language in many ways. In the chapter on Domain Specific Languages, you'll get the opportunity to see how to create your own specialized languages using HLA's macro facilities.

Even if you don't do exotic things like creating your own languages, HLA's macro facilities and compile-time language are really great for automating code generation in your programs. The HLA Standard Library, for example, makes heavy use of HLA's macro facilities; "procedures" like *stdout.put* and *stdin.get* would be very difficult (if not impossible) to create without the power of HLA macro facilities and the compile-time language. For some good examples of the possible complexity one can achieve with HLA's macros, you should scan through the `#include` files in the HLA Standard Library and look at some of the macros appearing therein.

This chapter serves as a basic introduction to HLA's macro facilities. As you use macros in your own programs you will gain even more insight into their power. So by all means, use macros as much as you can – they can help reduce the effort needed to develop programs.

Domain Specific Embedded Languages Chapter Nine

9.1 Chapter Overview

HLA's compile time language was designed with one purpose in mind: to give the HLA user the ability to change the syntax of the language in a user-defined manner. The compile-time language is actually so powerful that it lets you implement the syntax of other languages (not just an assembly language) within an HLA source file. This chapter discusses how to take this feature to an extreme and implement your own "mini-languages" within the HLA language.

9.2 Introduction to DSELS in HLA

One of the most interesting features of the HLA language is its ability to support *Domain Specific Embedded Languages* (or DSELS, for short, which you pronounce "D-cells"). A domain specific language is a language designed with a specific purpose in mind. Applications written in an appropriate domains specific language (DSL) are often much shorter and much easier to write than that same application written in a general purpose language (like C/C++, Java, or Pascal). Unfortunately, writing a compiler for a DSL is considerable work. Since most DSLs are so specific that few programs are ever written in them, it is generally cost-prohibitive to create a DSL for a given application. This economic fact has led to the popularity of domain specific *embedded* languages. The difference between a DSL and a DSEL is the fact that you don't write a new compiler for DSEL; instead, you provide some tools for use by an existing language translator to let the user extend the language as necessary for the specific application. This allows the language designer to use the features of the existing (i.e., *embedding*) language without having to write the translator for these features in the DSEL. The HLA language incorporates lots of features that let you extend the language to handle your own particular needs. This section discusses how to use these features to extend HLA as you choose.

As you probably suspect by now, the HLA compile-time language is the principle tool at your disposal for creating DSELS. HLA's multi-part macros let you easily create high level language-like control structures. If you need some new control structure that HLA does not directly support, it's generally an easy task to write a macro to implement that control structure. If you need something special, something that HLA's multi-part macros won't directly support, then you can write code in the HLA compile-time language to process portions of your source file as though they were simply string data. By using the compile-time string handling functions you can process the source code in just about any way you can imagine. While many such techniques are well beyond the scope of this text, it's reassuring to know that HLA can handle just about anything you want to do, even once you become an advanced assembly language programmer.

The following sections will demonstrate how to extend the HLA language using the compile-time language facilities. Don't get the idea that these simple examples push the limits of HLA's capabilities, they don't. You can accomplish quite a bit more with the HLA compile-time language; these examples must be fairly simple because of the assumed general knowledge level of the audience for this text.

9.2.1 Implementing the Standard HLA Control Structures

HLA supports a wide set of high level language-like control structures. These statements are not true assembly language statements, they are high level language statements that HLA compiles into the corresponding low-level machine instructions. They are general control statements, not "domain specific" (which is why HLA includes them) but they are quite typical of the types of statements one can add to HLA in order to extend the language. In this section we will look at how you could implement many of HLA's high-level control structures using the compile-time language. Although there is no real need to implement these state-

ments in this manner, their example should provide a template for implementing other types of control structures in HLA.

The following sections show how to implement the FOREVER..ENDFOR, WHILE..ENDWHILE, and IF..ELSEIF..ELSE..ENDIF statements. This text leaves the REPEAT..UNTIL and BEGIN..EXIT..EXITIF..END statements as exercises. The remaining high level language control structures (e.g., TRY..ENDTRY) are a little too complex to present at this point.

Because words like "if" and "while" are reserved by HLA, the following examples will use macro identifiers like "_if" and "_while". This will let us create recognizable statements using standard HLA identifiers (i.e., no conflicts with reserved words).

9.2.1.1 The FOREVER Loop

The FOREVER loop is probably the easiest control structure to implement. After all, the basic FOREVER loop simply consists of a label and a JMP instruction. So the first pass at implementing `_FOREVER.._ENDFOR` might look like the following:

```
#macro _forever: topOfLoop;
    topOfLoop:

#terminator _endfor;
    jmp topOfLoop;

#endmacro;
```

Unfortunately, there is a big problem with this simple implementation: you'll probably want the ability to exit the loop via `break` and `breakif` statements and you might want the equivalent of a `continue` and `continueif` statement as well. If you attempt to use the standard `BREAK`, `BREAKIF`, `CONTINUE`, and `CONTINUEIF` statements inside this `_forever` loop implementation, you'll quickly discover that they do not work. Those statements are valid only inside an HLA loop and the `_forever` macro above is not an HLA loop. Of course, we could easily solve this problem by defining `_FOREVER` thusly:

```
#macro _forever;
    forever

#terminator _endfor;
    endfor;

#endmacro;
```

Now you can use `BREAK`, `BREAKIF`, `CONTINUE`, and `CONTINUEIF` inside the `_forever.._endfor` statement. However, this solution is ridiculous. The purpose of this section is to show you how you could create this statement were it not present in the HLA language. Simply renaming `FOREVER` to `_forever` is not an interesting solution.

Probably the best way to implement these additional statements is via `KEYWORD` macros within the `_forever` macro. Not only is this easy to do, but it has the added benefit of not allowing the use of these statements outside a `_forever` loop.

Implementing a `_continue` statement is very easy. `Continue` must transfer control to the first statement at the top of the loop. Therefore, the `_continue` `#KEYWORD` macro will simply expand to a single `JMP` instruction that transfers control to the `topOfLoop` label. The complete implementation is the following:

```
keyword _continue;
    jmp topOfLoop;
```

Implementing `_continueif` is a little bit more difficult because this statement must evaluate a boolean expression and decide whether it must jump to the `topOfLoop` label. Fortunately, the HLA `JT` (jump if true) pseudo-instruction makes this a relatively trivial task. The `JT` pseudo-instruction expects a boolean expres-

sion (the same that CONTINUEIF allows) and transfers control to the corresponding target label if the result of the expression evaluation is true. The `_continueif` implementation is nearly trivial with JT:

```
keyword _continueif( ciExpr );
    JT( ciExpr ) topOfLoop;
```

You will implement the `_break` and `_breakif` #KEYWORD macros in a similar fashion. The only difference is that you must add a new label just beyond the JMP in the `_endfor` macro and the break statements should jump to this local label. The following program provides a complete implementation of the `_forever.._endfor` loop as well as a sample test program for the `_forever` loop.

```

/*****
/*
/* foreverMac.hla
/*
/* This program demonstrates how to use HLA's
/* "context-free" macros, along with the JT
/* "medium-level" instruction to create
/* the FOREVER..ENDFOR, BREAK, BREAKIF,
/* CONTINUE, and CONTINUEIF control statements.
/*
/*
/*****

program foreverDemo;
#include( "stdlib.hhf" )

    // Emulate the FOREVER..ENDFOR loop here, plus the
    // corresponding CONTINUE, CONTINUEIF, BREAK, and
    // BREAKIF statements.

macro _forever:foreverLbl, foreverbrk;

    // Target label for the top of the
    // loop. This is also the destination
    // for the _continue and _continueif
    // macros.

    foreverLbl:

    // The _continue and _continueif statements
    // transfer control to the label above whenever
    // they appear in a _forever.._endfor statement.
    // (Of course, _continueif only transfers control
    // if the corresponding boolean expression evaluates
    // true.)

keyword _continue;
    jmp foreverLbl;

keyword _continueif( cifExpr );
    jt( cifExpr ) foreverLbl;

```

```

// the _break and _breakif macros transfer
// control to the "foreverbrk" label which
// is at the bottom of the loop.

keyword _break;
    jmp foreverbrk;

keyword _breakif( bifExpr );
    jt( bifExpr ) foreverbrk;

// At the bottom of the _forever.._endfor
// loop this code must jump back to the
// label at the top of the loop. The
// _endfor terminating macro must also supply
// the target label for the _break and _breakif
// keyword macros:

terminator _endfor;
    jmp foreverLbl;
    foreverbrk:

endmacro;

begin foreverDemo;

// A simple main program that demonstrates the use of the
// statements above.

mov( 0, ebx );
_forever

    stdout.put( "Top of loop, ebx = ", (type uns32 ebx), nl );
    inc( ebx );

    // On first iteration, skip all further statements.
    _continueif( ebx = 1 );

    // On fourth iteration, stop.
    _breakif( ebx = 4 );

    _continue; // Always jumps to top of loop.
    _break;    // Never executes, just demonstrates use.

_endfor;

end foreverDemo;

```

Program 9.1 Macro Implementation of the FOREVER..ENDFOR Loop

9.2.1.2 The WHILE Loop

Once the FOREVER..ENDFOR loop is behind us, implementing other control structures like the WHILE..ENDWHILE loop is fairly easy. Indeed, the only notable thing about implementing the `_while.._endwhile` macros is that the code should implement this control structure as a REPEAT..UNTIL statement for efficiency reasons. The implementation appearing in this section takes a rather lazy approach to implementing the DO reserved word. The following code uses a #KEYWORD macro to implement a `_do` clause, but it does not enforce the (proper) use of this keyword. Instead, the code simply ignores the `_do` clause wherever it appears between the `_while` and `_endwhile`. Perhaps it would have been better to check for the presence of this statement (not too difficult to do) and verify that it immediately follows the `_while` clause and associated expression (somewhat difficult to do), but this just seems like a lot of work to check for the presence of an irrelevant keyword. So this implementation simply ignores the `_do`. The complete implementation appears in Program 9.2:

```

/*****
/*
/* whileMacs.hla
/*
/* This program demonstrates how to use HLA's
/* "context-free" macros, along with the JT and
/* JF "medium-level" instructions to create
/* the basic WHILE statement.
/*
/*
/*****

program whileDemo;
#include( "stdlib.hhf" )

    // Emulate the while..endwhile loop here.
    //
    // Note that this code implements the WHILE
    // loop as a REPEAT..UNTIL loop for efficiency
    // (though it inserts an extra jump so the
    // semantics remain the same as the WHILE loop).

macro _while( whlexpr ): repeatwhl, whltest, brkwhl;

    // Transfer control to the bottom of the loop
    // where the termination test takes place.

    jmp whltest;

    // Emit a label so we can jump back to the
    // top of the loop.

    repeatwhl:

    // Ignore the "_do" clause. Note that this
    // macro should really check to make sure
    // that "_do" follows the "_while" clause.
    // But it's not semantically important so
    // this code takes the lazy way out.

keyword _do;

```

```

// If we encounter "_break" inside this
// loop, transfer control to the first statement
// beyond the loop.

keyword _break;
    jmp brkwhl;

// Ditto for "_breakif" except, of course, we
// only exit the loop if the corresponding
// boolean expression evaluates true.

keyword _breakif( biwExpr );
    jt( biwExpr ) brkwhl;

// The "_continue" and "_continueif" statements
// should transfer control directly to the point
// where this loop tests for termination.

keyword _continue;
    jmp whltest;

keyword _continueif( ciwExpr );
    jt( ciwExpr ) whltest;

// The "_endwhile" clause does most of the work.
// First, it must emit the target label used by the
// "_while", "_continue", and "_continueif" clauses
// above. Then it must emit the code that tests the
// loop termination condition and transfers control
// to the top of the loop (the "repeatwhl" label)
// if the expression evaluates false. Finally,
// this code must emit the "brkwhl" label the "_break"
// and "_breakif" statements reference.

terminator _endwhile;

    whltest:
        jt( whlexpr ) repeatwhl;
        brkwhl:

endmacro;

begin whileDemo;

// Quick demo of the _while statement.
// Note that the _breakif in the nested
// _while statement only skips the
// inner-most _while, just as you should expect.

mov( 0, eax );
_while( eax < 10 ) _do

    stdout.put( "eax in loop = ", eax, " ebx=" );
    inc( eax );
    mov( 0, ebx );

```



```

_while( ebx < 4 ) _do

    stdout.puti32( ebx );
    _breakif( ebx = 3 );
    stdout.put( " , " );
    inc( ebx );

_endwhile;
stdout.newln();

_continueif( eax = 5 );
_breakif( eax = 8 );
_continue;
_break;

_endwhile

end whileDemo;

```

Program 9.2 Macro Implementation of the WHILE..ENDWHILE Loop

9.2.1.3 The IF Statement

Simulating the HLA IF.THEN.ELSEIF.ELSE.ENDIF statement using macros is a little bit more involved than the simulation of FOREVER or WHILE. The semantics of the ELSEIF and ELSE clauses complicate the code generation and require careful thought. While it is easy to write #KEYWORD macros for *_elseif* and *_else*, ensuring that these statements generate correct (and efficient) code is another matter altogether.

The basic *_if.._endif* statement, without the *_elseif* and *_else* clauses, is very easy to implement (even easier than the *_while.._endwhile* loop of the previous section). The complete implementation is

```

#macro _if( ifExpr ): onFalse;

    jf( ifExpr ) onFalse;

#keyword _then; // Just ignore _then.

#terminator _endif;

    onFalse:

#endmacro;

```

This macro generates code that tests the boolean expression you supply as a macro parameter. If the expression evaluates false, the code this macro emits immediately jumps to the point just beyond the *_endif* terminating macro. So this is a simple and elegant implementation of the IF..ENDIF statement, assuming you don't need an ELSE or ELSEIF clause.

Adding an ELSE clause to this statement introduces some difficulties. First of all, we need some way to emit the target label of the JF pseudo-instruction in the *_else* section if it is present and we need to emit this label in the terminator section if the *_else* section is not present.

A related problem is that the code after the *_if* clause must end with a JMP instruction that skips the *_else* section if it is present. This JMP must transfer control to the same location as the current *onFalse* label.

Another problem that occurs when we use #KEYWORD macros to implement the *_else* clause, is that we need some mechanism in place to ensure that at most one invocation of the *_else* macro appears in a given *_if.._endif* sequence.

We can easily solve these problems by introducing a compile-time variable (i.e., VAL object) into the macro. We will use this variable to indicate whether we've seen an *_else* section. This variable will tell us if we have more than one *_else* clause (which is an error) and it will tell us if we need to emit the onFalse label in the *_endif* macro. A reasonable implementation might be the following:

```
#macro _if( ifExpr ): onFalse, ifDone, hasElse;

    ?hasElse := False; // Haven't seen an _else clause yet.

    jf( ifExpr ) onFalse;

#keyword _then; // Just ignore _then.

#keyword _else;

// Check to see if this _if statement already has an _else clause:
#if( hasElse )

    #error( "Only one _else clause is legal in an _if statement" )

#endif

?hasElse := true; //Let the world know we've see an _else clause.

// Since we've just encountered the _else clause, we've just finished
// processing the statements in the _if section. The first thing we
// need to do is emit a JMP instruction that will skip around the
// _else statements (so the _if section doesn't fall in to the
// _else code).

    jmp ifDone;

// Okay, emit the onFalse label here so a false expression will transfer
// control to the _else statements:

onFalse:

#terminator _endif;

// If there was no _else section, we must emit the onFalse label
// so that the former JF instruction has a proper destination.
// If an _else section was present, we cannot emit this label
// (since the _else code has already done so) but we must emit
// the ifDone label.

#if( hasElse )

    ifdone:

#else

    onFalse:

#endif

#endmacro;
```

Adding the *_elseif* clause to the *_if.._endif* statement complicates things considerably. The problem is that *_elseif* can appear zero or more times in an *_if* statement and each occurrence needs to generate a unique *onFalse* label. Worse, if at least one *_elseif* clause appears in the sequence, then the JF instruction in the *_if* clause must transfer control to the first *_elseif*, not to the *_else* clause. Also, the last *_elseif* clause must transfer control to the *_else* clause (or to the first statement beyond the *_endif* clause) if its expression evaluates false. A straight-forward implementation just isn't going to work here.

A clever solution is to create a string variable that contains the name of the previous JF target label. Whenever you encounter an *_elseif* or an *_else* clause you simply emit this string to the source file as the target label. Then the only trick is "how do we generate a unique label whenever we need one?". Well, let's suppose that we have a string that is unique on each invocation of the *_if* macro. This being the case, we can generate a (source file wide) unique string by concatenating a counter value to the end of this base string. Each time we need a unique string, we simply bump the value of the counter up by one and create a new string. Consider the following macro:

```
#macro genLabel( base, number );

    @text( base + string( number ) );

#endmacro;
```

If the *base* parameter is a string value holding a valid HLA identifier and the *number* parameter is an integer numeric operand, then this macro will emit a valid HLA identifier that consists of the *base* string followed by a string representing the numeric constant. For example, 'genLabel("Hello", 52)' emits the label *Hello52*. Since we can easily create an *uns32* VAL object inside our *_if* macro and increment this each time we need a unique label, the only problem is to generate a unique base string on each invocation of the *_if* macro. Fortunately, HLA already does this for us.

Remember, HLA converts all local macro symbols to a unique identifier of the form "*_xxxx_*" where *xxxx* represents some four-digit hexadecimal value. Since local symbols are really nothing more than text constants initialized with these unique identifier strings, it's very easy to obtain a unique string in a macro invocation- just declare a local symbol (or use an existing local symbol) and apply the *@STRING:* operator to it to extract the unique name as a string. The following example demonstrates how to do this:

```
#macro uniqueIDs: counter, base;

    ?counter := 0;           // Increment this for each unique symbol you need.
    ?base := @string:base;  // base holds the base name to use.
    .
    .
    .

    // Generate a unique label at this point:

    genLabel( base, counter ): // Notice the colon. We're defining a
    ?counter := counter + 1;  // label at this point!
    .
    .
    .
    genLabel( base, counter ):
    ?counter := counter + 1;
    .
    .
    .
    etc.

#endmacro;
```

Once we have the capability to generate a sequence of unique labels throughout a macro, implementing the *_elseif* clause simply becomes the task of emitting the last referenced label at the beginning of each

`_elseif` (or `_else`) clause and jumping if false to the next unique label in the series. Program 9.3 implements the `_if.._then.._elseif.._else.._endif` statement using exactly this technique.

```

/*****
/*
/* IFmacs.hla
/*
/* This program demonstrates how to use HLA's
/* "context-free" macros, along with the JT and
/* JF "medium-level" instructions to create
/* an IF statement.
/*
/*
/*****

program IFDemo;
#include( "stdlib.hhf" )

    // genlabel-
    //
    // This macro creates an HLA-compatible
    // identifier of the form "_xxxx_n" where
    // "_xxxx_" is the string associated with
    // the "base" parameter and "n" represents
    // some numeric value that the caller. The
    // combination of the base and the n values
    // will produce a unique label in the
    // program if base's string is unique for
    // each invocation of the "_if" macro.

macro genLabel( base, number );

    @text( base + string( number ) )

endmacro;

/*
** Emulate the if..elseif..else..endif statement here.
*/

macro _if( ifexpr ):elseLbl, ifDone, hasElse, base;

    // This macro must create a unique ID string
    // in base. One sneaky way to do this is
    // to use the converted name HLA generates
    // for the "base" object (this is generally
    // a string of the form "_xxxx_" where "xxxx"
    // is a four-digit hexadecimal value).

    ?base := @string:base;

    // This macro may need to generate a large set
    // of different labels (one for each _elseif
    // clause). This macro uses the elseLbl
    // value, along with the value of "base" above,
    // to generate these unique labels.

    ?elseLbl := 0;

```

```

// hasElse determines if we have an _else clause
// present in this statement. This macro uses
// this value to determine if it must emit a
// final else label when it encounters _endif.

?hasElse := false;

// For an IF statement, we must evaluate the
// boolean expression and jump to the current
// else label if the expression evaluates false.

jf( ifexpr ) genLabel( base, elseLbl );

// Just ignore the _then keyword.
// A slightly better implementation would require
// this keyword, the current implementation lets
// you write an "_if" clause without the "_then"
// clause. For that matter, the current implementation
// lets you arbitrarily sprinkle "_then" clauses
// throughout the "_if" statement; we will ignore
// this for this example.

keyword _then;

// Handle the "_elseif" clause here.

keyword _elseif(elsex);

// _elseif clauses are illegal after
// an _else clause in the statement.
// Enforce that here.

#if( hasElse )

    #error( "Unexpected '_elseif' clause" )

#endif

// We've just finished the "_if" clause
// or a previous "_elseif" clause. So
// the first thing we have to do is jump
// to the code just beyond this "_if"
// statement.

jmp ifDone;

// Okay, this is where the previous "_if" or
// "_elseif" statement must jump if its boolean
// expression evaluates false. Emit the target
// label. Next, because we're about to jump
// to our own target label, bump up the elseLbl
// value by one to prevent jumping back to the
// label we're about to emit. Finally, emit
// the code that tests the boolean expression and
// transfers control to the next _elseif or _else
// clause if the result is false.

```

```

genLabel( base, elseLbl ):
    ?elseLbl := elseLbl+1;
    jf(elsex) genLabel( base, elseLbl );

keyword _else;

    // Only allow a single "_else" clause in this
    // "_if" statement:

    #if( hasElse )

        #error( "Unexpected '_else' clause" )

    #endif

    // As above, we've just finished the previous "_if"
    // or "_elseif" clause, so jump directly to the end
    // of the "_if" statement.

    jmp ifDone;

    // Okay, emit the current 'else' label so that
    // the failure of the previous "_if" or "_elseif"
    // test will transfer control here. Also set
    // 'hasElse' to true to catch additional "_elseif"
    // and "_else" clauses.

genLabel( base, elseLbl ):
    ?hasElse := true;

terminator _endif;

    // At the end of the _if statement we must emit the
    // destination label that the _if and _elseif sections
    // jump to. Also, if there was no _else section, this
    // code has to emit the last deployed else label.

    ifDone:
    #if( !hasElse )

        genLabel( base, elseLbl );

    #endif

endmacro;

begin IFDemo;

    // Quick demo of the use of the above statements.

for( mov( 0, eax ); eax < 5; inc( eax ) ) do

    _if( eax = 0 ) _then

        stdout.put( "in _if statement" nl );

```

```

    _elseif( eax = 1 ) _then

        stdout.put( "in first _elseif clause" nl );

    _elseif( eax = 2 ) _then

        stdout.put( "in second _elseif clause" nl );

    _else

        stdout.put( "in _else clause" nl );
        _if( eax > 3 ) _then

            stdout.put( "in second _if statement" nl );

        _endif;

    _endif;

endfor;

end IFDemo;

```

Program 9.3 Macro Implementation of the IF..ENDIF Statement

9.2.2 The HLA SWITCH/CASE Statement

HLA doesn't support a selection statement (SWITCH or CASE statement). Instead, HLA's SWITCH..CASE..DEFAULT..ENDSWITCH statement exists only as a macro in the HLA Standard Library HLL.HHF file. This section discusses HLA's macro implementation of the SWITCH statement.

The SWITCH statement is very complex so it should come as no surprise that the macro implementation is long, involved, and complex. The example appearing in this section is slightly simplified over the standard HLA version, but not by much. This discussion assumes that you're familiar with the low-level implementation of the SWITCH..CASE..DEFAULT..ENDSWITCH statement. If you are not comfortable with that implementation, or feel a little rusty, you may want to take another look at "SWITCH/CASE Statements" on page 776 before attempting to read this section. The discussion in this section is somewhat advanced and assumes a fair amount of programming skill. If you have trouble following this discussion, you may want to skip this section until you gain some more experience.

There are several different ways to implement a SWITCH statement. In this section we will assume that the *_switch.._endswitch* macro we are writing will implement the SWITCH statement using a jump table. Implementation as a sequence of *if..elseif* statements is fairly trivial and is left as an exercise. Other schemes are possible as well, this section will not consider them.

A typical SWITCH statement implementation might look like the following:

```

readonly
    JumpTbl:dword[3] := [ &stmt5, &stmt6, &stmt7 ];
    .
    .
    .

```

```

// switch( i )

mov( i, eax );           // Check to see if "i" is outside the range
cmp( eax, 5 );           // 5..7 and transfer control directly to the
jb EndCase               // DEFAULT case if it is.
cmp( eax, 7 );
ja EndCase;
jmp( JumpTbl[ eax*4 - 5*@size(dword)] );

// case( 5 )
    Stmt5:
        stdout.put( "I=5" );
        jmp EndCase;

// Case( 6 )
    Stmt6:
        stdout.put( "I=6" );
        jmp EndCase;

// Case( 7 )
    Stmt7:
        stdout.put( "I=7" );

EndCase:

```

If you study this code carefully, with an eye to writing a macro to implement this statement, you'll discover a couple of major problems. First of all, it is exceedingly difficult to determine how many cases and the range of values those cases cover before actually processing each CASE in the SWITCH statement. Therefore, it is really difficult to emit the range check (for values outside the range 5..7) and the indirect jump before processing all the cases in the SWITCH statement. You can easily solve this problem, however, by moving the checks and the indirect jump to the bottom of the code and inserting a couple of extra JMP instructions. This produces the following implementation:

```

readonly
    JumpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];
    .
    .
    .

// switch( i )

    jmp DoSwitch;           // First jump inserted into this code.

// case( 5 )
    Stmt5:
        stdout.put( "I=5" );
        jmp EndCase;

// Case( 6 )
    Stmt6:
        stdout.put( "I=6" );
        jmp EndCase;

// Case( 7 )
    Stmt7:
        stdout.put( "I=7" );
        jmp EndCase;           // Second jump inserted into this code.

DoSwitch:                   // Insert this label and move the range
    mov( i, eax );           // checks and indirect jump down here.

```



```

    cmp( eax, 5 );
    jb EndCase
    cmp( eax, 7 );
    ja EndCase;
    jmp( JumpTbl[ eax*4 - 5*@size(dword)] );

// All the cases (including the default case) jump down here:

EndCase:

```

Since the range check code appears after all the cases, the macro can now process those cases and easily determine the bounds on the cases by the time it must emit the CMP instructions above that check the bounds of the SWITCH value. However, this implementation still has a problem. The entries in the *JumpTbl* table refer to labels that can only be determined by first processing all the cases in the SWITCH statement. Therefore, a macro cannot emit this table in a READONLY section that appears earlier in the source file than the SWITCH statement. Fortunately, HLA lets you embed data in the middle of the code section using the READONLY..ENDREADONLY and STATIC..ENDSTATIC directives¹. Taking advantage of this feature allows use to rewrite the SWITCH implementation as follows:

```

// switch( i )

    jmp DoSwitch;                // First jump inserted into this code.

// case( 5 )
    Stmt5:
        stdout.put( "I=5" );
        jmp EndCase;

// Case( 6 )
    Stmt6:
        stdout.put( "I=6" );
        jmp EndCase;

// Case( 7 )
    Stmt7:
        stdout.put( "I=7" );
        jmp EndCase;           // Second jump inserted into this code.

DoSwitch:                        // Insert this label and move the range
    mov( i, eax );                // checks and indirect jump down here.
    cmp( eax, 5 );
    jb EndCase
    cmp( eax, 7 );
    ja EndCase;
    jmp( JumpTbl[ eax*4 - 5*@size(dword)] );

// All the cases (including the default case) jump down here:

EndCase:

readonly
    JumpTbl:dword[3] := [ &Stmt5, &Stmt6, &Stmt7 ];
endreadonly;

```

HLA's macros can produce code like this when processing a SWITCH macro. So this is the type of code we will generate with a *_switch.._case.._default.._endswitch* macro.

Since we're going to need to know the minimum and maximum case values (in order to generate the appropriate operands for the CMP instructions above), the *_case #KEYWORD* macro needs to compare the

1. HLA actually moves the data to the appropriate segment in memory, the data is not stored directly in the CODE section.

current case value(s) against the global minimum and maximum case values for all cases. If the current case value is less than the global minimum or greater than the global maximum, then the `_case` macro must update these global values accordingly. The `_endswitch` macro will use these global minimum and maximum values in the two CMP instructions it generates for the range checking sequence.

For each case value appearing in a `_switch` statement, the `_case` macros must save the case value and an identifying label for that case value. This is necessary so that the `_endswitch` macro can generate the jump table. What is really needed is an arbitrary list of records, each record containing a value field and a label field. Unfortunately, the HLA compile-time language does not support arbitrary lists of objects, so we will have to implement the list using a (fixed size) array of record constants. The record declaration will take the following form:

```
caseRecord:
    record
        value:uns32;
        label:uns32;
    endrecord;
```

The `value` field will hold the current case value. The `label` field will hold a unique integer value for the corresponding `_case` that the macros can use to generate statement labels. The implementation of the `_switch` macro in this section will use a variant of the trick found in the section on the `_if` macro; it will convert a local macro symbol to a string and append an integer value to the end of that string to create a unique label. The integer value appended will be the value of the `label` field in the `caseRecord` list.

Processing the `_case` macro becomes fairly easy at this point. All the `_case` macro has to do is create an entry in the `caseRecord` list, bump a few counters, and emit an appropriate case label prior to the code emission. The implementation in this section uses Pascal semantics, so all but the first case in the `_switch.._endswitch` statement must first emit a jump to the statement following the `_endswitch` so the previous case's code doesn't fall into the current case.

The real work in implementing the `_switch.._endswitch` statement lies in the generation of the jump table. First of all, there is no requirement that the cases appear in ascending order in the `_switch.._endswitch` statement. However, the entries in the jump table must appear in ascending order. Second, there is no requirement that the cases in the `_switch.._endswitch` statement be consecutive. Yet the entries in the jump table must be consecutive case values². The code that emits the jump table must handle these inconsistencies.

The first task is to sort the entries in the `caseRecord` list in ascending order. This is easily accomplished by writing a little `SortCases` macro to sort all the `caseRecord` entries once the `_switch.._endswitch` macro has processed all the cases. `SortCases` doesn't have to be fancy. In fact, a bubblesort algorithm is perfect for this because:

- Bubble sort is easy to implement
- Bubble sort is efficient when sorting small lists and most SWITCH statements only have a few cases.
- Bubble sort is especially efficient on nearly sorted data and most programmers put their cases in ascending order.

After sorting the cases, only one problem remains: there may be gaps in the case values. This problem is easily handled by stepping through the `caseRecord` elements one by one and synthesizing consecutive entries whenever a gap appears in the list. Program 9.4 provides the full `_switch.._case.._default.._endswitch` macro implementation.

```

/*****
/*
/* switch.hla-
/*
/*****/
```

2. Of course, if there are gaps in the case values, the jump table entries for the missing items should contain the address of the default case.

```

/* This program demonstrates how to implement the */
/* _switch.._case.._default.._endswitch statement */
/* using macros. */
/* */
/*****/

program demoSwitch;
#include( "stdlib.hhf" )

const

    // Because this code uses an array to implement
    // the caseRecord list, we have to specify a fixed
    // number of cases. The following constant defines
    // the maximum number of possible cases in a
    // _switch statement.

    maxCases := 256;

type

    // The following data type hold the case value
    // and statement label information for each
    // case appearing in a _switch statement.

    caseRecord:
        record

            value:uns32;
            lbl:uns32;

        endrecord;

    // SortCases
    //
    // This routine does a bubble sort on an array
    // of caseRecord objects. It sorts in ascending
    // order using the "value" field as the key.
    //
    // This is a good old fashioned bubble sort which
    // turns out to be very efficient because:
    //
    // (1) The list of cases is usually quite small, and
    // (2) The data is usually already sorted (or mostly sorted).

macro SortCases( sort_array, sort_size ):
    sort_i,
    sort_bnd,
    sort_didswap,
    sort_temp;

    ?sort_bnd := sort_size - 1;
    ?sort_didswap := true;
    #while( sort_didswap )

        ?sort_didswap := false;
        ?sort_i := 0;
        #while( sort_i < sort_bnd )

```

```

        #if
        (
            sort_array[sort_i].value >
                sort_array[sort_i+1].value
        )

            ?sort_temp := sort_array[sort_i];
            ?sort_array[sort_i] := sort_array[sort_i+1];
            ?sort_array[sort_i+1] := sort_temp;
            ?sort_didswap := true;

        #elseif
        (
            sort_array[sort_i].value =
                sort_array[sort_i+1].value
        )

            #error
            (
                "Two cases have the same value: (" +
                string( sort_array[sort_i].value ) +
                ")"
            )

        #endif
        ?sort_i := sort_i + 1;

    #endwhile
    ?sort_bnd := sort_bnd - 1;

#endwhile;

endmacro;

// HLA Macro to implement a C SWITCH statement (using
// Pascal semantics). Note that the switch parameter
// must be a 32-bit register.

macro _switch( switch_reg ):
    switch_minval,
    switch_maxval,
    switch_otherwise,
    switch_endcase,
    switch_jmptbl,
    switch_cases,
    switch_caseIndex,
    switch_doCase,
    switch_hasotherwise;           // Just used to generate unique names.

// Verify that we have a register operand.

```

```

#if( !@isReg32( switch_reg ) )

    #error( "Switch operand must be a 32-bit register" )

#endif

// Create the switch_cases array.  Allow, at most, 256 cases.

?switch_cases:caseRecord[ maxCases ];

// General initialization for processing cases.

?switch_caseIndex := 0;           // Index into switch_cases array.
?switch_minval := $FFFF_FFFF;    // Minimum case value.
?switch_maxval := 0;             // Maximum case value.
?switch_hasotherwise := false;   // Determines if DEFAULT section present.

// We need to process the cases to collect information like
// switch_minval prior to emitting the indirect jump.  So move the
// indirect jump to the bottom of the case statement.

jmp switch_doCase;

// "case" keyword macro handles each of the cases in the
// case statement.  Note that this syntax allows you to
// specify several cases in the same _case macro, e.g.,
// _case( 2, 3, 4 ).  Such a situation tells this macro
// that these three values all execute the same code.

keyword _case( switch_parms[] ):
    switch_parmIndex,
    switch_parmCount,
    switch_constant;

?switch_parmCount:uns32;
?switch_parmCount := @elements( switch_parms );

#if( switch_parmCount <= 0 )

    #error( "Must have at least one case value" );
    ?switch_parms:uns32[1] := [0];

#endif

// If we have at least one case already, terminate
// the previous case by transferring control to the
// first statement after the endcase macro.  Note
// that these semantics match Pascal's CASE statement,
// not C/C++'s SWITCH statement which would simply
// fall through to the next CASE.

#if( switch_caseIndex <> 0 )

    jmp switch_endcase;

#endif

// The following loop processes each case value

```

```

// supplied to the _case macro.

?switch_parmIndex:uns32;
?switch_parmIndex := 0;
#while( switch_parmIndex < switch_parmCount )

    ?switch_constant: uns32;
    ?switch_constant: uns32 :=
        uns32( @text( switch_parms[ switch_parmIndex ]));

    // Update minimum and maximum values based on the
    // current case value.

    #if( switch_constant < switch_minval )

        ?switch_minval := switch_constant;

    #endif
    #if( switch_constant > switch_maxval )

        ?switch_maxval := switch_constant;

    #endif

    // Emit a unique label to the source code for this case:

    @text
    (
        "_case"
        + @string:switch_caseIndex
        + string( switch_caseIndex )
    ):

    // Save away the case label and the case value so we
    // can build the jump table later on.

    ?switch_cases[ switch_caseIndex ].value := switch_constant;
    ?switch_cases[ switch_caseIndex ].lbl := switch_caseIndex;

    // Bump switch_caseIndex value because we've just processed
    // another case.

    ?switch_caseIndex := switch_caseIndex + 1;
    #if( switch_caseIndex >= maxCases )

        #error( "Too many cases in statement" );

    #endif

    ?switch_parmIndex := switch_parmIndex + 1;

#endwhile

// Handle the default keyword/macro here.

keyword _default;

// If there was not a preceding case, this is an error.
// If so, emit a jmp instruction to skip over the

```

```

// default case.

#if( switch_caseIndex < 1 )

    #error( "Must have at least one case" );

#endif

    jmp switch_endcase;

// Emit the label for this default case and set the
// switch_hasotherwise flag to true.

switch_otherwise:
?switch_hasotherwise := true;

// The endswitch terminator/macro checks to see if
// this is a reasonable switch statement and emits
// the jump table code if it is.

terminator _endswitch:
switch_i_,
switch_j_,
switch_curCase_;

// If the difference between the smallest and
// largest case values is great, the jump table
// is going to be fairly large.  If the difference
// between these two values is greater than 256 but
// less than 1024, warn the user that the table will
// be large.  If it's greater than 1024, generate
// an error.
//
// Note: these are arbitrary limits.  Feel free to
// adjust them if you like.

#if( (switch_maxval - switch_minval) > 256 )

    #if( (switch_maxval - switch_minval) > 1024 )

        // Perhaps in the future, this macro could
        // switch to generating an if..elseif..elseif...
        // chain if the range between the values is
        // too great.

        #error( "Range of cases is too great" );

    #else

        #print( "Warning: Range of cases is large" );

    #endif

#endif

// Table emission algorithm requires that the switch_cases
// array be sorted by the case values.

```

```

SortCases( switch_cases, switch_caseIndex );

// Build a string of the form:
//
//      switch_jmptbl:dword[ xx ] := [&case1, &case2, &case3...&casen];
//
// so we can output the jump table.

readonly

switch_jmptbl:dword[ switch_maxval - switch_minval + 2] := [

?switch_i_ := 0;
#while( switch_i_ < switch_caseIndex )

    ?switch_curCase_ := switch_cases[ switch_i_ ].value;
    // Emit the label associated with the current case:

    @text
    (
        "&"
        + "_case"
        + @string:switch_caseIndex
        + string( switch_cases[ switch_i_ ].lbl )
        + ", "
    )

    // Emit "&switch_otherwise" table entries for any gaps present
    // in the table:

    ?switch_j_ := switch_cases[ switch_i_ + 1 ].value;
    ?switch_curCase_ := switch_curCase_ + 1;

    #while( switch_curCase_ < switch_j_ )

        &switch_otherwise,
        ?switch_curCase_ := switch_curCase_ + 1;

    #endwhile
    ?switch_i_ := switch_i_ + 1;

#endwhile

// Emit a dummy entry to terminate the table:

&switch_otherwise];

endreadonly;

#if( switch_caseIndex < 1 )

    #error( "Must have at least one case" );

#endif

// After the default case, or after the last
// case entry, jump over the code that does
// the conditional jump.

```



```

        jmp switch_endcase;

// Okay, here's the code that does the conditional jump.

switch_doCase:

    // If the minimum case value is zero, we don't
    // need to emit a CMP instruction for it.

    #if( switch_minval <> 0 )

        cmp( switch_reg, switch_minval );
        jb switch_otherwise;

    #endif
    cmp( switch_reg, switch_maxval );
    ja switch_otherwise;
    jmp( switch_jmptbl[ switch_reg*4 - switch_minval*4 ] );

// If there was no default case, transfer control
// to the first statement after the "endcase" clause.

#if( !switch_hasotherwise )

    switch_otherwise:

#endif

// When each of the cases complete execution,
// transfer control down here.

switch_endcase:

// The following statement deallocates the storage
// associated with the switch_cases array (this saves
// memory at compile time, it does not affect the
// execution of the resulting machine code).

?switch_cases := 0;

endmacro;

begin demoSwitch;

// A simple demonstration of the _switch.._endswitch statement:

for( mov( 0, eax ); eax < 8; inc( eax )) do

    _switch( eax )

        _case( 0 )

            stdout.put( "eax = 0" nl );

        _case( 1, 2 )

```

```

        stdout.put( "eax = 1 or 2" nl );

    _case( 3, 4, 5 )

        stdout.put( "eax = 3, 4, or 5" nl );

    _case( 6 )

        stdout.put( "eax = 6" nl );

    _default

        stdout.put( "eax is not in the range 0-6" nl );

    _endswitch;

endfor;

end demoSwitch;

```

Program 9.4 Macro Implementation of the SWITCH..ENDSWITCH Statement

9.2.3 A Modified WHILE Loop

The previous sections have shown you how to implement statements that are already available in HLA or the HLA Standard Library. While this approach lets you work with familiar statements that you should be comfortable with, it doesn't really demonstrate that you can create *new* control statements with HLA's compile-time language. In this section you will see how to create a variant of the WHILE statement that is not simply a rehash of HLA's WHILE statement. This should amply demonstrate that there are some useful control structures that HLA (and high level languages) don't provide and that you can easily use HLA compile-time language to implement specialized control structures as needed.

A common use of a WHILE loop is to search through a list and stop upon encountering some desired value or upon hitting the end of the list. A typical HLA example might take the following form:

```

while( <<There are more items in the list>> ) do

    breakif( <<This was the item we're looking for>> );
    << select the next item in the list>>

endwhile;

```

The problem with this approach is that when the statement immediately following the ENDWHILE executes, that code doesn't know whether the loop terminated because it found the desired value or because it exhausted the list. The typical solution is to test to see if the loop exhausted the list and deal with that accordingly:

```

while( <<There are more items in the list>> ) do

    breakif( <<This was the item we're looking for>> );
    << select the next item in the list>>

```

```

endwhile;
if( <<The list wasn't exhausted>> ) then

    << do something with the item we found >>

endif;

```

The problem with this "solution" should be obvious if you think about it a moment. We've already tested to see if the loop is empty, immediately after leaving the loop we repeat this same test. This is somewhat inefficient. A better solution would be to have something like an "else" clause in the WHILE loop that executes if you break out of the loop and doesn't execute if the loop terminates because the boolean expression evaluated false. Rather than use the keyword ELSE, let's invent a new (more readable) term: *onbreak*. The ONBREAK section of a WHILE loop executes (only once) if a BREAK or BREAKIF statement was the reason for the loop termination. With this ONBREAK clause, you could recode the previous WHILE loop a little bit more elegantly as follows:

```

while( <<There are more items in the list>> ) do

    breakif( <<This was the item we're looking for>> );
    << select the next item in the list>>

    onbreak

    << do something with the item we found >>

endwhile;

```

Note that if the ONBREAK clause is present, the WHILE's loop body ends at the ONBREAK keyword. The ONBREAK clause executes at most once per execution of this WHILE statement.

Implementing a `_while.._onbreak.._endwhile` statement is very easy using HLA's multi-part macros. Program 9.5 provides the complete implementation of this statement:

```

/*****
/*
/* while.hla
/*
/* This program demonstrates a variant of the
/* WHILE loop that provides a special "onbreak"
/* clause. The _onbreak clause executes if the
/* program executes a _break clause or it executes
/* a _breakif clause and the corresponding
/* boolean expression evaluates true. The _onbreak
/* section does not execute if the loop terminates
/* due to the _while boolean expression evaluating
/* false.
/*
/*
*****/

program Demo_while;
#include( "stdlib.hhf" )

// _while semantics:
//
// _while( expr )
//
//     << stmts including optional _break, _breakif
//         _continue, and _continueif statements >>
//

```

```

//  _onbreak // This section is optional.
//
//  << stmts that only execute if program executes
//      a _break or _breakif (with true expression)
//      statement. >>
//
//  _endwhile;

macro _while( expr ):falseLbl, breakLbl, topOfLoop, hasOnBreak;

    // hasOnBreak keeps track of whether we've seen an _onbreak
    // section.
    ?hasOnBreak:boolean:=false;

    // Here's the top of the WHILE loop.
    // Implement this as a straight-forward WHILE (test for
    // loop termination at the top of the loop).

    topOfLoop:
        jf( expr ) falseLbl;

    // Ignore the _do keyword.

    keyword _do;

    // _continue and _continueif (with a true expression)
    // transfer control to the top of the loop where the
    // _while code retests the loop termination condition.

    keyword _continue;
        jmp topOfLoop;

    keyword _continueif( expr1 );
        jt( expr1 ) topOfLoop;

    // Unlike the _break or _breakif in a standard WHILE
    // statement, we don't immediately exit the WHILE.
    // Instead, this code transfers control to the optional
    // _onbreak section if it is present.  If it is not
    // present, control transfers to the first statement
    // beyond the _endwhile.

    keyword _break;
        jmp breakLbl;

    keyword _breakif( expr2 );
        jt( expr2 ) breakLbl;

    // If we encounter an _onbreak section, this marks
    // the end of the while loop body.  Emit a jump that
    // transfers control back to the top of the loop.
    // This code also has to verify that there is only
    // one _onbreak section present.  Any code following
    // this clause is going to execute only if the _break
    // or _breakif statements execute and transfer control
    // down here.

    keyword _onbreak;

```

```

#if( hasOnBreak )

    #error( "Extra _onbreak clause encountered" )

#else

    jmp topOfLoop;
    ?hasOnBreak := true;

    breakLbl:

#endif

terminator _endwhile;

// If we didn't have an _onbreak section, then
// this is the bottom of the _while loop body.
// Emit the jump to the top of the loop and emit
// the "breakLbl" label so the execution of a
// _break or _breakif transfers control down here.

#if( !hasOnBreak )

    jmp topOfLoop;
    breakLbl:

#endif
falseLbl:

endmacro;

static
    i:int32;

begin Demo_while;

    // Demonstration of standard while loop

    mov( 0, i );
    _while( i < 10 ) _do

        stdout.put( "1: i=", i, nl );
        inc( i );

    _endwhile;

    // Demonstration with BREAKIF:

    mov( 5, i );
    _while( i < 10 ) _do

        stdout.put( "2: i=", i, nl );
        _breakif( i = 7 );
        inc( i );

    _endwhile

    // Demonstration with _BREAKIF and _ONBREAK:

    mov( 0, i );

```

```

_while( i < 10 ) _do

    stdout.put( "3: i=", i, nl );
    _breakif( i = 4 );
    inc( i );

_onbreak

    stdout.put( "Breakif was true at i=", i, nl );

_endwhile
stdout.put( "All Done" nl );

end Demo_while;

```

Program 9.5 The Implementation of `_while.._onbreak.._endwhile`

9.2.4 A Modified IF..ELSE..ENDIF Statement

The IF statement is another statement that doesn't always do exactly what you want. Like the `_while.._onbreak.._endwhile` example above, it's quite possible to redefine the IF statement so that it behaves the way we want it to. In this section you'll see how to implement a variant of the IF..ELSE..ENDIF statement that nests differently than the standard IF statement.

It is possible to simulate short-circuit boolean evaluation involving conjunction and disjunction without using the "&&" and "||" operators if you carefully structure your code. Consider the following example:

```

// "C" code employing logical-AND operator:

if( expr1 && expr2 )
{
    << statements >>
}

// Equivalent HLA version:

if( expr1 ) then

    if( expr2 ) then

        << statements >>

    endif;

endif;

```

In both cases ("C" and HLA) the `<< statements >>` block executes only if both `expr1` and `expr2` evaluate true. So other than the extra typing involved, it is often very easy to simulate logical conjunction by using two IF statements in HLA.

There is one very big problem with this scheme. Consider what happens if you modify the "C" code to be the following:

```

// "C" code employing logical-AND operator:

if( expr1 && expr2 )
{

```

```

    << 'true' statements >>
  }
else
{
    << 'false' statements >>
}

```

Before describing how to create this new type of IF statement, we must digress for a moment and explore an interesting feature of HLA's multi-part macro expansion: #KEYWORD macros do not have to use unique names. Whenever you declare an HLA #KEYWORD macro, HLA accepts whatever name you choose. If that name happens to be already defined, then the #KEYWORD macro name takes precedence as long as the macro is active (that is, from the point you invoke the macro name until HLA encounters the #TERMINATOR macro). Therefore, the #KEYWORD macro name hides the previous definition of that name until the termination of the macro. This feature applies even to the original macro name; that is, it is possible to define a #KEYWORD macro with the same name as the original macro to which the #KEYWORD macro belongs. This is a very useful feature because it allows you to change the definition of the macro within the scope of the opening and terminating invocations of the macro.

Although not pertinent to the IF statement we are constructing, you should note that parameter and local symbols in a macro also override any previously defined symbols of the same name. So if you use that symbol between the opening macro and the terminating macro, you will get the value of the local symbol, not the global symbol. E.g.,

```

var
    i:int32;
    j:int32;
    .
    .
    .
#macro abc:i;
    ?i:text := "j";
    .
    .
#terminator xyz;
    .
    .
#endmacro;
    .
    .
    .
    mov( 25, i );
    mov( 10, j );
    abc
        mov( i, eax ); // Loads j's value (10), not 25 into eax.
    xyz;

```

The code above loads 10 into EAX because the "mov(i, eax);" instruction appears between the opening and terminating macros *abc..xyz*. Between those two macros the local definition of *i* takes precedence over the global definition. Since *i* is a text constant that expands to *j*, the aforementioned MOV statement is really equivalent to "mov(j, eax);" That statement, of course, loads 10 into EAX. Since this problem is difficult to see while reading your code, you should choose local symbols in multi-part macros very carefully. A good convention to adopt is to combine your local symbol name with the macro name, e.g.,

```
#macro abc : i_abc;
```

You may wonder why HLA allows something so crazy to happen in your source code, in a moment you'll see why this behavior is useful (and now, with this brief message out of the way, back to our regularly scheduled discussion).

Before we digressed to discuss this interesting feature in HLA multi-part macros, we were trying to figure out how to efficiently simulate the conjunction and disjunction operators in an IF statement without actually using these operators in our code. The problem in the example appearing earlier in this section is that you would have to duplicate some code in order to convert the IF.ELSE statement properly. The following code shows this problem:

```
// "C" code employing logical-AND operator:

if( expr1 && expr2 )
{
    << 'true' statements >>
}
else
{
    << 'false' statements >>
}

// Corresponding HLA code using the "nested-IF" algorithm:

if( expr1 ) then

    if( expr2 ) then

        << 'true' statements >>

    else

        << 'false' statements >>

    endif;

else

    << 'false' statements >>

endif;
```

Note that this code must duplicate the "<< 'false' statements >>" section if the logic is to exactly match the original "C" code. This means that the program will be larger and harder to read than is absolutely necessary.

One solution to this problem is to create a new kind of IF statement that doesn't nest the same way standard IF statements nest. In particular, if we define the statement such that all IF clauses nested with an outer IF.ENDIF block share the same ELSE and ENDIF clauses. If this were the case, then you could implement the code above as follows:

```
if( expr1 ) then

    if( expr2 ) then

        << 'true' statements >>

    else

        << 'false' statements >>

    endif;

endif;
```


If *expr1* is false, control immediately transfers to the ELSE clause. If the value of *expr1* is true, the control falls through to the next IF statement.

If *expr2* evaluates false, then the program jumps to the single ELSE clause that all IFs share in this statement. Notice that a single ELSE clause (and corresponding 'false' statements) appear in this code; hence the code does not necessarily expand in size. If *expr2* evaluates true, then control falls through to the 'true' statements, exactly like a standard IF statement.

Notice that the nested IF statement above does not have a corresponding ENDIF. Like the ELSE clause, all nested IFs in this structure share the same ENDIF. Syntactically, there is no need to end the nested IF statement; the end of the THEN section ends with the ELSE clause, just as the outer IF statement's THEN block ends.

Of course, we can't actually define a new macro named "if" because you cannot redefine HLA reserved words. Nor would it be a good idea to do so even if these were legal (since it would make your programs very difficult to comprehend if the IF keyword had different semantics in different parts of the program. The following program uses the identifiers "_if", "_then", "_else", and "_endif" instead. It is questionable if these are good identifiers in production code (perhaps something a little more different would be appropriate). The following code example uses these particular identifiers so you can easily correlate them with the corresponding high level statements.

```

/*****
/*
/* if.hla
/*
/* This program demonstrates a modification of
/* the IF..ELSE..ENDIF statement using HLA's
/* multi-part macros.
/*
/*
/*****

program newIF;
#include( "stdlib.hhf" )

// Macro implementation of new form of if..then..else..endif.
//
// In this version, all nested IF statements transfer control
// to the same ELSE clause if any one of them have a false
// boolean expression. Syntax:
//
// _if( expression ) _then
//
//     <<statements including nested _if clauses>>
//
// _else // this is optional
//
//     <<statements, but _if clauses are not allowed here>>
//
// _endif
//
//
// Note that nested _if clauses do not have a corresponding
// _endif clause. This is because the single _else and/or
// _endif clauses terminate all the nested _if clauses
// including the first one. Of course, once the code
// encounters an _endif another _if statement may begin.

```

```

// Macro to handle the main "_if" clause.
// This code just tests the expression and jumps to the _else
// clause if the expression evaluates false.

macro _if( ifExpr ):elseLbl, hasElse, ifDone;

    ?hasElse := false;
    jf(ifExpr) elseLbl;

// Just ignore the _then keyword.

keyword _then;

// Nested _if clause (yes, HLA lets you replace the main
// macro name with a keyword macro). Identical to the
// above _if implementation except this one does not
// require a matching _endif clause. The single _endif
// (matching the first _if clause) terminates all nested
// _if clauses as well as the main _if clause.

keyword _if( nestedIfExpr );
    jf( nestedIfExpr ) elseLbl;

// If this appears within the _else section, report
// an error (we don't allow _if clauses nested in
// the else section, that would create a loop).

#if( hasElse )

    #error( "All _if clauses must appear before the _else clause" )

#endif

// Handle the _else clause here. All we need to is check to
// see if this is the only _else clause and then emit the
// jmp over the else section and output the elseLbl target.

keyword _else;
    #if( hasElse )

        #error( "Only one _else clause is legal per _if.._endif" )

    #else

        // Set hasElse true so we know that we've seen an _else
        // clause in this statement.

        ?hasElse := true;
        jmp ifDone;
        elseLbl:

    #endif

// _endif has two tasks. First, it outputs the "ifDone" label
// that _else uses as the target of its jump to skip over the
// else section. Second, if there was no else section, this
// code must emit the "elseLbl" label so that the false conditional(s)

```

```

// in the _if clause(s) have a legal target label.

terminator _endif;

    ifDone:
    #if( !hasElse )

        elseLbl:

    #endif

endmacro;

static
    tr:boolean := true;
    f:boolean := false;

begin newIF;

    // Real quick demo of the _if statement:

    _if( tr ) _then

        _if( tr ) _then
        _if( f ) _then

            stdout.put( "error" nl );

        _else

            stdout.put( "Success" );

        _endif

end newIF;

```

Program 9.6 Using Macros to Create a New IF Statement

Just in case you're wondering, this program prints "Success" and then quits. This is because the nested "_if" statements are equivalent to the expression "true && true && false" which, of course, is false. Therefore, the "_else" portion of this code should execute.

The only surprise in this macro is the fact that it redefines the *_if* macro as a keyword macro upon invocation of the main *_if* macro. The reason this code does this is so that any nested *_if* clauses do not require a corresponding *_endif* and don't support an *_else* clause.

Implementing an ELSEIF clause introduces some difficulties, hence its absence in this example. The design and implementation of an ELSEIF clause is left to the more serious reader³.

3. I.e., I don't even want to have to think about this problem!

9.3 Sample Program: A Simple Expression Compiler

This program's sample program is a bit complex. In fact, the theory behind this program is well beyond the scope of this text (since it involves compiler theory). However, this example is such a good demonstration of the capabilities of HLA's macro facilities and DSEL capabilities, it was too good not to include here. The following paragraphs will attempt to explain how this compile-time program operates. If you have difficulty understanding what's going on, don't feel too bad, this code isn't exactly the type of stuff that beginning assembly language programmers would normally develop on their own.

This program presents a (very) simple *expression compiler*. This code includes a macro, *u32expr*, that emits a sequence of instructions that compute the value of an arithmetic expression and leave that result sitting in one of the 80x86's 32-bit registers. The syntax for the *u32expr* macro invocation is the following:

```
u32expr( reg32, uns32_expression );
```

This macro emits the code that computes the following (HLL) statement:

```
reg32 := uns32_expression;
```

For example, the macro invocation "u32expr(eax, ebx+ecx*5 - edi);" computes the value of the expression "ebx+ecx*5 - edi" and leaves the result of this expression sitting in the EAX register.

The *u32expr* macro places several restrictions on the expression. First of all, as the name implies, it only computes the result of an *uns32* expression. No other data types may appear within the expression. During computation, the macro uses the EAX and EDX registers, so expressions should not contain these registers as their values may be destroyed by the code that computes the expression (EAX or EDX may safely appear as the first operand of the expression, however). Finally, expressions may only contain the following operators:

```
<, <=, >, >=, <>, !=, =, ==
      +, -
      *, /
      (, )
```

The "<>" and "!=" operators are equivalent (not equals) and the "=" and "==" operators are also equivalent (equals). The operators above are listed in order of increasing precedence; i.e., "*" has a higher precedence than "+" (as you would expect). You can override the precedence of an operator by using parentheses in the standard manner.

It is important to remember that *u32expr* is a macro, not a function. That is, the invocation of this macro results in a sequence of 80x86 assembly language instructions that compute the desired expression. The *u32expr* invocation is not a function call. to some routine that computes the result.

To understand how this macro works, it would be a good idea to review the section on "Converting Arithmetic Expressions to Postfix Notation" on page 635. That section discusses how to convert floating point expressions to reverse polish notation; although the *u32expr* macro works with *uns32* objects rather than floating point objects, the approach it uses to translate expressions into assembly language uses this same algorithm. So if you don't remember how to translate expressions into reverse polish notation, it might be worthwhile to review that section of this text.

Converting floating point expressions to reverse polish notation is especially easy because the 80x86's FPU uses a stack architecture. Alas, the integer instructions on the 80x86 use a register architecture and efficiently translating integer expression to assembly language is a bit more difficult (see "Arithmetic Expressions" on page 597). We'll solve this problem by translating the expressions to assembly code in a somewhat less than efficient manner; we'll simulate an integer stack architecture by using the 80x86's hardware stack to hold temporary results during an integer calculation.

To push an integer constant or variable onto the 80x86 hardware stack, we need only use a PUSH or PUSHED instruction. This operation is trivial.

To add two values sitting on the top of stack together, leaving their sum on the stack, all we need do is pop those two values into registers, add the register values, and then push the result back onto the stack. We can do this operation slightly more efficiently, since addition is commutative, by using the following code:

```
// Compute X+Y where X is on NOS (next on stack) and Y is on TOS (top of stack):

    pop( eax );           // Get Y's value.
    add( eax, [esp] );    // Add with X's value and leave sum on TOS.
```

Subtraction is identical to addition. Although subtraction is not commutative the operands just happen to be on the stack in the proper order to efficiently compute their difference. To compute "X-Y" where X is on NOS and Y is on TOS, we can use code like the following:

```
// Compute X-y where X is on NOS and Y is on TOS:

    pop( eax );
    sub( eax, [esp] );
```

Multiplication of the two items on the top of stack is a little more complicated since we must use the MUL instruction (the only unsigned multiplication instruction available) and the destination operand must be the EDX:EAX register pair. Fortunately, multiplication is a commutative operation, so we can compute the product of NOS (next on stack) and TOS (top of stack) using code like the following:

```
// Compute X*Y where X is on NOS and Y is on TOS:

    pop( eax );
    mul( [esp], eax );    // Note that this wipes out the EDX register.
    mov( eax, [esp] );
```

Division is problematic because it is not a commutative operation and its operands on the stack are not in a convenient order. That is, to compute X/Y it would be really convenient if X was on TOS and Y was in the NOS position. Alas, as you'll soon see, it turns out that X is at NOS and Y is on the TOS. To resolve this issue requires slightly less efficient code than the sequences we've used above. Since the DIV instruction is so slow anyway, this will hardly matter.

```
// Compute X/Y where X is on NOS and Y is on TOS:

    mov( [esp+4], eax );  // Get X from NOS.
    xor( edx, edx );     // Zero-extend EAX into EDX:EAX
    div( [esp], edx:eax ); // Compute their quotient.
    pop( edx );          // Remove unneeded Y value from the stack.
    mov( eax, [esp] );   // Store quotient to the TOS.
```

The remaining operators are the comparison operators. These operators compare the value on NOS with the value on TOS and leave true (1) or false (0) sitting on the stack based on the result of the comparison. While it is easy to work around the non-commutative aspect of many of the comparison operators, the big challenge is converting the result to true or false. The SETcc instructions are convenient for this purpose, but they only work on byte operands. Therefore, we will have to zero extend the result of the SETcc instructions to obtain an *uns32* result we can push onto the stack. Ultimately, the code we must emit for a comparison is similar to the following:

```
// Compute X <= Y where X is on NOS and Y is on TOS.

    pop( eax );
    cmp( [esp], eax );
    setbe( al );         // This instruction changes for other operators.
    movzx( al, eax );
```

```
mov( eax, [esp] );
```

As it turns out, the appearance of parentheses in an expression only affects the order of the instructions appearing in the sequence, it does not affect the number of type of instructions that correspond to the calculation of an expression. As you'll soon see, handling parentheses is an especially trivial operation.

With this short description of how to emit code for each type of arithmetic operator, it's time to discuss exactly how we will write a macro to automate this translation. Once again, a complete discussion of this topic is well beyond the scope of this text, however a simple introduction to compiler theory will certainly ease the understanding the *u32expr* macro.

For efficiency and reasons of convenience, most compilers are broken down into several components called *phases*. A compiler phase is collection of logically related activities that take place during compilation. There are three general compiler phases we are going to consider here: (1) *lexical analysis* (or *scanning*), (2) *parsing*, and (3) *code generation*. It is important to realize that these three activities occur concurrently during compilation; that is, they take place at the same time rather than as three separate, serial, activities. A compiler will typically run the lexical analysis phase for a short period, transfer control to the parsing phase, do a little code generation, and then, perhaps, do some more scanning and parsing and code generation (not necessarily in that order). Real compilers have additional phases, the *u32expr* macro will only use these three phases (and if you look at the macro, you'll discover that it's difficult to separate the parsing and code generation phases).

Lexical analysis is the process of breaking down a string of characters, representing the expression to compile, into a sequence of *tokens* for use by the parser. For example, an expression of the form "MaxVal - x <= \$1c" contains five distinct tokens:

- MaxVal
- -
- x
- <=
- \$1c

Breaking any one of these tokens into smaller objects would destroy the intent of the expression (e.g., converting MaxVal to "Max" and "Val" or converting "<=" into "<" and "="). The job of the lexical analyzer is to break the string down into a sequence of constituent tokens and return this sequence of tokens to the parser (generally one token at a time, as the parser requests new tokens). Another task for the lexical analyzer is to remove any extra white space from the string of symbols (since expressions may generally contain an arbitrary amount of white space).

Fortunately, it is easy to extract the next available token in the input string by skipping all white space characters and then look at the current character. Identifiers always begin with an alphabetic character or an underscore, numeric values always begin with a decimal digit, a dollar sign ("\$"), or a percent sign ("%"). Operators always begin with the corresponding punctuation character that represents the operator. There are only two major issues here: how do we classify these tokens and how do we differentiate two or more distinct tokens that start with the same character (e.g., "<", "<=", and "<>")? Fortunately, HLA's compile-time functions provide the tools we need to do this.

Consider the declaration of the *u32expr* macro:

```
#macro u32expr( reg, expr ):sexpr;
```

The *expr* parameter is a text object representing the expression to compile. The *sexpr* local symbol will contain the string equivalent of this text expression. The macro translates the text *expr* object to a string with the following statement:

```
?sexpr := @string:expr;
```

From this point forward, the macro works with the string in *sexpr*.

The *lexer* macro (compile-time function) handles the lexical analysis operation. This macro expects a single string parameter from which it extracts a single token and removes the string associated with that

token from the front of the string. For example, the following macro invocation returns "2" as the function result and leaves "+3" in the parameter string (*str2Lex*):

```
?str2Lex := "2+3";
?TokenResult := lexer( str2Lex );
```

The *lexer* function actually returns a little more than the string it extracts from its parameter. The actual return value is a record constant that has the definition:

```
tokType:
  record

    lexeme:string;
    tokClass:tokEnum;

  endrecord;
```

The *lexeme* field holds that actual string (e.g., "2" in this example) that the *lexer* macro returns. The *tokClass* field holds a small numeric value (see the *tokEnum* enumerated data type) that specifies that type of the token. In this example, the call to *lexer* stores the value *intconst* into the *tokClass* field. Having a single value (like *intconst*) when the *lexeme* could take on a large number of different forms (e.g., "2", "3", "4", ...) will help make the parser easier to write. The call to *lexer* in the previous example produces the following results:

```
str2lex : "+3"
TokenResult.lexeme: "2"
TokenResult.tokClass: intconst
```

A subsequent call to *lexer*, immediately after the call above, will process the next available character in the string and return the following values:

```
str2lex : "3"
TokenResult.lexeme: "+"
TokenResult.tokClass: plusOp
```

To see how *lexer* works, consider the first few lines of the *lexer* macro:

```
#macro lexer( input ):theLexeme,boolResult;

  ?theLexeme:string;      // Holds the string we scan.
  ?boolResult:boolean;   // Used only as a dummy value.

  // Check for an identifier.

  #if( @peekCset( input, tok1stIDChar ) )

    // If it began with a legal ID character, extract all
    // ID characters that follow. The extracted string
    // goes into "theLexeme" and this call also removes
    // those characters from the input string.

    ?boolResult := @oneOrMoreCset( input, tokIDChars, input, theLexeme );

    // Return a tokType constant with the identifier string and
    // the "identifier" token value:

    tokType:[ theLexeme, identifier ]

  // Check for a decimal numeric constant.
```

```

#elseif( @peekCset( input, digits )
.
.
.

```

The real work begins with the #IF statement where the code uses the `@peekCset` function to see if the first character of the `input` parameter is a member of the `tok1stIDChar` set (which is the alphabetic characters plus an underscore, i.e., the set of character that may appear as the first character of an identifier). If so, the code executes the `@oneOrMoreCset` function to extract all legal identifier characters (alphanumerics plus underscore), storing the result in the `theLexeme` string variable. Note that this function call to `@oneOrMoreCset` also removes the string it matches from the front of the `input` string (see the description of `@oneOrMoreCset` for more details). This macro returns a `tokType` result by simply specifying a `tokType` constant containing `theLexeme` and the enum constant `identifier`.

If the first character of the input string is not in the `tok1stIDChar` set, then the `lexer` macro checks to see if the first character is a legal decimal digit. If so, then this macro processes that string of digits in a manner very similar to identifiers. The code handles hexadecimal and binary constants in a similar fashion. About the only thing exciting in the whole macro is the way it differentiates tokens that begin with the same symbol. Once it determines that a token begins with a character common to several lexemes, it calls `@matchStr` to attempt to match the longer tokens before settling on the shorter lexeme (i.e., `lexer` attempts to match "`<=`" or "`<>`" before it decides the lexeme is just "`<`"). Other than this complication, the operation of the `lexer` is really quite simple.

The operation of the parser/code generation phases is a bit more complex, especially since these macros are indirectly recursive; to simplify matters we will explore the parser/code generator in a bottom-up fashion.

The parser/code generator phases consist of four separate macros: `doTerms`, `doMulOps`, `doAddOps`, and `doCmpOps`. The reason for these four separate macros is to handle the different precedences of the arithmetic operators and the parentheses. An explanation of how these four macros handle the different arithmetic precedences is beyond the scope of this text; we'll just look at how these four macros do their job.

The `doTerms` macro is responsible for handling identifiers, numeric constants, and subexpressions surrounded by parentheses. The single parameter is the current input string whose first (non-blank) character sequence is an identifier, constant, or parenthetical expression. Here is the full text for this macro:

```

#macro doTerms( expr ):termToken;

    // Begin by removing any leading white space from the string:

    ?expr := @trim( expr, 0 );

    // Okay, call the lexer to extract the next token from the input:

    ?termToken:tokType := lexer( expr );

    // See if the current token is an identifier.  If so, assume that
    // it's an uns32 identifier and emit the code to push its value onto
    // the stack.

    #if( termToken.tokClass = identifier )

        // If we've got an identifier, emit the code to
        // push that identifier onto the stack.

        push( @text( termToken.lexeme ) );

    // If it wasn't an identifier, see if it's a numeric constant.
    // If so, emit the code that will push this value onto the stack.

```



```

#elseif( termToken.tokClass = intconst )

    // If we've got a constant, emit the code to push
    // that constant onto the stack.

    pushd( @text( termToken.lexeme ) );

    // If it's not an identifier or an integer constant, see if it's
    // a parenthesized subexpression.  If so, invoke the doCmpOps macro
    // to do the real work of code generation for the subexpression.
    // The call to the doCmpOps macro emits all the code needed to push
    // the result of the subexpression onto the stack; note that this
    // macro doesn't need to emit any code for the parenthetical expression,
    // all the code emission is handled by doCmpOps.

#elseif( termToken.tokClass = lparen )

    // If we've got a parenthetical expression, emit
    // the code to leave the parenthesized expression
    // sitting on the stack.

    doCmpOps( expr );

    // We must have a closing right parentheses after the subexpression.
    // Skip any white space and check for the closing ")" here.

    ?expr := @trim( expr, 0 );
    ?termToken:tokType := lexer( expr );
    #if( termToken.tokClass <> rparen )

        #error( "Expected closing parenthesis: " + termToken.lexeme )

    #endif

    // If we get to this point, then the lexer encountered something besides
    // an identifier, a numeric constant, or a parenthetical expression.

#else

    #error( "Unexpected term: '" + termToken.lexeme + "'" )

#endif

#endmacro;

```

The *doTerms* macro is responsible for leaving a single item sitting on the top of the 80x86 hardware stack. That stack item is either the value of an *uns32* identifier, the value of an *uns32* expression, or the value left on the stack via a parenthesized subexpression. The important thing to remember is that you can think of *doTerms* as a function that emits code that leaves a single item on the top of the 80x86 stack.

The *doMulOps* macro handles expressions consisting of a single term (items handled by the *doTerms* macro) optionally followed by zero or more pairs consisting of a multiplicative operator ("*" or "/") and a second term. It is especially important to remember that the *doMulOps* macro does not require the presence of a multiplicative operator; it will legally process a single term (identifier, numeric constant, or parenthetical expression). If one or more multiplicative operator and term pairs are present, the *doMulOps* macro will emit the code that will multiply the values of the two terms together and push the result onto the stack. E.g., consider the following:

$$x * 5$$

Since there is a multiplicative operator present ("*"), the *doMulOps* macro will call *doTerms* to process the two terms (pushing *X* and then *Y* onto the stack) and then the *doMulOps* macro will emit the code to multiply the two values on the stack leaving their product on the stack. The complete code for the *doMulOps* macro is the following:

```
#macro doMulOps( sexpr ):opToken;

    // Process the leading term (not optional). Note that
    // this expansion leaves an item sitting on the stack.

    doTerms( sexpr );

    // Process all the MULOPS at the current precedence level.
    // (these are optional, there may be zero or more of them.)
    // Begin by removing any leading white space.

    ?sexpr := @trim( sexpr, 0 );
    #while( @peekCset( sexpr, MulOps ) )

        // Save the operator so we know what code we should
        // generate later.

        ?opToken := lexer( sexpr );

        // Get the term following the operator.

        doTerms( sexpr );

        // Okay, the code for the two terms is sitting on
        // the top of the stack (left operand at [esp+4] and
        // the right operand at [esp]). Emit the code to
        // perform the specified operation.

        #if( opToken.lexeme = "*" )

            // For multiplication, compute
            // [esp+4] = [esp] * [esp+4] and
            // then pop the junk off the top of stack.

            pop( eax );
            mul( (type dword [esp]) );
            mov( eax, [esp] );

        #elseif( opToken.lexeme = "/" )

            // For division, compute
            // [esp+4] = [esp+4] / [esp] and
            // then pop the junk off the top of stack.

            mov( [esp+4], eax );
            xor( edx, edx );
            div( [esp], edx:eax );
            pop( edx );
            mov( eax, [esp] );

        #endif

        ?sexpr := @trim( sexpr, 0 );

    #endwhile

#endmacro;
```

Note the simplicity of the code generation. This macro assumes that *doTerms* has done its job leaving two values sitting on the top of the stack. Therefore, the only code this macro has to generate is the code to pop these two values off the stack and then multiply or divide them, depending on the actual operator that is present. The code generation uses the sequences appearing earlier in this section.

The *doAddOps* and *doCmpOps* macros work in a manner nearly identical to *doMulOps*. The only difference is the operators these macros handle (and, of course, the code that they generate). See Program 9.7, below, for details concerning these macros.

Once we've got the lexer and the four parser/code generation macros written, writing the *u32expr* macro is quite easy. All that *u32expr* needs to do is call the *doCmpOps* macro to compile the expression and then pop the result off the stack and store it into the destination register appearing as the first operand. This requires little more than a single POP instruction.

About the only thing interesting in the *u32expr* macro is the presence of the RETURNS statement. This HLA statement takes the following form:

```
returns( { statements }, string_expression )
```

This statement simply compiles the sequence of statements appearing between the braces in the first operand and then it uses the second *string_expression* operand as the "returns" value for this statement. As you may recall from the discussion of instruction composition (see "Instruction Composition in HLA" on page 558), HLA substitutes the "returns" value of a statement in place of that statement if it appears as an operand to another expression. The RETURNS statement appearing in the *u32expr* macro returns the register you specify as the first parameter as the "returns" value for the macro invocation. This lets you invoke the *u32expr* macro as an operand to many different instructions (that accept a 32-bit register as an operand). For example, the following *u32expr* macro invocations are all legal:

```
mov( u32expr( eax, i*j+k/15 - 2), m );
if( u32expr( edx, eax < (ebx-2)*ecx ) then ... endif;
funcCall( u32expr( eax, (x*x + y*y)/z*z ), 16, 2 );
```

Well, without further ado, here's the complete code for the *u32expr* compiler and some test code that checks out the operation of this macro:

```
// u32expr.hla
//
// This program demonstrates how to write an "expression compiler"
// using the HLA compile-time language. This code defines a macro
// (u32expr) that accepts an arithmetic expression as a parameter.
// This macro compiles that expression into a sequence of HLA
// machine language instructions that will compute the result of
// that expression at run-time.
//
// The u32expr macro does have some severe limitations.
// First of all, it only support uns32 operands.
// Second, it only supports the following arithmetic
// operations:
//
// +, -, *, /, <, <=, >, >=, =, <>.
//
// The comparison operators produce zero (false) or
// one (true) depending upon the result of the (unsigned)
// comparison.
//
// The syntax for a call to u32expr is
//
// u32expr( register, expression )
```

```

//
// The macro computes the result of the expression and
// leaves this result sitting in the register specified
// as the first operand. This register overwrites the
// values in the EAX and EDX registers (though these
// two registers are fine as the destination for the
// result).
//
// This macro also returns the first (register) parameter
// as its "returns" value, so you may use u32expr anywhere
// a 32-bit register is legal, e.g.,
//
//     if( u32expr( eax, (i*3-2) < j )) then
//
//         << do something if (i*3-2) < j >>
//
//     endif;
//
// The statement above computes true or false in EAX and the
// "if" statement processes this result accordingly.

```

```

program TestExpr;
#include( "stdlib.hhf" )

// Some special character classifications the lexical analyzer uses.

const

    // tok1stIDChar is the set of legal characters that
    // can begin an identifier. tokIDChars is the set
    // of characters that may follow the first character
    // of an identifier.

    tok1stIDChar := { 'a'..'z', 'A'..'Z', '_' };
    tokIDChars := { 'a'..'z', 'A'..'Z', '0'..'9', '_' };

    // digits, hexDigits, and binDigits are the sets
    // of characters that are legal in integer constants.
    // Note that these definitions don't allow underscores
    // in numbers, although it would be a simple fix to
    // allow this.

    digits := { '0'..'9' };
    hexDigits := { '0'..'9', 'a'..'f', 'A'..'F' };
    binDigits := { '0'..'1' };

    // CmpOps, PlusOps, and MulOps are the sets of
    // operator characters legal at three levels
    // of precedence that this parser supports.

    CmpOps := { '>', '<', '=', '!' };
    PlusOps := { '+', '-' };
    MulOps := { '*', '/' };

```

```

type

```

```

// tokEnum-
//
// Data values the lexical analyzer returns to quickly
// determine the classification of a lexeme. By
// classifying the token with one of these values, the
// parser can more quickly process the current token.
// I.e., rather than having to compare a scanned item
// against the two strings "+" and "-", the parser can
// simply check to see if the current item is a "plusOp"
// (which indicates that the lexeme is "+" or "-").
// This speeds up the compilation of the expression since
// only half the comparisons are needed and they are
// simple integer comparisons rather than string comparisons.

tokEnum:    enum
            {
                identifier,
                intconst,
                lparen,
                rparen,
                plusOp,
                mulOp,
                cmpOp
            };

// tokType-
//
// This is the "token" type returned by the lexical analyzer.
// The "lexeme" field contains the string that matches the
// current item scanned by the lexer. The "tokClass" field
// contains a generic classification for the symbol (see the
// "tokEnum" type above).

tokType:
    record

        lexeme:string;
        tokClass:tokEnum;

    endrecord;

// lexer-
//
// This is the lexical analyzer. On each call it extracts a
// lexical item from the front of the string passed to it as a
// parameter (it also removes this item from the front of the
// string). If it successfully matches a token, this macro
// returns a tokType constant as its return value.

macro lexer( input ):theLexeme,boolResult;

    ?theLexeme:string;    // Holds the string we scan.
    ?boolResult:boolean;  // Used only as a dummy value.

    // Check for an identifier.

    #if( @peekCset( input, tok1stIDChar ))

```

```

// If it began with a legal ID character, extract all
// ID characters that follow. The extracted string
// goes into "theLexeme" and this call also removes
// those characters from the input string.

?boolResult := @oneOrMoreCset( input, tokIDChars, input, theLexeme );

// Return a tokType constant with the identifier string and
// the "identifier" token value:

tokType:[ theLexeme, identifier ]

// Check for a decimal numeric constant.

#elseif( @peekCset( input, digits ) )

// If the current item began with a decimal digit, extract
// all the following digits and put them into "theLexeme".
// Also remove these characters from the input string.

?boolResult := @oneOrMoreCset( input, digits, input, theLexeme );

// Return an integer constant as the current token.

tokType:[ theLexeme, intconst ]

// Check for a hexadecimal numeric constant.

#elseif( @peekChar( input, '$' ) )

// If we had a "$" symbol, grab it and any following
// hexadecimal digits. Set boolResult true if there
// is at least one hexadecimal digit. As usual, extract
// the hex value to "theLexeme" and remove the value
// from the input string:

?boolResult := @oneChar( input, '$', input ) &
               @oneOrMoreCset( input, hexDigits, input, theLexeme );

// Returns the hex constant string as an intconst object:

tokType:[ '$' + theLexeme, intconst ]

// Check for a binary numeric constant.

#elseif( @peekChar( input, '%' ) )

// See the comments for hexadecimal constants. This boolean
// constant scanner works the same way.

?boolResult := @oneChar( input, '%', input ) &
               @oneOrMoreCset( input, binDigits, input, theLexeme );
tokType:[ '%' + theLexeme, intconst ]

```

```

// Handle the "+" and "-" operators here.

#elseif( @peekCset( input, PlusOps ))

    // If it was a "+" or "-" sign, extract it from the input
    // and return it as a "plusOp" token.

    ?boolResult := @oneCset( input, PlusOps, input, theLexeme );
    tokType:[ theLexeme, plusOp ]

// Handle the "*" and "/" operators here.

#elseif( @peekCset( input, MulOps ))

    // If it was a "*" or "/" sign, extract it from the input
    // and return it as a "mulOp" token.

    ?boolResult := @oneCset( input, MulOps, input, theLexeme );
    tokType:[ theLexeme, mulOp ]

// Handle the "=" ("=="), "<>" ("!="), "<", "<=", ">", and ">="
// operators here.

#elseif( @peekCset( input, CmpOps ))

    // Note that we must check for two-character operators
    // first so we don't confuse them with the single
    // character operators:

    #if
    (
        @matchStr( input, ">=", input, theLexeme )
        | @matchStr( input, "<=", input, theLexeme )
        | @matchStr( input, "<>", input, theLexeme )
    )

    tokType:[ theLexeme, cmpOp ]

#elseif( @matchStr( input, "!=", input, theLexeme ))

    tokType:[ "<>", cmpOp ]

#elseif( @matchStr( input, "==", input, theLexeme ))

    tokType:[ "=", cmpOp ]

#elseif( @oneCset( input, {'>', '<', '='}, input, theLexeme ))

    tokType:[ theLexeme, cmpOp ]

#else

    #error( "Illegal comparison operator: " + input )

#endif

```

```

// Handle the parentheses down here.

#elseif( @oneChar( input, '(', input, theLexeme ))

    tokType:[ "(", lparen ]

#elseif( @oneChar( input, ')', input, theLexeme ))

    tokType:[ ")", rparen ]

// Anything else is an illegal character.

#else

    #error
    (
        "Illegal character in expression: '" +
        @substr( input, 0, 1 ) +
        "' ($" +
        string( dword( @substr( input, 0, 1 ))) +
        ")"
    )
    ?input := @substr( input, 1, @length(input) - 1 );

#endif

endmacro;

// Handle identifiers, constants, and sub-expressions within
// parentheses within this macro.
//
// terms-> identifier | intconst | '(' CmpOps ')'
//
// This compile time function does the following:
//
// (1) If it encounters an identifier, it emits the
//     following instruction to the code stream:
//
//         push( identifier );
//
// (2) If it encounters an (unsigned) integer constant, it emits
//     the following instruction to the code stream:
//
//         pushd( constant_value );
//
// (3) If it encounters an expression surrounded by parentheses,
//     then it emits whatever instruction sequence is necessary
//     to leave the value of that (unsigned integer) expression
//     sitting on the top of the stack.
//
// (4) If the current lexeme is none of the above, then this
//     macro prints an appropriate error message.
//
// The end result of the execution of this macro is the emission
// of some code that leaves a single 32-bit unsigned value sitting
// on the top of the 80x86 stack (assuming no error).

```



```

macro doTerms( expr ):termToken;

    ?expr := @trim( expr, 0 );
    ?termToken:tokType := lexer( expr );
    #if( termToken.tokClass = identifier )

        // If we've got an identifier, emit the code to
        // push that identifier onto the stack.

        push( @text( termToken.lexeme ) );

    #elseif( termToken.tokClass = intconst )

        // If we've got a constant, emit the code to push
        // that constant onto the stack.

        pushd( @text( termToken.lexeme ) );

    #elseif( termToken.tokClass = lparen )

        // If we've got a parenthetical expression, emit
        // the code to leave the parenthesized expression
        // sitting on the stack.

        doCmpOps( expr );
        ?expr := @trim( expr, 0 );
        ?termToken:tokType := lexer( expr );
        #if( termToken.tokClass <> rparen )

            #error( "Expected closing parenthesis: " + termToken.lexeme )

        #endif

    #else

        #error( "Unexpected term: '" + termToken.lexeme + "'" )

    #endif

endmacro;

// Handle the multiplication, division, and modulo operations here.
//
// MulOps-> terms ( mulOp terms )*
//
// The above grammar production tells us that a "MulOps" consists
// of a "terms" expansion followed by zero or more instances of a
// "mulop" followed by a "terms" expansion (like wildcard filename
// expansions, the "*" indicates zero or more copies of the things
// inside the parentheses).
//
// This code assumes that "terms" leaves whatever operands/expressions
// it processes sitting on the 80x86 stack at run time. If there is
// a single term (no optional mulOp/term following), then this code
// does nothing (it leaves the result on the stack that was pushed
// by the "terms" expansion). If one or more mulOp/terms pairs are
// present, then for each pair this code assumes that the two "terms"
// expansions left some value on the stack. This code will pop

```

```

// those two values off the stack and multiply or divide them and
// push the result back onto the stack (sort of like the way the
// FPU multiplies or divides values on the FPU stack).
//
// If there are three or more operands in a row, separated by
// mulops ("*" or "/") then this macro will process them in
// a left-to-right fashion, popping each pair of values off the
// stack, operating on them, pushing the result, and then processing
// the next pair. E.g.,
//
//      i * j * k
//
// yields:
//
//      push( i ); // From the "terms" macro.
//      push( j ); // From the "terms" macro.
//
//      pop( eax ); // Compute the product of i*j
//      mul( (type dword [esp]));
//      mov( eax, [esp]);
//
//      push( k ); // From the "terms" macro.
//
//      pop( eax ); // Pop K
//      mul( (type dword [esp])); // Compute K* (i*j) [i*j is value on TOS].
//      mov( eax, [esp]); // Save product on TOS.

macro doMulOps( sexpr ):opToken;

    // Process the leading term (not optional). Note that
    // this expansion leaves an item sitting on the stack.

    doTerms( sexpr );

    // Process all the MULOPs at the current precedence level.
    // (these are optional, there may be zero or more of them.)

    ?sexpr := @trim( sexpr, 0 );
    #while( @peekCset( sexpr, MulOps )

        // Save the operator so we know what code we should
        // generate later.

        ?opToken := lexer( sexpr );

        // Get the term following the operator.

        doTerms( sexpr );

        // Okay, the code for the two terms is sitting on
        // the top of the stack (left operand at [esp+4] and
        // the right operand at [esp]). Emit the code to
        // perform the specified operation.

        #if( opToken.lexeme = "*" )

            // For multiplication, compute
            // [esp+4] = [esp] * [esp+4] and
            // then pop the junk off the top of stack.

```

```

        pop( eax );
        mul( (type dword [esp]) );
        mov( eax, [esp] );

#elseif( opToken.lexeme = "/" )

        // For division, compute
        // [esp+4] = [esp+4] / [esp] and
        // then pop the junk off the top of stack.

        mov( [esp+4], eax );
        xor( edx, edx );
        div( [esp], edx:eax );
        pop( edx );
        mov( eax, [esp] );

#endif
?sexpr := @trim( sexpr, 0 );

#endwhile

endmacro;

// Handle the addition, and subtraction operations here.
//
// AddOps-> MulOps ( addOp MulOps )*
//
// The above grammar production tells us that an "AddOps" consists
// of a "MulOps" expansion followed by zero or more instances of an
// "addOp" followed by a "MulOps" expansion.
//
// This code assumes that "MulOps" leaves whatever operands/expressions
// it processes sitting on the 80x86 stack at run time. If there is
// a single MulOps item then this code does nothing. If one or more
// addOp/MulOps pairs are present, then for each pair this code
// assumes that the two "MulOps" expansions left some value on the
// stack. This code will pop those two values off the stack and
// add or subtract them and push the result back onto the stack.

macro doAddOps( sexpr ):opToken;

    // Process the first operand (or subexpression):

    doMulOps( sexpr );

    // Process all the ADDOPS at the current precedence level.

    ?sexpr := @trim( sexpr, 0 );
    #while( @peekCset( sexpr, PlusOps )

        // Save the operator so we know what code we should
        // generate later.

        ?opToken := lexer( sexpr );

        // Get the MulOp following the operator.

        doMulOps( sexpr );

```

```

// Okay, emit the code associated with the operator.

#if( opToken.lexeme = "+" )

    pop( eax );
    add( eax, [esp] );

#elseif( opToken.lexeme = "-" )

    pop( eax );
    sub( eax, [esp] );

#endif

#endwhile

endmacro;

// Handle the comparison operations here.
//
// CmpOps-> addOps ( cmpOp AddOps )*
//
// The above grammar production tells us that a "CmpOps" consists
// of an "AddOps" expansion followed by zero or more instances of an
// "cmpOp" followed by an "AddOps" expansion.
//
// This code assumes that "MulOps" leaves whatever operands/expressions
// it processes sitting on the 80x86 stack at run time. If there is
// a single MulOps item then this code does nothing. If one or more
// addOp/MulOps pairs are present, then for each pair this code
// assumes that the two "MulOps" expansions left some value on the
// stack. This code will pop those two values off the stack and
// add or subtract them and push the result back onto the stack.

macro doCmpOps( sexpr ):opToken;

    // Process the first operand:

    doAddOps( sexpr );

    // Process all the CMPOPs at the current precedence level.

    ?sexpr := @trim( sexpr, 0 );
    #while( @peekCset( sexpr, CmpOps ))

        // Save the operator for the code generation task later.

        ?opToken := lexer( sexpr );

        // Process the item after the comparison operator.

        doAddOps( sexpr );

        // Generate the code to compare [esp+4] against [esp]
        // and leave true/false sitting on the stack in place
        // of these two operands.

```

```

#if( opToken.lexeme = "<" )

    pop( eax );
    cmp( [esp], eax );
    setb( al );
    movzx( al, eax );
    mov( eax, [esp] );

#elif( opToken.lexeme = "<=" )

    pop( eax );
    cmp( [esp], eax );
    setbe( al );
    movzx( al, eax );
    mov( eax, [esp] );

#elif( opToken.lexeme = ">" )

    pop( eax );
    cmp( [esp], eax );
    seta( al );
    movzx( al, eax );
    mov( eax, [esp] );

#elif( opToken.lexeme = ">=" )

    pop( eax );
    cmp( [esp], eax );
    setae( al );
    movzx( al, eax );
    mov( eax, [esp] );

#elif( opToken.lexeme = "=" )

    pop( eax );
    cmp( [esp], eax );
    sete( al );
    movzx( al, eax );
    mov( eax, [esp] );

#elif( opToken.lexeme = "<>" )

    pop( eax );
    cmp( [esp], eax );
    setne( al );
    movzx( al, eax );
    mov( eax, [esp] );

#endif

#endwhile

endmacro;

// General macro that does the expression compilation.
// The first parameter must be a 32-bit register where
// this macro will leave the result. The second parameter

```

```

// is the expression to compile. The expression compiler
// will destroy the value in EAX and may destroy the value
// in EDX (though EDX and EAX make fine destination registers
// for this macro).
//
// This macro generates poor machine code. It is more a
// "proof of concept" rather than something you should use
// all the time. Nevertheless, if you don't have serious
// size or time constraints on your code, this macro can be
// quite handy. Writing an optimizer is left as an exercise
// to the interested reader.

```

```

macro u32expr( reg, expr):sexpr;

    // The "returns" statement processes the first operand
    // as a normal sequence of statements and then returns
    // the second operand as the "returns" value for this
    // macro.

    returns
    (
        {

            ?sexpr:string := @string:expr;
            #if( !@IsReg32( reg ) )

                #error( "Expected a 32-bit register" )

            #else

                // Process the expression and leave the
                // result sitting in the specified register.

                doCmpOps( sexpr );
                pop( reg );

            #endif
        },

        // Return the specified register as the "returns"
        // value for this compilation:

        @string:reg
    )

```

```
endmacro;
```

```

// The following main program provides some examples of the
// use of the above macro:

```

```

static
    x:uns32;
    v:uns32 := 5;

```

```
begin TestExpr;
```

```

mov( 10, x );
mov( 12, ecx );

// Compute:
//
// edi := (x*3/v + %1010 == 16) + ecx;
//
// This is equivalent to:
//
// edi := (10*3/5 + %1010 == 16) + 12
//      := ( 30/5 + %1010 == 16) + 12
//      := ( 6 + 10 == 16) + 12
//      := ( 16 == 16) + 12
//      := ( 1 ) + 12
//      := 13
//
// This macro invocation emits the following code:
//
// push(x);
// pushd(3);
// pop(eax);
// mul( (type dword [esp]) );
// mov( eax, [esp] );
// push( v );
// mov( [esp+4], eax );
// xor edx, edx
// div( [esp], edx:eax );
// pop( edx );
// mov( eax, [esp] );
// pushd( 10 );
// pop( eax );
// add( eax, [esp] );
// pushd( 16 );
// pop( eax );
// cmp( [esp], eax );
// sete( al );
// movzx( al, eax );
// mov( eax, [esp+0] );
// push( ecx );
// pop( eax );
// add( eax, [esp] );
// pop( edi );

u32expr( edi, (x*3/v+%1010 == 16) + ecx );
stdout.put( "Sum = ", (type uns32 edi), nl );

```

```

// Now compute:
//
// eax := x + ecx/2
//      := 10 + 12/2
//      := 10 + 6
//      := 16
//
// This macro emits the following code:
//
// push( x );
// push( ecx );
// pushd( 2 );

```

```

// mov( [esp+4], eax );
// xor( edx, edx );
// div( [esp], edx:eax );
// pop( edx );
// mov( eax, [esp] );
// pop( eax );
// add( eax, [esp] );
// pop( eax );

u32expr( eax, x+ecx/2 );
stdout.put( "x=", x, " ecx=", (type uns32 ecx), " v=", v, nl );
stdout.put( "x+ecx/2 = ", (type uns32 eax ), nl );

// Now determine if (x+ecx/2) < v
// (it is not since (x+ecx/2)=16 and v = 5.)
//
// This macro invocation emits the following code:
//
// push( x );
// push( ecx );
// pushd( 2 );
// mov( [esp+4], eax );
// xor( edx, edx );
// div( [esp], edx:eax );
// pop( edx );
// mov( eax, [esp] );
// pop( eax );
// add( eax, [esp] );
// push( v );
// pop( eax );
// cmp( eax, [esp+0] );
// setb( al );
// movzx( al, eax );
// mov( eax, [esp+0] );
// pop( eax );

if( u32expr( eax, x+ecx/2 < v ) ) then

    stdout.put( "x+ecx/2 < v" nl );

else

    stdout.put( "x+ecx/2 >= v" nl );

endif;

end TestExpr;

```

Program 9.7 Uns32 Expression Compiler

9.4 Putting It All Together

The ability to extend the HLA language is one of the most powerful features of the HLA language. In this chapter you got to explore the use of several tools that allow you to extend the base language. Although a complete treatise on language design and implementation is beyond the scope of this chapter, further study in the area of compiler construction will help you learn new techniques for extending the HLA language. Later volumes in this text, including the volume on advanced string handling, will cover additional topics of interest to those who want to design and implement their own language constructs.

Classes and Objects

Chapter Ten

10.1 Chapter Overview

Many modern imperative high level languages support the notion of classes and objects. C++ (an object version of C), Java, and Delphi (an object version of Pascal) are two good examples. Of course, these high level language compilers translate their high level source code into low-level machine code, so it should be pretty obvious that some mechanism exists in machine code for implementing classes and objects.

Although it has always been possible to implement classes and objects in machine code, most assemblers provide poor support for writing object-oriented assembly language programs. Of course, HLA does not suffer from this drawback as it provides good support for writing object-oriented assembly language programs. This chapter discusses the general principles behind object-oriented programming (OOP) and how HLA supports OOP.

10.2 General Principles

Before discussing the mechanisms behind OOP, it is probably a good idea to take a step back and explore the benefits of using OOP (especially in assembly language programs). Most texts describing the benefits of OOP will mention buzz-words like “code reuse,” “abstract data types,” “improved development efficiency,” and so on. While all of these features are nice and are good attributes for a programming paradigm, a good software engineer would question the use of assembly language in an environment where “improved development efficiency” is an important goal. After all, you can probably obtain far better efficiency by using a high level language (even in a non-OOP fashion) than you can by using objects in assembly language. If the purported features of OOP don’t seem to apply to assembly language programming, why bother using OOP in assembly? This section will explore some of those reasons.

The first thing you should realize is that the use of assembly language does not negate the aforementioned OOP benefits. OOP in assembly language does promote code reuse, it provides a good method for implementing abstract data types, and it can improve development efficiency *in assembly language*. In other words, if you’re dead set on using assembly language, there are benefits to using OOP.

To understand one of the principle benefits of OOP, consider the concept of a global variable. Most programming texts strongly recommend against the use of global variables in a program (as does this text). Interprocedural communication through global variables is dangerous because it is difficult to keep track of all the possible places in a large program that modify a given global object. Worse, it is very easy when making enhancements to accidentally reuse a global object for something other than its intended purpose; this tends to introduce defects into the system.

Despite the well-understood problems with global variables, the semantics of global objects (extended lifetimes and accessibility from different procedures) are absolutely necessary in various situations. Objects solve this problem by letting the programmer decide on the lifetime of an object¹ as well as allow access to data fields from different procedures. Objects have several advantages over simple global variables insofar as objects can control access to their data fields (making it difficult for procedures to accidentally access the data) and you can also create multiple *instances* of an object allowing two separate sections of your program to use their own unique “global” object without interference from the other section.

Of course, objects have many other valuable attributes. One could write several volumes on the benefits of objects and OOP; this single chapter cannot do this subject justice. The following subsections present objects with an eye towards using them in HLA/assembly programs. However, if you are a beginning to

1. That is, the time during which the system allocates memory for an object.

OOP or wish more information about the object-oriented paradigm, you should consult other texts on this subject.

An important use for classes and objects is to create *abstract data types* (ADTs). An abstract data type is a collection of data objects and the functions (which we'll call *methods*) that operate on the data. In a pure abstract data type, the ADT's methods are the only code that has access to the data fields of the ADT; external code may only access the data using function calls to get or set data field values (these are the ADT's *accessor* methods). In real life, for efficiency reasons, most languages that support ADTs allow, at least, limited access to the data fields of an ADT by external code.

Assembly language is not a language most people associate with ADTs. Nevertheless, HLA provides several features to allow the creation of rudimentary ADTs. While some might argue that HLA's facilities are not as complete as those in a language such as C++ or Java, keep in mind that these differences exist because HLA is assembly language.

True ADTs should support *information hiding*. This means that the ADT does not allow the user of an ADT access to internal data structures and routines which manipulate those structures. In essence, information hiding restricts access to an ADT to only the accessor methods provided by the ADT. Assembly language, of course, provides very few restrictions. If you are dead set on accessing an object directly, there is very little HLA can do to prevent you from doing this. However, HLA has some facilities which will provide a small amount of information hiding capabilities. Combined with some care on your part, you will be able to enjoy many of the benefits of information hiding within your programs.

The primary facility HLA provides to support information hiding is separate compilation, linkable modules, and the `#INCLUDE/#INCLUDEONCE` directives. For our purposes, an abstract data type definition will consist of two sections: an *interface* section and an *implementation* section.

The interface section contains the definitions which must be visible to the application program. In general, it should not contain any specific information which would allow the application program to violate the information hiding principle, but this is often impossible given the nature of assembly language. Nevertheless, you should attempt to only reveal what is absolutely necessary within the interface section.

The implementation section contains the code, data structures, etc., to actually implement the ADT. While some of the methods and data types appearing in the implementation section may be public (by virtue of appearance within the interface section), many of the subroutines, data items, and so on will be private to the implementation code. The implementation section is where you hide all the details from the application program.

If you wish to modify the abstract data type at some point in the future, you will only have to change the interface and implementation sections. Unless you delete some previously visible object which the applications use, there will be no need to modify the applications at all.

Although you could place the interface and implementation sections directly in an application program, this would not promote information hiding or maintainability, especially if you have to include the code in several different applications. The best approach is to place the implementation section in an include file that any interested application reads using the HLA `#INCLUDE` directive and to place the implementation section in a separate module that you link with your applications.

The include file would contain `EXTERNAL` directives, any necessary macros, and other definitions you want made public. It generally would not contain 80x86 code except, perhaps, in some macros. When an application wants to make use of an ADT it would include this file.

The separate assembly file containing the implementation section would contain all the procedures, functions, data objects, etc., to actually implement the ADT. Those names which you want to be public should appear in the interface include file and have the `EXTERNAL` attribute. You should also include the interface include file in the implementation file so you do not have to maintain two sets of `EXTERNAL` directives.

One problem with using procedures for data access methods is the fact that many accessor methods are especially trivial (typically just a `MOV` instruction) and the overhead of the call and return instructions is expensive for such trivial operations. For example, suppose you have an ADT whose data object is a structure, but you do not want to make the field names visible to the application and you really do not want to

allow the application to access the fields of the data structure directly (because the data structure may change in the future). The normal way to handle this is to supply a method *GetField* which returns the desired field of the object. However, as pointed out above, this can be very slow. An alternative, for simple access methods is to use a macro to emit the code to access the desired field. Although code to directly access the data object appears in the application program (via macro expansion), it will be automatically updated if you ever change the macro in the interface section by simply assembling your application.

Although it is quite possible to create ADTs using nothing more than separate compilation and, perhaps, RECORDs, HLA does provide a better solution: the class. Read on to find out about HLA's support for classes and objects as well as how to use these to create ADTs.

10.3 Classes in HLA

HLA's classes provide a good mechanism for creating abstract data types. Fundamentally, a class is little more than a RECORD declaration that allows the definition of fields other than data fields (e.g., procedures, constants, and macros). The inclusion of other program declaration objects in the class definition dramatically expands the capabilities of a class over that of a record. For example, with a class it is now possible to easily define an ADT since classes may include data and methods that operate on that data (procedures).

The principle way to create an abstract data type in HLA is to declare a class data type. Classes in HLA always appear in the TYPE section and use the following syntax:

```
classname : class
    << Class declaration section >>
endclass;
```

The class declaration section is very similar to the local declaration section for a procedure insofar as it allows CONST, VAL, VAR, and STATIC variable declaration sections. Classes also let you define macros and specify procedure, iterator, and *method* prototypes (method declarations are legal only in classes). Conspicuously absent from this list is the TYPE declaration section. You cannot declare new types within a class.

A method is a special type of procedure that appears only within a class. A little later you will see the difference between procedures and methods, for now you can treat them as being one and the same. Other than a few subtle details regarding class initialization and the use of pointers to classes, their semantics are identical². Generally, if you don't know whether to use a procedure or method in a class, the safest bet is to use a method.

You do not place procedure/iterator/method code within a class. Instead you simply supply *prototypes* for these routines. A routine prototype consists of the PROCEDURE, ITERATOR, or METHOD reserved word, the routine name, any parameters, and a couple of optional procedure attributes (@USE, RETURNS, and EXTERNAL). The actual routine definition (i.e., the body of the routine and any local declarations it needs) appears outside the class.

The following example demonstrates a typical class declaration appearing in the TYPE section:

```
TYPE
  TypicalClass: class
    const
      TCconst := 5;
    val
```

2. Note, however, that the difference between procedures and methods makes all the difference in the world to the object-oriented programming paradigm. Hence the inclusion of methods in HLA's class definitions.

```

    TCval := 6;

var
    TCvar : uns32;          // Private field used only by TCproc.

static
    TCstatic : int32;

procedure TCproc( u:uns32 ); returns( "eax" );
iterator TCiter( i:int32 ); external;
method TCmethod( c:char );

endclass;

```

As you can see, classes are very similar to records in HLA. Indeed, you can think of a record as being a class that only allows VAR declarations. HLA implements classes in a fashion quite similar to records insofar as it allocates sequential data fields in sequential memory locations. In fact, with only one minor exception, there is almost no difference between a RECORD declaration and a CLASS declaration that only has a VAR declaration section. Later you'll see exactly how HLA implements classes, but for now you can assume that HLA implements them the same as it does records and you won't be too far off the mark.

You can access the *TCvar* and *TCstatic* fields (in the class above) just like a record's fields. You access the CONST and VAL fields in a similar manner. If a variable of type *TypicalClass* has the name *obj*, you can access the fields of *obj* as follows:

```

mov ( obj.TCconst, eax );
mov( obj.TCval, ebx );
add( obj.TCvar, eax );
add( obj.TCstatic, ebx );
obj.TCproc( 20 );          // Calls the TCproc procedure in TypicalClass.
etc.

```

If an application program includes the class declaration above, it can create variables using the *TypicalClass* type and perform operations using the above methods. Unfortunately, the application program can also access the fields of the ADT data type with impunity. For example, if a program created a variable *MyClass* of type *TypicalClass*, then it could easily execute instructions like "MOV(MyClass.TCvar, eax);" even though this field might be private to the implementation section. Unfortunately, if you are going to allow an application to declare a variable of type *TypicalClass*, the field names will have to be visible. While there are some tricks we could play with HLA's class definitions to help hide the private fields, the best solution is to thoroughly comment the private fields and then exercise some restraint when accessing the fields of that class. Specifically, this means that ADTs you create using HLA's classes cannot be "pure" ADTs since HLA allows direct access to the data fields. However, with a little discipline, you can simulate a pure ADT by simply electing not to access such fields outside the class' methods, procedures, and iterators.

Prototypes appearing in a class are effectively FORWARD declarations. Like normal forward declarations, all procedures, iterators, and methods you define in a class must have an actual implementation later in the code. Alternately, you may attach the EXTERNAL keyword to the end of a procedure, iterator, or method declaration within a class to inform HLA that the actual code appears in a separate module. As a general rule, class declarations appear in header files and represent the interface section of an ADT. The procedure, iterator, and method bodies appear in the implementation section which is usually a separate source file that you compile separately and link with the modules that use the class.

The following is an example of a sample class procedure implementation:

```

procedure TypicalClass.TCproc( u:uns32 ); nodisplay;
    << Local declarations for this procedure >>
begin TCproc;

    << Code to implement whatever this procedure does >>

end TCProc;

```

There are several differences between a standard procedure declaration and a class procedure declaration. First, and most obvious, the procedure name includes the class name (e.g., *TypicalClass.TCproc*). This differentiates this class procedure definition from a regular procedure that just happens to have the name *TCproc*. Note, however, that you do not have to repeat the class name before the procedure name in the BEGIN and END clauses of the procedure (this is similar to procedures you define in HLA NAMESPACES).

A second difference between class procedures and non-class procedures is not obvious. Some procedure attributes (@USE, EXTERNAL, RETURNS, @CDECL, @PASCAL, and @STDCALL) are legal only in the prototype declaration appearing within the class while other attributes (@NOFRAME, @NODISPLAY, @NOALIGNSTACK, and ALIGN) are legal only within the procedure definition and not within the class. Fortunately, HLA provides helpful error messages if you stick the option in the wrong place, so you don't have to memorize this rule.

If a class routine's prototype does not have the EXTERNAL option, the compilation unit (that is, the PROGRAM or UNIT) containing the class declaration must also contain the routine's definition or HLA will generate an error at the end of the compilation. For small, local, classes (i.e., when you're embedding the class declaration and routine definitions in the same compilation unit) the convention is to place the class' procedure, iterator, and method definitions in the source file shortly after the class declaration. For larger systems (i.e., when separately compiling a class' routines), the convention is to place the class declaration in a header file by itself and place all the procedure, iterator, and method definitions in a separate HLA unit and compile them by themselves.

10.4 Objects

Remember, a class definition is just a type. Therefore, when you declare a class type you haven't created a variable whose fields you can manipulate. An *object* is an *instance* of a class; that is, an object is a variable that is some class type. You declare objects (i.e., class variables) the same way you declare other variables: in a VAR, STATIC, or STORAGE section³. A pair of sample object declarations follow:

```
var
  T1: TypicalClass;
  T2: TypicalClass;
```

For a given class object, HLA allocates storage for each variable appearing in the VAR section of the class declaration. If you have two objects, *T1* and *T2*, of type *TypicalClass* then *T1.TCvar* is unique as is *T2.TCvar*. This is the intuitive result (similar to RECORD declarations); most data fields you define in a class will appear in the VAR declaration section.

Static data objects (e.g., those you declare in the STATIC section of a class declaration) are not unique among the objects of that class; that is, HLA allocates only a single static variable that all variables of that class share. For example, consider the following (partial) class declaration and object declarations:

```
type
  sc: class
    var
      i:int32;
    static
      s:int32;
      .
      .
      .
  endclass;

var
```

3. Technically, you could also declare an object in a READONLY section, but HLA does not allow you to define class constants, so there is little utility in declaring class objects in the READONLY section.

```
s1: sc;
s2: sc;
```

In this example, *s1.i* and *s2.i* are different variables. However, *s1.s* and *s2.s* are aliases of one another. Therefore, an instruction like “`mov(5, s1.s);`” also stores five into *s2.s*. Generally you use static class variables to maintain information about the whole class while you use class VAR objects to maintain information about the specific object. Since keeping track of class information is relatively rare, you will probably declare most class data fields in a VAR section.

You can also create dynamic instances of a class and refer to those dynamic objects via pointers. In fact, this is probably the most common form of object storage and access. The following code shows how to create pointers to objects and how you can dynamically allocate storage for an object:

```
var
  pSC: pointer to sc;
  .
  .
  .
  malloc( @size( sc ) );
  mov( eax, pSC );
  .
  .
  .
  mov( pSC, ebx );
  mov( (type sc [ebx]).i, eax );
```

Note the use of type coercion to cast the pointer in EBX as type *sc*.

10.5 Inheritance

Inheritance is one of the most fundamental ideas behind object-oriented programming. The basic idea behind inheritance is that a class inherits, or copies, all the fields from some class and then possibly expands the number of fields in the new data type. For example, suppose you created a data type *point* which describes a point in the planar (two dimensional) space. The class for this point might look like the following:

```
type
  point: class

    var
      x:int32;
      y:int32;

    method distance;

endclass;
```

Suppose you want to create a point in 3D space rather than 2D space. You can easily build such a data type as follows:

```
type
  point3D: class inherits( point );

    var
      z:int32;

endclass;
```


The INHERITS option on the CLASS declaration tells HLA to insert the fields of *point* at the beginning of the class. In this case, *point3D* inherits the fields of *point*. HLA always places the inherited fields at the beginning of a class object. The reason for this will become clear a little later. If you have an instance of *point3D* which you call *P3*, then the following 80x86 instructions are all legal:

```
mov( P3.x, eax );
add( P3.y, eax );
mov( eax, P3.z );
P3.distance();
```

Note that the *P3.distance* method invocation in this example calls the *point.distance* method. You do not have to write a separate *distance* method for the *point3D* class unless you really want to do so (see the next section for details). Just like the *x* and *y* fields, *point3D* objects inherit *point*'s methods.

10.6 Overriding

Overriding is the process of replacing an existing method in an inherited class with one more suitable for the new class. In the *point* and *point3D* examples appearing in the previous section, the *distance* method (presumably) computes the distance from the origin to the specified point. For a point on a two-dimensional plane, you can compute the distance using the function:

$$\text{dist} = \sqrt{x^2+y^2}$$

However, the distance for a point in 3D space is given by the equation:

$$\text{dist} = \sqrt{x^2+y^2+z^2}$$

Clearly, if you call the *distance* function for *point* for a *point3D* object you will get an incorrect answer. In the previous section, however, you saw that the *P3* object calls the distance function inherited from the *point* class. Therefore, this would produce an incorrect result.

In this situation the *point3D* data type must override the *distance* method with one that computes the correct value. You cannot simply redefine the *point3D* class by adding a *distance* method prototype:

```
type
point3D:  class inherits( point )

          var
            z:int32;

          method distance;  // This doesn't work!

endclass;
```

The problem with the *distance* method declaration above is that *point3D* already has a distance method – the one that it inherits from the *point* class. HLA will complain because it doesn't like two methods with the same name in a single class.

To solve this problem, we need some mechanism by which we can override the declaration of *point.distance* and replace it with a declaration for *point3D.distance*. To do this, you use the OVERRIDE keyword before the method declaration:

```
type
point3D:  class inherits( point )

          var
            z:int32;

          override method distance;  // This will work!

endclass;
```

The `OVERRIDE` prefix tells HLA to ignore the fact that `point3D` inherits a method named `distance` from the `point` class. Now, any call to the `distance` method via a `point3D` object will call the `point3D.distance` method rather than `point.distance`. Of course, once you override a method using the `OVERRIDE` prefix, you must supply the method in the implementation section of your code, e.g.,

```
method point3D.distance; nodisplay;

    << local declarations for the distance function >>

begin distance;

    << Code to implement the distance function >>

end distance;
```

10.7 Virtual Methods vs. Static Procedures

A little earlier, this chapter suggested that you could treat class methods and class procedures the same. There are, in fact, some major differences between the two (after all, why have methods if they're the same as procedures?). As it turns out, the differences between methods and procedures is crucial if you want to develop object-oriented programs. Methods provide the second feature necessary to support true polymorphism: virtual procedure calls⁴. A virtual procedure call is just a fancy name for an indirect procedure call (using a pointer associated with the object). The key benefit of virtual procedures is that the system automatically calls the right method when using pointers to generic objects.

Consider the following declarations using the `point` class from the previous sections:

```
var
    P2: point;
    P: pointer to point;
```

Given the declarations above, the following assembly statements are all legal:

```
mov( P2.x, eax );
mov( P2.y, ecx );
P2.distance();           // Calls point3D.distance.

lea( ebx, P2 );         // Store address of P2 into P.
mov( ebx, P );
P.distance();           // Calls point.distance.
```

Note that HLA lets you call a method via a pointer to an object rather than directly via an object variable. This is a crucial feature of objects in HLA and a key to implementing *virtual method calls*.

The magic behind polymorphism and inheritance is that object pointers are *generic*. In general, when your program references data indirectly through a pointer, the value of the pointer should be the address of the underlying data type associated with that pointer. For example, if you have a pointer to a 16-bit unsigned integer, you wouldn't normally use that pointer to access a 32-bit signed integer value. Similarly, if you have a pointer to some record, you would not normally cast that pointer to some other record type and access the fields of that other type⁵. With pointers to class objects, however, we can lift this restriction a bit. Pointers to objects may legally contain the address of the object's type *or the address of any object that inherits the fields of that type*. Consider the following declarations that use the `point` and `point3D` types from the previous examples:

```
var
```

4. Polymorphism literally means "many-faced." In the context of object-oriented programming polymorphism means that the same method name, e.g., `distance`, and refer to one of several different methods.

5. Of course, assembly language programmers break rules like this all the time. For now, let's assume we're playing by the rules and only access the data using the data type associated with the pointer.

```

P2: point;
P3: point3D;
p: pointer to point;
.
.
.
lea( ebx, P2 );
mov( ebx, p );
p.distance();           // Calls the point.distance method.
.
.
.
lea( ebx, P3 );
mov( ebx, p );         // Yes, this is semantically legal.
p.distance();         // Surprise, this calls point3D.distance.

```

Since *p* is a pointer to a *point* object, it might seem intuitive for *p.distance* to call the *point.distance* method. However, methods are *polymorphic*. If you've got a pointer to an object and you call a method associated with that object, the system will call the actual (overridden) method associated with the object, not the method specifically associated with the pointer's class type.

Class procedures behave differently than methods with respect to overridden procedures. When you call a class procedure indirectly through an object pointer, the system will always call the procedure associated with the underlying class associated with the pointer. So had *distance* been a procedure rather than a method in the previous examples, the “*p.distance()*” invocation would always call *point.distance*, even if *p* is pointing at a *point3D* object. The section on Object Initialization, later in this chapter, explains why methods and procedures are different (see “Object Implementation” on page 1071).

Note that iterators are also virtual; so like methods an object iterator invocation will always call the (overridden) iterator associated with the actual object whose address the pointer contains. To differentiate the semantics of methods and iterators from procedures, we will refer to the method/iterator calling semantics as *virtual procedures* and the calling semantics of a class procedure as a *static procedure*.

10.8 Writing Class Methods, Iterators, and Procedures

For each class procedure, method, and iterator prototype appearing in a class definition, there must be a corresponding procedure, method, or iterator appearing within the program (for the sake of brevity, this section will use the term *routine* to mean procedure, method, or iterator from this point forward). If the prototype does not contain the EXTERNAL option, then the code must appear in the same compilation unit as the class declaration. If the EXTERNAL option does follow the prototype, then the code may appear in the same compilation unit or a different compilation unit (as long as you link the resulting object file with the code containing the class declaration). Like external (non-class) procedures and iterators, if you fail to provide the code the linker will complain when you attempt to create an executable file. To reduce the size of the following examples, they will all define their routines in the same source file as the class declaration.

HLA class routines must always follow the class declaration in a compilation unit. If you are compiling your routines in a separate unit, the class declarations must still precede the code with the class declaration (usually via an #INCLUDE file). If you haven't defined the class by the time you define a routine like *point.distance*, HLA doesn't know that *point* is a class and, therefore, doesn't know how to handle the routine's definition.

Consider the following declarations for a *point2D* class:

```

type
  point2D: class

      const

```

```

    UnitDistance: real32 := 1.0;

var
    x: real32;
    y: real32;

static
    LastDistance: real32;

method distance( fromX: real32; fromY:real32 ); returns( "st0" );
procedure InitLastDistance;

endclass;

```

The distance function for this class should compute the distance from the object's point to (fromX,fromY). The following formula describes this computation:

$$\sqrt{(x - \text{fromX})^2 + (y - \text{fromY})^2}$$

A first pass at writing the distance method might produce the following code:

```

method point2D.distance( fromX:real32; fromY:real32 ); nodisplay;
begin distance;

    fld( x );           // Note: this doesn't work!
    fld( fromX );       // Compute (x-fromX)
    fsub();
    fld( st0 );         // Duplicate value on TOS.
    fmul();             // Compute square of difference.

    fld( y );           // This doesn't work either.
    fld( fromY );       // Compute (y-fromY)
    fsub();
    fld( st0 );         // Compute the square of the difference.
    fmul();

    fsqrt();

end distance;

```

This code probably looks like it should work to someone who is familiar with an object-oriented programming language like C++ or Delphi. However, as the comments indicate, the instructions that push the *x* and *y* variables onto the FPU stack don't work – HLA doesn't automatically define the symbols associated with the data fields of a class within that class' routines.

To learn how to access the data fields of a class within that class' routines, we need to back up a moment and discover some very important implementation details concerning HLA's classes. To do this, consider the following variable declarations:

```

var
    Origin: point2D;
    PtInSpace: point2D;

```

Remember, whenever you create two objects like *Origin* and *PtInSpace*, HLA reserves storage for the *x* and *y* data fields for both of these objects. However, there is only one copy of the *point2D.distance* method in memory. Therefore, were you to call *Origin.distance* and *PtInSpace.distance*, the system would call the same routine for both method invocations. Once inside that method, one has to wonder what an instruction like “fld(x);” would do. How does it associate *x* with *Origin.x* or *PtInSpace.x*? Worse still, how would this code differentiate between the data field *x* and a global object *x*? In HLA, the answer is “it doesn't.” You do

not specify the data field names within a class routine by simply using their names as though they were common variables.

To differentiate *Origin.x* from *PtInSpace.x* within class routines, HLA automatically passes a pointer to an object's data fields whenever you call a class routine. Therefore, you can reference the data fields indirectly off this pointer. HLA passes this object pointer in the ESI register. This is one of the few places where HLA-generated code will modify one of the 80x86 registers behind your back: **anytime you call a class routine, HLA automatically loads the ESI register with the object's address.** Obviously, you cannot count on ESI's value being preserved across class routine class nor can you pass parameters to the class routine in the ESI register (though it is perfectly reasonable to specify "@USE ESI;" to allow HLA to use the ESI register when setting up other parameters). For class methods and iterators (but not procedures), HLA will also load the EDI register with the address of the class' *virtual method table* (see "Virtual Method Tables" on page 1073). While the virtual method table address isn't as interesting as the object address, keep in mind that **HLA-generated code will overwrite any value in the EDI register when you call a method or an iterator.** Again, "EDI" is a good choice for the @USE operand for methods since HLA will wipe out the value in EDI anyway.

Upon entry into a class routine, ESI contains a pointer to the (non-static) data fields associated with the class. Therefore, to access fields like *x* and *y* (in our *point2D* example), you could use an address expression like the following:

```
(type point2D [esi]).x
```

Since you use ESI as the base address of the object's data fields, it's a good idea not to disturb ESI's value within the class routines (or, at least, preserve ESI's value if you need to access the objects data fields after some point where you must use ESI for some other purpose). Note that if you call an iterator or a method you do not have to preserve EDI (unless, for some reason, you need access to the virtual method table, which is unlikely).

Accessing the fields of a data object within a class' routines is such a common operation that HLA provides a shorthand notation for casting ESI as a pointer to the class object: **THIS**. Within a class in HLA, the reserved word **THIS** automatically expands to a string of the form "(type *classname* [esi])" substituting, of course, the appropriate class name for *classname*. Using the **THIS** keyword, we can (correctly) rewrite the previous distance method as follows:

```
method point2D.distance( fromX:real32; fromY:real32 ); nodisplay;
begin distance;

    fld( this.x );
    fld( fromX );      // Compute (x-fromX)
    fsub();
    fld( st0 );       // Duplicate value on TOS.
    fmul();           // Compute square of difference.

    fld( this.y );
    fld( fromY );     // Compute (y-fromY)
    fsub();
    fld( st0 );       // Compute the square of the difference.
    fmul();

    fsqrt();

end distance;
```

Don't forget that calling a class routine wipes out the value in the ESI register. This isn't obvious from the syntax of the routine's invocation. It is especially easy to forget this when calling some class routine from inside some other class routine; don't forget that if you do this the internal call wipes out the value in ESI and on return from that call ESI no longer points at the original object. Always push and pop ESI (or otherwise preserve ESI's value) in this situation, e.g.,

```

.
.
fld( this.x );    // ESI points at current object.
.
.
.
push( esi );      // Preserve ESI across this method call.
SomeObject.SomeMethod();
pop( esi );
.
.
.
lea( ebx, this.x );    // ESI points at original object here.

```

The **THIS** keyword provides access to the class variables you declare in the VAR section of a class. You can also use **THIS** to call other class routines associated with the current object, e.g.,

```

this.distance( 5.0, 6.0 );

```

To access class constants and **STATIC** data fields you generally do not use the **THIS** pointer. HLA associates constant and static data fields with the whole class, not a specific object. To access these class members, just use the class name in place of the object name. For example, to access the *UnitDistance* constant in the *point2D* class you could use a statement like the following:

```

fld( point2D.UnitDistance );

```

As another example, if you wanted to update the *LastDistance* field in the *point2D* class each time you computed a distance, you could rewrite the *point2D.distance* method as follows:

```

method point2D.distance( fromX:real32; fromY:real32 ); nodisplay;
begin distance;

    fld( this.x );
    fld( fromX );    // Compute (x-fromX)
    fsub();
    fld( st0 );      // Duplicate value on TOS.
    fmul();          // Compute square of difference.

    fld( this.y );
    fld( fromY );    // Compute (y-fromY)
    fsub();
    fld( st0 );      // Compute the square of the difference.
    fmul();

    fsqrt();

    fst( point2D.LastDistance );    // Update shared (STATIC) field.

end distance;

```

To understand why you use the class name when referring to constants and static objects but you use **THIS** to access VAR objects, check out the next section.

Class procedures are also static objects, so it is possible to call a class procedure by specifying the class name rather than an object name in the procedure invocation, e.g., both of the following are legal:

```

Origin.InitLastDistance();
point2D.InitLastDistance();

```

There is, however, a subtle difference between these two class procedure calls. The first call above loads **ESI** with the address of the *Origin* object prior to actually calling the *InitLastDistance* procedure. The second call, however, is a direct call to the class procedure without referencing an object; therefore, HLA doesn't

know what object address to load into the ESI register. In this case, HLA loads NULL (zero) into ESI prior to calling the *InitLastDistance* procedure. Because you can call class procedures in this manner, it's always a good idea to check the value in ESI within your class procedures to verify that HLA contains an object address. Checking the value in ESI is a good way to determine which calling mechanism is in use. Later, this chapter will discuss constructors and object initialization; there you will see a good use for static procedures and calling those procedures directly (rather than through the use of an object).

10.9 Object Implementation

In a high level object-oriented language like C++ or Delphi, it is quite possible to master the use of objects without really understanding how the machine implements them. One of the reasons for learning assembly language programming is to fully comprehend low-level implementation details so one can make educated decisions concerning the use of programming constructs like objects. Further, since assembly language allows you to poke around with data structures at a very low-level, knowing how HLA implements objects can help you create certain algorithms that would not be possible without a detailed knowledge of object implementation. Therefore, this section, and its corresponding subsections, explains the low-level implementation details you will need to know in order to write object-oriented HLA programs.

HLA implements objects in a manner quite similar to records. In particular, HLA allocates storage for all VAR objects in a class in a sequential fashion, just like records. Indeed, if a class consists of only VAR data fields, the memory representation of that class is nearly identical to that of a corresponding RECORD declaration. Consider the Student record declaration taken from Volume Three and the corresponding class:

```
type
  student:  record
            Name: char[65];
            Major: int16;
            SSN:  char[12];
            Midterm1: int16;
            Midterm2: int16;
            Final: int16;
            Homework: int16;
            Projects: int16;
            endrecord;

  student2: class
            Name: char[65];
            Major: int16;
            SSN:  char[12];
            Midterm1: int16;
            Midterm2: int16;
            Final: int16;
            Homework: int16;
            Projects: int16;
            endclass;
```

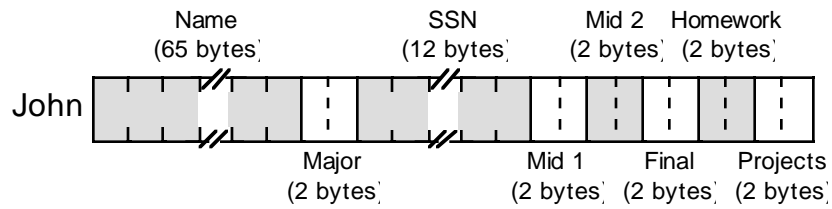


Figure 10.1 Student RECORD Implementation in Memory

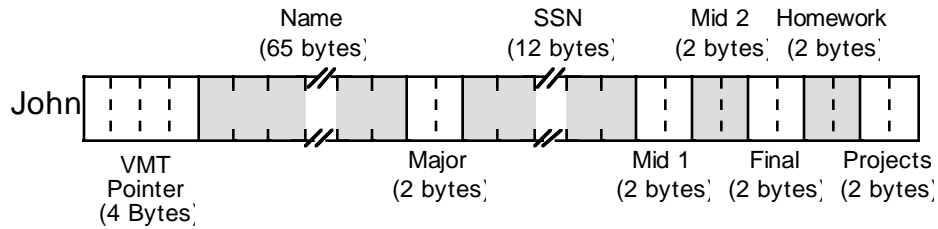


Figure 10.2 Student CLASS Implementation in Memory

If you look carefully at these two figures, you'll discover that the only difference between the class and the record implementations is the inclusion of the VMT (virtual method table) pointer field at the beginning of the class object. This field, which is always present in a class, contains the address of the class' virtual method table which, in turn, contains the addresses of all the class' methods and iterators. The VMT field, by the way, is present even if a class doesn't contain any methods or iterators.

As pointed out in previous sections, HLA does not allocate storage for STATIC objects within the object's storage. Instead, HLA allocates a single instance of each static data field that all objects share. As an example, consider the following class and object declarations:

```

type
  tHasStatic: class

    var
      i:int32;
      j:int32;
      r:real32;

    static
      c:char[2];
      b:byte;

  endclass;

var
  hs1: tHasStatic;
  hs2: tHasStatic;

```

Figure 10.3 shows the storage allocation for these two objects in memory.

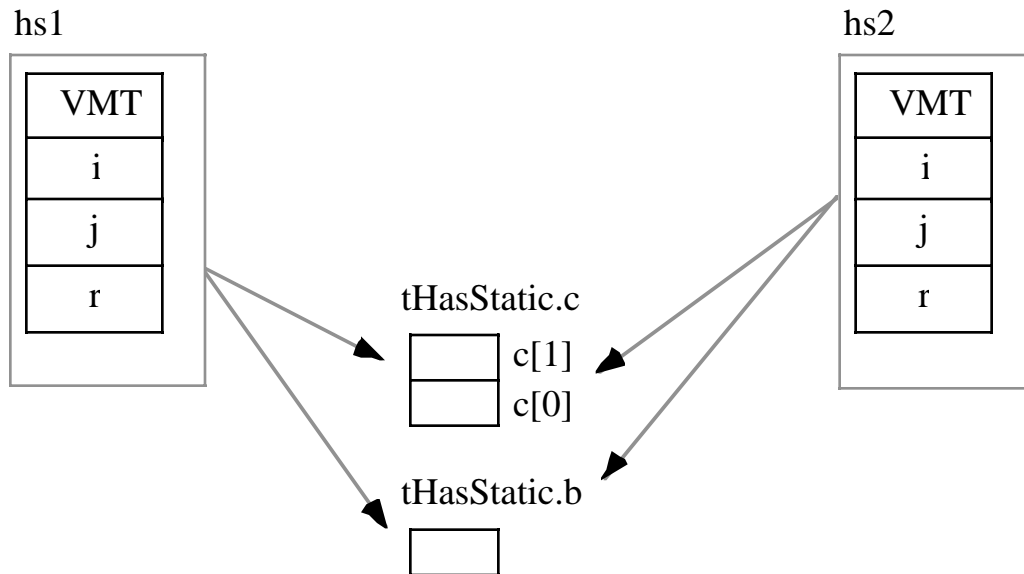


Figure 10.3 Object Allocation with Static Data Fields

Of course, `CONST`, `VAL`, and `#MACRO` objects do not have any run-time memory requirements associated with them, so HLA does not allocate any storage for these fields. Like the `STATIC` data fields, you may access `CONST`, `VAL`, and `#MACRO` fields using the class name as well as an object name. Hence, even if `tHasStatic` has these types of fields, the memory organization for `tHasStatic` objects would still be the same as shown in Figure 10.3.

Other than the presence of the virtual method table pointer (VMT), the presence of methods, iterators, and procedures has no impact on the storage allocation of an object. Of course, the machine instructions associated with these routines does appear somewhere in memory. So in a sense the code for the routines is quite similar to static data fields insofar as all the objects share a single instance of the routine.

10.9.1 Virtual Method Tables

When HLA calls a class procedure, it directly calls that procedure using a `CALL` instruction, just like any normal non-class procedure call. Methods and iterators are another story altogether. Each object in the system carries a pointer to a virtual method table which is an array of pointers to all the methods and iterators appearing within the object's class.

SomeObject

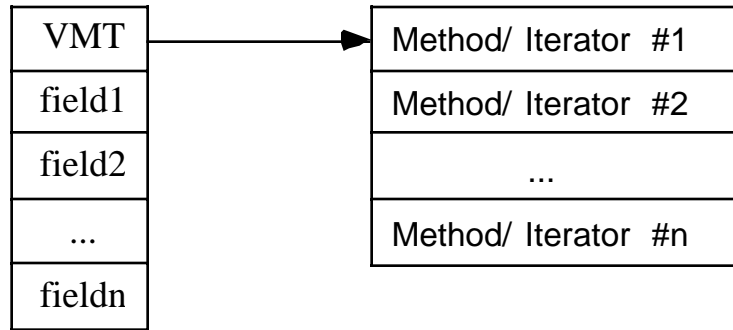


Figure 10.4 Virtual Method Table Organization

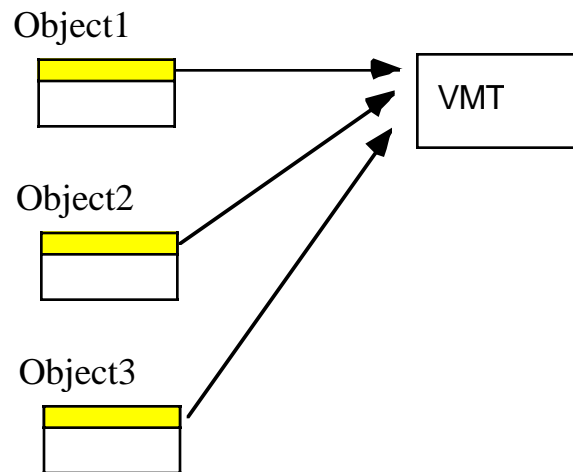
Each iterator or method you declare in a class has a corresponding entry in the virtual method table. That dword entry contains the address of the first instruction of that iterator or method. To call a class method or iterator is a bit more work than calling a class procedure (it requires one additional instruction plus the use of the EDI register). Here is a typical calling sequence for a method:

```

mov( ObjectAdrs, ESI );           // All class routines do this.
mov( [esi], edi );               // Get the address of the VMT into EDI
call( (type dword [edi+n]));     // "n" is the offset of the method's entry
                                 // in the VMT.

```

For a given class there is only one copy of the VMT in memory. This is a static object so all objects of a given class type share the same VMT. This is reasonable since all objects of the same class type have exactly the same methods and iterators (see Figure 10.5).



Note: Objects are all the same class type

Figure 10.5 All Objects That are the Same Class Type Share the Same VMT

Although HLA builds the VMT record structure as it encounters methods and iterators within a class, HLA does not automatically create the actual run-time virtual method table for you. You must explicitly

declare this table in your program. To do this, you include a statement like the following in a `STATIC` or `READONLY` declaration section of your program, e.g.,

```
readonly
    VMT( classname );
```

Since the addresses in a virtual method table should never change during program execution, the `READONLY` section is probably the best choice for declaring VMTs. It should go without saying that changing the pointers in a VMT is, in general, a really bad idea. So putting VMTs in a `STATIC` section is usually not a good idea.

A declaration like the one above defines the variable `classname._VMT_`. In section 10.10 (see “Constructors and Object Initialization” on page 1079) you see that you’ll need this name when initializing object variables. The class declaration automatically defines the `classname._VMT_` symbol as an external static variable. The declaration above just provides the actual definition of this external symbol.

The declaration of a VMT uses a somewhat strange syntax because you aren’t actually declaring a new symbol with this declaration, you’re simply supplying the data for a symbol that you previously declared implicitly by defining a class. That is, the class declaration defines the static table variable `classname._VMT_`, all you’re doing with the VMT declaration is telling HLA to emit the actual data for the table. If, for some reason, you would like to refer to this table using a name other than `classname._VMT_`, HLA does allow you to prefix the declaration above with a variable name, e.g.,

```
readonly
    myVMT: VMT( classname );
```

In this declaration, `myVMT` is an alias of `classname._VMT_`. As a general rule, you should avoid aliases in a program because they make the program more difficult to read and understand. Therefore, it is unlikely that you would ever really need to use this type of declaration.

Like any other global static variable, there should be only one instance of a VMT for a given class in a program. The best place to put the VMT declaration is in the same source file as the class’ method, iterator, and procedure code (assuming they all appear in a single file). This way you will automatically link in the VMT whenever you link in the routines for a given class.

10.9.2 Object Representation with Inheritance

Up to this point, the discussion of the implementation of class objects has ignored the possibility of inheritance. Inheritance only affects the memory representation of an object by adding fields that are not explicitly stated in the class declaration.

Adding inherited fields from a *base class* to another class must be done carefully. Remember, an important attribute of a class that inherits fields from a base class is that you can use a pointer to the base class to access the inherited fields from that base class in another class. As an example, consider the following classes:

```
type
    tBaseClass: class
        var
            i:uns32;
            j:uns32;
            r:real32;

        method mBase;
    endclass;

    tChildClassA: class inherits( tBaseClass );
        var
            c:char;
            b:boolean;
```

```

        w:word;

    method mA;
endclass;

tChildClassB: class inherits( tBaseClass );
    var
        d:dword;
        c:char;
        a:byte[3];
endclass;

```

Since both *tChildClassA* and *tChildClassB* inherit the fields of *tBaseClass*, these two child classes include the *i*, *j*, and *r* fields as well as their own specific fields. Furthermore, whenever you have a pointer variable whose base type is *tBaseClass*, it is legal to load this pointer with the address of any child class of *tBaseClass*; therefore, it is perfectly reasonable to load such a pointer with the address of a *tChildClassA* or *tChildClassB* variable, e.g.,

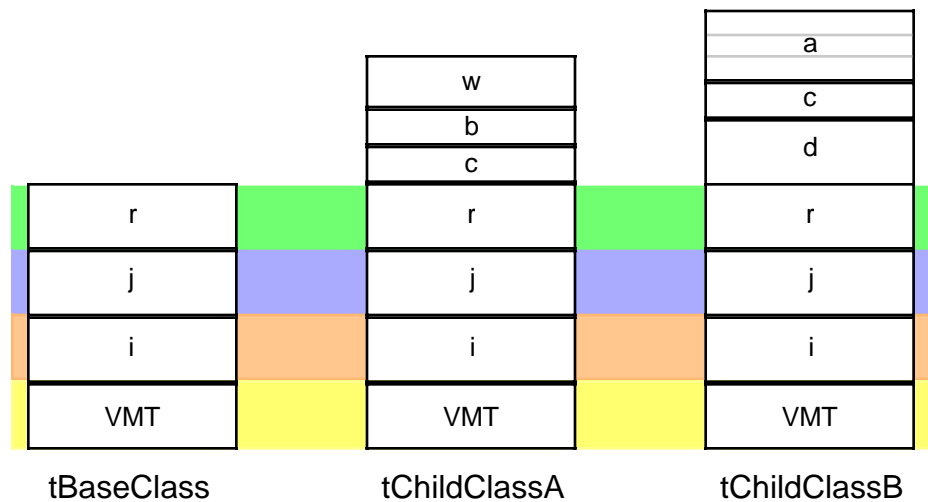
```

var
    B1: tBaseClass;
    CA: tChildClassA;
    CB: tChildClassB;
    ptr: pointer to tBaseClass;
    .
    .
    .
    lea( ebx, B1 );
    mov( ebx, ptr );
    << Use ptr >>
    .
    .
    .
    lea( eax, CA );
    mov( ebx, ptr );
    << Use ptr >>
    .
    .
    .
    lea( eax, CB );
    mov( eax, ptr );
    << Use ptr >>

```

Since *ptr* points at an object of *tBaseClass*, you may legally (from a semantic sense) access the *i*, *j*, and *r* fields of the object where *ptr* is pointing. It is not legal to access the *c*, *b*, *w*, or *d* fields of the *tChildClassA* or *tChildClassB* objects since at any one given moment the program may not know exactly what object type *ptr* references.

In order for inheritance to work properly, the *i*, *j*, and *r* fields must appear at the same offsets all child classes as they do in *tBaseClass*. This way, an instruction of the form “mov((type tBaseClass [ebx]).i, eax);” will correct access the *i* field even if EBX points at an object of type *tChildClassA* or *tChildClassB*. Figure 10.6 shows the layout of the child and base classes:



Derived (child) classes locate their inherited fields at the same offsets as those fields in the base class.

Figure 10.6 Layout of Base and Child Class Objects in Memory

Note that the new fields in the two child classes bear no relation to one another, even if they have the same name (e.g., field *c* in the two child classes does not lie at the same offset). Although the two child classes share the fields they inherit from their common base class, any new fields they add are unique and separate. Two fields in different classes share the same offset only by coincidence.

All classes (even those that aren't related to one another) place the pointer to the virtual method table at offset zero within the object. There is a single VMT associated with each class in a program; even classes that inherit fields from some base class have a VMT that is (generally) different than the base class' VMT. shows how objects of type `tBaseClass`, `tChildClassA` and `tChildClassB` point at their specific VMTs:

```

var
  B1: tBaseClass;
  CA: tChildClassA;
  CB: tChildClassB;
  CB2: tChildClassB;
  CA2: tChildClassA;

```

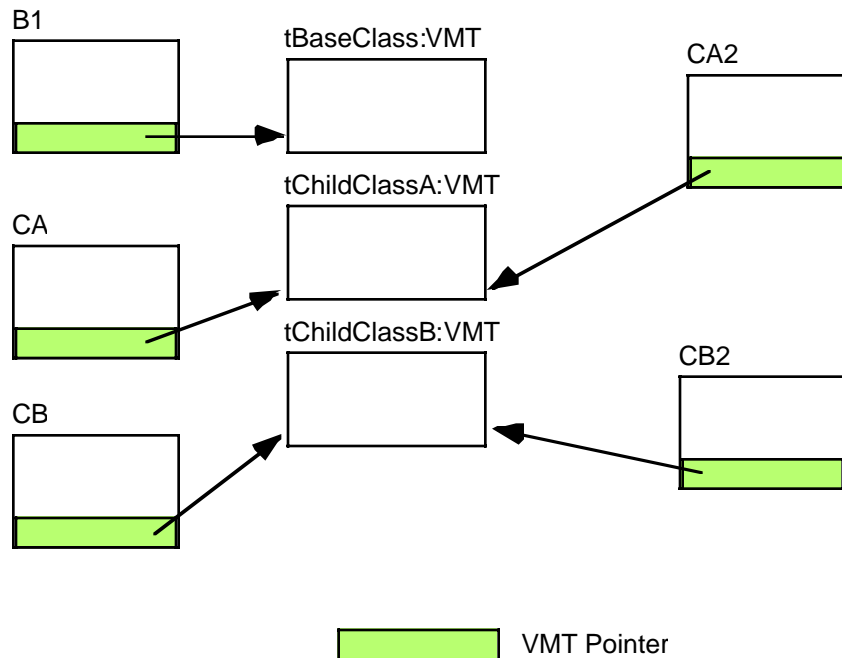


Figure 10.7 Virtual Method Table References from Objects

A virtual method table is nothing more than an array of pointers to the methods and iterators associated with a class. The address of the first method or iterator appearing in a class is at offset zero, the address of the second appears at offset four, etc. You can determine the offset value for a given iterator or method by using the `@offset` function. If you want to call a method or iterator directly (using 80x86 syntax rather than HLA's high level syntax), you code use code like the following:

```

var
  sc: tBaseClass;
  .
  .
  .
  lea( esi, sc );           // Get the address of the object (& VMT).
  mov( [esi], edi );       // Put address of VMT into EDI.
  call( (type dword [edi+@offset( tBaseClass.mBase )] ) );

```

Of course, if the method has any parameters, you must push them onto the stack before executing the code above. Don't forget, when making direct calls to a method, that you must load ESI with the address of the object. Any field references within the method will probably depend upon ESI containing this address. The choice of EDI to contain the VMT address is nearly arbitrary. Unless you're doing something tricky (like using EDI to obtain run-time type information), you could use any register you please here. As a general rule, you should use EDI when simulating class iterator/method calls because this is the convention that HLA employs and most programmers will expect this.

Whenever a child class inherits fields from some base class, the child class' VMT also inherits entries from the base class' VMT. For example, the VMT for class *tBaseClass* contains only a single entry – a pointer to method *tBaseClass.mBase*. The VMT for class *tChildClassA* contains two entries: a pointer to *tBaseClass.mBase* and *tChildClassA.mA*. Since *tChildClassB* doesn't define any new methods or iterators, *tChildClassB*'s VMT contains only a single entry, a pointer to the *tBaseClass.mBase* method. Note that *tChildClassB*'s VMT is identical to *tBaseClass*' VMT. Nevertheless, HLA produces two distinct VMTs. This is a critical fact that we will make use of a little later. Figure 10.8 shows the relationship between these VMTs:

Virtual Method Tables for Derived (inherited) Classes

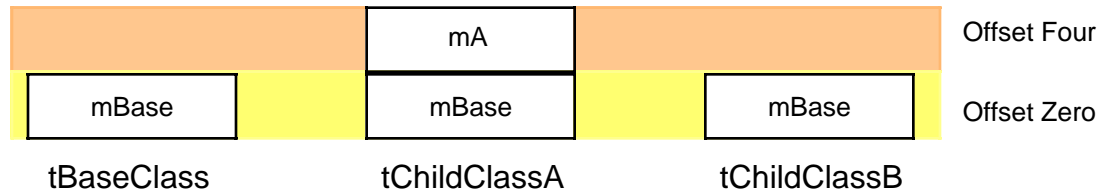


Figure 10.8 Virtual Method Tables for Inherited Classes

Although the VMT always appears at offset zero in an object (and, therefore, you can access the VMT using the address expression “[ESI]” if ESI points at an object), HLA actually inserts a symbol into the symbol table so you may refer to the VMT symbolically. The symbol *_pVMT_* (pointer to Virtual Method Table) provides this capability. So a more readable way to access the VMT pointer (as in the previous code example) is

```
lea( esi, sc );
mov( (type tBaseClass [esi])._pVMT_, edi );
call( (type dword [edi+@offset( tBaseClass.mBase )] ) );
```

If you need to access the VMT directly, there are a couple ways to do this. Whenever you declare a class object, HLA automatically includes a field named *_VMT_* as part of that class. *_VMT_* is a static array of double word objects. Therefore, you may refer to the VMT using an identifier of the form *class-name._VMT_*. Generally, you shouldn't access the VMT directly, but as you'll see shortly, there are some good reasons why you need to know the address of this object in memory.

10.10 Constructors and Object Initialization

If you've tried to get a little ahead of the game and write a program that uses objects prior to this point, you've probably discovered that the program inexplicably crashes whenever you attempt to run it. We've covered a lot of material in this chapter thus far, but you are still missing one crucial piece of information – how to properly initialize objects prior to use. This section will put the final piece into the puzzle and allow you to begin writing programs that use classes.

Consider the following object declaration and code fragment:

```
var
  bc: tBaseClass;
  .
  .
  .
  bc.mBase( );
```

Remember that variables you declare in the VAR section are uninitialized at run-time. Therefore, when the program containing these statements gets around to executing *bc.mBase*, it executes the three-statement sequence you've seen several times already:

```
lea( esi, bc);
mov( [esi], edi );
call( (type dword [edi+@offset( tBaseClass.mBase )] ) );
```

The problem with this sequence is that it loads EDI with an undefined value assuming you haven't previously initialized the *bc* object. Since EDI contains a garbage value, attempting to call a subroutine at address "[EDI+@offset(tBaseClass.mBase)]" will likely crash the system. Therefore, before using an object, you must initialize the *_pVMT_* field with the address of that object's VMT. One easy way to do this is with the following statement:

```
mov( &tBaseClass._VMT_, bc._pVMT_ );
```

Always remember, **before using an object, be sure to initialize the virtual method table pointer for that field.**

Although you must initialize the virtual method table pointer for all objects you use, this may not be the only field you need to initialize in those objects. Each specific class may have its own application-specific initialization that is necessary. Although the initialization may vary by class, you need to perform the same initialization on each object of a specific class that you use. If you ever create more than a single object from a given class, it is probably a good idea to create a procedure to do this initialization for you. This is such a common operation that object-oriented programmers have given these initialization procedures a special name: *constructors*.

Some object-oriented languages (e.g., C++) use a special syntax to declare a constructor. Others (e.g., Delphi) simply use existing procedure declarations to define a constructor. One advantage to employing a special syntax is that the language knows when you define a constructor and can automatically generate code to call that constructor for you (whenever you declare an object). Languages, like Delphi, require that you explicitly call the constructor; this can be a minor inconvenience and a source of defects in your programs. HLA does not use a special syntax to declare constructors – you define constructors using standard class procedures. As such, you will need to explicitly call the constructors in your program; however, you'll see an easy method for automating this in a later section of this chapter.

Perhaps the most important fact you must remember is that **constructors must be class procedures**. You must not define constructors as methods (or iterators). The reason is quite simple: one of the tasks of the constructor is to initialize the pointer to the virtual method table and you cannot call a class method or iterator until after you've initialized the VMT pointer. Since class procedures don't use the virtual method table, you can call a class procedure prior to initializing the VMT pointer for an object.

By convention, HLA programmers use the name *Create* for the class constructor. There is no requirement that you use this name, but by doing so you will make your programs easier to read and follow by other programmers.

As you may recall, you can call a class procedure via an object reference or a class reference. E.g., if *clsProc* is a class procedure of class *tClass* and *Obj* is an object of type *tClass*, then the following two class procedure invocations are both legal:

```
tClass.clsProc( );
Obj.clsProc( );
```

There is a big difference between these two calls. The first one calls *clsProc* with ESI containing zero (NULL) while the second invocation loads the address of *Obj* into ESI before the call. We can use this fact to determine within a method the particular calling mechanism.

10.10.1 Dynamic Object Allocation Within the Constructor

As it turns out, most programs allocate objects dynamically using *malloc* and refer to those objects indirectly using pointers. This adds one more step to the initialization process – allocating storage for the object. The constructor is the perfect place to allocate this storage. Since you probably won't need to allocate all objects dynamically, you'll need two types of constructors: one that allocates storage and then initializes the object, and another that simply initializes an object that already has storage.

Another constructor convention is to merge these two constructors into a single constructor and differentiate the type of constructor call by the value in ESI. On entry into the class' *Create* procedure, the program checks the value in ESI to see if it contains NULL (zero). If so, the constructor calls *malloc* to allocate storage for the object and returns a pointer to the object in ESI. If ESI does not contain NULL upon entry into the procedure, then the constructor assumes that ESI points at a valid object and skips over the memory allocation statements. At the very least, a constructor initializes the pointer to the VMT; therefore, the minimalist constructor will look like the following:

```
procedure tBaseClass.mBase; nodisplay;
begin mBase;

    if( ESI = 0 ) then

        push( eax ); // Malloc returns its result here, so save it.
        malloc( @size( tBaseClass ) );
        mov( eax, esi ); // Put pointer into ESI;
        pop( eax );

    endif;

    // Initialize the pointer to the VMT:
    // (remember, "this" is shorthand for (type tBaseClass [esi])"

    mov( &tBaseClass._VMT_, this._pVMT_ );

    // Other class initialization would go here.

end mBase;
```

After you write a constructor like the one above, you choose an appropriate calling mechanism based on whether your object's storage is already allocated. For pre-allocated objects (i.e., those you've declared in VAR, STATIC, or STORAGE sections⁶ or those you've previously allocated storage for via *malloc*) you simply load the address of the object into ESI and call the constructor. For those objects you declare as a variable, this is very easy – just call the appropriate *Create* constructor:

```
var
    bc0: tBaseClass;
    bcp: pointer to tBaseClass;
    .
    .
    .
    bc0.Create(); // Initializes pre-allocated bc0 object.
    .
    .
    .
    malloc( @size( tBaseClass ) ); // Allocate storage for bcp object.
    mov( eax, bcp );
    .
    .
```

6. You generally do not declare objects in READONLY sections because you cannot initialize them.

```
bcp.Create(); // Initializes pre-allocated bcp object.
```

Note that although *bcp* is a pointer to a *tBaseClass* object, the *Create* method does not automatically allocate storage for this object. The program already allocates the storage earlier. Therefore, when the program calls *bcp.Create* it loads ESI with the address contained within *bcp*; since this is not NULL, the *tBaseClass.Create* procedure does not allocate storage for a new object. By the way, the call to *bcp.Create* emits the following sequence of machine instructions:

```
mov( bcp, esi );
call tBaseClass.Create;
```

Until now, the code examples for a class procedure call always began with an LEA instruction. This is because all the examples to this point have used object variables rather than pointers to object variables. Remember, a class procedure (method/iterator) call passes the address of the object in the ESI register. For object variables HLA emits an LEA instruction to obtain this address. For pointers to objects, however, the actual object address is the *value* of the pointer variable; therefore, to load the address of the object into ESI, HLA emits a MOV instruction that copies the value of the pointer into the ESI register.

In the example above, the program preallocates the storage for an object prior to calling the object constructor. While there are several reasons for preallocating object storage (e.g., you're creating a dynamic array of objects), you can achieve most simple object allocations like the one above by calling a standard *Create* method (i.e., one that allocates storage for an object if ESI contains NULL). The following example demonstrates this:

```
var
  bcp2: pointer to tBaseClass;
  .
  .
  .
tBaseClass.Create(); // Calls Create with ESI=NULL.
mov( esi, bcp2 ); // Save pointer to new class object in bcp2.
```

Remember, a call to a *tBaseClass.Create* constructor returns a pointer to the new object in the ESI register. It is the caller's responsibility to save the pointer this function returns into the appropriate pointer variable; the constructor does not automatically do this for you.

10.10.2 Constructors and Inheritance

Constructors for derived (child) classes that inherit fields from a base class represent a special case. Each class must have its own constructor but needs the ability to call the base class constructor. This section explains the reasons for this and how to do this.

A derived class inherits the *Create* procedure from its base class. However, you must override this procedure in a derived class because the derived class probably requires more storage than the base class and, therefore, you will probably need to use a different call to *malloc* to allocate storage for a dynamic object. Hence, it is very unusual for a derived class not to override the definition of the *Create* procedure.

However, overriding a base class' *Create* procedure has problems of its own. When you override the base class' *Create* procedure, you take the full responsibility of initializing the (entire) object, including all the initialization required by the base class. At the very least, this involves putting duplicate code in the overridden procedure to handle the initialization usually done by the base class constructor. In addition to make your program larger (by duplicating code already present in the base class constructor), this also violates information hiding principles since the derived class must be aware of all the fields in the base class (including those that are logically private to the base class). What we need here is the ability to call a base class' constructor from within the derived class' destructor and let that call do the lower-level initialization of the base class' fields. Fortunately, this is an easy thing to do in HLA.

Consider the following class declarations (which does things the hard way):

```

type
  tBase: class
    var
      i:uns32;
      j:int32;

    procedure Create(); returns( "esi" );
  endclass;

  tDerived: class inherits( tBase );
    var
      r: real64;

    override procedure Create(); returns( "esi" );
  endclass;

procedure tBase.Create; @nodisplay;
begin Create;

  if( esi = 0 ) then

    push( eax );
    mov( malloc( @size( tBase ) ), esi );
    pop( eax );

  endif;
  mov( &tBase._VMT_, this._pVMT_ );
  mov( 0, this.i );
  mov( -1, this.j );

end Create;

procedure tDerived.Create; @nodisplay;
begin Create;

  if( esi = 0 ) then

    push( eax );
    mov( malloc( @size( tDerived ) ), esi );
    pop( eax );

  endif;

  // Initialize the VMT pointer for this object:

  mov( &tDerived._VMT_, this._pVMT_ );

  // Initialize the "r" field of this particular object:

  fldz();
  fstp( this.r );

  // Duplicate the initialization required by tBase.Create:

  mov( 0, this.i );
  mov( -1, this.j );

end Create;

```

Let's take a closer look at the *tDerived.Create* procedure above. Like a conventional constructor, it begins by checking ESI and allocates storage for a new object if ESI contains NULL. Note that the size of a

tDerived object includes the size required by the inherited fields, so this properly allocates the necessary storage for all fields in a *tDerived* object.

Next, the *tDerived.Create* procedure initializes the VMT pointer field of the object. Remember, each class has its own VMT and, specifically, derived classes do not use the VMT of their base class. Therefore, this constructor must initialize the `_pVMT_` field with the address of the *tDerived* VMT.

After initializing the VMT pointer, the *tDerived* constructor initializes the value of the *r* field to 0.0 (remember, `FLDZ` loads zero onto the FPU stack). This concludes the *tDerived*-specific initialization.

The remaining instructions in *tDerived.Create* are the problem. These statements duplicate some of the code appearing in the *tBase.Create* procedure. The problem with code duplication becomes really apparent when you decide to modify the initial values of these fields; if you've duplicated the initialization code in derived classes, you will need to change the initialization code in more than one *Create* procedure. More often than not, this results in defects in the derived class *Create* procedures, especially if those derived classes appear in different source files than the base class.

Another problem with burying base class initialization in derived class constructors is the violation of the information hiding principle. Some fields of the base class may be *logically private*. Although HLA does not explicitly support the concept of public and private fields in a class (as, say, C++ does), well-disciplined programmers will still partition the fields as private or public and then only use the private fields in class routines belonging to that class. Initializing these private fields in derived classes is not acceptable to such programmers. Doing so will make it very difficult to change the definition and implementation of some base class at a later date.

Fortunately, HLA provides an easy mechanism for calling the inherited constructor within a derived class' constructor. All you have to do is call the base constructor using the classname syntax, e.g., you could call *tBase.Create* directly from within *tDerived.Create*. By calling the base class constructor, your derived class constructors can initialize the base class fields without worrying about the exact implementation (or initial values) of the base class.

Unfortunately, there are two types of initialization that every (conventional) constructor does that will affect the way you call a base class constructor: all conventional constructors allocate memory for the class if `ESI` contains zero and all conventional constructors initialize the VMT pointer. Fortunately, it is very easy to deal with these two problems

The memory required by an object of some most base class is usually less than the memory required for an object of a class you derive from that base class (because the derived classes usually add more fields). Therefore, you cannot allow the base class constructor to allocate the storage when you call it from inside the derived class' constructor. This problem is easily solved by checking `ESI` within the derived class constructor and allocating any necessary storage for the object *before* calling the base class constructor.

The second problem is the initialization of the VMT pointer. When you call the base class' constructor, it will initialize the VMT pointer with the address of the base class' virtual method table. A derived class object's `_pVMT_` field, however, must point at the virtual method table for the derived class. Calling the base class constructor will always initialize the `_pVMT_` field with the wrong pointer; to properly initialize the `_pVMT_` field with the appropriate value, the derived class constructor must store the address of the derived class' virtual method table into the `_pVMT_` field after the call to the base class constructor (so that it overwrites the value written by the base class constructor).

The *tDerived.Create* constructor, rewritten to call the *tBase.Create* constructors, follows:

```
procedure tDerived.Create; @nodisplay;
begin Create;

    if( esi = 0 ) then

        push( eax );
        mov( malloc( @size( tDerived ) ), esi );
        pop( eax );

    endif;
```

```

// Call the base class constructor to do any initialization
// needed by the base class. Note that this call must follow
// the object allocation code above (so ESI will always contain
// a pointer to an object at this point and tBase.Create will
// never allocate storage).

tBase.Create();

// Initialize the VMT pointer for this object. This code
// must always follow the call to the base class constructor
// because the base class constructor also initializes this
// field and we don't want the initial value supplied by
// tBase.Create.

mov( &tDerived._VMT_, this._pVMT_ );

// Initialize the "r" field of this particular object:

fldz();
fstp( this.r );

end Create;

```

This solution solves all the above concerns with derived class constructors.

10.10.3 Constructor Parameters and Procedure Overloading

All the constructor examples to this point have not had any parameters. However, there is nothing special about constructors that prevent the use of parameters. Constructors are procedures therefore you can specify any number and types of parameters you choose. You can use these parameter values to initialize certain fields or control how the constructor initializes the fields. Of course, you may use constructor parameters for any purpose you'd use parameters in any other procedure. In fact, about the only issue you need concern yourself with is the use of parameters whenever you have a derived class. This section deals with those issues.

The first, and probably most important, problem with parameters in derived class constructors actually applies to all overridden procedures, iterators, and methods: the parameter list of an overridden routine must exactly match the parameter list of the corresponding routine in the base class. In fact, HLA doesn't even give you the chance to violate this rule because **OVERRIDE** routine prototypes don't allow parameter list declarations – they automatically inherit the parameter list of the base routine. Therefore, you cannot use a special parameter list in the constructor prototype for one class and a different parameter list for the constructors appearing in base or derived classes. Sometimes it would be nice if this weren't the case, but there are some sound and logical reasons why HLA does not support this⁷.

Some languages, like C++, support function overloading letting you specify several different constructors whose parameter list specifies which constructor to use. HLA does not directly support procedure overloading in this manner, but you can use macros to simulate this language feature (see “Simulating Function Overloading with Macros” on page 990). To use this trick with constructors you would create a macro with the name *Create*. The actual constructors could have names that describe their differences (e.g., *CreateDefault*, *CreateSetIJ*, etc.). The *Create* macro would parse the actual parameter list to determine which routine to call.

7. Calling virtual methods and iterators would be a real problem since you don't really know which routine a pointer references. Therefore, you couldn't know the proper parameter list. While the problems with procedures aren't quite as drastic, there are some subtle problems that could creep into your code if base or derived classes allowed overridden procedures with different parameter lists.

HLA does not support macro overloading. Therefore, you cannot override a macro in a derived class to call a constructor unique to that derived class. In certain circumstances you can create a small workaround by defining empty procedures in your base class that you intend to override in some derived class (this is similar to an abstract method, see “Abstract Methods” on page 1091). Presumably, you would never call the procedure in the base class (in fact, you would probably want to put an error message in the body of the procedure just in case you accidentally call it). By putting the empty procedure declaration in the base class, the macro that simulates function overloading can refer to that procedure and you can use that in derived classes later on.

10.11 Destructors

A destructor is a class routine that cleans up an object once a program finishes using that object. Like constructors, HLA does not provide a special syntax for creating destructors nor does HLA automatically call a destructor; unlike constructors, a destructor is usually a method rather than a procedure (since virtual destructors make a lot of sense while virtual constructors do not).

A typical destructor will close any files opened by the object, free the memory allocated during the use of the object, and, finally, free the object itself if it was created dynamically. The destructor also handles any other clean-up chores the object may require before it ceases to exist.

By convention, most HLA programmers name their destructors *Destroy*. Destructors generally do not have any parameters, so the issue of overloading the parameter list rarely arises. About the only code that most destructors have in common is the code to free the storage associated with the object. The following destructor demonstrates how to do this:

```
procedure tBase.Destroy; nodisplay;
begin Destroy;

    push( eax ); // isInHeap uses this

    // Place any other clean up code here.
    // The code to free dynamic objects should always appear last
    // in the destructor.

    /*****/

    // The following code assumes that ESI still contains the address
    // of the object.

    if( isInHeap( esi ) ) then

        free( esi );

    endif;
    pop( eax );

end Destroy;
```

The HLA Standard Library routine *isInHeap* returns true if its parameter is an address that *malloc* returned. Therefore, this code automatically frees the storage associated with the object if the program originally allocated storage for the object by calling *malloc*. Obviously, on return from this method call, ESI will no longer point at a legal object in memory if you allocated it dynamically. Note that this code will not affect the value in ESI nor will it modify the object if the object wasn't one you've previously allocated via a call to *malloc*.

10.12 HLA's “_initialize_” and “_finalize_” Strings

Although HLA does not automatically call constructors and destructors associated with your classes, HLA does provide a mechanism whereby you can cause these calls to happen automatically: by using the `_initialize_` and `_finalize_` compile-time string variables (i.e., VAL constants) HLA automatically declares in every procedure.

Whenever you write a procedure, iterator, or method, HLA automatically declares several local symbols in that routine. Two such symbols are `_initialize_` and `_finalize_`. HLA declares these symbols as follows:

```
val
  _initialize_: string := "";
  _finalize_: string := "";
```

HLA emits the `_initialize_` string as text at the very beginning of the routine's body, i.e., immediately after the routine's BEGIN clause⁸. Similarly, HLA emits the `_finalize_` string at the very end of the routine's body, just before the END clause. This is comparable to the following:

```
procedure SomeProc;
  << declarations >>
begin SomeProc;

  @text( _initialize_ );

  << procedure body >>

  @text( _finalize_ );

end SomeProc;
```

Since `_initialize_` and `_finalize_` initially contain the empty string, these expansions have no effect on the code that HLA generates unless you explicitly modify the value of `_initialize_` prior to the BEGIN clause or you modify `_finalize_` prior to the END clause of the procedure. So if you modify either of these string objects to contain a machine instruction, HLA will compile that instruction at the beginning or end of the procedure. The following example demonstrates how to use this technique:

```
procedure SomeProc;
  ?_initialize_ := "mov( 0, eax );";
  ?_finalize_   := "stdout.put( eax );";
begin SomeProc;

  // HLA emits "mov( 0, eax );" here in response to the _initialize_
  // string constant.

  add( 5, eax );

  // HLA emits "stdout.put( eax );" here.

end SomeProc;
```

Of course, these examples don't save you much. It would be easier to type the actual statements at the beginning and end of the procedure than assign a string containing these statements to the `_initialize_` and `_finalize_` compile-time variables. However, if we could automate the assignment of some string to these variables, so that you don't have to explicitly assign them in each procedure, then this feature might be useful. In a moment, you'll see how we can automate the assignment of values to the `_initialize_` and `_finalize_` strings. For the time being, consider the case where we load the name of a constructor into the `_initialize_`

8. If the routine automatically emits code to construct the activation record, HLA emits `_initialize_`'s text after the code that builds the activation record.

string and we load the name of a destructor in to the `_finalize_` string. By doing this, the routine will “automatically” call the constructor and destructor for that particular object.

The example above has a minor problem. If we can automate the assignment of some value to `_initialize_` or `_finalize_`, what happens if these variables already contain some value? For example, suppose we have two objects we use in a routine and the first one loads the name of its constructor into the `_initialize_` string; what happens when the second object attempts to do the same thing? The solution is simple: don’t directly assign any string to the `_initialize_` or `_finalize_` compile-time variables, instead, always concatenate your strings to the end of the existing string in these variables. The following is a modification to the above example that demonstrates how to do this:

```
procedure SomeProc;
  ?_initialize_ := _initialize_ + "mov( 0, eax );";
  ?_finalize_ := _finalize_ + "stdout.put( eax );"
begin SomeProc;

  // HLA emits "mov( 0, eax );" here in response to the _initialize_
  // string constant.

  add( 5, eax );

  // HLA emits "stdout.put( eax );" here.

end SomeProc;
```

When you assign values to the `_initialize_` and `_finalize_` strings, HLA almost guarantees that the `_initialize_` sequence will execute upon entry into the routine. Sadly, the same is not true for the `_finalize_` string upon exit. HLA simply emits the code for the `_finalize_` string at the end of the routine, immediately before the code that cleans up the activation record and returns. Unfortunately, “falling off the end of the routine” is not the only way that one could return from that routine. One could explicitly return from somewhere in the middle of the code by executing a RET instruction. Since HLA only emits the `_finalize_` string at the very end of the routine, returning from that routine in this manner bypassing the `_finalize_` code. Unfortunately, other than manually emitting the `_finalize_` code, there is nothing you can do about this⁹. Fortunately, this mechanism for exiting a routine is completely under your control; if you never exit a routine except by “falling off the end” then you won’t have to worry about this problem (note that you can use the EXIT control structure to transfer control to the end of a routine if you really want to return from that routine from somewhere in the middle of the code).

Another way to prematurely exit a routine which, unfortunately, you have no control over, is by raising an exception. Your routine could call some other routine (e.g., a standard library routine) that raises an exception and then transfers control immediately to whomever called your routine. Fortunately, you can easily trap and handle exceptions by putting a TRY..ENDTRY block in your procedure. Here is an example that demonstrates this:

```
procedure SomeProc;
  << declarations that modify _initialize_ and _finalize_ >>
begin SomeProc;

  << HLA emits the code for the _initialize_ string here. >>

  try // Catch any exceptions that occur:

    << Procedure Body Goes Here >>

  anyexception

    push( eax ); // Save the exception #.
    @text( _finalize_ ); // Execute the _finalize_ code here.
```

9. Note that you can manually emit the `_finalize_` code using the statement “@text(`_finalize_`);”.


```

    pop( eax );          // Restore the exception #.
    raise( eax );       // Reraise the exception.

endtry;

// HLA automatically emits the _finalize_ code here.

end SomeProc;

```

Although the code above handles some problems that exist with `_finalize_`, by no means that this handle every possible case. Always be on the look out for ways your program could inadvertently exit a routine without executing the code found in the `_finalize_` string. You should explicitly expand `_finalize_` if you encounter such a situation.

There is one important place you can get into trouble with respect to exceptions: within the code the routine emits for the `_initialize_` string. If you modify the `_initialize_` string so that it contains a constructor call and the execution of that constructor raises an exception, this will probably force an exit from that routine without executing the corresponding `_finalize_` code. You could bury the TRY.ENDTRY statement directly into the `_initialize_` and `_finalize_` strings but this approach has several problems, not the least of which is the fact that one of the first constructors you call might raise an exception that transfers control to the exception handler that calls the destructors for all objects in that routine (including those objects whose constructors you have yet to call). Although no single solution that handles all problems exists, probably the best approach is to put a TRY.ENDTRY block around each constructor call if it is possible for that constructor to raise some exception that is possible to handle (i.e., doesn't require the immediate termination of the program).

Thus far this discussion of `_initialize_` and `_finalize_` has failed to address one important point: why use this feature to implement the "automatic" calling of constructors and destructors since it apparently involves more work than simply calling the constructors and destructors directly? Clearly there must be a way to automate the assignment of the `_initialize_` and `_finalize_` strings or this section wouldn't exist. The way to accomplish this is by using a macro to define the class type. So now it's time to take a look at another HLA feature that makes it possible to automate this activity: the FORWARD keyword.

You've seen how to use the FORWARD reserved word to create procedure and iterator prototypes (see "Forward Procedures" on page 567), it turns out that you can declare forward CONST, VAL, TYPE, and variable declarations as well. The syntax for such declarations takes the following form:

```
ForwardSymbolName: forward( undefinedID );
```

This declaration is completely equivalent to the following:

```
?undefinedID: text := "ForwardSymbolName";
```

Especially note that this expansion does not actually define the symbol `ForwardSymbolName`. It just converts this symbol to a string and assigns this string to the specified TEXT object (`undefinedID` in this example).

Now you're probably wonder how something like the above is equivalent to a forward declaration. The truth is, it isn't. However, FORWARD declarations let you create macros that simulate type names by allowing you to defer the actual declaration of an object's type until some later point in the code. Consider the following example:

```

type
  myClass: class
    var
      i:int32;

    procedure Create; returns( "esi" );
    procedure Destroy;
  endclass;

#macro _myClass: varID;

```

```

forward( varID );
?_initialize_ := _initialize_ + @string:varID + ".Create(); ";
?_finalize_ := _finalize_ + @string:varID + ".Destroy(); ";
varID: myClass
#endmacro;

```

Note, and this is very important, that a semicolon does not follow the “varID: myClass” declaration at the end of this macro. You’ll find out why this semicolon is missing in a little bit.

If you have the class and macro declarations above in your program, you can now declare variables of type `_myClass` that automatically invoke the constructor and destructor upon entry and exit of the routine containing the variable declarations. To see how, take a look at the following procedure shell:

```

procedure HasmyClassObject;
var
    mco: _myClass;
begin HasmyClassObject;

    << do stuff with mco here >>

end HasmyClassObject;

```

Since `_myClass` is a macro, the procedure above expands to the following text during compilation:

```

procedure HasmyClassObject;
var
    mco:                // Expansion of the _myClass macro:
        forward( _0103_ ); // _0103_ symbol is and HLA supplied text symbol
                            // that expands to "mco".

    ?_initialize_ := _initialize_ + "mco" + ".Create(); ";
    ?_finalize_ := _finalize_ + "mco" + ".Destroy(); ";
    mco: myClass;

begin HasmyClassObject;

    mco.Create(); // Expansion of the _initialize_ string.

    << do stuff with mco here >>

    mco.Destroy(); // Expansion of the _finalize_ string.

end HasmyClassObject;

```

You might notice that a semicolon appears after “mco: myClass” declaration in the example above. This semicolon is not actually a part of the macro, instead it is the semicolon that follows the “mco: _myClass;” declaration in the original code.

If you want to create an array of objects, you could legally declare that array as follows:

```

var
    mcoArray: _myClass[10];

```

Because the last statement in the `_myClass` macro doesn’t end with a semicolon, the declaration above will expand to something like the following (almost correct) code:

```

mcoArray:                // Expansion of the _myClass macro:
    forward( _0103_ ); // _0103_ symbol is and HLA supplied text symbol
                        // that expands to "mcoArray".

?_initialize_ := _initialize_ + "mcoArray" + ".Create(); ";
?_finalize_ := _finalize_ + "mcoArray" + ".Destroy(); ";
mcoArray: myClass[10];

```

The only problem with this expansion is that it only calls the constructor for the first object of the array. There are several ways to solve this problem; one is to append a macro name to the end of `_initialize_` and `_finalize_` rather than the constructor name. That macro would check the object's name (`mcoArray` in this example) to determine if it is an array. If so, that macro could expand to a loop that calls the constructor for each element of the array (the implementation appears as a programming project at the end of this chapter).

Another solution to this problem is to use a macro parameter to specify the dimensions for arrays of `myClass`. This scheme is easier to implement than the one above, but it does have the drawback of requiring a different syntax for declaring object arrays (you have to use parentheses around the array dimension rather than square brackets).

The `FORWARD` directive is quite powerful and lets you achieve all kinds of tricks. However, there are a few problems of which you should be aware. First, since HLA emits the `_initialize_` and `_finalize_` code transparently, you can be easily confused if there are any errors in the code appearing within these strings. If you start getting error messages associated with the `BEGIN` or `END` statements in a routine, you might want to take a look at the `_initialize_` and `_finalize_` strings within that routine. The best defense here is to always append very simple statements to these strings so that you reduce the likelihood of an error.

Fundamentally, HLA doesn't support automatic constructor and destructor calls. This section has presented several tricks to attempt to automate the calls to these routines. However, the automation isn't perfect and, indeed, the aforementioned problems with the `_finalize_` strings limit the applicability of this approach. The mechanism this section presents is probably fine for simple classes and simple programs. However, one piece of advice is probably worth following: if your code is complex or correctness is critical, it's probably a good idea to explicitly call the constructors and destructors manually.

10.13 Abstract Methods

An *abstract base class* is one that exists solely to supply a set of common fields to its derived classes. You never declare variables whose type is an abstract base class, you always use one of the derived classes. The purpose of an abstract base class is to provide a template for creating other classes, nothing more. As it turns out, the only difference in syntax between a standard base class and an abstract base class is the presence of at least one *abstract method* declaration. An abstract method is a special method that does not have an actual implementation in the abstract base class. Any attempt to call that method will raise an exception. If you're wondering what possible good an abstract method could be, well, keep on reading...

Suppose you want to create a set of classes to hold numeric values. One class could represent unsigned integers, another class could represent signed integers, a third could implement BCD values, and a fourth could support *real64* values. While you could create four separate classes that function independently of one another, doing so passes up an opportunity to make this set of classes more convenient to use. To understand why, consider the following possible class declarations:

```
type
  uint: class
    var
      TheValue: dword;

    method put;
    << other methods for this class >>
  endclass;

  sint: class
    var
      TheValue: dword;

    method put;
    << other methods for this class >>
  endclass;
```

```

r64: class
  var
    TheValue: real64;

  method put;
  << other methods for this class >>
endclass;

```

The implementation of these classes is not unreasonable. They have fields for the data, they have a *put* method (which, presumably, writes the data to the standard output device), Presumably they have other methods and procedures in implement various operations on the data. There is, however, two problems with these classes, one minor and one major, both occurring because these classes do not inherit any fields from a common base class.

The first problem, which is relatively minor, is that you have to repeat the declaration of several common fields in these classes. For example, the *put* method declaration appears in each of these classes¹⁰. This duplication of effort involves results in a harder to maintain program because it doesn't encourage you to use a common name for a common function since it's easy to use a different name in each of the classes.

A bigger problem with this approach is that it is not generic. That is, you can't create a generic pointer to a "numeric" object and perform operations like addition, subtraction, and output on that value (regardless of the underlying numeric representation).

We can easily solve these two problems by turning the previous class declarations into a set of derived classes. The following code demonstrates an easy way to do this:

```

type
numeric: class
  procedure put;
  << Other common methods shared by all the classes >>
endclass;

uint: class inherits( numeric )
  var
    TheValue: dword;

  override method put;
  << other methods for this class >>
endclass;

sint: class inherits( numeric )
  var
    TheValue: dword;

  override method put;
  << other methods for this class >>
endclass;

r64: class inherits( numeric )
  var
    TheValue: real64;

  override method put;
  << other methods for this class >>
endclass;

```

This scheme solves both the problems. First, by inheriting the *put* method from *numeric*, this code encourages the derived classes to always use the name *put* thereby making the program easier to maintain. Second, because this example uses derived classes, it's possible to create a pointer to the *numeric* type and

10. Note, by the way, that *TheValue* is not a common class because this field has a different type in the *r64* class.

load this pointer with the address of a *uint*, *sint*, or *r64* object. That pointer can invoke the methods found in the *numeric* class to do functions like addition, subtraction, or numeric output. Therefore, the application that uses this pointer doesn't need to know the exact data type, it only deals with numeric values in a generic fashion.

One problem with this scheme is that it's possible to declare and use variables of type *numeric*. Unfortunately, such numeric variables don't have the ability to represent any type of number (notice that the data storage for the numeric fields actually appears in the derived classes). Worse, because you've declared the *put* method in the *numeric* class, you've actually got to write some code to implement that method even though one should never really call it; the actual implementation should only occur in the derived classes. While you could write a dummy method that prints an error message (or, better yet, raises an exception), there shouldn't be any need to write "dummy" procedures like this. Fortunately, there *is* no reason to do so – if you use *abstract* methods.

The **ABSTRACT** keyword, when it follows a method declaration, tells HLA that you are not going to provide an implementation of the method for this class. Instead, it is the responsibility of all derived class to provide a concrete implementation for the abstract method. HLA will raise an exception if you attempt to call an abstract method directly. The following is the modification to the *numeric* class to convert *put* to an abstract method:

```
type
  numeric: class
    method put; abstract;
    << Other common methods shared by all the classes >>
  endclass;
```

An abstract base class is a class that has at least one abstract method. Note that you don't have to make all methods abstract in an abstract base class; it is perfectly legal to declare some standard methods (and, of course, provide their implementation) within the abstract base class.

Abstract method declarations provide a mechanism by which a base class enforces the methods that the derived classes must implement. In theory, all derived classes must provide concrete implementations of all abstract methods or those derived classes are themselves abstract base classes. In practice, it's possible to bend the rules a little and use abstract methods for a slightly different purpose.

A little earlier, you read that one should never create variables whose type is an abstract base class. For if you attempt to execute an abstract method the program would immediately raise an exception to complain about this illegal method call. In practice, you actually can declare variables of an abstract base type and get away with this as long as you don't call any abstract methods. We can use this fact to provide a better form of method overloading (that is, providing several different routines with the same name but different parameter lists). Remember, the standard trick in HLA to overload a routine is to write several different routines and then use a macro to parse the parameter list and determine which actual routine to call (see "Simulating Function Overloading with Macros" on page 990). The problem with this technique is that you cannot override a macro definition in a class, so if you want to use a macro to override a routine's syntax, then that macro must appear in the base class. Unfortunately, you may not need a routine with a specific parameter list in the base class (for that matter, you may only need that particular version of the routine in a single derived class), so implementing that routine in the base class and in all the other derived classes is a waste of effort. This isn't a big problem. Just go ahead and define the abstract method in the base class and only implement it in the derived class that needs that particular method. As long as you don't call that method in the base class or in the other derived classes that don't override the method, everything will work fine.

One problem with using abstract methods to support overloading is that this trick does not apply to procedures - only methods and iterators. However, you can achieve the same effect with procedures by declaring a (non-abstract) procedure in the base class and overriding that procedure only in the class that actually uses it. You will have to provide an implementation of the procedure in the base class, but that is a minor issue (the procedure's body, by the way, should simply raise an exception to indicate that you should have never called it).

An example of routine overloading in a class appears in this chapter's sample program.

10.14 Run-time Type Information (RTTI)

When working with an object variable (as opposed to a pointer to an object), the type of that object is obvious: it's the variable's declared type. Therefore, at both compile-time and run-time the program trivially knows the type of the object. When working with pointers to objects you cannot, in the general case, determine the type of an object a pointer references. However, at run-time it is possible to determine the object's actual type. This section discusses how to detect the underlying object's type and how to use this information.

If you have a pointer to an object and that pointer's type is some base class, at run-time the pointer could point at an object of the base class or any derived type. At compile-time it is not possible to determine the exact type of an object at any instant. To see why, consider the following short example:

```
ReturnSomeObject();           // Returns a pointer to some class in ESI.
mov( esi, ptrToObject );
```

The routine *ReturnSomeObject* returns a pointer to an object in ESI. This could be the address of some base class object or a derived class object. At compile-time there is no way for the program to know what type of object this function returns. For example, *ReturnSomeObject* could ask the user what value to return so the exact type could not be determined until the program actually runs and the user makes a selection.

In a perfectly designed program, there probably is no need to know a generic object's actual type. After all, the whole purpose of object-oriented programming and inheritance is to produce general programs that work with lots of different objects without having to make substantial changes to the program. In the real world, however, programs may not have a perfect design and sometimes it's nice to know the exact object type a pointer references. Run-time type information, or RTTI, gives you the capability of determining an object's type at run-time, even if you are referencing that object using a pointer to some base class of that object.

Perhaps the most fundamental RTTI operation you need is the ability to ask if a pointer contains the address of some specific object type. Many object-oriented languages (e.g., Delphi) provide an *IS* operator that provides this functionality. *IS* is a boolean operator that returns true if its left operand (a pointer) points at an object whose type matches the left operand (which must be a type identifier). The typical syntax is generally the following:

```
ObjectPointerOrVar is ClassType
```

This operator would return true if the variable is of the specified class, it returns false otherwise. Here is a typical use of this operator (in the Delphi language)

```
if( ptrToNumeric is uint ) then begin
.
.
.
end;
```

It's actually quite simple to implement this functionality in HLA. As you may recall, each class is given its own virtual method table. Whenever you create an object, you must initialize the pointer to the VMT with the address of that class' VMT. Therefore, the VMT pointer field of all objects of a given class type contain the same pointer value and this pointer value is different from the VMT pointer field of all other classes. We can use this fact to see if an object is some specific type. The following code demonstrates how to implement the Delphi statement above in HLA:

```
mov( ptrToNumeric, esi );
if( (type uint [esi])._pVMT_ = &uint._VMT_ ) then
.
.
```

```

    .
endif;

```

This IF statement simply compares the object's `_pVMT_` field (the pointer to the VMT) against the address of the desired class' VMT. If they are equal, then the `ptrToNumeric` variable points at an object of type `uint`.

Within the body of a class method or iterator, there is a slightly easier way to see if the object is a certain class. Remember, upon entry into a method or an iterator, the EDI register contains the address of the virtual method table. Therefore, assuming you haven't modified EDI's value, you can easily test to see if THIS (ESI) is a specific class type using an IF statement like the following:

```

    if( EDI = &uint._VMT_ ) then
    .
    .
    .
endif;

```

10.15 Calling Base Class Methods

In the section on constructors you saw that it is possible to call an ancestor class' procedure within the derived class' overridden procedure. To do this, all you needed to do was to invoke the procedure using the call "classname.procedureName(parameters);". On occasion you may want to do this same operation with a class' methods as well as its procedures (that is, have an overridden method call the corresponding base class method in order to do some computation you'd rather not repeat in the derived class' method). Unfortunately, HLA does not let you directly call methods as it does procedures. You will need to use an indirect mechanism to achieve this; specifically, you will have to call the function using the address in the base class' virtual method table. This section describes how to do this.

Whenever your program calls a method it does so indirectly, using the address found in the virtual method table for the method's class. The virtual method table is nothing more than an array of 32-bit pointers with each entry containing the address of one of that class' methods. So to call a method, all you need is the index into this array (or, more properly, the offset into the array) of the address of the method you wish to call. The HLA compile-time function `@offset` comes to the rescue- it will return the offset into the virtual method table of the method whose name you supply as a parameter. Combined with the CALL instruction, you can easily call any method associated with a class. Here's an example of how you would do this:

```

type
  myCls: class
    .
    .
    .
    method m;
    .
    .
    .
  endclass;
.
.
.
call( myCls._VMT_[ @offset( myCls.m ) ] );

```

The CALL instruction above calls the method whose address appears at the specified entry in the virtual method table for `myCls`. The `@offset` function call returns the offset (i.e., index times four) of the address of `myCls.m` within the virtual method table. Hence, this code indirectly calls the `m` method by using the virtual method table entry for `m`.

There is one major drawback to calling methods using this scheme: you don't get to use the high level syntax for procedure/method calls. Instead, you must use the low-level CALL instruction. In the example

above, this isn't much of an issue because the *m* procedure doesn't have any parameters. If it did have parameters, you would have to manually push those parameters onto the stack yourself (see "Passing Parameters on the Stack" on page 822). Fortunately, you'll rarely need to call ancestor class methods from a derived class, so this won't be much of an issue in real-world programs.

10.16 Putting It All Together

HLA's class declarations provide a powerful tool for creating object-oriented assembly language programs. Although object-oriented programming is not as popular in assembly as in high level languages, part of the reason has been the lack of assemblers that support object-oriented programming in a reasonable fashion and an even greater lack of tutorial information on object-oriented programming in assembly language.

While this chapter cannot go into great detail about the object-oriented programming paradigm (space limitations prevent this), this chapter does explain the object-oriented facilities that HLA provides and supplies several example programs that use those facilities. From here on, it's up to you to utilize these facilities in your programs and gain experience writing object oriented assembly code.

The MMX Instruction Set

Chapter Eleven

11.1 Chapter Overview

While working on the Pentium and Pentium Pro processors, Intel was also developing an instruction set architecture extension for multimedia applications. By studying several existing multimedia applications, developing lots of multimedia related algorithms, and through simulation, Intel developed 57 instructions that would greatly accelerate the execution of multimedia applications. The end result was their multimedia extensions to the Pentium processor that Intel calls the MMX Technology Instructions.

Prior to the invention of the MMX enhancements, good quality multimedia systems required separate digital signal processors and special electronics to handle much of the multimedia workload¹. The introduction of the MMX instruction set allowed later Pentium processors to handle these multimedia tasks without these expensive digital signal processors (DSPs), thus lowering the cost of multimedia systems. So later Pentiums, Pentium II, Pentium III, and Pentium IV processors all have the MMX instruction set. Earlier Pentiums (and CPUs prior to the Pentium) and the Pentium Pro do not have these instructions available. Since the instruction set has been available for quite some time, you can probably use the MMX instructions without worrying about your software failing on many machines.

In this chapter we will discuss the MMX Technology instructions and how to use them in your assembly language programs. The use of MMX instructions, while not completely limited to assembly language, is one area where assembly language truly shines since most high level languages do not make good use of MMX instructions except in library routines. Therefore, writing fast code that uses MMX instructions is mainly the domain of the assembly language programmer. Hence, it's a good idea to learn these instructions if you're going to write much assembly code.

11.2 Determining if a CPU Supports the MMX Instruction Set

While it's almost a given that any modern CPU your software will run on will support the MMX extended instruction set, there may be times when you want to write software that will run on a machine even in the absence of MMX instructions. There are two ways to handle this problem – either provide two versions of the program, one with MMX support and one without (and let the user choose which program they wish to run), or the program can dynamically determine whether a processor supports the MMX instruction set and skip the MMX instructions if they are not available.

The first situation, providing two different programs, is the easiest solution from a software development point of view. You don't actually create two source files, of course; what you do is use conditional compilation statements (i.e., #IF..#ELSE..#ENDIF) to selectively compile MMX or standard instructions depending on the presence of an identifier or value of a boolean constant in your program. See “Conditional Compilation (Compile-Time Decisions)” on page 962 for more details.

Another solution is to dynamically determine the CPU type at run-time and use program logic to skip over the MMX instructions and execute equivalent standard code if the CPU doesn't support the MMX instruction set. If you're expecting the software to run on an Intel Pentium or later CPU, you can use the CPUID instruction to determine whether the processor supports the MMX instruction set. If MMX instructions are available, the CPUID instruction will return bit 23 as a one in the feature flags return result.

The following code illustrates how to use the CPUID instruction. This example does not demonstrate the entire CPUID sequence, but shows the portion used for detection of MMX technology.

1. A good example was the Apple Quadra 660AV and 840AV computer systems; they were built around the Motorola 68040 processor rather than a Pentium, but the 68040 was no more capable of handling multimedia applications than the Pentium. However, an on-board DSP (digital signal processor) CPU allowed the Quodras to easily handle audio applications that the 68040 could not.

```
// For a perfectly general routine, you should determine if this
// is a Pentium or later processor. We'll assume at least a Pentium
// for now, since most OSes expect a Pentium or better processor.

    mov( 1, eax );           // Request for CPUID feature flags.
    CPUID();                // Get the feature flags into EDX.
    test( $80_0000, edx );  // Is bit 23 set?
    jnz HasMMX;
```

This code assumes at least the presence of a Pentium Processor. If your code needs to run on a 486 or 386 processor, you will have to detect that the system is using one of these processors. There is tons of code on the net that detects different processors, but most of it will not run under 32-bit OSes since the code typically uses protected (non-user-mode) instructions. We'll not go into the details here because 99% of the users out there that are running modern operating systems have a CPU that supports the MMX instruction set or, at least, the CPUID instruction.

11.3 The MMX Programming Environment

The MMX architecture extends the Pentium architecture by adding the following:

- Eight MMX registers (MM0..MM7).
- Four MMX data types (packed bytes, packed words, packed double words, and quad word).
- 57 MMX Instructions.

11.3.1 The MMX Registers

The MMX architecture adds eight 64-bit registers to the Pentium. The MMX instructions refer to these registers as MM0, MM1, MM2, MM3, MM4, MM5, MM6, and MM7. These are strictly data registers, you cannot use them to hold addresses nor are they suitable for calculations involving addresses.

Although MM0..MM7 appear as separate registers in the Intel Architecture, the Pentium processors alias these registers with the FPU's registers (ST0..ST7). Each of the eight MMX 64-bit registers is physically equivalent to the L.O. 64-bits of each of the FPU's registers (see Figure 11.1). The MMX registers overlay the FPU registers in much the same way that the 16-bit general purpose registers overlay the 32-bit general purpose registers.

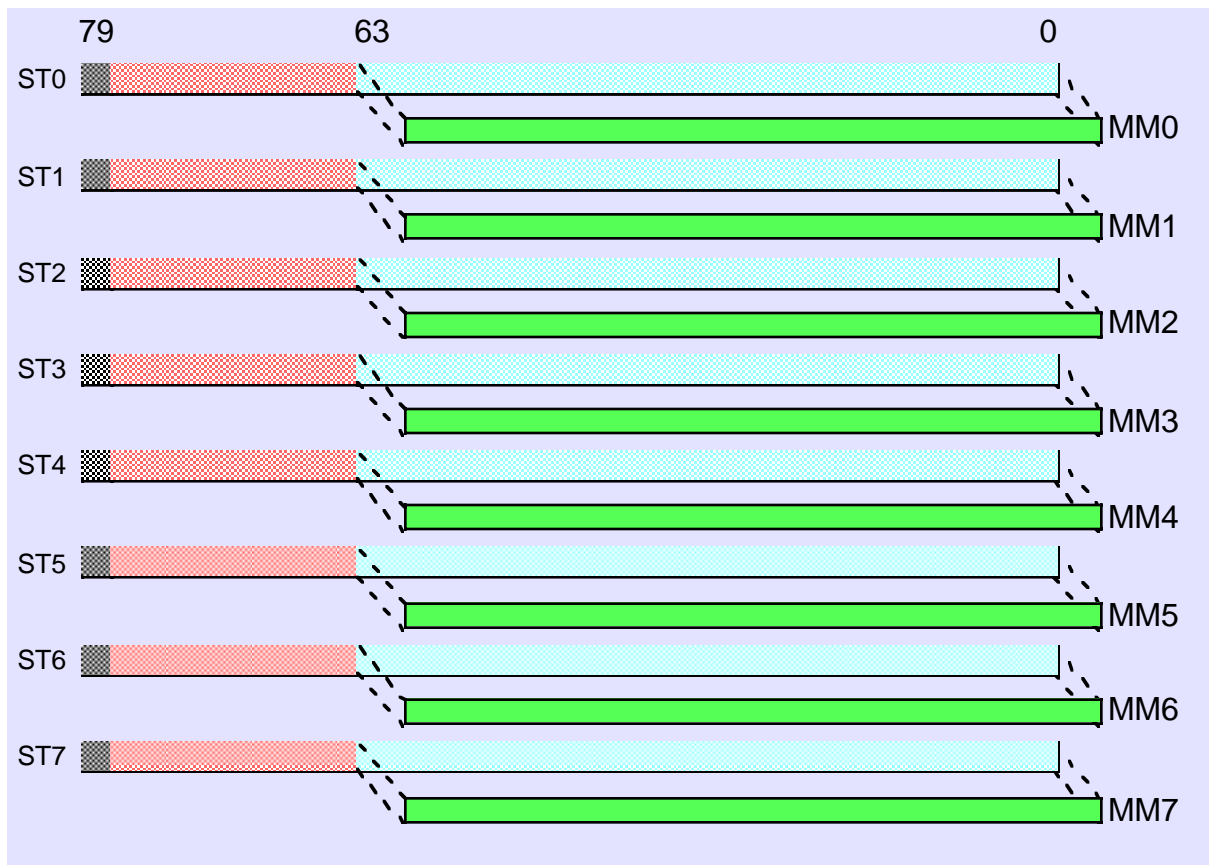


Figure 11.1 MMX and FPU Register Aliasing

Because the MMX registers overlay the FPU registers, you cannot mix FPU and MMX instructions in the same computation sequence. You can begin executing an MMX instruction sequence at any time; however, once you execute an MMX instruction you cannot execute another FPU instruction until you execute a special MMX instruction, EMMS (Exit MMX Machine State). This instruction resets the FPU so you may begin a new sequence of FPU calculations. The CPU does not save the FPU state across the execution of the MMX instructions; executing EMMS clears all the FPU registers. Because saving FPU state is very expensive, and the EMMS instruction is quite slow, it's not a good idea to frequently switch between MMX and FPU calculations. Instead, you should attempt to execute the MMX and FPU instructions at different times during your program's execution.

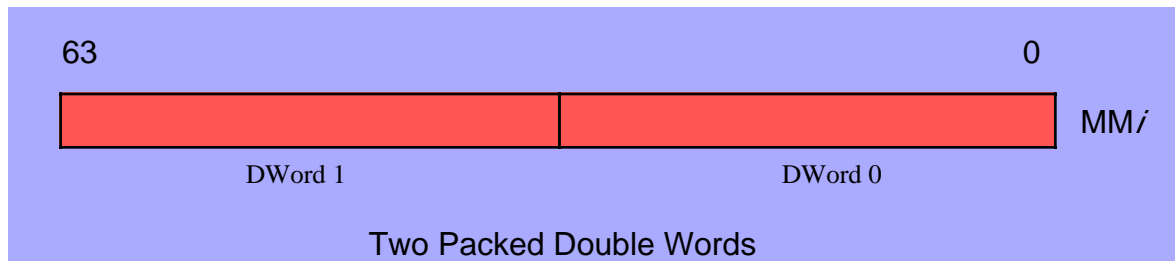
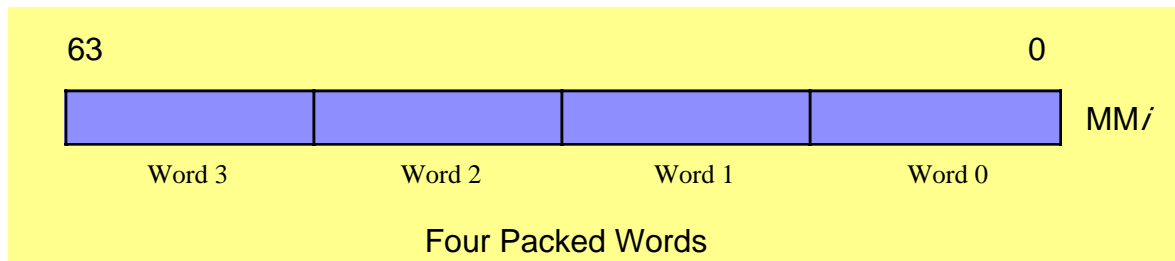
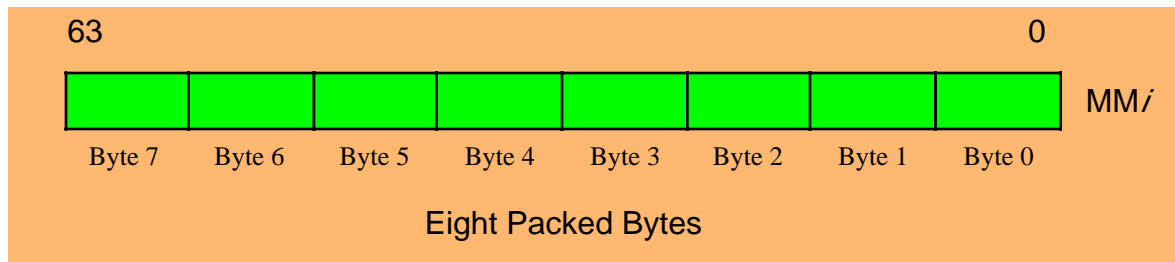
You're probably wondering why Intel chose to alias the MMX registers with the FPU registers. Intel, in their literature, brags constantly about what a great idea this was. You see, by aliasing the MMX registers with the FPU registers, Microsoft and other multitasking OS vendors did not have to write special code to save the MMX state when the CPU switched from one process to another. The fact that the OS automatically saved the FPU state means that the CPU would automatically save the MMX state as well. This meant that the new Pentium chips with MMX technology that Intel created were automatically compatible with Windows 95, Windows NT, and Linux without any changes to the operating system code.

Of course, those operating systems have long since been upgraded and Microsoft (and Linux developers) could have easily provided a "service pack" to handle the new registers (had Intel chosen not to alias the FPU and MMX registers). So while aliasing MMX with the FPU provided a very short-lived and temporary benefit, in retrospect Intel made a big mistake with this decision. They've obviously realized their mistake, because as they've introduced new "streaming" instructions (the floating point equivalent of the MMX instruction set) they've added new registers (XMM0..XMM7) without using this trick. It's too bad they

don't fix the problem in their current CPUs (there is no technical reason why they can't create separate MMX and FPU registers at this point). Oh well, you'll just have to live with the fact that you can't execute interleaved FPU and MMX instructions.

11.3.2 The MMX Data Types

The MMX instruction set supports four different data types: an eight-byte array, a four-word array, a two element double word array, and a quadword object. Each MMX register processes one of these four data types (see Figure 11.2).



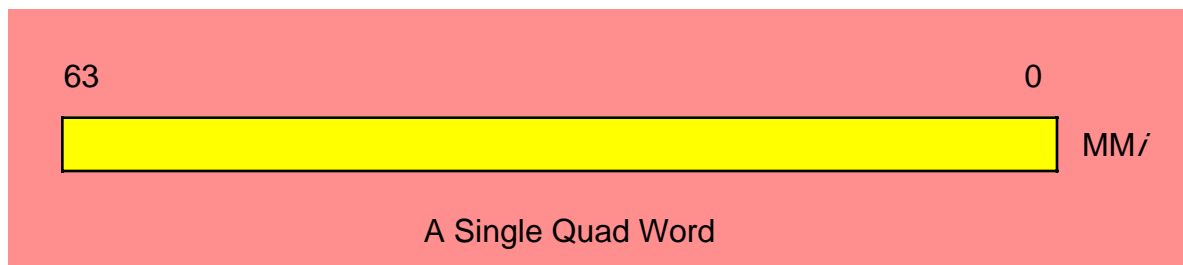


Figure 11.2 The MMX Data Types

Despite the presence of 64-bit registers, the MMX instruction set does not extend the 32-bit Pentium processor to 64-bits. Instead, after careful study Intel added only those 64-bit instructions that were useful for multimedia operations. For example, you cannot add or subtract two 64-bit integers with the MMX instruction set. In fact, only the logical and shift operations directly manipulate 64 bits.

The MMX instruction set was not designed to provide general 64-bit capabilities to the Pentium. Instead, the MMX instruction set provides the Pentium with the capability of performing multiple eight-, sixteen-, or thirty-two bit operations simultaneously. In other words, the MMX instructions are generally SIMD (Single Instruction Multiple Data) instructions (see “Parallel Processing” on page 268 for an explanation of SIMD). For example, a single MMX instruction can add eight separate pairs of byte values together. This is not the same as adding two 64-bit values since the overflow from the individual bytes does not carry over into the higher order bytes. This can accelerate a program that needs to add a long string of bytes together since a single MMX instruction can do the work of eight regular Pentium instructions. This is how the MMX instruction set speeds up multimedia applications – by processing multiple data objects in parallel with a single instruction. Given the data types the MMX instruction set supports, you can process up to eight byte objects in parallel, four word objects in parallel, or two double words in parallel.

11.4 The Purpose of the MMX Instruction Set

The Single Instruction Multiple Data model the MMX architecture supports may not look all that impressive when viewed with a SISD (Single Instruction, Single Data) bias. Once you’ve mastered the basic integer instructions on the 80x86, it’s difficult to see the application of the MMX’s SIMD instruction set. However, the MMX instructions directly address the needs of modern media, communications, and graphics applications, which often use sophisticated algorithms that perform the same operations on a large number of small data types (bytes, words, and double words).

For example, most programs use a stream of bytes or words to represent audio and video data. The MMX instructions can operate on eight bytes or four words with a single instruction, thus accelerating the program by almost a factor of four or eight.

One drawback to the MMX instruction set is that it is not general purpose. Intel’s research that led to the development of these new instructions specifically targeted audio, video, graphics, and another multimedia applications. Although some of the instructions are applicable in many general programs, you’ll find that many of the instructions have very little application outside their limited domain. Although, with a lot of deep thought, you can probably dream up some novel uses of many of these instructions that have nothing whatsoever at all to do with multimedia, you shouldn’t get too frustrated if you cannot figure out why you would want to use a particular instruction; that instruction probably has a specific purpose and if you’re not trying to code a solution for that problem, you may not be able to use the instruction. If you’re questioning why Intel would put such limited instructions in their instruction set, just keep in mind that although you can use the instruction(s) for lots of different purposes, they are invaluable for the few purposes they are uniquely suited.

11.5 Saturation Arithmetic and Wraparound Mode

The MMX instruction set supports saturating arithmetic (see “Sign Extension, Zero Extension, Contraction, and Saturation” on page 73). When manipulating standard integer values and an overflow occurs, the standard integer instructions maintain the correct L.O. bits of the value in the integer while truncating any overflow². This form of arithmetic is known as *wraparound* mode since the L.O. bits wrap back around to zero. For example, if you add the two eight-bit values \$02 and \$FF you wind up with a carry and the result \$01. The actual sum is \$101, but the operation truncates the ninth bit and the L.O. byte wraps around to \$01.

In saturation mode, results of an operation that overflow or underflow are clipped (saturated) to some maximum or minimum value depending on the size of the object and whether it is signed or unsigned. The result of an operation that exceeds the range of a data-type saturates to the maximum value of the range. A result that is less than the range of a data type saturates to the minimum value of the range.

Table 1:

Data Type	Decimal		Hexadecimal	
	Lower Limit	Upper Limit	Lower Limit	Upper Limit
Signed Byte	-128	+127	\$80	\$7f
Unsigned Byte	0	255	0	\$ff
Signed Word	-32768	+32767	\$8000	\$7fff
Unsigned Word	0	65535	0	\$ffff

For example, when the result exceeds the data range limit for signed bytes, it is saturated to \$7f; if a value is less than the data range limit, it is saturated to \$80 for signed bytes. If a value exceeds the range for unsigned bytes, it is saturated to \$ff or \$00.

This saturation effect is very useful for audio and video data. For example, if you are amplifying an audio signal by multiplying the words in the CD-quality 44.1 kHz audio stream by 1.5, clipping the value at +32767, while introducing distortion, sounds far better than allowing the waveform to wrap around to -32768. Similarly, if you are mixing colors in a 24-bit graphic or video image, saturating to white produces much more meaningful results than wrap-around.

Since Intel created the MMX architecture to support audio, graphics, and video, it should come as no surprise that the MMX instruction set supports saturating arithmetic. For those applications that require saturating arithmetic, having the CPU automatically handle this process (rather than having to explicitly check after each calculation) is another way the MMX architecture speeds up multimedia applications.

11.6 MMX Instruction Operands

Most MMX instructions operate on two operands, a source and a destination operand. A few instructions have three operands with the third operand being a small immediate (constant) value. In this section we'll take a look at the common MMX instruction operands.

2. For some instructions the overflow may appear in another register or the carry flag, but in the destination register the high order bits are lost.

The destination operand is almost always an MMX register. In fact, the only exceptions are those instructions that store an MMX register into memory. The MMX instructions always leave the result of MMX calculations in an MMX register.

The source operand can be an MMX register or a memory location. The memory location is usually a quad word entity, but certain instructions operate on double word objects. Note that, in this context, “quad word” and “double word” mean eight or four consecutive bytes in memory; they do not necessarily imply that the MMX instruction is operating on a qword or dword object. For example, if you add eight bytes together using the PADDDB (packed add bytes) instruction, PADDDB references a qword object in memory, but actually adds together eight separate bytes.

For most MMX instructions, the generic HLA syntax is one of the following:

```
mmxInstr( source, dest );
```

The specific forms are

```
mmxInstr( mmi, mmi ); // i=0..7
mmxInstr( mem, mmi ); // i=0..7
```

MMX instructions access memory using the same addressing modes as the standard integer instructions. Therefore, any legal 80x86 addressing mode is usable in an MMX instruction. For those instructions that reference a 64-bit memory location, HLA requires that you specify an anonymous memory object (e.g., “[ebx]” or “[ebp+esi*8+6]”) or a qword variable.

A few instructions require a small immediate value (or constant). For example, the shift instructions let you specify a shift count as an immediate value in the range 0..63. Another instruction uses the immediate value to specify a set of four different count values in the range 0..3 (i.e., four two-bit count values). These instructions generally take the following form:

```
mmxInstr( imm8, source, dest );
```

Note that, in general, MMX instructions do not allow you to specify immediate constants as operands except for a few special cases (such as shift counts). In particular, the source operand to an MMX instruction has to be a register or a quad word variable, it cannot be a 64-bit constant. To achieve the same effect as specifying a constant as the source operand, you must initialize a quad word variable in the READONLY (or STATIC) section of your program and specify this variable as the source operand. Unfortunately, HLA does not support 64-bit constants, so initializing the value is going to be a bit of a problem. There are two solutions to this problem: break the constant into smaller pieces (bytes, words, or double words) and emit the constant in pieces that HLA can process; or you can write your own numeric conversion routine(s) using the HLA compile-time language to allow the emission of a 64-bit constant. We’ll explore both of those approaches here.

The first approach is the one you will most commonly use. Very few MMX instructions actually operate on 64-bit data operands; instead, they typically operate on a (small) array of bytes, words, or double words. Since HLA provides good support for byte, word, and double word constant expressions, specifying a 64-bit MMX memory operand as a short array of objects is probably the best way to create this data. Since the MMX instructions that fetch a source value from memory expect a 64-bit operand, you must declare such objects as qword variables, e.g.,

```
static
    mmxVar:qword;
```

The big problem with this declaration is that the qword type does not allow an initializer (since HLA cannot handle 64-bit constant expressions). Since this declaration occurs in the STATIC segment, HLA will initialize *mmxVar* with zero; probably not the value you’re interested in supplying here.

There are two ways to solve this problem. The first way is to attach the @NOSTORAGE option to the MMX variable declarations in the STATIC segment. The data declarations that immediately follow the variable definition provide the initial data for that variable. Here’s an example of such a declaration:

```
static
    mmxDVar: qword; @nostorage;
```

```
dword $1234_5678, $90ab_cdef;
```

Note that the `DWORD` directive above stores the double word constants in successive memory locations. Therefore, `$1234_5678` will appear in the L.O. double word of the 64-bit value and `$90ab_cdef` will appear in the H.O. double word of the 64-bit value. Always keep in mind that the L.O. objects come first in the list following the `DWORD` (or `BYTE`, or `WORD`, or ???) directive; this is opposite of the way you're used to reading 64-bit values.

The example above used a `DWORD` directive to provide the initialization constant. However, you can use any data declaration directive, or even a combination of directives, as long as you allocate at least eight bytes (64-bits) for each qword constant. The following data declaration, for example, initializes eight eight-bit constants for an MMX operand; this would be perfect for a `PADDB` instruction or some other instruction that operates on eight bytes in parallel:

```
static
  eightBytes: qword; @nostorage;
  byte 0, 1, 2, 3, 4, 5, 6, 7;
```

Although most MMX instructions operate on small arrays of bytes, words, or double words, a few actually do operate on 64-bit quantities. For such memory operands you would probably prefer to specify a 64-bit constant rather than break it up into its constituent double word values. This way, you don't have to remember to put the L.O. double word first and perform other mental adjustments.

Although HLA does not support 64-bit constants in the compile time language, HLA is flexible enough to allow you to extend the language to handle such declarations. Program 11.1 demonstrates how to write a macro to accept a 64-bit hexadecimal constant. This macro will automatically emit two `DWORD` declarations containing the L.O. and H.O. components of the 64-bit value you specify as the *qword16* (quadword constant, base 16) macro parameter. You would typically use the *qword16* macro as follows:

```
static
  H0Ones: qword; @nostorage;
  qword16( $FFFF_FFFF_0000_0000 );
```

The `qword16` macro would emit the following:

```
dword 0;
dword $FFFF_FFFF;
```

Without further ado, here's the macro (and a sample test program):

```
program qwordConstType;
#include( "stdlib.hhf" )

// The following macro accepts a 64-bit hexadecimal constant
// and emits two dword objects in place of the constant.

macro qword16( theHexVal ):hs, len, dwval, mplier, curch, didLO;

  // Remove whitespace around the macro parameter (shouldn't
  // be any, but just in case something weird is going on) and
  // convert all lower case characters to upper case.

  ?hs := @uppercase( @trim( @string:theHexVal, 0 ), 0 );

  // If there is a leading "$" symbol, strip it from the string.

  #if( @substr( hs, 0, 1 ) = "$" )

    ?hs := @substr( hs, 1, 256 );
```



```

#endif

// Process each character in the string from the L.O. digit
// through to the H.O. digit.  Add the digit, multiplied by
// some successive power of 16, to the current sum we're
// accumulating in dwval.  When we cross a dword boundary,
// emit the L.O. dword and start over.

?len := @length( hs );      // Number of characters to process.
?dwval:dword := 0;         // Accumulate value here.
?mplier:dword := 1;       // Power of 16 to multiply by.
?didLO:boolean := false;  // Checks for overflow.
#while( len > 0 )         // Repeat for each char in string.

    // For each character in the string, verify that it is
    // a legal hexadecimal character and merge it in with the
    // current accumulated value if it is.  Print an error message
    // if we come across an illegal character.

    ?len := len - 1;       // Next available char.
    ?curch := char( @substr( hs, len, 1 ) ); // Get the character.
    #if( curch in { '0'..'9' } ) // See if valid decimal digit.

        // Accumulate result if decimal digit.

        ?dwval := dwval +
            (uns8( curch ) - uns8( '0' )) * mplier;

    #elseif( curch in { 'A'..'F' } ) // See if valid hex digit.

        // Accumulate result if a hexadecimal digit.

        ?dwval := dwval +
            (uns8( curch ) - uns8( 'A' ) + 10) * mplier;

    // Ignore underscore characters and report an error for anything
    // else we find in the string.

    #elseif( curch <> '_' )

        #error( "Illegal character in 64-bit hexadecimal constant" )
        #print( "Character = '", curch, "' Rest of string: '", hs, "' )

#endif

// If it's not an underscore character, adjust the multiplier value.
// If we cross a dword boundary, emit the L.O. value as a dword
// and reset everything for the H.O. dword.

#if( curch <> '_' )

    // If the current value fits in 32 bits, process this
    // as though it were a dword object.

    #if( mplier < $1000_0000 )

        ?mplier := mplier * 16;

    #elseif( len > 0 )

```

```

// Down here we've just processed the last hex
// digit that will fit into 32 bits. So emit the
// L.O. dword and reset the multiplier and dwval constants.

?multiplier := 1;
dword dwval;
?dwval := 0;

// If we've been this way before, we've got an
// overflow.

#if( didLO )

    #error( "64-bit overflow in constant" );

#endif
?didLO := true;

#endif

#endif

#endif

// Emit the H.O. dword here.

dword dwval;

// If the constant only consumed 32 bits, we've got to emit a zero
// for the H.O. dword at this point.

#if( !didLO )

    dword 0;

#endif

endmacro;

static
    x:qword; @nostorage;
    qword16( $1234_5678_90ab_cdef );
    qword16( 100 );

begin qwordConstType;

    stdout.put( "64-bit value of x = $" );
    stdout.putq( x );
    stdout.newln();

end qwordConstType;

```

Program 11.1 qword16 Macro to Process 64-bit Hexadecimal Constants

Although it's a little bit more difficult, you could also write a *qword10* macro that lets you specify decimal constants as the macro operand rather than hexadecimal constants. The implementation of *qword10* is left as a programming exercise at the end of this volume.

11.7 MMX Technology Instructions

The following subsections describe each of the MMX instructions in detail. The organization is as follows:

- Data Transfer Instructions,
- Conversion Instructions,
- Packed Arithmetic Instructions,
- Comparisons,
- Logical Instructions,
- Shift and Rotate Instructions,
- the EMMS Instruction.

These sections describe *what* these instructions do, not *how* you would use them. Later sections will provide examples of how you can use several of these instructions.

11.7.1 MMX Data Transfer Instructions

```
movd( reg32, mmi );
movd( mem32, mmi );
movd( mmi, reg32 );
movd( mmi, mem32 );

movq( mem64, mmi );
movq( mmi, mem64 );
movq( mmi, mmi );
```

The MOVD (move double word) instruction copies data between a 32-bit integer register or double word memory location and an MMX register. If the destination is an MMX register, this instruction zero-extends the value while moving it. If the destination is a 32-bit register or memory location, this instruction copies the L.O. 32-bits of the MMX register to the destination.

The MOVQ (move quadword) instruction copies data between two MMX registers or between an MMX register and memory. If either the source or destination operand is a memory object, it must be a qword variable or HLA will complain.

11.7.2 MMX Conversion Instructions

```
packssdw( mem64, mmi );
packssdw( mmi, mmi );

packsswb( mem64, mmi );
packsswb( mmi, mmi );

packusdw( mem64, mmi );
packusdw( mmi, mmi );
```

```

packuswb( mem64, mmi );
packuswb( mmi, mmi );

punpckhbw( mem64, mmi );
punpckhbw( mmi, mmi );

punpckhdq( mem64, mmi );
punpckhdq( mmi, mmi );

punpckhwd( mem64, mmi );
punpckhwd( mmi, mmi );

punpcklbw( mem64, mmi );
punpcklbw( mmi, mmi );

punpckldq( mem64, mmi );
punpckldq( mmi, mmi );

punpcklwd( mem64, mmi );
punpcklwd( mmi, mmi );

```

The `PACKSSxx` instructions pack and saturate signed values. They convert a sequence of larger values to a sequence of smaller values via saturation. Those instructions with the *dw* suffix pack four double words into four words; those with the *wb* suffix saturate and pack eight signed words into eight signed bytes.

The `PACKSSDW` instruction takes the two double words in the source operand and the two double words in the destination operand and converts these to four signed words via saturation. The instruction packs these four words together and stores the result in the destination MMX register. See Figure 11.3 for details.

The `PACKSSWB` instruction takes the four words from the source operand and the four signed words from the destination operand and converts, via signed saturation, these values to eight signed bytes. This instruction leaves the eight bytes in the destination MMX register. See Figure 11.4 for details.

One application for these pack instructions is to convert UNICODE to ASCII (ANSI). You can convert UNICODE (16-bit) character to ANSI (8-bit) character if the H.O. eight bits of each UNICODE character is zero. The `PACKUSWB` instruction will take eight UNICODE characters and pack them into a string that is eight bytes long with a single instruction. If the H.O. byte of any UNICODE character contains a non-zero value, then the `PACKUSWB` instruction will store \$FF in the respective byte; therefore, you can use \$FF as a conversion error indication.

Another use for the `PACKSSWB` instruction is to translate a 16-bit audio stream to an eight-bit stream. Assuming you've scaled your sixteen-bit values to produce a sequence of values in the range -128..+127, you can use the `PACKSSWB` instruction to convert that sequence of 16-bit values into a packed sequence of eight bit values.

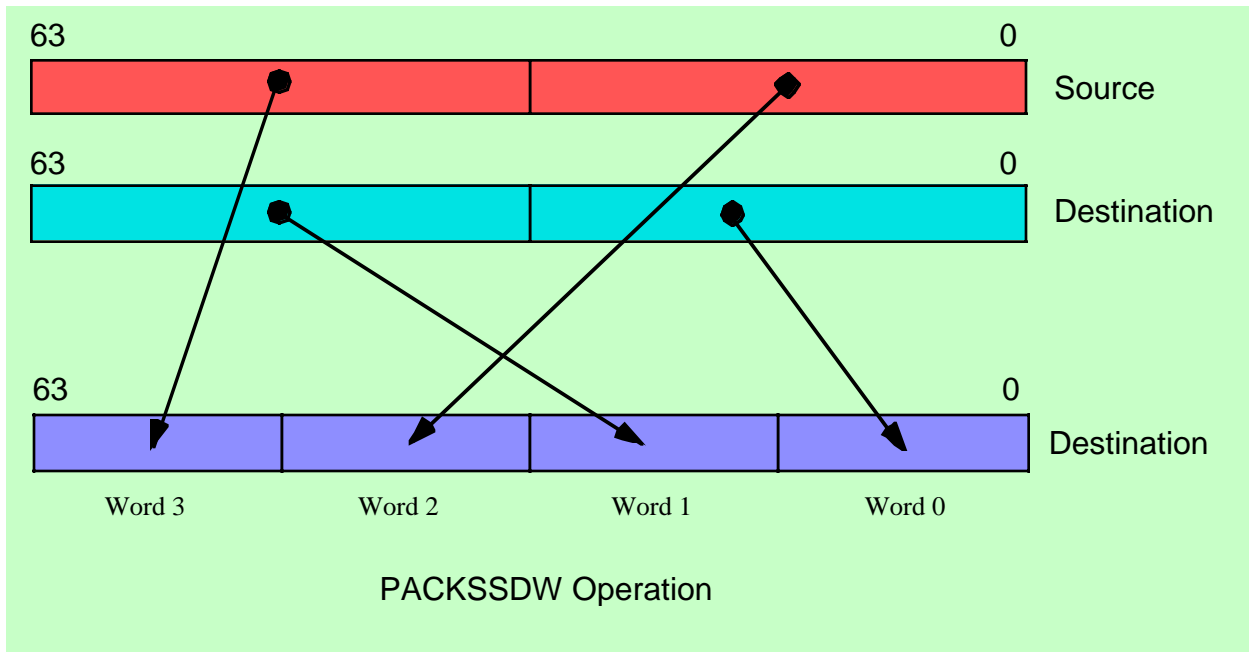


Figure 11.3 PACKSSDW Instruction

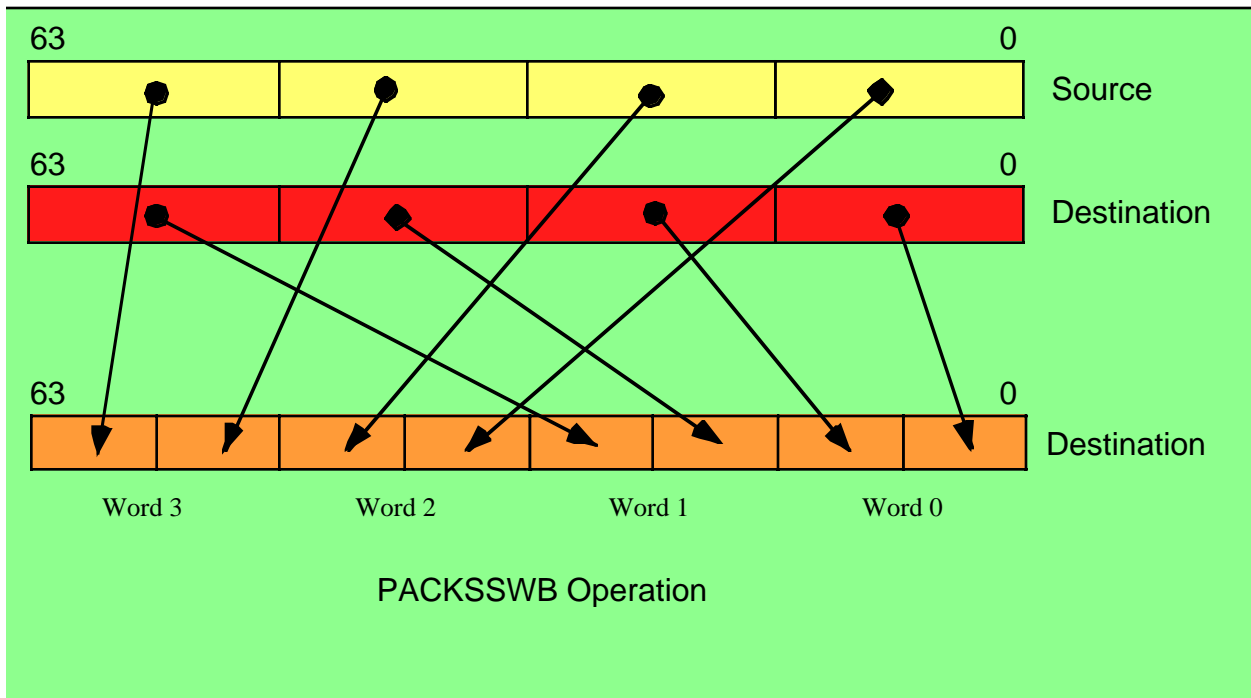


Figure 11.4 PACKSSWB Instruction

The unpack instructions (PUNPCKxxx) provide the converse operation to the pack instructions. The

unpack instructions take a sequence of smaller, packed, values and translate them into larger values. There is one problem with this conversion, however. Unlike the pack instructions, where it took two 64-bit operands to generate a single 64-bit result, the unpack operations will produce a 64-bit result from a single 32-bit result. Therefore, these instructions cannot operate directly on full 64-bit source operands. To overcome this limitation, there are two sets of unpack instructions: one set unpacks the data from the L.O. double word of a 64-bit object, the other set of instructions unpacks the H.O. double word of a 64-bit object. By executing one instruction from each set you can unpack a 64-bit object into a 128-bit object.

The PUNPCKLBW, PUNPCKLWD, and PUNPCKLDQ instructions merge (unpack) the L.O. double words of their source and destination operands and store the 64-bit result into their destination operand.

The PUNPCKLBW instruction unpacks and interleaves the low-order four bytes of the source (first) and destination (second) operands. It places the L.O. four bytes of the destination operand at the even byte positions in the destination and it places the L.O. four bytes of the source operand in the odd byte positions of the destination operand.(see Figure 11.5).

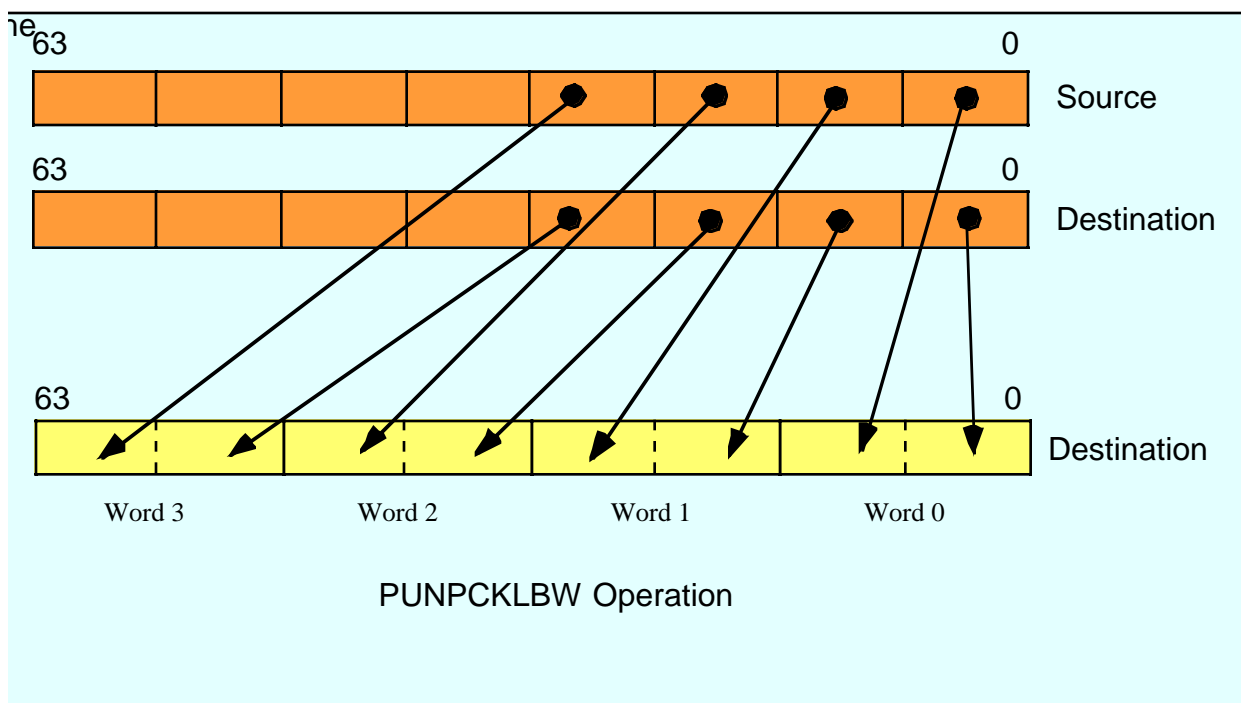


Figure 11.5 UNPCKLBW Instruction

The PUNPCKLWD instruction unpacks and interleaves the low-order two words of the source (first) and destination (second) operands. It places the L.O. two words of the destination operand at the even word positions in the destination and it places the L.O. words of the source operand in the odd word positions of the destination operand (see Figure 11.6).

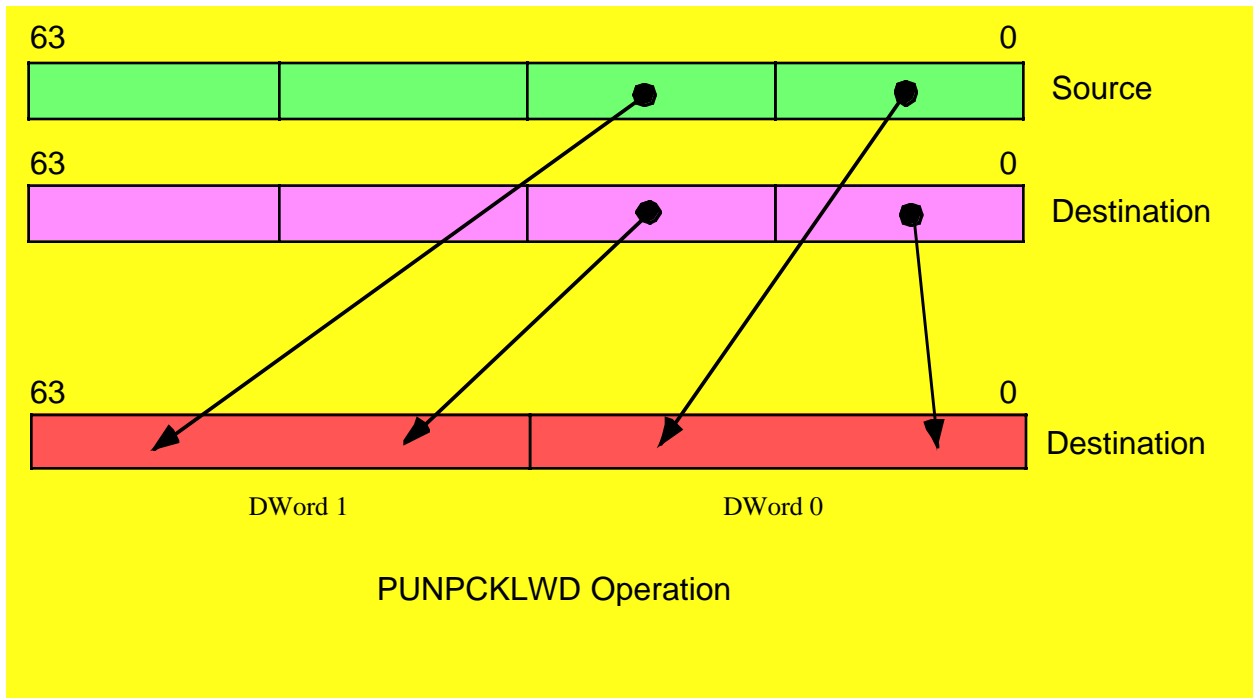


Figure 11.6 The PUNPCKLWD Instruction

The PUNPCKDQ instruction copies the L.O. dword of the source operand to the L.O. dword of the destination operand and it copies the (original) L.O. dword of the destination operand to the L.O. dword of the destination (i.e., it doesn't change the L.O. dword of the destination, see Figure 11.7).

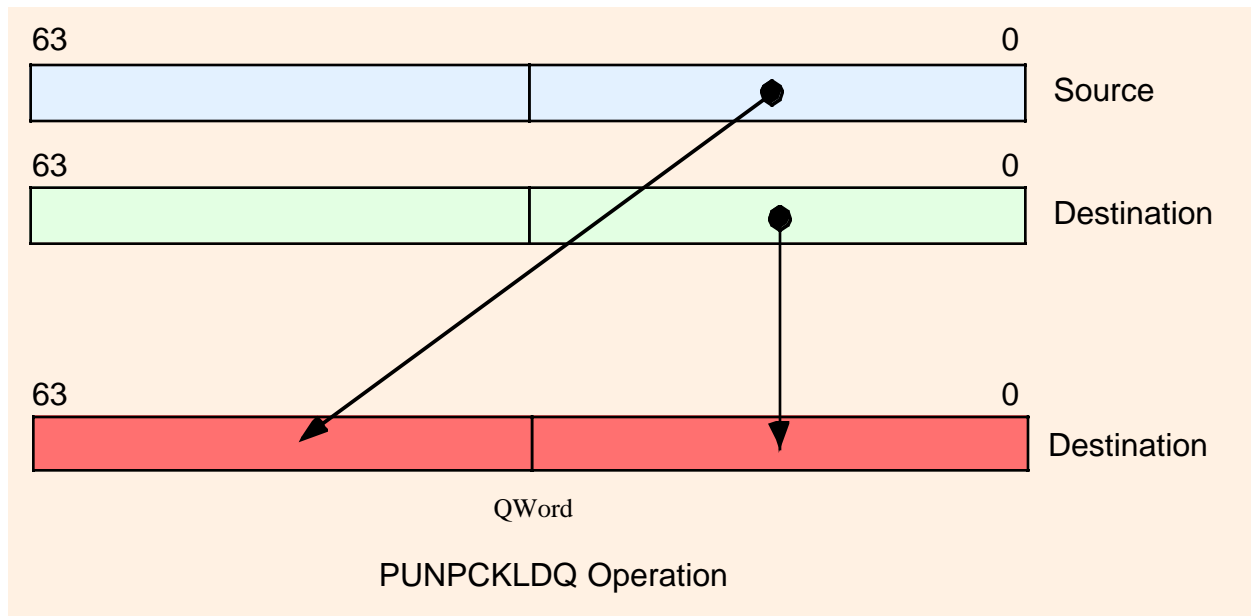


Figure 11.7 PUNPCKLDQ Instruction

The PUNPCKHBW instruction is quite similar to the PUNPCKLBW instruction. The difference is that it unpacks and interleaves the high-order four bytes of the source (first) and destination (second) operands. It places the H.O. four bytes of the destination operand at the even byte positions in the destination and it places the H.O. four bytes of the source operand in the odd byte positions of the destination operand (see Figure 11.8).

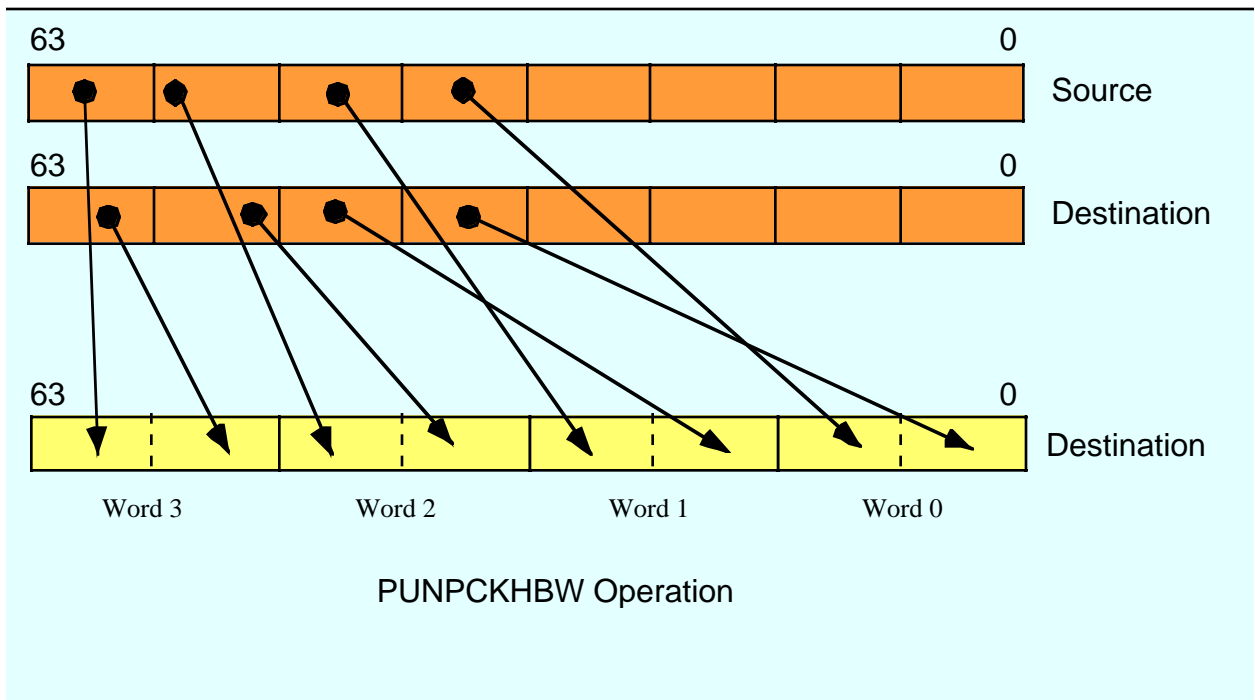


Figure 11.8 PUNPCKHBW Instruction

The PUNPCKHWD instruction unpacks and interleaves the low-order two words of the source (first) and destination (second) operands. It places the L.O. two words of the destination operand at the even word positions in the destination and it places the L.O. words of the source operand in the odd word positions of the destination operand (see Figure 11.9)

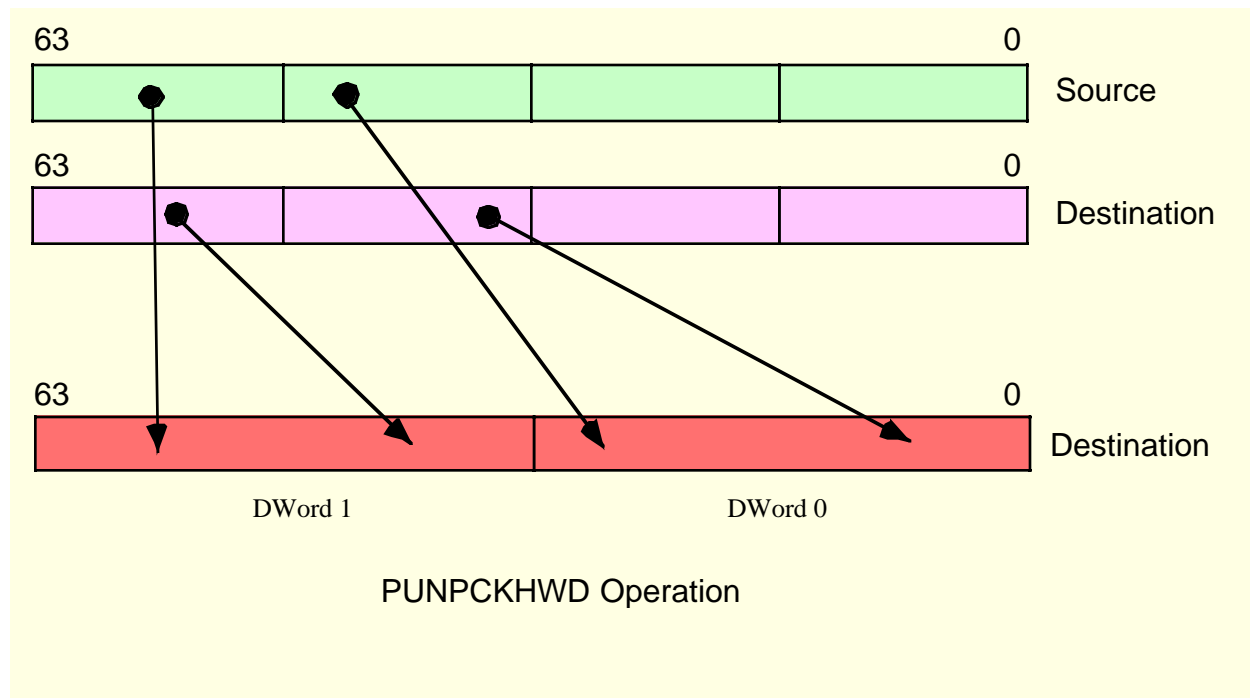


Figure 11.9 PUNPCKHWD Instruction

The PUNPCKHDQ instruction copies the H.O. dword of the source operand to the H.O. dword of the destination operand and it copies the (original) H.O. dword of the destination operand to the L.O. dword of the destination (see Figure 11.10).

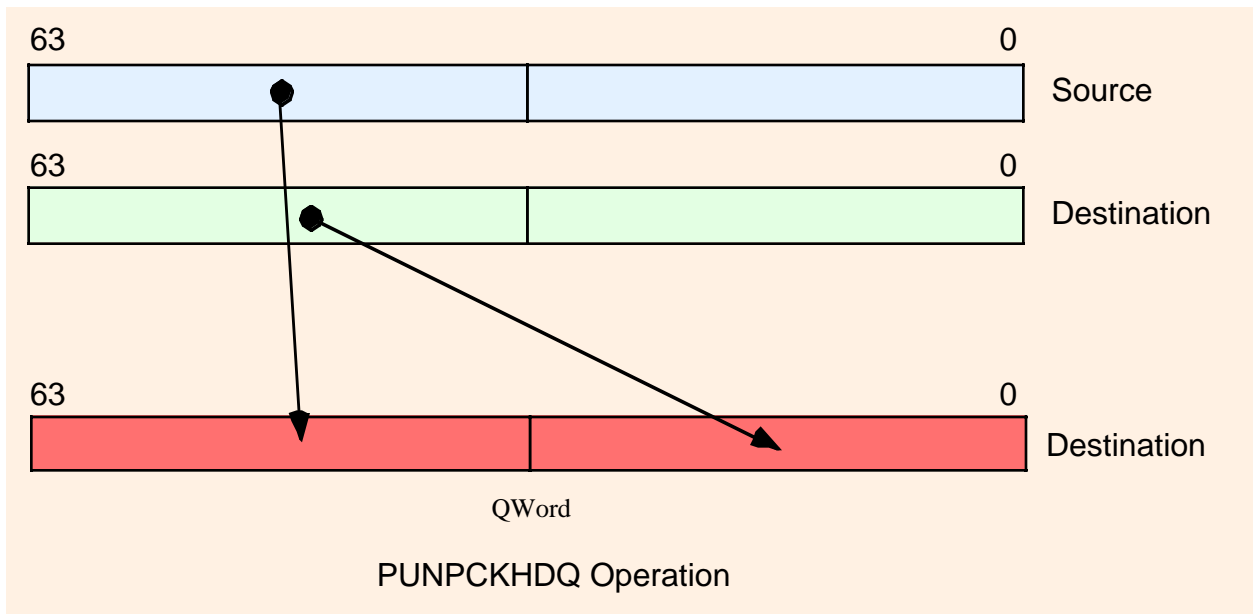


Figure 11.10 PUNPCKDQ Instruction

Since the unpack instructions provide the converse operation of the pack instructions, it should come as no surprise that you can use these instructions to perform the inverse algorithms of the examples given earlier for the pack instructions. For example, if you have a string of eight-bit ANSI characters, you can convert them to their UNICODE equivalents by setting one MMX register (the source) to all zeros. You can convert each four characters of the ANSI string to UNICODE by loading those four characters into the L.O. double word of an MMX register and executing the PUNPCKLBW instruction. This will interleave each of the characters with a zero byte, thus converting them from ANSI to UNICODE.

Of course, the unpack instructions are quite valuable any time you need to interleave data. For example, if you have three separate images containing the blue, red, and green components of a 24-bit image, it is possible to merge these three bytes together using the PUNPCKLBW instruction³.

11.7.3 MMX Packed Arithmetic Instructions

```
paddb( mem64, mmi );
paddb( mmi, mmi );
```

```
paddd( mem64, mmi );
paddd( mmi, mmi );
```

```
paddd( mem64, mmi );
paddd( mmi, mmi );
```

```
paddsb( mem64, mmi );
paddsb( mmi, mmi );
```

```
paddsw( mem64, mmi );
paddsw( mmi, mmi );
```

3. Typically you would merge in a fourth byte of zero and then store the resulting double word every three bytes in memory to overwrite the zeros.

```

paddusb( mem64, mmi );
paddusb( mmi, mmi );

paddusw( mem64, mmi );
paddusw( mmi, mmi );

psubb( mem64, mmi );
psubb( mmi, mmi );

psubw( mem64, mmi );
psubw( mmi, mmi );

psubd( mem64, mmi );
psubd( mmi, mmi );

psubsb( mem64, mmi );
psubsb( mmi, mmi );

psubsw( mem64, mmi );
psubsw( mmi, mmi );

psubusb( mem64, mmi );
psubusb( mmi, mmi );

psubusw( mem64, mmi );
psubusw( mmi, mmi );

pmulhuw( mem64, mmi );
pmulhuw( mmi, mmi );

pmulhw( mem64, mmi );
pmulhw( mmi, mmi );

pmullw( mem64, mmi );
pmullw( mmi, mmi );

pmaddwd( mem64, mmi );
pmaddwd( mmi, mmi );

```

The packed arithmetic instructions operate on a set of bytes, words, or double words within a 64-bit block. For example, the PADDW instruction computes four 16-bit sums of two operand simultaneously. None of these instructions affect the CPU's FLAGS register. Therefore, there is no indication of overflow, underflow, zero result, negative result, etc. If you need to test a result after a packed arithmetic computation, you will need to use one of the packed compare instructions (see "MMX Comparison Instructions" on page 1134).

The PADDB, PADDW, and PADDD instructions add the individual bytes, words, or double words in the two 64-bit operands using a wrap-around (i.e., non-saturating) addition. Any carry out of a sum is lost; it is your responsibility to ensure that overflow never occurs. As for the integer instructions, these packed add instructions add the values in the source operand to the destination operand, leaving the sum in the destination operand. These instructions produce correct results for signed or unsigned operands (assuming overflow/underflow does not occur).

The PADDSB and PADDSW instructions add the eight eight-bit or four 16-bit operands in the source and destination locations together using signed saturation arithmetic. The PADDUSB and PADDUSW instructions add their eight eight-bit or four 16-bit operands together using unsigned saturation arithmetic. Notice that you must use different instructions for signed and unsigned value since saturation arithmetic is different depending upon whether you are manipulating signed or unsigned operands. Also note that the instruction set does not support the saturated addition of double word values.

The PSUBB, PSUBW, and PSUBD instructions work just like their addition counterparts, except of course, they compute the wrap-around difference rather than the sum. These instructions compute `dest=dest-src`. Likewise, the PSUBSB, PSUBSW, PSUBUSB, and PSUBUSW instruction compute the difference of the destination and source operands using saturation arithmetic.

While addition and subtraction can produce a one-bit carry or borrow, multiplication of two n-bit operands can produce as large as a 2*n bit result. Since overflow is far more likely in multiplication than in addition or subtraction, the MMX packed multiply instructions work a little differently than their addition and subtraction counterparts. To successfully multiply two packed values requires two instructions - one to compute the L.O. component of the result and one to produce the H.O. component of the result. The PMULLW, PMULHW, and PMULHUW instructions handle this task.

The PMULLW instruction multiplies the four words of the source operand by the four words of the destination operand and stores the four L.O. words of the four double word results into the destination operand. This instruction ignores the H.O. words of the results. Used by itself, this instruction computes the wrap-around product of an unsigned or signed set of operands; this is also the L.O. words of the four products.

The PMULHW and PMULHUW instructions complete the calculation. After computing the L.O. words of the four products with the PMULLW instruction, you use either the PMULHW or PMULHUW instruction to compute the H.O. words of the products. These two instruction multiply the four words in the source by the four words in the destination and then store the H.O. words of the results in the destination MMX register. The difference between the two is that you use PMULHW for signed operands and PMULHUW for unsigned operands. If you compute the full product by using a PMULLW and a PMULHW (or PMULHUW) instruction pair, then there is no overflow possible, hence you don't have to worry about wrap-around or saturation arithmetic.

The PMADDWD instruction multiplies the four words in the source operand by the four words in the destination operand to produce four double word products. Then it adds the two L.O. double words together and stores the result in the L.O. double word of the destination MMX register; it also adds together the two H.O. double words and stores their sum in the H.O. word of the destination MMX register.

11.7.4 MMX Logic Instructions

```
pand( mem64, mmi );
pand( mmi, mmi );
```

```
pandn( mem64, mmi );
pandn( mmi, mmi );
```

```
por( mem64, mmi );
por( mmi, mmi );
```

```
pxor( mem64, mmi );
pxor( mmi, mmi );
```

The packed logic instructions are some examples of MMX instructions that actually operate on 64-bit values. There are no packed byte, packed word, or packed double word versions of these instructions. Of course, there is no need for special byte, word, or double word versions of these instructions since they would all be equivalent to the 64-bit logic instruction. Hence, if you want to logically AND eight bytes together in parallel, you use the PAND instruction; likewise, if you want to logically AND four words or two double words together, you just use the PAND instruction.

The PAND, POR, and PXOR instructions do the same thing as their 32-bit integer instruction counterparts (AND, OR, XOR) except, of course, they operate on two 64-bit MMX operands. Hence, no further discussion of these instructions is really necessary here. The PANDN (AND NOT) instruction is a new logic instruction, so it bears a little bit of a discussion. The PANDN instruction computes the following result:

```
dest := dest and (not source);
```

As you may recall from the chapter on Introduction to Digital Design, this is the inhibition function. If the destination operand is B and the source operand is A, this function computes $B = BA'$. (see “Boolean Functions and Truth Tables” on page 205 for details of the inhibition function). If you’re wondering why Intel chose to include such a weird function in the MMX instruction set, well, this instruction has one very useful property: it forces bits to zero in the destination operand everywhere there is a one bit in the source operand. This is an extremely useful function for merging to 64-bit quantities together. The following code sequence demonstrates this:

```
readonly
    AlternateNibbles: qword; nostorage;
    qword16( $F0F0_F0F0_F0F0_F0F0 ); // Note: needs qword16 macro!
    .
    .
    .
// Create a 64-bit value in MM0 containing the Odd nibbles from MM1 and
// the even nibbles from MM0:

    pandn( AlternateNibbles, mm0 ); // Clear the odd numbered nibbles.
    pand( AlternateNibbles, mm1 ); // Clear the even numbered nibbles.
    por( mm1, mm0 ); // Merge the two.
```

The PANDN operation is also useful for compute the set difference of two character sets. You could implement the *cs.difference* function using only six MMX instructions:

```
// Compute csdest := csdest - cssrc;

    movq( (type qword csdest), mm0 );
    pandn( (type qword cssrc), mm0 );
    movq( mm0, (type qword csdest) );
    movq( (type qword csdest[8]), mm0 );
    pandn( (type qword cssrc[8]), mm0 );
    movq( mm0, (type qword csdest[8]) );
```

Of course, if you want to improve the performance of the HLA Standard Library character set functions, you can use the MMX logic instructions throughout that module. Examples of such code appear later in this chapter.

11.7.5 MMX Comparison Instructions

```
pcmpeqb( mem64, mmi );
pcmpeqb( mmi, mmi );

pcmpq( mem64, mmi );
pcmpq( mmi, mmi );

pcmpeqd( mem64, mmi );
pcmpeqd( mmi, mmi );

pcmpgtb( mem64, mmi );
pcmpgtb( mmi, mmi );

pcmpgtw( mem64, mmi );
pcmpgtw( mmi, mmi );

pcmpgtd( mem64, mmi );
pcmpgtd( mmi, mmi );
```

The packed comparison instructions compare the destination (second) operand to the source (first) operand and to test for equality or greater than. These instructions compare eight pairs of bytes (PCMPEQB, PCMPGTB), four pairs of words (PCMPEQW, PCMPGTW), or two pairs of double words (PCMPEQD, PCMPGTD).

The first big difference to notice about these packed comparison instructions is that they compare the second operand to the first operand. This is exactly opposite of the standard CMP instruction (that compares the first operand to the second operand). The reason for this will become clear in a moment; however, you do have to keep in mind when using these instructions that the operands are opposite what you would normally expect. If this ordering bothers you, you can create macros to reverse the operands; we will explore this possibility a little later in this section.

The second big difference between the packed comparisons and the standard integer comparison is that these instructions test for a specific condition (equality or greater than) rather than doing a generic comparison. This is because these instructions, like the other MMX instructions, do not affect any condition code bits in the FLAGS register. This may seem contradictory, after all the whole purpose of the CMP instruction is to set the condition code bits. However, keep in mind that these instructions simultaneously compare two, four, or eight operands; that implies that you would need two, four, or eight sets of condition code bits to hold the results of the comparisons. Since the FLAGS register maintains only one set of condition code bits, it is not possible to reflect the comparison status in the FLAGS. This is why the packed comparison instructions test a specific condition - so they can return true or false to indicate the result of their comparison.

Okay, so where do these instructions return their true or false values? In the destination operand, of course. This is the third big difference between the packed comparisons and the standard integer CMP instruction - the packed comparisons modify their destination operand. Specifically, the PCMPEQB and PCMPGTB instruction compare each pair of bytes in the two operands and write false (\$00) or true (\$FF) to the corresponding byte in the destination operand, depending on the result of the comparison. For example, the instruction “pcmpgtb(MM1, MM0);” compares the L.O. byte of MM0 (A) with the L.O. byte of MM1 (B) and writes \$00 to the L.O. byte of MM0 if A is not greater than B. It writes \$FF to the L.O. byte of MM0 if A is greater than B (see Figure 11.11).

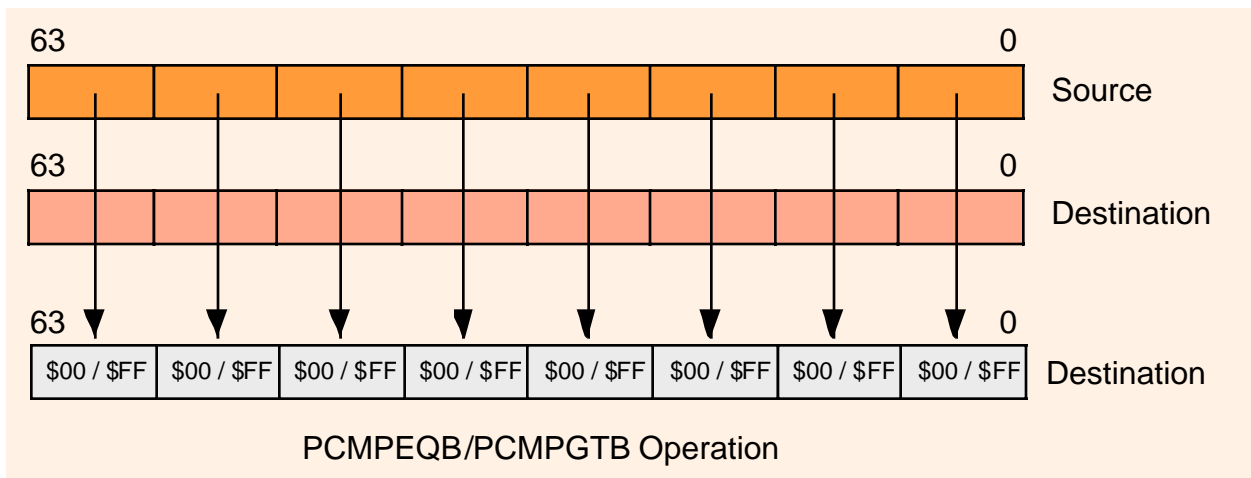


Figure 11.11 PCMPEQB and PCMPGTB Instructions

The PCMPEQW, PCMPGTW, PCMPEQD, and PCMPGTD instructions work in an analogous fashion except, of course, they compare words and double words rather than bytes (see Figure 11.12 and Figure 11.13).

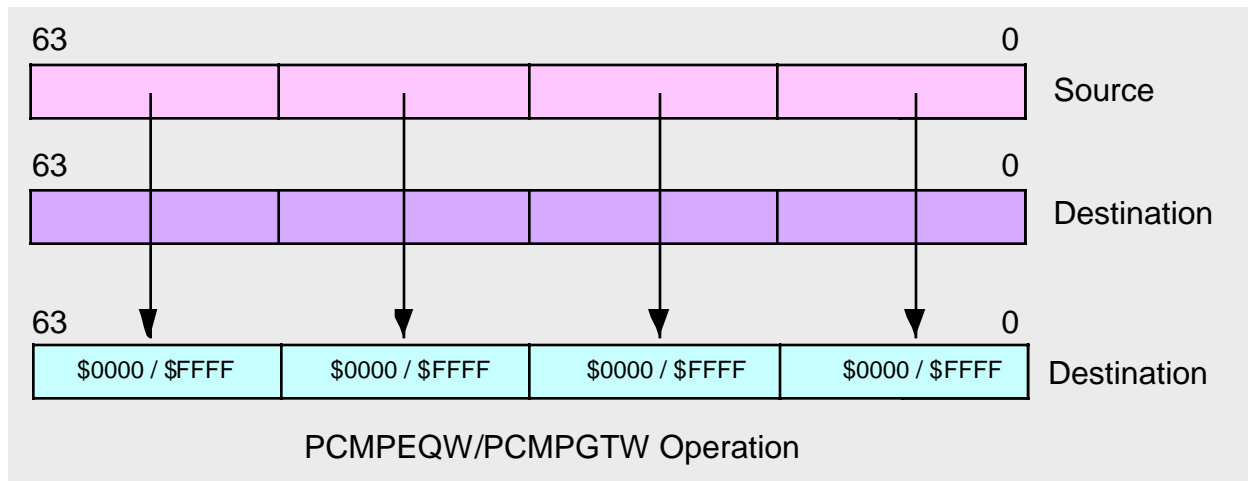


Figure 11.12 PCMPEQW and PCMPGTW Instructions

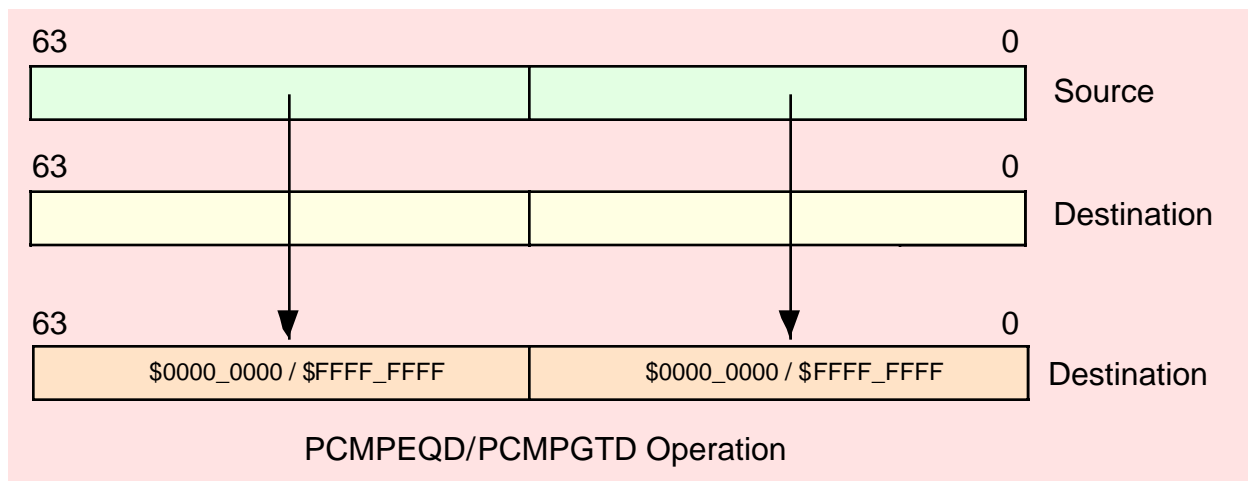


Figure 11.13 PCMPEQD and PCMPGTD Instructions

You've probably already noticed that there isn't a set of PCMPPLTx instructions. Intel chose not to provide these instructions because you can simulate them with the PCMPGTx instructions by reversing the operands. That is, $A > B$ implies $B < A$. Therefore, if you want to do a concurrent comparison of multiple operands for less than, you can use the PCMPGTx instructions to do this by simply reversing the operands. The only time this isn't directly possible is if your source operand is a memory operand; since the destination operand of the packed comparison instructions has to be an MMX register, you would have to move the memory operand into an MMX register before comparing them.

In addition to the lack of a packed less than comparison, you're also missing the not equals, less than or equal, and greater than or equal comparisons. You can easily synthesize these comparisons by executing a PXOR or POR instruction after the packed comparison.

To simulate a PCMPNEx instruction, all you've got to do is invert all the bits in the destination operand after executing a PCMPEQx instruction, e.g.,


```

pcmpeqb( mm1, mm0 );
pxor( AllOnes, mm0 ); // Assumption: AllOnes is a qword variable
                        // containing $FFFF_FFFF_FFFF_FFFF.

```

Of course, you can save the PXOR instruction by testing for zeros in the destination operand rather than ones (that is, use your program's logic to invert the result rather than actually computing the inverse).

To simulate the PCMPGEx and PCMPLEx instructions, you must do two comparisons, one for equality and one for greater than or less than, and then logically OR the results. Here's an example that computes $MM0 \leq MM1$:

```

movq( mm1, mm2 ); // Need a copy of destination operand.
pcmpgtb( mm0, mm1 ); // Remember: A<B is equal to B>A, so we're
pcmpeqb( mm0, mm2 ); // MM0<MM1 and MM0=MM1 here.
por( mm2, mm1 ); // Leaves boolean results in MM1.

```

If it really bothers you to have to reverse the operands, you can create macros to create your own PCMP-PLTx instructions. The following example demonstrates how to create the PCMPPLTB macro:

```

#macro pcmppltb( mmOp1, mmOp2 );

    pcmpgtb( mmOp2, mmOp1 );

#endmacro

```

Of course, you must keep in mind that there are two very big differences between this PCMPPLTB "instruction" and a true PCMPPLTB instruction. First, this form leaves the result in the first operand, not the second operand, hence the semantics of this "instruction" are different than the other packed comparisons. Second, the first operand has to be an MMX register while the second operand can be an MMX register or a quad word variable; again, just the opposite of the other packed instructions. The fact that this instruction's operands behave differently than the PCMPGTB instruction may create some problems. So you will have to carefully consider whether you really want to use this scheme to create a PCMPPLTB "instruction" for use in your programs. If you decide to do this, it would help tremendously if you always commented each invocation of the macro to point out that the first operand is the destination operand, e.g.,

```

pcmppltb( mm0, mm1 ); // Computes mm0 := mm1<mm0!

```

If the fact that the packed comparison instruction's operands are reversed bothers you, you can also use macros to swap those operands. The following example demonstrates how to write such macros for the PEQB (PCMPEQB), PGTB (PCMPGTB), and PLTB (packed less than, byte) instructions.

```

#macro peqb( leftOp, rightOp );

    pcmpeqb( rightOp, leftOp );

#endmacro

#macro pgtb( leftOp, rightOp );

    pcmpgtb( rightOp, leftOp );

#endmacro

#macro pltb( leftOp, rightOp );

    pcmpgtb( leftOp, rightOp );

#endmacro

```

Note that these macros don't solve the PLTB problem of having the wrong operand as the destination. However, these macros do compare the first operand to the second operand, just like the standard CMP instruction.

Of course, once you obtain a boolean result in an MMX register, you'll probably want to test the results at one point or another. Unfortunately, the MMX instructions only provide a couple of ways to move comparison information in and out of the MMX processor – you can store an MMX register value into memory or you can copy 32-bits of an MMX register to a general-purpose integer register. Since the comparison instructions produce a 64-bit result, writing the destination of a comparison to memory is the easiest way to gain access to the comparison results in your program. Typically, you'd use an instruction sequence like the following:

```
pcmpeqb( mm1, mm0 );           // Compare 8 bytes in mm1 to mm0.
movq( mm0, qwordVar );        // Write comparison results to memory.
if((type boolean qwordVar )) then

    << do this if byte #0 contained true ($FF, which is non-zero). >>

endif;
if((type boolean qwordVar[1])) then

    << do this if byte #1 contained true. >>

endif;
etc.
```

11.7.6 MMX Shift Instructions

```
pshllw( mmi, mmi );
pshllw( imm8, mmi );

pshlld( mmi, mmi );
pshlld( imm8, mmi );

pshllq( mmi, mmi );
pshllq( imm8, mmi );

pshrlw( mmi, mmi );
pshrlw( imm8, mmi );

pshrld( mmi, mmi );
pshrld( imm8, mmi );

pshrq( mmi, mmi );
pshrq( imm8, mmi );

pshraw( mmi, mmi );
pshraw( imm8, mmi );

pshrad( mmi, mmi );
pshrad( imm8, mmi );
```

The MMX shift, like the arithmetic instructions, allow you to simultaneously shift several different values in parallel. The PSHLLx instructions perform a packed shift left logical operation, the PSHLRx instructions do a packed logical shift right operation, and the PSRAx instruction do a packed arithmetic shift right operation. These instructions operate on word, double word, and quad word operands. Note that Intel does not provide a version of these instructions that operate on bytes.

The first operand to these instructions specifies a shift count. This should be an unsigned integer value in the range 0..15 for word shifts, 0..31 for double word operands, and 0..63 for quadword operands. If the

shift count is outside these ranges, then these instructions set their destination operands to all zeros. If the count (first) operand is not an immediate constant, then it must be an MMX register.

The PSLW instruction simultaneously shifts the four words in the destination MMX register to the left the number of bit positions specified by the source operand. The instruction shifts zero into the L.O. bit of each word and the bit shifted out of the H.O. bit of each word is lost. There is no carry from one word to the other (since that would imply a larger shift operation). This instruction, like all the other MMX instructions, does not affect the FLAGS register (including the carry flag).

The PSLD instruction simultaneously shifts the two double words in the destination MMX register to the left one bit position. Like the PSLW instruction, this instruction shifts zeros into the L.O. bits and any bits shifted out of the H.O. positions are lost.

The PSLQ is one of the few MMX instructions that operates on 64-bit quantities. This instruction shifts the entire 64-bit destination register to the left the number of bits specified by the count (source) operand. In addition to allowing you to manipulate 64-bit integer quantities, this instruction is especially useful for moving data around in MMX registers so you can pack or unpack data as needed.

Although there is no PSLLB instruction to shift bits, you can simulate this instruction using a PSLW and a PANDN instruction. After shifting the word values to the left the specified number of bits, all you've got to do is clear the L.O. n bits of each byte, where n is the shift count. For example, to shift the bytes in MM0 to the left three positions you could use the following two instructions:

```
static
ThreeBitsZero: byte; @nostorage;
    byte $F8, $F8, $F8, $F8, $F8, $F8, $F8, $F8;
    .
    .
    .
    psllw( 3, mm0 );
    pandn( ThreeBitsZero, mm0 );
```

The PSLRW, PSLRD, and PSLRQ instructions work just like their left shift counterparts except that these instructions shift their operands to the right rather than to the left. They shift zeros into the vacated H.O. positions of the destination values and bits they shift out of the L.O. bits are lost. As with the shift left instructions, there is no PSLRB instruction but you can easily simulate this with a PSLRW and a PANDN instruction.

The PSRAW and PSRAD instructions do an arithmetic shift right operation on the words or double words in the destination MMX register. Note that there isn't a PSRAQ instruction. While shifting data to the right, these instructions replicate the H.O. bit of each word, double word, or quad word rather than shifting in zeros. As for the logical shift right instructions, bits that these instructions shift out of the L.O. bits are lost forever.

The PSLQ and PSLRQ instructions provide a convenient way to shift a quad word to the left or right. However, the MMX shift instructions are not generally useful for extended precision shifts since all data shifted out of the operands is lost. If you need to do an extended precision shift other than 64 bits, you should stick with the SHLD and SHRD instructions. The MMX shift instructions are mainly useful for shifting several values in parallel or (PSLQ and PSLRQ) repositioning data in an MMX register.

11.8 The EMMS Instruction

```
emms();
```

The EMMS (Empty MMX Machine State) instruction restores the FPU status on the CPU so that it can begin processing FPU instructions again after an MMX instruction sequence. You should always execute the EMMS instruction once you complete some MMX sequence. Failure to do so may cause any following floating point instructions to fail.

When an MMX instruction executes, the floating point tag word is marked valid (00s). Subsequent floating-point instructions that will be executed may produce unexpected results because the floating-point stack seems to contain valid data. The EMMS instruction marks the floating point tag word as empty. This must occur before the execution of any following floating point instructions.

Of course, you don't have to execute the EMMS instruction immediately after an MMX sequence if you're going to execute some additional MMX instructions prior to executing any FPU instructions, but you must take care to execute this instruction if

- You call any library routines or OS APIs (that might possibly use the FPU).
- You switch tasks in a cooperative fashion (for example, see the chapter on Coroutines in the Volume on Advanced Procedures).
- You execute any FPU instructions.

If the EMMS instruction is not used when trying to execute a floating-point instruction, the following may occur:

- Depending on the exception mask bits of the floating-point control word, a floating point exception event may be generated.
- A "soft exception" may occur. In this case floating-point code continues to execute, but generates incorrect results.

The EMMS instruction is rather slow, so you don't want to unnecessarily execute it, but it is critical that you execute it at the appropriate times. Of course, better safe than sorry; if you're not sure you're going to execute more MMX instructions before any FPU instructions, then go ahead and execute the EMMS instruction to clear the state.

11.9 The MMX Programming Paradigm

In general, you don't learn scalar (non-MMX) 80x86 assembly language programming and then use that same mindset when writing programs using the MMX instruction set. While it is possible to directly use various MMX instructions the same way you would the general purpose integer instructions, one phrase comes to mind when working with MMX: think parallel. This text has spent many hundreds of pages up to this point attempting to get you to think in assembly language; to think that this small section can teach you how to design optimal MMX sequence would be ludicrous. Nonetheless, a few simple examples are useful to help start you thinking about how to use the MMX instructions to your benefit in your programs. This section will begin by presenting some fairly obvious uses for the MMX instruction set, and then it will attempt to present some examples that exploit the inherent parallelism of the MMX instructions.

Since the MMX registers are 64-bits wide, you can double the speed of certain data movement operations by using MMX registers rather than the 32-bit general purpose registers. For example, consider the following code from the HLA Standard Library that copies one character set object to another:

```
procedure cs.cpy( src:cset; var dest:cset ); nodisplay;
begin cpy;

    push( eax );
    push( ebx );
    mov( dest, ebx );
    mov( (type dword src), eax );
    mov( eax, [ebx] );
    mov( (type dword src[4]), eax );
    mov( eax, [ebx+4] );
    mov( (type dword src[8]), eax );
    mov( eax, [ebx+8] );
```

```

mov( (type dword src[12]), eax );
mov( eax, [ebx+12] );
pop( ebx );
pop( eax );

end cpy;

```

Program 11.2 HLA Standard Library *cs.cpy* Routine

This is a relatively simple code sequence. Indeed, a fair amount of the execution time is spent copying the parameters (20 bytes) onto the stack, calling the routine, and returning from the routine. This entire sequence can be reduced to the following four MMX instructions:

```

movq( (type qword src), mm0 );
movq( (type qword src[8]), mm1 );
movq( mm0, (type qword dest));
movq( mm1, (type qword dest[8]));

```

Of course, this sequence assumes two things: (1) it's okay to wipe out the values in MM0 and MM1, and (2) you'll execute the EMMS instruction a little later on after the execution of some other MMX instructions. If either, or both, of these assumptions is incorrect, the performance of this sequence won't be quite as good (though probably still better than the *cs.cpy* routine). However, if these two assumptions do hold, then it's relatively easy to implement the *cs.cpy* routine as an in-line function (i.e., a macro) and have it run much faster. If you really need this operation to occur inside a procedure and you need to preserve the MMX registers, and you don't know if any MMX instructions will execute shortly thereafter (i.e., you'll need to execute EMMS), then it's doubtful that using the MMX instructions will help here. However, in those cases when you can put the code in-line, using the MMX instructions will be faster.

Warning: don't get too carried away with the MMX MOVQ instruction. Several programmers have gone to great extremes to use this instruction as part of a high performance MOVSD replacement. However, except in very special cases on very well designed systems, the limiting factor for a block move is the speed of memory. Since Intel has optimized the operation of the MOVSD instruction, you're best off using the MOVSD instructions when moving blocks of memory around.

Earlier, this chapter used the *cs.difference* function as an example when discussing the PANDN instruction. Here's the original HLA Standard Library implementation of this function:

```

procedure cs.difference( src:cset; var dest:cset ); nodisplay;
begin difference;

    push( eax );
    push( ebx );
    mov( dest, ebx );
    mov( (type dword src), eax );
    not( eax );
    and( eax, [ebx] );
    mov( (type dword src[4]), eax );
    not( eax );
    and( eax, [ebx+4] );
    mov( (type dword src[8]), eax );
    not( eax );
    and( eax, [ebx+8] );
    mov( (type dword src[12]), eax );
    not( eax );
    and( eax, [ebx+12] );
    pop( ebx );

```

```

    pop( eax );

end difference;

```

Program 11.3 HLA Standard Library *cs.difference* Routine

Once again, the high-level nature of HLA is hiding the fact that calling this function is somewhat expensive. A typical call to *cs.difference* emits five or more instructions just to push the parameters (it takes four 32-bit PUSH instructions to pass the *src* character set because it is a value parameter). If you're willing to wipe out the values in MM0 and MM1, and you don't need to execute an EMMS instruction right away, it's possible to compute the set difference with only six instructions – that's about the same number of instructions (and often fewer) than are needed to call this routine, much less do the actual work. Here are those six instructions:

```

    movq( dest, mm0 );
    movq( dest[8], mm1 );
    pandn( src, mm0 );
    pandn( src[8], mm1 );
    movq( mm0, dest );
    movq( mm1, dest[8] );

```

These six instructions replace 12 of the instructions in the body of the function. The sequence is sufficiently short that it's reasonable to code it in-line rather than in a function. However, were you to bury this code in the *cs.difference* routine, you needed to preserve MM0 and MM1⁴, and you needed to execute EMMS afterwards, this would cost more than it's worth. As an in-line macro, however, it is going to be significantly faster since it avoids passing parameters and the call/return sequence.

If you want to compute the intersection of two character sets, the instruction sequence is identical to the above except you substitute PAND for PANDN. Similarly, if you want to compute the union of two character sets, use the code sequence above substituting POR for PANDN. Again, both approaches pay off handsomely if you insert the code in-line rather than burying it in a procedure and you don't need to preserve MMX registers or execute EMMS afterwards.

We can continue with this exercise of working our way through the HLA Standard Library character set (and other) routines substituting MMX instructions in place of standard integer instructions. As long as we don't need to preserve the MMX machine state (i.e., registers) and we don't have to execute EMMS, most of the character set operations will be short enough to code in-line. Unfortunately, we're not buying that much over code the standard implementations of these functions in-line from a performance point of view (though the code would be quite a bit shorter). The problem here is that we're not "thinking in MMX." We're still thinking in scalar (non-parallel mode) and the fact that the MMX instruction set requires a lot of set-up (well, "tear-down" actually) negates many of the advantages of using MMX instructions in our programs.

The MMX instructions are most appropriate when you compute multiple results in parallel. The problem with the character set examples above is that we're not even processing a whole data object with a single instruction; we're actually only processing a half of a character set with a sequence of three MMX instructions (i.e., it requires six instructions to compute the intersection, union, or difference of two character sets). At best, we can only expect the code to run about twice as fast since we're processing 64 bits at a time instead of 32 bits. Executing EMMS (and, God help us, having to preserve MMX registers) negates much of what we might gain by using the MMX instructions. Again, we're only going to see a speed improvement if we process multiple objects with a single MMX instruction. We're not going to do that manipulating large objects like character sets.

One data type that will let us easily manipulate up to eight objects at one time is a character string. We can speed up many character string operations by operating on eight characters in the string at one time. Consider the HLA Standard Library *str.uppercase* procedure. This function steps through each character of

4. Actually, the code could be rewritten easily enough to use only one MMX register.

a string, tests to see if it's a lower case character, and if so, converts the lower case character to upper case. A good question to ask is "can we process eight characters at a time using the MMX instructions?" The answer turns out to be yes and the MMX implementation of this function provides an interesting perspective on writing MMX code.

At first glance it might seem impractical to use the MMX instructions to test for lower case characters and convert them to upper case. Consider the typical scalar approach that tests and converts a single character at a time:

```
<< Get character to convert into the AL register >>

    cmp( al, 'a' );
    jb noConversion;
    cmp( al, 'z' );
    ja noConversion;
    sub( $20, al );    // Could also use AND($5f, al); here.
noConversion:
```

This code first checks the value in AL to see if it's actually a lower case character (that's the CMP and Jcc instructions in the code above). If the character is outside the range 'a'..'z' then this code skips over the conversion (the SUB instruction); however, if the code is in the specified range, then the sequence above drops through to the SUB instruction and converts the lower case character to upper case by subtracting \$20 from the lower case character's ASCII code (since lower case characters always have bit #5 set, subtracting \$20 always clears this bit).

Any attempt to convert this code directly to an MMX sequence is going to fail. Comparing and branching around the conversion instruction only works if you're converting one value at a time. When operating on eight characters simultaneously, any mixture of the eight characters may or may not require conversion from lower case to upper case. Hence, we need to be able to perform some calculation that is benign if the character is not lower case (i.e., doesn't affect the character's value) while converting the character to upper case if it was lower case to begin with. Worse, we have to do this with pure computation since flow of control isn't going to be particularly effective here (if we test each individual result in our MMX register we won't really save anything over the scalar approach). To save you some suspense, yes, such a calculation does exist.

Consider the following algorithm that converts lower case characters to upper case:

```
<< Get character to test into AL >>
    cmp( al, 'a' );
    setae( bl );    // bl := al >= 'a'
    cmp( al, 'z' );
    setbe( bh );    // bh := al <= 'z'
    and( bh, bl ); // bl := (al >= 'a') && (al <= 'z' );
    dec( bl );     // bl := $FF/$00 if false/true.
    not( bl );     // bl := $FF/$00 if true/false.
    and( $20, bl ); // bl := $20/$00 if true/false.
    sub( bl, al ); // subtract $20 if al was lowercase.
```

This code sequence is fairly straight-forward up until the DEC instruction above. It computes true/false in BL depending on whether AL is in the range 'a'..'z'. At the point of the DEC instruction, BL contains one if AL is a lower case character, it contains zero if AL's value is not lower case. After the DEC instruction, BL contains \$FF for false (AL is not lower case) and \$00 for true (AL is lowercase). The code is going to use this as a mask a little later, but it really needs true to be \$FF and false \$00, hence the NOT instruction that follows. The (second) AND instruction above converts true to \$20 and false to \$00 and the final SUB instruction subtracts \$20 if AL contained lower case, it subtracts \$00 from AL if AL did not contain a lower case character (subtracting \$20 from a lower case character will convert it to upper case).

Whew! This sequence probably isn't very efficient when compared to the simpler code given previously. Certainly there are more instructions in this version (nearly twice as many). Whether this code without any branches runs faster or slower than the earlier code with two branches is a good question. The important thing to note here, though, is that we converted the lower case characters to upper case (leaving

other characters unchanged) using only a calculation; no program flow logic is necessary. This means that the code sequence above is a good candidate for conversion to MMX. Even if the code sequence above is slower than the previous algorithm when converting one character at a time to upper case, it's positively going to scream when it converts eight characters at a shot (since you'll only need to execute the sequence one-eighth as many times).

The following is the code sequence that will convert the eight characters starting at location [EDI] in memory to upper case:

```
static
  A:qword; @nostorage;
    byte $60, $60, $60, $60, $60, $60, $60, $60; // Note: $60 = 'a'-1.
  Z:qword; @nostorage;
    byte $7B, $7B, $7B, $7B, $7B, $7B, $7B, $7B; // Note: $7B = 'z' + 1.
  ConvFactor:qword; @nostorage;
    byte $20, $20, $20, $20, $20, $20, $20, $20; // Magic value for lc->UC.
    .
    .
    .
  movq( ConvFactor, mm4 ); // Eight copies of conversion value.
  movq( A, mm2 ); // Put eight "a" characters in mm2.
  movq( Z, mm3 ); // Put eight "z" characters in mm3.
  movq( [edi], mm0 ); // Get next eight characters of our string.
  movq( mm0, mm1 ); // We need two copies.
  pcmptgb( mm2, mm1 ); // Generate 1's in MM1 everywhere chars >= 'a'
  pcmptgb( mm0, mm3 ); // Generate 1's in MM3 everywhere chars <= 'z'
  pand( mm3, mm1 ); // Generate 1's in MM1 when 'a'<=chars<='z'
  pand( mm4, mm1 ); // Generates $20 in each spot we have a l.c. char
  psubb( mm1, mm0 ); // Convert l.c. chars to U.C. by adding $20.
  movq( mm0, [edi]);
```

Note that this code compares the characters that [EDI] points at to 'a'-1 and 'z'+1 because we only have a greater than comparison rather than a greater or equal comparison (this saves a few extra instructions). Other than setting up the MMX registers and taking advantage of the fact that the PCMPGTB instructions automatically produce \$FF for true and \$00 for false, this is a faithful reproduction of the previous algorithm except it operates on eight bytes simultaneously. So if we put this code in a loop and execute it once for each eight characters in the string, there will be one-eighth the iterations of a similar loop using the scalar instructions.

Of course, there is one problem with this code. Not all strings have lengths that are an even multiple of eight bytes. Therefore, we've got to put some special case code into our algorithm to handle strings that are less than eight characters long and handle strings whose length is not an even multiple of eight characters. In the following program, the *mmxupper* function simply borrows the scalar code from the HLA Standard Library's *str.upper* procedure to handle the leftover characters. The following example program provides both an MMX and a scalar solution with a main program that compares the running time of both. If you're wondering, the MMX version is about three times faster (on a Pentium III) for strings around 35 characters long, containing mostly lower case (mostly lower case favors the scalar algorithm since fewer branches are taken with lower case characters; longer strings favor the MMX algorithm since it spends more time in the MMX code compared to the scalar code at the end).

```
program UpperCase;
#include( "stdlib.hhf" )

// The following code was stolen from the
// HLA Standard Library's str.upper function.
// It is not optimized, but then none of this
// code is optimized other than to use the MMX
// instruction set (later).
```



```

procedure strupper( dest: string ); @nodisplay;
begin strupper;

    push( edi );
    push( eax );

    mov( dest, edi );
    if( edi = 0 ) then

        raise( ex.AttemptToDerefNULL );

    endif;

    // Until we encounter a zero byte, convert any lower
    // case characters to upper case.

    forever

        mov( [edi], al );
        breakif( al = 0 );      // Quit when we find a zero byte.

        // If a lower case character, convert it to upper case
        // and store the result back into the destination string.

        if
        (#{
            cmp( al, 'a' );
            jb false;
            cmp( al, 'z' );
            ja false;
        }#) then

            and( $5f, al );    // Magic lc->UC translation.
            mov( al, [edi] );  // Save result.

        endif;

        // Move on to the next character.

        inc( edi );

    endfor;

    pop( edi );
    pop( eax );

end strupper;

```

```

procedure mmxupper( dest: string ); @nodisplay;
const
    zCh:char := char( uns8( 'z' ) + 1 );
    aCh:char := char( uns8( 'a' ) - 1 );

static

    // Create eight copies of the A-1 and Z+1 characters
    // so we can compare eight characters at once:

```

```

A:qword; @nostorage;
    byte aCh, aCh, aCh, aCh, aCh, aCh, aCh, aCh;

Z:qword; @nostorage;
    byte zCh, zCh, zCh, zCh, zCh, zCh, zCh, zCh;

// Conversion factor: UC := LC - $20.

ConvFactor: qword; @nostorage;
    byte $20, $20, $20, $20, $20, $20, $20, $20;

begin mmxupper;

    push( edi );
    push( eax );

    mov( dest, edi );
    if( edi = 0 ) then

        raise( ex.AttemptToDerefNULL );

    endif;

    // Some invariant operations (things that don't
    // change on each iteration of the loop):

    movq( A, mm2 );
    movq( ConvFactor, mm4 );

    // Get the string length from the length field:

    mov( (type str.strRec [edi]).length, eax );

    // Process the string in blocks of eight characters:

    while( (type int32 eax) >= 8 ) do

        movq( [edi], mm0 ); // Get next eight characters of our string.
        movq( mm0, mm1 ); // We need two copies.
        movq( Z, mm3 ); // Need to refresh on each loop.
        pcmprgtb( mm2, mm1 ); // Generate 1's in MM1 everywhere chars >= 'a'
        pcmprgtb( mm0, mm3 ); // Generate 1's in MM3 everywhere chars <= 'z'
        pand( mm3, mm1 ); // Generate 1's in MM1 when 'a'<=chars<='z'
        pand( mm4, mm1 ); // Generates $20 in each spot we have a l.c. char
        psubb( mm1, mm0 ); // Convert l.c. chars to U.C. by adding $20.
        movq( mm0, (type qword [edi]));

        // Move on to the next eight characters in the string.

        sub( 8, eax );
        add( 8, edi );

    endwhile;

    // If we're processing less than eight characters, do it the old-fashioned
    // way (one character at a time). This also handles the last 1..7 chars
    // if the number of characters is not an even multiple of eight. This
    // code was swiped directly from the HLA str.upper function (above).

    if( eax != 0 ) then

```

```

    forever

        mov( [edi], al );
        breakif( al = 0 );      // Quit when we find a zero byte.

        // If a lower case character, convert it to upper case
        // and store the result back into the destination string.

        if
        (#{
            cmp( al, 'a' );
            jb false;
            cmp( al, 'z' );
            ja false;
        }#) then

            and( $5f, al );      // Magic lc->UC translation.
            mov( al, [edi] );    // Save result.

        endif;

        // Move on to the next character.

        inc( edi );

    endfor;

endif;
emms(); // Clean up MMX state.

pop( edi );
pop( eax );

end mmxupper;

static
    MyStr: string := "Hello There, MMX Uppercase Routine!";
    destStr:string;
    mmxCycles:qword;
    strCycles:qword;

begin UpperCase;

    // Charge up the cache (prefetch the code and data
    // to avoid cache misses later).

    mov( str.a_cpy( MyStr ), destStr );
    mmxupper( destStr );
    strupper( destStr );

    // Okay, time the execution of the MMX version:

    mov( str.a_cpy( MyStr ), destStr );

    rdtsc();
    mov( eax, (type dword mmxCycles));

```

```

mov( edx, (type dword mmxCycles[4]));
mmxupper( destStr );
rdtsc();
sub( (type dword mmxCycles), eax );
sbb( (type dword mmxCycles[4]), edx );
mov( eax, (type dword mmxCycles));
mov( edx, (type dword mmxCycles[4]));

stdout.put( "Dest String = `", destStr, "`", nl );

// Okay, time the execution of the HLA version:

mov( str.a_cpy( MyStr ), destStr );

rdtsc();
mov( eax, (type dword strCycles));
mov( edx, (type dword strCycles[4]));
strupper( destStr );
rdtsc();
sub( (type dword strCycles), eax );
sbb( (type dword strCycles[4]), edx );
mov( eax, (type dword strCycles));
mov( edx, (type dword strCycles[4]));

stdout.put( "Dest String(2) = `", destStr, "`", nl );

stdout.put( "MMX cycles:" );
stdout.puti64( mmxCycles );
stdout.put( nl "HLA cycles: " );
stdout.puti64( strCycles );
stdout.newln();

end UpperCase;

```

Program 11.4 MMX Implementation of the HLA Standard Library str.upper Procedure

Other string functions, like a case insensitive string comparison, can greatly benefit from the use of parallel computation via the MMX instruction set. Implementation of other string functions is left as an exercise to the reader; interested readers should consider converting string functions that involve calculations and tests on each individual characters in a string as candidates for optimization via MMX.

11.10 Putting It All Together

Intel's MMX enhancements to the basic Pentium instruction set allow the acceleration of certain algorithms. Unfortunately, the MMX instruction set isn't generally applicable to a wide range of problems. The MMX instructions, with their SIMD orientation, are generally useful for manipulating a large amount of data organized as byte, word, or double word arrays where the MMX instructions can calculate several values in parallel. Learning to effectively use the MMX instruction set requires a paradigm shift on the part of the programmer. You don't apply the same rules for scalar 80x86 instructions to the MMX instructions. However, if you take the time to master parallel programming techniques with the MMX instructions, then you will be able to accelerate many of your applications.

12.1 Chapter Overview

Most assembly language code doesn't appear in a stand-alone assembly language program. Instead, most assembly code is actually part of a library package that programs written in a high level language wind up calling. Although HLA makes it really easy to write standalone assembly applications, at one point or another you'll probably want to call an HLA procedure from some code written in another language or you may want to call code written in another language from HLA. This chapter discusses the mechanisms for doing this in three languages: low-level assembly (i.e., MASM or Gas), C/C++, and Delphi/Kylix. The mechanisms for other languages are usually similar to one of these three, so the material in this chapter will still apply even if you're using some other high level language.

12.2 Mixing HLA and MASM/Gas Code in the Same Program

It may seem kind of weird to mix MASM or Gas and HLA code in the same program. After all, they're both assembly languages and almost anything you can do with MASM or Gas can be done in HLA. So why bother trying to mix the two in the same program? Well, there are three reasons:

- You've already got a lot of code written in MASM or Gas and you don't want to convert it to HLA's syntax.
- There are a few things MASM and Gas do that HLA cannot, and you happen to need to do one of those things.
- Someone else has written some MASM or Gas code and they want to be able to call code you've written using HLA.

In this section, we'll discuss two ways to merge MASM/Gas and HLA code in the same program: via in-line assembly code and through linking object files.

12.2.1 In-Line (MASM/Gas) Assembly Code in Your HLA Programs

As you're probably aware, the HLA compiler doesn't actually produce machine code directly from your HLA source files. Instead, it first compiles the code to a MASM or Gas-compatible assembly language source file and then it calls MASM or Gas to assemble this code to object code. If you're interested in seeing the MASM or Gas output HLA produces, just edit the *filename.ASM* file that HLA creates after compiling your *filename.HLA* source file. The output assembly file isn't amazingly readable, but it is fairly easy to correlate the assembly output with the HLA source file.

HLA provides two mechanisms that let you inject raw MASM or Gas code directly into the output file it produces: the `#ASM..#ENDASM` sequence and the `#EMIT` statement. The `#ASM..#ENDASM` sequence copies all text between these two clauses directly to the assembly output file, e.g.,

```
#asm

    mov eax, 0          ;MASM/Gas syntax for MOV( 0, EAX );
    add eax, ebx       ; " " " " ADD( ebx, eax );

#endasm
```

The `#ASM..#ENDASM` sequence is how you inject in-line (MASM or Gas) assembly code into your HLA programs. For the most part there is very little need to use this feature, but in a few instances it is valuable.

Note, when using Gas, that HLA specifies the “.intel_syntax” directive, so you should use Intel syntax when supplying Gas code between #asm and #endasm.

For example, if you’re writing structured exception handling code under Windows, you’ll need to access the double word at address FS:[0] (offset zero in the segment pointed at by the 80x86’s FS segment register). Unfortunately, HLA does not support segmentation and the use of segment registers. However, you can drop into MASM for a statement or two in order to access this value:

```
#asm
  mov ebx, fs:[0]      ; Loads process pointer into EBX
#endasm
```

At the end of this instruction sequence, EBX will contain the pointer to the process information structure that Windows maintains.

HLA blindly copies all text between the #ASM and #ENDASM clauses directly to the assembly output file. HLA does not check the syntax of this code or otherwise verify its correctness. If you introduce an error within this section of your program, the assembler will report the error when HLA assembles your code by calling MASM or Gas.

The #EMIT statement also writes text directly to the assembly output file. However, this statement does not simply copy the text from your source file to the output file; instead, this statement copies the value of a string (constant) expression to the output file. The syntax for this statement is as follows:

```
#emit( string_expression );
```

This statement evaluates the expression and verifies that it’s a string expression. Then it copies the string data to the output file. Like the #ASM/#ENDASM statement, the #EMIT statement does not check the syntax of the MASM statement it writes to the assembly file. If there is a syntax error, MASM or Gas will catch it later on when HLA assembles the output file.

When HLA compiles your programs into assembly language, it does not use the same symbols in the assembly language output file that you use in the HLA source files. There are several technical reasons for this, but the bottom line is this: you cannot easily reference your HLA identifiers in your in-line assembly code. The only exception to this rule are external identifiers. HLA external identifiers use the same name in the assembly file as in the HLA source file. Therefore, you can refer to external objects within your in-line assembly sequences or in the strings you output via #EMIT.

One advantage of the #EMIT statement is that it lets you construct MASM or Gas statements under (compile-time) program control. You can write an HLA compile-time program that generates a sequence of strings and emits them to the assembly file via the #EMIT statement. The compile-time program has access to the HLA symbol table; this means that you can extract the identifiers that HLA emits to the assembly file and use these directly, even if they aren’t external objects.

The @StaticName compile-time function returns the name that HLA uses to refer to most static objects in your program. The following program demonstrates a simple use of this compile-time function to obtain the assembly name of an HLA procedure:

```
program emitDemo;
#include( "stdlib.hhf" )

  procedure myProc;
  begin myProc;

    stdout.put( "Inside MyProc" nl );

  end myProc;

begin emitDemo;

  ?stmt:string := "call " + @StaticName( myProc );
```



```

    #emit( stmt );

end emitDemo;

```

Program 12.1 Using the @StaticName Function

This example creates a string value (*stmt*) that contains something like “call ?741_myProc” and emits this assembly instruction directly to the source file (“?741_myProc” is typical of the type of name mangling that HLA does to static names it writes to the output file). If you compile and run this program, it should display “Inside MyProc” and then quit. If you look at the assembly file that HLA emits, you will see that it has given the *myProc* procedure the same name it appends to the CALL instruction¹.

The @StaticName function is only valid for static symbols. This includes STATIC, READONLY, and STORAGE variables, procedures, and iterators. It does not include VAR objects, constants, macros, class iterators, or methods.

You can access VAR variables by using the [EBP+offset] addressing mode, specifying the offset of the desired local variable. You can use the @offset compile-time function to obtain the offset of a VAR object or a parameter. The following program demonstrates how to do this:

```

program offsetDemo;
#include( "stdlib.hhf" )

var
    i:int32;

begin offsetDemo;

    mov( -255, i );
    ?stmt := "mov eax, [ebp+( " + string( @offset( i ) ) + " )]";
    #print( "Emitting '", stmt, "'" );
    #emit( stmt );
    stdout.put( "eax = ", (type int32 eax), nl );

end offsetDemo;

```

Program 12.2 Using the @Offset Compile-Time Function

This example emits the statement “mov eax, [ebp+(-8)]” to the assembly language source file. It turns out that -8 is the offset of the *i* variable in the offsetDemo program’s activation record.

Of course, the examples of #EMIT up to this point have been somewhat ridiculous since you can achieve the same results by using HLA statements. One very useful purpose for the #emit statement, however, is to create some instructions that HLA does not support. For example, as of this writing HLA does not support the LES instruction because you can’t really use it under most 32-bit operating systems. However, if

1. HLA may assign a different name than “?741_myProc” when you compile the program. The exact symbol HLA chooses varies from version to version of the assembler (it depends on the number of symbols defined prior to the definition of *myProc*). In this example, there were 741 static symbols defined in the HLA Standard Library before the definition of *myProc*.

you found a need for this instruction, you could easily write a macro to emit this instruction and appropriate operands to the assembly source file. Using the #EMIT statement gives you the ability to reference HLA objects, something you cannot do with the #ASM..#ENDASM sequence.

12.2.2 Linking MASM/Gas-Assembled Modules with HLA Modules

Although you can do some interesting things with HLA's in-line assembly statements, you'll probably never use them. Further, future versions of HLA may not even support these statements, so you should avoid them as much as possible even if you see a need for them. Of course, HLA does most of the stuff you'd want to do with the #ASM/#ENDASM and #EMIT statements anyway, so there is very little reason to use them at all. If you're going to combine MASM/Gas (or other assembler) code and HLA code together in a program, most of the time this will occur because you've got a module or library routine written in some other assembly language and you would like to take advantage of that code in your HLA programs. Rather than convert the other assembler's code to HLA, the easy solution is to simply assemble that other code to an object file and link it with your HLA programs.

Once you've compiled or assembled a source file to an object file, the routines in that module are callable from almost any machine code that can handle the routines' calling sequences. If you have an object file that contains a SQRT function, for example, it doesn't matter whether you compiled that function with HLA, MASM, TASM, NASM, Gas, or even a high level language; if it's object code and it exports the proper symbols, you can call it from your HLA program.

Compiling a module in MASM or Gas and linking that with your HLA program is little different than linking other HLA modules with your main HLA program. In the assembly source file you will have to export some symbols (using the PUBLIC directive in MASM or the .GLOBAL directive in Gas) and in your HLA program you've got to tell HLA that those symbols appear in a separate module (using the EXTERNAL option).

Since the two modules are written in assembly language, there is very little language imposed structure on the calling sequence and parameter passing mechanisms. If you're calling a function written in MASM or Gas from your HLA program, then all you've got to do is to make sure that your HLA program passes parameters in the same locations where the MASM/Gas function is expecting them.

About the only issue you've got to deal with is the case of identifiers in the two programs. By default, MASM and Gas are case insensitive. HLA, on the other hand, enforces case neutrality (which, essentially, means that it is case sensitive). If you're using MASM, there is a MASM command line option ("/Cp") that tells MASM to preserve case in all public symbols. It's a real good idea to use this option when assembling modules you're going to link with HLA so that MASM doesn't mess with the case of your identifiers during assembly.

Of course, since MASM and Gas process symbols in a case sensitive manner, it's possible to create two separate identifiers that are the same except for alphabetic case. HLA enforces case neutrality so it won't let you (directly) create two different identifiers that differ only in case. In general, this is such a bad programming practice that one would hope you never encounter it (and God forbid you actually do this yourself). However, if you inherit some MASM or Gas code written by a C hacker, it's quite possible the code uses this technique. The way around this problem is to use two separate identifiers in your HLA program and use the extended form of the EXTERNAL directive to provide the external names. For example, suppose that in MASM you have the following declarations:

```

public  AVariable
public  avariable
.
.
.
.data
AVariable dword  ?
avariable byte   ?

```

If you assemble this code with the “/Cp” or “/Cx” (total case sensitivity) command line options, MASM will emit these two external symbols for use by other modules. Of course, were you to attempt to define variables by these two names in an HLA program, HLA would complain about a duplicate symbol definition. However, you can connect two different HLA variables to these two identifiers using code like the following:

```
static
  AVariable: dword; external( "AVariable" );
  AnotherVar: byte; external( "avariabLe" );
```

HLA does not check the strings you supply as parameters to the EXTERNAL clause. Therefore, you can supply two names that are the same except for case and HLA will not complain. Note that when HLA calls MASM to assemble its output file, HLA specifies the “/Cp” option that tells MASM to preserve case in public and global symbols. Of course, you would use this same technique in Gas if the Gas programmer has exported two symbols that are identical except for case.

The following program demonstrates how to call a MASM subroutine from an HLA main program:

```
// To compile this module and the attendant MASM file, use the following
// command line:
//
//      ml -c masmupper.masm
//      hla masmdemol.hla masmupper.obj
//
// Sorry about no make file for this code, but these two files are in
// the HLA Vol4/Ch12 subdirectory that has its own makefile for building
// all the source files in the directory and I wanted to avoid confusion.

program MasmDemol;
#include( "stdlib.hhf" )

    // The following external declaration defines a function that
    // is written in MASM to convert the character in AL from
    // lower case to upper case.

    procedure masmUpperCase( c:char in al ); external( "masmUpperCase" );

static
  s: string := "Hello World!";

begin MasmDemol;

  stdout.put( "String converted to uppercase: \" );
  mov( s, edi );
  while( mov( [edi], al ) <> #0 ) do

    masmUpperCase( al );
    stdout.putc( al );
    inc( edi );

  endwhile;
  stdout.put( "\"" nl );

end MasmDemol;
```

Program 12.3 Main HLA Program to Link with a MASM Program

```

; MASM source file to accompany the MasmDemol.HLA source
; file. This code compiles to an object module that
; gets linked with an HLA main program. The function
; below converts the character in AL to upper case if it
; is a lower case character.

        .586
        .model flat, pascal

        .code
        public  masmUpperCase
masmUpperCase  proc  near32
        .if al >= 'a' && al <= 'z'
            and al, 5fh
        .endif
        ret
masmUpperCase  endp
        end

```

Program 12.4 Calling a MASM Procedure from an HLA Program: MASM Module

It is also possible to call an HLA procedure from a MASM or Gas program (this should be obvious since HLA compiles its source code to an assembly source file and that assembly source file can call HLA procedures such as those found in the HLA Standard Library). There are a few restrictions when calling HLA code from some other language. First of all, you can't easily use HLA's exception handling facilities in the modules you call from other languages (including MASM or Gas). The HLA main program initializes the exception handling system; this initialization is probably not done by your non-HLA assembly programs. Further, the HLA main program exports a couple of important symbols needed by the exception handling subsystem; again, it's unlikely your non-HLA main assembly program provides these public symbols. In the volume on Advanced Procedures this text will discuss how to deal with HLA's Exception Handling subsystem. However, that topic is a little too advanced for this chapter. Until you get to the point you can write code in MASM or Gas to properly set up the HLA exception handling system, you should not execute any code that uses the TRY..ENDTRY, RAISE, or any other exception handling statements.

Warning: a large percentage of the HLA Standard Library routines include exception handling statements or call other routines that use exception handling statements. Unless you've set up the HLA exception handling subsystem properly, you should not call any HLA Standard Library routines from non-HLA programs.

Other than the issue of exception handling, calling HLA procedures from standard assembly code is really easy. All you've got to do is put an EXTERNAL prototype in the HLA code to make the symbol you wish to access public and then include an EXTERN (or EXTERNDEF) statement in the MASM/Gas source file to provide the linkage. Then just compile the two source files and link them together.

About the only issue you need concern yourself with when calling HLA procedures from assembly is the parameter passing mechanism. Of course, if you pass all your parameters in registers (the best place), then communication between the two languages is trivial. Just load the registers with the appropriate param-

eters in your MASM/Gas code and call the HLA procedure. Inside the HLA procedure, the parameter values will be sitting in the appropriate registers (sort of the converse of what happened in Program 12.4).

If you decide to pass parameters on the stack, note that HLA normally uses the PASCAL language calling model. Therefore, you push parameters on the stack in the order they appear in a parameter list (from left to right) and it is the called procedure's responsibility to remove the parameters from the stack. Note that you can specify the PASCAL calling convention for use with MASM's INVOKE statement using the ".model" directive, e.g.,

```
.586
.model flat, pascal
.
.
.
```

Of course, if you manually push the parameters on the stack yourself, then the specific language model doesn't really matter. Gas users, of course, don't have the INVOKE statement, so they have to manually push the parameters themselves anyway.

This section is not going to attempt to go into gory details about MASM or Gas syntax. There is an appendix in this text that contrasts the HLA language with MASM (and Gas when using the ".intel_syntax" directive); you should be able to get a rough idea of MASM/Gas syntax from that appendix if you're completely unfamiliar with these assemblers. Another alternative is to read a copy of the DOS/16-bit edition of this text that uses the MASM assembler. That text describes MASM syntax in much greater detail, albeit from a 16-bit perspective. Finally, this section isn't going to go into any further detail because, quite frankly, the need to call MASM or Gas code from HLA (or vice versa) just isn't that great. After all, most of the stuff you can do with MASM and Gas can be done directly in HLA so there really is little need to spend much more time on this subject. Better to move on to more important questions, like how do you call HLA routines from C or Pascal...

12.3 Programming in Delphi/Kylix and HLA

Delphi is a marvelous language for writing Win32 GUI-based applications. Kylix is the companion product that runs under Linux. Their support for Rapid Application Design (RAD) and visual programming is superior to almost every other Windows or Linux programming approach available. However, being Pascal-based, there are some things that just cannot be done in Delphi/Kylix and many things that cannot be done as efficiently in Delphi/Kylix as in assembly language. Fortunately, Delphi/Kylix lets you call assembly language procedures and functions so you can overcome Delphi's limitations.

Delphi provides two ways to use assembly language in the Pascal code: via a built-in assembler (BASM) or by linking in separately compiled assembly language modules. The built-in "Borland Assembler" (BASM) is a very weak Intel-syntax assembler. It is suitable for injecting a few instructions into your Pascal source code or perhaps writing a very short assembly language function or procedure. It is not suitable for serious assembly language programming. If you know Intel syntax and you only need to execute a few machine instructions, then BASM is perfect. However, since this is a text on assembly language programming, the assumption here is that you want to write some serious assembly code to link with your Pascal/Delphi code. To do that, you will need to write the assembly code and compile it with a different assembler (e.g., HLA) and link the code into your Delphi application. That is the approach this section will concentrate on. For more information about BASM, check out the Delphi documentation.

Before we get started discussing how to write HLA modules for your Delphi programs, you must understand two very important facts:

HLA's exception handling facilities are not directly compatible with Delphi's. This means that you cannot use the TRY..ENDTRY and RAISE statements in the HLA code you intend to link to a Delphi program. This also means that you cannot call library functions

that contain such statements. Since the HLA Standard Library modules use exception handling statements all over the place, this effectively prevents you from calling HLA Standard Library routines from the code you intend to link with Delphi².

Although you can write console applications with Delphi, 99% of Delphi applications are GUI applications. You cannot call console-related functions (e.g., `stdin.xxxx` or `stdout.xxxx`) from a GUI application. Even if HLA's console and standard input/output routines didn't use exception handling, you wouldn't be able to call them from a standard Delphi application.

Given the rich set of language features that Delphi supports, it should come as no surprise that the interface between Delphi's Object Pascal language and assembly language is somewhat complex. Fortunately there are two facts that reduce this problem. First, HLA uses many of the same calling conventions as Pascal; so much of the complexity is hidden from sight by HLA. Second, the other complex stuff you won't use very often, so you may not have to bother with it.

Note: the following sections assume you are already familiar with Delphi programming. They make no attempt to explain Delphi syntax or features other than as needed to explain the Delphi assembly language interface. If you're not familiar with Delphi, you will probably want to skip this section.

12.3.1 Linking HLA Modules With Delphi Programs

The basic unit of interface between a Delphi program and assembly code is the procedure or function. That is, to combine code between the two languages you will write procedures in HLA (that correspond to procedures or functions in Delphi) and call these procedures from the Delphi program. Of course, there are a few mechanical details you've got to worry about, this section will cover those.

To begin with, when writing HLA code to link with a Delphi program you've got to place your HLA code in an HLA UNIT. An HLA PROGRAM module contains start up code and other information that the operating system uses to determine where to begin program execution when it loads an executable file from disk. However, the Delphi program also supplies this information and specifying two starting addresses confuses the linker, therefore, you must place all your HLA code in a UNIT rather than a PROGRAM module.

Within the HLA UNIT you must create EXTERNAL procedure prototypes for each procedure you wish to call from Delphi. If you prefer, you can put these prototype declarations in a header file and #INCLUDE them in the HLA code, but since you'll probably only reference these declarations from this single file, it's okay to put the EXTERNAL prototype declarations directly in the HLA UNIT module. These EXTERNAL prototype declarations tell HLA that the associated functions will be public so that Delphi can access their names during the link process. Here's a typical example:

```
unit LinkWithDelphi;

    procedure prototype; external;

    procedure prototype;
    begin prototype;

        << Code to implement prototype's functionality >>

    end prototype;

end LinkWithDelphi;
```

After creating the module above, you'd compile it using HLA's "-c" (compile to object only) command line option. This will produce an object (".o") file.

2. Note that the HLA Standard Library source code is available; feel free to modify the routines you want to use and remove any exception handling statements contained therein.

Once you've created the HLA code and compiled it to an object file, the next step is to tell Delphi that it needs to call the HLA/assembly code. There are two steps needed to achieve this: You've got to inform Delphi that a procedure (or function) is written in assembly language (rather than Pascal) and you've got to tell Delphi to link in the object file you've created when compiling the Delphi code.

The second step above, telling Delphi to include the HLA object module, is the easiest task to achieve. All you've got to do is insert a compiler directive of the form “`{ $L objectFileName.obj }`” in the Delphi program before declaring and calling your object module. A good place to put this is after the **implementation** reserved word in the module that calls your assembly procedure. The code examples a little later in this section will demonstrate this.

The next step is to tell Delphi that you're supplying an external procedure or function. This is done using the Delphi EXTERNAL directive on a procedure or function prototype. For example, a typical external declaration for the *prototype* procedure appearing earlier is

```
procedure prototype; external; // This may look like HLA code, but it's
                               // really Delphi code!
```

As you can see here, Delphi's syntax for declaring external procedures is nearly identical to HLA's (in fact, in this particular example the syntax is identical). This is not an accident, much of HLA's syntax was borrowed directly from Pascal.

The next step is to call the assembly procedure from the Delphi code. This is easily accomplished using standard Pascal procedure calling syntax. The following two listings provide a complete, working, example of an HLA procedure that a Delphi program can call. This program doesn't accomplish very much other than to demonstrate how to link in an assembly procedure. The Delphi program contains a form with a single button on it. Pushing the button calls the HLA procedure, whose body is empty and therefore returns immediately to the Delphi code without any visible indication that it was ever called. Nevertheless, this code does provide all the syntactical elements necessary to create and call an assembly language routine from a Delphi program.

```
unit LinkWithKylix;

    procedure CalledFromKylix; external;

    procedure CalledFromKylix;
    begin CalledFromKylix;
    end CalledFromKylix;

end LinkWithKylix;
```

Program 12.5 CalledFromKylix.hla Module Containing the Assembly Code

```
unit KylixEx1;

interface

uses
```

```

SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs,
QStdCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.xfm}
{$L CalledFromKylix.o }

procedure CalledFromKylix; external;

procedure TForm1.Button1Click(Sender: TObject);
begin
    CalledFromKylix();
end;

end.

```

Program 12.6 DelphiEx1– Delphi Source Code that Calls an Assembly Procedure

The full Delphi and HLA source code for the programs appearing in Program 12.5 and Program 12.6 accompanies the HLA software distribution in the appropriate subdirectory for this chapter in the Example code module. If you've got a copy of Delphi 5 or later, you might want to load this module and try compiling it. To compile the HLA code for this example, you would use the following commands from the command prompt:

```
hla -c CalledFromDelphi.hla
```

After producing the CalledFromDelphi object module with the two commands above, you'd enter the Delphi Integrated Development Environment and tell it to compile the DelphiEx1 code (i.e., you'd load the DelphiEx1Project file into Delphi and compile the code). This process automatically links in the HLA code and when you run the program you can call the assembly code by simply pressing the single button on the Delphi form.

12.3.2 Register Preservation

Delphi code expects all procedures to preserve the EBX, ESI, EDI, and EBP registers. Routines written in assembly language may freely modify the contents of EAX, ECX, and EDX without preserving their values. The HLA code will have to modify the ESP register to remove the activation record (and, possibly,

some parameters). Of course, HLA procedures (unless you specify the `@NOFRAME` option) automatically preserve and set up EBP for you, so you don't have to worry about preserving this register's value; of course, you will not usually manipulate EBP's value since it points at your procedure's parameters and local variables.

Although you can modify EAX, ECX, and EDX to your heart's content and not have to worry about preserving their values, don't get the idea that these registers are available for your procedure's exclusive use. In particular, Delphi may pass parameters into a procedure within these registers and you may need to return function results in some of these registers. Details on the further use of these registers appears in later sections of this chapter.

Whenever Delphi calls a procedure, that procedure can assume that the direction flag is clear. On return, all procedures must ensure that the direction flag is still clear. So if you manipulate the direction flag in your assembly code (or call a routine that might set the direction flag), be sure to clear the direction flag before returning to the Delphi code.

If you use any MMX instructions within your assembly code, be sure to execute the EMMS instruction before returning. Delphi code assumes that it can manipulate the floating point stack without running into problems.

Although the Delphi documentation doesn't explicitly state this, experiments with Delphi code seem to suggest that you don't have to preserve the FPU (or MMX) registers across a procedure call other than to ensure that you're in FPU mode (versus MMX mode) upon return to Delphi.

12.3.3 Function Results

Delphi generally expects functions to return their results in a register. For ordinal return results, a function should return a byte value in AL, a word value in AX, or a double word value in EAX. Functions return pointer values in EAX. Functions return real values in ST0 on the FPU stack. The code example in this section demonstrates each of these parameter return locations.

For other return types (e.g., arrays, sets, records, etc.), Delphi generally passes an extra VAR parameter containing the address of the location where the function should store the return result. We will not consider such return results in this text, see the Delphi documentation for more details.

The following Delphi/HLA program demonstrates how to return different types of scalar (ordinal and real) parameters to a Delphi program from an assembly language function. The HLA functions return boolean (one byte) results, word results, double word results, a pointer (PChar) result, and a floating point result when you press an appropriate button on the form. See the DelphiEx2 example code in the HLA/Art of Assembly examples code for the full project. Note that the following code doesn't really do anything useful other than demonstrate how to return Function results in EAX and ST0.

```

unit KylixEx2;

interface

uses
  SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs,
  QStdCtrls;

type
  TForm1 = class(TForm)
    BooleanBtn: TButton;
    WordBtn: TButton;
    DWordBtn: TButton;
    PointerBtn: TButton;
    RealBtn: TButton;
  end;

```

```

    BooleanLbl: TLabel;
    WordLbl: TLabel;
    DWordLbl: TLabel;
    PointerLbl: TLabel;
    RealLbl: TLabel;
    procedure BooleanBtnClick(Sender: TObject);
    procedure WordBtnClick(Sender: TObject);
    procedure DWordBtnClick(Sender: TObject);
    procedure PointerBtnClick(Sender: TObject);
    procedure RealBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.xfm}
{$L ReturnValues.o }

function ReturnBoolean:boolean; external;
function ReturnWord:word; external;
function ReturnDWord:dword; external;
function ReturnPtr:pchar; external;
function ReturnReal:extended; external;

procedure TForm1.BooleanBtnClick(Sender: TObject);
begin
    if( ReturnBoolean() ) then
        BooleanLbl.caption := 'true'
    else
        BooleanLbl.caption := 'false';
end;

procedure TForm1.WordBtnClick(Sender: TObject);
var
    w:word;
    s:string;
begin
    w := ReturnWord();
    s := format( '%x', [w] );
    WordLbl.Caption := s;
end;

procedure TForm1.DWordBtnClick(Sender: TObject);
var
    dw:dword;
    s:string;
begin

```

```

    dw := ReturnDWord();
    s := format( '$%x', [dw] );
    DWordLbl.Caption := s;

end;

procedure TForm1.PointerBtnClick(Sender: TObject);
begin

    PointerLbl.caption := ReturnPtr();

end;

procedure TForm1.RealBtnClick(Sender: TObject);
var
    r:extended;
    s:string;

begin

    r := ReturnReal();
    s := format( '%10e', [r] );
    RealLbl.caption := s;

end;

end.

```

Program 12.7 KylixEx2: Pascal Code for Assembly Return Results Example

```

// ReturnUnit-
//
// Provides the ReturnXXXX functions for the KylixEx2 program.

unit ReturnUnit;

// Tell HLA that the ReturnXXXXXX symbols are public:

procedure ReturnBoolean; external;
procedure ReturnWord; external;
procedure ReturnDWord; external;
procedure ReturnReal; external;
procedure ReturnPtr; external;

// Demonstration of a function that returns a byte value in AL.
// This function simply returns a boolean result that alternates
// between true and false on each call.

procedure ReturnBoolean; @nodisplay; @noalignstack; @noframe;
static b:boolean:=false;

```

```

begin ReturnBoolean;

    xor( 1, b );    // Invert boolean status
    and( 1, b );   // Force to zero (false) or one (true).
    mov( b, al );  // Function return result comes back in AL.
    ret();

end ReturnBoolean;

procedure ReturnWord; @nodisplay; @noalignstack; @noframe;
static w:int16 := 1234;
begin ReturnWord;

    // Increment the static value by one on each
    // call and return the new result as the function
    // return value.

    inc( w );
    mov( w, ax );
    ret();

end ReturnWord;

// Same code as ReturnWord except this one returns a 32-bit value
// in EAX rather than a 16-bit value in AX.

procedure ReturnDWord; @nodisplay; @noalignstack; @noframe;
static
    d:int32 := -7;
begin ReturnDWord;

    inc( d );
    mov( d, eax );
    ret();

end ReturnDWord;

procedure ReturnPtr; @nodisplay; @noalignstack; @noframe;
static
    stringData: byte; @nostorage;
    byte "Pchar object", 0;

begin ReturnPtr;

    lea( eax, stringData );
    ret();

end ReturnPtr;

procedure ReturnReal; @nodisplay; @noalignstack; @noframe;
static
    realData: real80 := 1.234567890;

begin ReturnReal;

    fld( realData );
    ret();

end ReturnReal;

```

```
end ReturnUnit;
```

Program 12.8 ReturnReal: Demonstrates Returning a Real Value in ST0

The second thing to note is the `#code`, `#static`, etc., directives at the beginning of each file to change the segment name declarations. You'll learn the reason for these segment renaming directives a little later in this chapter.

12.3.4 Calling Conventions

Delphi supports five different calling mechanisms for procedures and functions: **register**, **pascal**, **cdecl**, and **safecall**. The **register** and **pascal** calling methods are very similar except that the **pascal** parameter passing scheme always passes all parameters on the stack while the **register** calling mechanism passes the first three parameters in CPU registers. We'll return to these two mechanisms shortly since they are the primary mechanisms we'll use. The **cdecl** calling convention uses the C/C++ programming language calling convention. We'll study this scheme more in the section on interfacing C/C++ with HLA. There is no need to use this scheme when calling HLA procedures from Delphi. If you must use this scheme, then see the section on the C/C++ languages for details. **Safecall** is another specialized calling convention that we will not use. See, we've already reduced the complexity from five mechanisms to two! Seriously, though, when calling assembly language routines from Delphi code that you're writing, you only need to use the **pascal** and **register** conventions.

The calling convention options specify how Delphi passes parameters between procedures and functions as well as who is responsible for cleaning up the parameters when a function or procedure returns to its caller. The **pascal** calling convention passes all parameters on the stack and makes it the procedure or function's responsibility to remove those parameters from the stack. The pascal calling convention mandates that the caller push parameters in the order the compiler encounters them in the parameter list (i.e., left to right). This is exactly the calling convention that HLA uses (assuming you don't use the "IN register" parameter option). Here's an example of a Delphi external procedure declaration that uses the **pascal** calling convention:

```
procedure UsesPascal( parm1:integer; parm2:integer; parm3:integer );
```

The following program provides a quick example of a Delphi program that calls an HLA procedure (function) using the **pascal** calling convention.

```
unit KylixEx3;

interface

uses
  SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs,
  QStdCtrls;

type
  TForm1 = class(TForm)
    UsesPascalBtn: TButton;
    UsesPascalLbl: TLabel;
  end;
end;
```

```

    procedure UsesPascalBtnClick(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

{$R *.xfm}
{$L UsesPascal.o}

function UsesPascal
(
    parm1:integer;
    parm2:integer;
    parm3:integer
):integer; pascal; external;

procedure TForm1.UsesPascalBtnClick(Sender: TObject);
var
    i:    integer;
    strVal: string;
begin
    i := UsesPascal( 5, 6, 7 );
    str( i, strVal );
    UsesPascalLbl.caption := 'Uses Pascal = ' + strVal;

end;

end.

```

Program 12.9 KylixEx3 – Sample Program that Demonstrates the **pascal** Calling Convention

```

// UsesPascalUnit-
//
// Provides the UsesPascal function for the KylixEx3 program.

unit UsesPascalUnit;

// Tell HLA that UsesPascal is a public symbol:

procedure UsesPascal( parm1:int32; parm2:int32; parm3:int32 ); external;

// Demonstration of a function that uses the PASCAL calling convention.
// This function simply computes parm1+parm2-parm3 and returns the
// result in EAX. Note that this function does not have the
// "NOFRAME" option because it needs to build the activation record
// (stack frame) in order to access the parameters. Furthermore, this

```

```

// code must clean up the parameters upon return (another chore handled
// automatically by HLA if the "NOFRAME" option is not present).

procedure UsesPascal( parm1:int32; parm2:int32; parm3:int32 );
    @nodisplay;
    @noalignstack;

begin UsesPascal;

    mov( parm1, eax );
    add( parm2, eax );
    sub( parm3, eax );

end UsesPascal;

end UsesPascalUnit;

```

Program 12.10 UsesPascal – HLA Function the Previous Kylix Code Will Call

To compile the HLA code, you would use the following command from the shell:

```
hla -c UsesPascal.hla
```

Once you produce the .o file with the above two commands, you can get into Delphi and compile the Pascal code.

The **register** calling convention also processes parameters from left to right and requires the procedure/function to clean up the parameters upon return; the difference is that procedures and functions that use the **register** calling convention will pass their first three (ordinal) parameters in the EAX, EDX, and ECX registers (in that order) rather than on the stack. You can use HLA's "IN *register*" syntax to specify that you want the first three parameters passed in this registers, e.g.,

```

procedure UsesRegisters
(
    parm1:int32 in EAX;
    parm2:int32 in EDX;
    parm3:int32 in ECX
);

```

If your procedure had four or more parameters, you would not specify registers as their locations. Instead, you'd access those parameters on the stack. Since most procedures have three or fewer parameters, the **register** calling convention will typically pass all of a procedure's parameters in a register.

Although you can use the **register** keyword just like **pascal** to force the use of the **register** calling convention, the register calling convention is the default mechanism in Delphi. Therefore, a Delphi declaration like the following will automatically use the **register** calling convention:

```

procedure UsesRegisters
(
    parm1:integer;
    parm2:integer;
    parm3:integer
); external;

```

The following program is a modification of the previous program in this section that uses the **register** calling convention rather than the **pascal** calling convention.

```
unit KylixEx4;

interface

uses
  SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs,
  QStdCtrls;

type
  TForm1 = class(TForm)
    RegisterBtn: TButton;
    UsesRegisterLabel: TLabel;
    procedure RegisterBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.xfm}
{$L UsesRegister.o}

function UsesRegister
(
  parm1:integer;
  parm2:integer;
  parm3:integer;
  parm4:integer
):integer; external;

procedure TForm1.RegisterBtnClick(Sender: TObject);
var
  i:      integer;
  strVal: string;
begin
  i := UsesRegister( 5, 6, 7, 3 );
  str( i, strVal );
  UsesRegisterLabel.caption := 'Uses Register = ' + strVal;

end;

end.
```

Program 12.11 KylixEx4 – Using the **register** Calling Convention

```

// UsesRegisterUnit-
//
// Provides the UsesRegister function for the DelphiEx4 program.

unit UsesRegisterUnit;

// Tell HLA that UsesRegister is a public symbol:

procedure UsesRegister
(
    parm1:int32 in eax;
    parm2:int32 in edx;
    parm3:int32 in ecx;
    parm4:int32
); external;

// Demonstration of a function that uses the REGISTER calling convention.
// This function simply computes (parm1+parm2-parm3)*parm4 and returns the
// result in EAX. Note that this function does not have the
// "NOFRAME" option because it needs to build the activation record
// (stack frame) in order to access the fourth parameter. Furthermore, this
// code must clean up the fourth parameter upon return (another chore handled
// automatically by HLA if the "NOFRAME" option is not present).

procedure UsesRegister
(
    parm1:int32 in eax;
    parm2:int32 in edx;
    parm3:int32 in ecx;
    parm4:int32
); @nodisplay; @noalignstack;

begin UsesRegister;

    mov( parm1, eax );
    add( parm2, eax );
    sub( parm3, eax );
    intmul( parm4, eax );

end UsesRegister;

end UsesRegisterUnit;

```

Program 12.12 HLA Code to support the KylixEx4 Program

To compile the HLA code, you would use the following shell command:

```
hla -c UsesRegister.hla
```

Once you produce the .o file with the above command, you can get into Delphi and compile the Pascal code.

12.3.5 Pass by Value, Reference, CONST, and OUT in Delphi

A Delphi program can pass parameters to a procedure or function using one of four different mechanisms: pass by value, pass by reference, CONST parameters, and OUT parameters. The examples up to this point in this chapter have all used Delphi's (and HLA's) default pass by value mechanism. In this section we'll look at the other parameter passing mechanisms.

HLA and Delphi also share a (mostly) common syntax for pass by reference parameters. The following two lines provide an external declaration in Delphi and the corresponding external (public) declaration in HLA for a pass by reference parameter using the **pascal** calling convention:

```
procedure HasRefParm( var refparm: integer ); pascal; external; // Delphi
procedure HasRefParm( var refparm: int32 ); external; // HLA
```

Like HLA, Delphi will pass the 32-bit address of whatever actual parameter you specify when calling the *HasRefParm* procedure. Don't forget, inside the HLA code, that you must dereference this pointer to access the actual parameter data. See the chapter on Intermediate Procedures for more details (see "Pass by Reference" on page 817).

The CONST and OUT parameter passing mechanisms are virtually identical to pass by reference. Like pass by reference these two schemes pass a 32-bit address of their actual parameter. The difference is that the called procedure is not supposed to write to CONST objects since they're, presumably, constant. Conversely, the called procedure is supposed to write to an OUT parameter (and not assume that it contains any initial value of consequence) since the whole purpose of an OUT parameter is to return data from a procedure or function. Other than the fact that the Delphi compiler will check procedures and functions (written in Delphi) for compliance with these rules, there is no difference between CONST, OUT, and reference parameters. Delphi passes all such parameters by reference to the procedure or function. Note that in HLA you would declare all CONST and OUT parameters as pass by reference parameters. HLA does not enforce the readonly attribute of the CONST object nor does it check for an attempt to access an uninitialized OUT parameter; those checks are the responsibility of the assembly language programmer.

As you learned in the previous section, by default Delphi uses the **register** calling convention. If you pass one of the first three parameters by reference to a procedure or function, Delphi will pass the address of that parameter in the EAX, EDX, or ECX register. This is very convenient as you can immediately apply the register indirect addressing mode without first loading the parameter into a 32-bit register.

Like HLA, Delphi lets you pass untyped parameters by reference (or by CONST or OUT). The syntax to achieve this in Delphi is the following:

```
procedure UntypedRefParm( var parm1; const parm2; out parm3 ); external;
```

Note that you do not supply a type specification for these parameters. Delphi will compute the 32-bit address of these objects and pass them on to the *UntypedRefParm* procedure without any further type checking. In HLA, you can use the VAR keyword as the data type to specify that you want an untyped reference parameter. Here's the corresponding prototype for the *UntypedRefParm* procedure in HLA:

```
procedure UntypedRefParm( var parm1:var; var parm2:var; var parm3:var );
external;
```

As noted above, you use the VAR keyword (pass by reference) when passing CONST and OUT parameters. Inside the HLA procedure it's your responsibility to use these pointers in a manner that is reasonable given the expectations of the Delphi code.

12.3.6 Scalar Data Type Correspondence Between Delphi and HLA

When passing parameters between Delphi and HLA procedures and functions, it's very important that the calling code and the called code agree on the basic data types for the parameters. In this section we will draw a correspondence between the Delphi scalar data types and the HLA (v1.x) data types³.

Assembly language supports any possible data format, so HLA's data type capabilities will always be a superset of Delphi's. Therefore, there may be some objects you can create in HLA that have no counterpart in Delphi, but the reverse is not true. Since the assembly functions and procedures you write are generally manipulating data that Delphi provides, you don't have to worry too much about not being able to process some data passed to an HLA procedure by Delphi⁴.

Delphi provides a wide range of different integer data types. The following table lists the Delphi types and the HLA equivalents:

Table 1: Delphi and HLA Integer Types

Delphi	HLA Equivalent	Range	
		Minimum	Maximum
integer	int32 ^a	-2147483648	2147483647
cardinal	uns32 ^b	0	4294967295
shortint	int8	-128	127
smallint	int16	-32768	32767
longint	int32	-2147483648	2147483647
int64	qword	-2^{63}	$(2^{63}-1)$
byte	uns8	0	255
word	uns16	0	65535
longword	uns32	0	4294967295
subrange types	Depends on range	minimum range value	maximum range value

a. Int32 is the implementation of integer in Delphi. Though this may change in later releases.

b. Uns32 is the implementation of cardinal in Delphi. Though this may change in later releases.

In addition to the integer values, Delphi supports several non-integer ordinal types. The following table provides their HLA equivalents:

3. Scalar data types are the ordinal, pointer, and real types. It does not include strings or other composite data types.

4. Delphi string objects are an exception. For reasons that have nothing to do with data representation, you should not manipulate string parameters passed in from Delphi to an HLA routine. This section will explain the problems more fully a little later.

Table 2: Non-integer Ordinal Types in Delphi and HLA

Delphi	HLA	Range	
		Minimum	Maximum
char	char	#0	#255
widechar	word	chr(0)	chr(65535)
boolean	boolean	false (0)	true(1)
bytebool	byte	0(false)	255 (non-zero is true)
wordbool	word	0 (false)	65535 (non-zero is true)
longbool	dword	0 (false)	4294967295 (non-zero is true)
enumerated types	enum, byte, or word	0	Depends on number of items in the enumeration list. Usually the upper limit is 256 symbols

Like the integer types, Delphi supports a wide range of real numeric formats. The following table presents these types and their HLA equivalents.

Table 3: Real Types in Delphi and HLA

Delphi	HLA	Range	
		Minimum	Maximum
real	real64	5.0 E-324	1.7 E+308
single	real32	1.5 E-45	3.4 E+38
double	real64	5.0 E-324	1.7 E+308
extended	real80	3.6 E-4951	1.1 E+4932
comp	real80	$-2^{63}+1$	$2^{63}-1$
currency	real80	-922337203685477.5808	922337203685477.5807

The last scalar type of interest is the pointer type. Both HLA and Delphi use a 32-bit address to represent pointers, so these data types are completely equivalent in both languages.

12.3.7 Passing String Data Between Delphi and HLA Code

Delphi supports a couple of different string formats. The native string format is actually very similar to HLA's string format. A string object is a pointer that points at a zero terminated sequence of characters. In the four bytes preceding the first character of the string, Delphi stores the current dynamic length of the string (just like HLA). In the four bytes before the length, Delphi stores a reference count (unlike HLA, which stores a maximum length value in this location). Delphi uses the reference count to keep track of how many different pointers contain the address of this particular string object. Delphi will automatically free the storage associated with a string object when the reference count drops to zero (this is known as garbage collection).

The Delphi string format is just close enough to HLA's to tempt you to use some HLA string functions in the HLA Standard Library. This will fail for two reasons: (1) many of the HLA Standard Library string functions check the maximum length field, so they will not work properly when they access Delphi's reference count field; (2) HLA Standard Library string functions have a habit of raising string overflow (and other) exceptions if they detect a problem (such as exceeding the maximum string length value). Remember, the HLA exception handling facility is not directly compatible with Delphi's, so you should never call any HLA code that might raise an exception.

Of course, you can always grab the source code to some HLA Standard Library string function and strip out the code that raises exceptions and checks the maximum length field (this is usually the same code that raises exceptions). However, you could still run into problems if you attempt to manipulate some Delphi string. In general, it's okay to read the data from a string parameter that Delphi passes to your assembly code, but you should never change the value of such a string. To understand the problem, consider the following HLA code sequence:

```
static
  s:string := "Hello World";
  sref:string;
  scopy:string;
  .
  .
  .
  str.a_cpy( s, scopy ); // scopy has its own copy of "Hello World"

  mov( s, eax ); // After this sequence, s and sref point at
  mov( eax, sref ); // the same character string in memory.
```

After the code sequence above, any change you would make to the *scopy* string would affect only *scopy* because it has its own copy of the "Hello World" string. On the other hand, if you make any changes to the characters that *s* points at, you'll also be changing the string that *sref* points at because *sref* contains the same pointer value as *s*; in other words, *s* and *sref* are aliases of the same data. Although this aliasing process can lead to the creation of some killer defects in your code, there is a big advantage to using copy by reference rather than copy by value: copy by reference is much quicker since it only involves copying a single four-byte pointer. If you rarely change a string variable after you assign one string to that variable, copy by reference can be very efficient.

Of course, what happens if you use copy by reference to copy *s* to *sref* and then you want to modify the string that *sref* points at without changing the string that *s* points at? One way to do this is to make a copy of the string at the time you want to change *sref* and then modify the copy. This is known as *copy on write semantics*. In the average program, copy on write tends to produce faster running programs because the typical program tends to assign one string to another without modification more often than it assigns a string value and then modifies it later. Of course, the real problem is "how do you know whether multiple string variables are pointing at the same string in memory?" After all, if only one string variable is pointing at the string data, you don't have to make a copy of the data, you can manipulate the string data directly. The *ref-*

erence counter field that Delphi attaches to the string data solves this problem. Each time a Delphi program assigns one string variable to another, the Delphi code simply copies a pointer and then increments the reference counter. Similarly, if you assign a string address to some Delphi string variable and that variable was previously pointing at some other string data, Delphi decrements the reference counter field of that previous string value. When the reference count hits zero, Delphi automatically deallocates storage for the string (this is the garbage collection operation).

Note that Delphi strings don't need a maximum length field because Delphi dynamically allocates (standard) strings whenever you create a new string. Hence, string overflow doesn't occur and there is no need to check for string overflow (and, therefore, no need for the maximum length field). For literal string constants (which the compiler allocates statically, not dynamically on the heap), Delphi uses a reference count field of -1 so that the compiler will not attempt to deallocate the static object.

It wouldn't be that hard to take the HLA Standard Library strings module and modify it to use Delphi's dynamically allocated string format. There is, however, one problem with this approach: Borland has not published the internal string format for Delphi strings (the information appearing above is the result of sleuthing through memory with a debugger). They have probably withheld this information because they want the ability to change the internal representation of their string data type without breaking existing Delphi programs. So if you poke around in memory and modify Delphi string data (or allocate or deallocate these strings on your own), don't be surprised if your program malfunctions when a later version of Delphi appears (indeed, this information may already be obsolete).

Like HLA strings, a Delphi string is a pointer that happens to contain the address of the first character of a zero terminated string in memory. As long as you don't modify this pointer, you don't modify any of the characters in that string, and you don't attempt to access any bytes before the first character of the string or after the zero terminating byte, you can safely access the string data in your HLA programs. Just remember that you cannot use any Standard Library routines that check the maximum string length or raise any exceptions. If you need the length of a Delphi string that you pass as a parameter to an HLA procedure, it would be wise to use the Delphi *Length* function to compute the length and pass this value as an additional parameter to your procedure. This will keep your code working should Borland ever decide to change their internal string representation.

Delphi also supports a *ShortString* data type. This data type provides backwards compatibility with older versions of Borland's Turbo Pascal (Borland Object Pascal) product. *ShortString* objects are traditional length-prefixed strings (see "Character Strings" on page 419). A short string variable is a sequence of one to 256 bytes where the first byte contains the current dynamic string length (a value in the range 0..255) and the following *n* bytes hold the actual characters in the string (*n* being the value found in the first byte of the string data). If you need to manipulate the value of a string variable within an assembly language module, you should pass that parameter as a *ShortString* variable (assuming, of course, that you don't need to handle strings longer than 256 characters). For efficiency reasons, you should always pass *ShortString* variables by reference (or CONST or OUT) rather than by value. If you pass a short string by value, Delphi must copy all the characters allocated for that string (even if the current length is shorter) into the procedure's activation record. This can be very slow. If you pass a *ShortString* by reference, then Delphi will only need to pass a pointer to the string's data; this is very efficient.

Note that *ShortString* objects do not have a zero terminating byte following the string data. Therefore, your assembly code should use the length prefix byte to determine the end of the string, it should not search for a zero byte in the string.

If you need the maximum length of a *ShortString* object, you can use the Delphi *high* function to obtain this information and pass it to your HLA code as another parameter. Note that the *high* function is a compiler intrinsic much like HLA's @size function. Delphi simply replaces this "function" with the equivalent constant at compile-time; this isn't a true function you can call. This maximum size information is not available at run-time (unless you've used the Delphi *high* function) and you cannot compute this information within your HLA code.

12.3.8 Passing Record Data Between HLA and Delphi

Records in HLA are (mostly) compatible with Delphi records. Syntactically their declarations are very similar and if you've specified the correct Delphi compiler options you can easily translate a Delphi record to an HLA record. In this section we'll explore how to do this and learn about the incompatibilities that exist between HLA records and Delphi records.

For the most part, translating Delphi records to HLA is a no brainer. The two record declarations are so similar syntactically that conversion is trivial. The only time you really run into a problem in the conversion process is when you encounter case variant records in Delphi; fortunately, these don't occur very often and when they do, HLA's anonymous unions within a record come to the rescue.

Consider the following Pascal record type declaration:

```
type
  recType =
    record

      day: byte;
      month:byte;
      year:integer;
      dayOfWeek:byte;

    end;
```

The translation to an HLA record is, for the most part, very straight-forward. Just translate the field types accordingly and use the HLA record syntax (see "Records" on page 483) and you're in business. The translation is the following:

```
type
  recType:
    record

      day: byte;
      month: byte;
      year:int32;
      dayOfWeek:byte;

    endrecord;
```

There is one minor problem with this example: data alignment. By default Delphi aligns each field of a record on the size of that object and pads the entire record so its size is an even multiple of the largest (scalar) object in the record. This means that the Delphi declaration above is really equivalent to the following HLA declaration:

```
type
  recType:
    record

      day: byte;
      month: byte;
      padding:byte[2];      // Align year on a four-byte boundary.
      year:int32;
      dayOfWeek:byte;
      morePadding: byte[3]; // Make record an even multiple of four bytes.

    endrecord;
```

Of course, a better solution is to use HLA's ALIGN directive to automatically align the fields in the record:

```

type
  recType:
    record

      day: byte;
      month: byte;
      align( 4 );      // Align year on a four-byte boundary.
      year: int32;
      dayOfWeek: byte;
      align(4);      // Make record an even multiple of four bytes.

    endrecord;

```

Alignment of the fields is good insofar as access to the fields is faster if they are aligned appropriately. However, aligning records in this fashion does consume extra space (five bytes in the examples above) and that can be expensive if you have a large array of records whose fields need padding for alignment.

The alignment parameters for an HLA record should be the following:

Table 4: Alignment of Record Fields

Data Type	Alignment
Ordinal Types	Size of the type: 1, 2, or 4 bytes.
Real Types	2 for real48 and extended, 4 bytes for other real types
ShortString	1
Arrays	Same as the element size
Records	Same as the largest alignment of all the fields.
Sets	1 or two if the set has fewer than 8 or 16 elements, 4 otherwise
All other types	4

Another possibility is to tell Delphi not to align the fields in the record. There are two ways to do this: use the **packed** reserved word or use the {\$A-} compiler directive.

The packed keyword tells Delphi not to add padding to a specific record. For example, you could declare the original Delphi record as follows:

```

type
  recType =
    packed record

      day: byte;
      month: byte;
      year: integer;
      dayOfWeek: byte;

    end;

```


With the **packed** reserved word present, Delphi does not add any padding to the fields in the record. The corresponding HLA code would be the original record declaration above, e.g.,

```
type
  recType:
    record

        day: byte;
        month: byte;
        year: int32;
        dayOfWeek: byte;

    endrecord;
```

The nice thing about the **packed** keyword is that it lets you explicitly state whether you want data alignment/padding in a record. On the other hand, if you've got a lot of records and you don't want field alignment on any of them, you'll probably want to use the “{\$A-}” (turn data alignment off) option rather than add the **packed** reserved word to each record definition. Note that you can turn data alignment back on with the “{\$A+}” directive if you want a sequence of records to be packed and the rest of them to be aligned.

While it's far easier (and syntactically safer) to use packed records when passing record data between assembly language and Delphi, you will have to determine on a case-by-case basis whether you're willing to give up the performance gain in exchange for using less memory (and a simpler interface). It is certainly the case that packed records are easier to maintain in HLA than aligned records (since you don't have to carefully place ALIGN directives throughout the record in the HLA code). Furthermore, on new x86 processors most mis-aligned data accesses aren't particularly expensive (the cache takes care of this). However, if performance really matters you will have to measure the performance of your program and determine the cost of using packed records.

Case variant records in Delphi let you add mutually exclusive fields to a record with an optional tag field. Here are two examples:

```
type
  r1=
    record

        stdField: integer;
        case choice: boolean of
            true: ( i: integer );
            false: ( r: real );
        end;

  r2=
    record
        s2: real;
        case boolean of // Notice no tag object here.
            true: ( s: string );
            false: ( c: char );
        end;
```

HLA does not support the case variant syntax, but it does support anonymous unions in a record that let you achieve the same semantics. The two examples above, converted to HLA (assuming “{A-}”) are

```
type
  r1:
    record

        stdField: int32;
        choice: boolean; // Notice that the tag field is just another field
        union
```

```

        i:int32;
        r:real64;

    endunion;

endrecord;

r2:
record

    s2:real64;
    union

        s: string;
        c: char;

    endunion;

endrecord;

```

Again, you should insert appropriate `ALIGN` directives if you're not creating a packed record. Note that you shouldn't place any `ALIGN` directives inside the anonymous union section; instead, place a single `ALIGN` directive before the `UNION` reserved word that specifies the size of the largest (scalar) object in the union as given by the table "Alignment of Record Fields" on page 1179.

In general, if the size of a record exceeds about 16-32 bytes, you should pass the record by reference rather than by value.

12.3.9 Passing Set Data Between Delphi and HLA

Sets in Delphi can have between 1 and 256 elements. Delphi implements sets using an array of bits, exactly as HLA implements character sets (see "Character Sets" on page 441). Delphi reserves one to 32 bytes for each set; the size of the set (in bytes) is $(\text{Number_of_elements} + 7) \text{ div } 8$. Like HLA's character sets, Delphi uses a set bit to indicate that a particular object is a member of the set and a zero bit indicates absence from the set. You can use the bit test (and set/complement/reset) instructions and all the other bit manipulation operations to manipulate character sets. Furthermore, the MMX instructions might provide a little added performance boost to your set operations (see "The MMX Instruction Set" on page 1113). For more details on the possibilities, consult the Delphi documentation and the chapters on character sets and the MMX instructions in this text.

Generally, sets are sufficiently short (maximum of 32 bytes) that passing the by value isn't totally horrible. However, you will get slightly better performance if you pass larger sets by reference. Note that HLA often passes character sets by value (16 bytes per set) to various Standard Library routines, so don't be totally afraid of passing sets by value.

12.3.10 Passing Array Data Between HLA and Delphi

Passing array data between some procedures written in Delphi and HLA is little different than passing array data between two HLA procedures. Generally, if the arrays are large, you'll want to pass the arrays by reference rather than value. Other than that, you should declare an appropriate array type in HLA to match the type you're passing in from Delphi and have at it. The following code fragments provide a simple example:

```

type
    PascalArray = array[0..127, 0..3] of integer;

procedure PassedArray( var ary: PascalArray ); external;

```

Corresponding HLA code:

```
type
  PascalArray: int32[ 128, 4];

procedure PassedArray( var ary: PascalArray ); external;
```

As the above examples demonstrate, Delphi's array declarations specify the starting and ending indicies while HLA's array bounds specify the number of elements for each dimension. Other than this difference, however, you can see that the two declarations are very similar.

Delphi uses row-major ordering for arrays. So if you're accessing elements of a Delphi multi-dimensional array in HLA code, be sure to use the row-major order computation (see "Row Major Ordering" on page 469).

12.3.11 Referencing Delphi Objects from HLA Code

Symbols you declare in the INTERFACE section of a Delphi program are public. Therefore, you can access these objects from HLA code if you declare those objects as external in the HLA program. The following sample program demonstrates this fact by declaring a structured constant (*y*) and a function (*callme*) that the HLA code uses when you press the button on a form. The HLA code calls the *callme* function (which returns the value 10) and then the HLA code stores the function return result into the *y* structured constant (which is really just a static variable).

```
unit DelphiEx5;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TDataTable = class(TForm)
    GetDataBtn: TButton;
    DataLabel: TLabel;
    procedure GetDataBtnClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

  // Here's a static variable that we will export to
  // the HLA source code (in Delphi, structured constants
  // are initialized static variables).

const
  y:integer = 12345;

var
  DataTable: TDataTable;

  // Here's the function we will export to the HLA code:
```

```

function callme:integer;

implementation

{$R *.DFM}
{$L TableData.obj }

function TableData:integer; external;

// This function will simply return 10 as the function
// result (remember, function results come back in EAX).

function callme;
begin

    callme := 10;

end;

procedure TDataTable.GetDataBtnClick(Sender: TObject);
var
    strVal: string;
    yVal:  string;
begin

    // Display the value that TableData returns.
    // Also display the value of y, which TableValue modifies

    str( TableData(), strVal );
    str( y, yVal );
    DataLabel.caption := 'Data = ' + strVal + ' y=' + yVal;

end;

end.

```

Program 12.13 DelphiEx5 – Static Data and Delphi Public Symbols Demonstration

```

unit TableDataUnit;

static
    y:int32; external;      // Static object from Delphi code

    //d:dataseg:nostorage; // All of our static variables are here.

    index: dword := -1; // index initial value;
    TheTable: dword[12] :=
        [ -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6]; // TheTable values.

// Interface to "callme" procedure found in the Delphi code:

```

```

procedure callme; external;

// Declare the procedure we're supplying to the Delphi code:

procedure TableData; external;
procedure TableData; @nodisplay; @noalignstack; @noframe;
begin TableData;

    callme();           // Call Delphi code.
    mov( eax, y );      // Store return result in Y.

    // Okay, on each successive call to this function, return
    // the next element (or wraparound to the first element) from
    // the "TheTable" array:

    inc( index );
    mov( index, eax );
    if( eax > 11 ) then

        xor( eax, eax );
        mov( eax, index );

    endif;
    mov( TheTable[ eax*4 ], eax );
    ret();

end TableData;

end TableDataUnit;

```

Program 12.14 HLA Code for DelphiEx5 Example

```

unit KylixEx5;

interface

uses
    SysUtils, Types, Classes, Variants, QGraphics, QControls, QForms, QDialogs,
    QStdCtrls;

type
    TForm1 = class(TForm)
        GetData: TButton;
        DataLabel: TLabel;
        procedure GetDataClick(Sender: TObject);
    private
        { Private declarations }
    public

```

```

    { Public declarations }
end;

// Here's a static variable that we will export to
// the HLA source code (in Delphi, structured constants
// are initialized static variables).

const
    y:integer = 12345;

var
    Form1: TForm1;

// Here's the function we will export to the HLA code:

    function callme:integer;

implementation

{$R *.xfm}
{$L TableData.o }

function TableData:integer; external;

// This function will simply return 10 as the function
// result (remember, function results come back in EAX).

function callme;
begin
    callme := 10;

end;

procedure TForm1.GetDataClick(Sender: TObject);
var
    strVal: string;
    yVal: string;
begin
    // Display the value that TableData returns.
    // Also display the value of y, which TableValue modifies

    str( TableData(), strVal );
    str( y, yVal );
    DataLabel.caption := 'Data = ' + strVal + ' y=' + yVal;

end;

end.

```

Program 12.15 KylixEx5 – Static Data and Delphi Public Symbols Demonstration

```

unit TableDataUnit;

static
  y:int32; external;      // Static object from Delphi code

  //d:dataseg:nostorage; // All of our static variables are here.

  index: dword := -1; // index initial value;
  TheTable: dword[12] :=
    [ -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6]; // TheTable values.

// Interface to "callme" procedure found in the Delphi code:

procedure callme; external;

// Declare the procedure we're supplying to the Delphi code:

procedure TableData; external;
procedure TableData; @nodisplay; @noalignstack; @noframe;
begin TableData;

  callme();          // Call Delphi code.
  mov( eax, y );     // Store return result in Y.

  // Okay, on each successive call to this function, return
  // the next element (or wraparound to the first element) from
  // the "TheTable" array:

  inc( index );
  mov( index, eax );
  if( eax > 11 ) then

    xor( eax, eax );
    mov( eax, index );

  endif;
  mov( TheTable[ eax*4 ], eax );
  ret();

end TableData;

end TableDataUnit;

```

Program 12.16 HLA Code for KylixEx5 Example

12.4 Programming in C/C++ and HLA

Unlike Delphi, that has only a single vendor, there are many different C/C++ compilers available on the market. Each vendor (Microsoft, Borland, Watcom, GNU, etc.) has their own ideas about how C/C++ should interface to external code. Many vendors have their own extensions to the C/C++ language to aid in the interface to assembly and other languages. For example, Borland provides a special keyword to let Borland C++ (and C++ Builder) programmers call Pascal code (or, conversely, allow Pascal code to call the C/C++ code). Microsoft, who stopped making Pascal compilers years ago, no longer supports this option. This is unfortunate since HLA uses the Pascal calling conventions. Fortunately, HLA provides a special interface to code that C/C++ systems generate.

Before we get started discussing how to write HLA modules for your C/C++ programs, you must understand two very important facts:

HLA's exception handling facilities are not directly compatible with C/C++'s exception handling facilities. This means that you cannot use the TRY..ENDTRY and RAISE statements in the HLA code you intend to link to a C/C++ program. This also means that you cannot call library functions that contain such statements. Since the HLA Standard Library modules use exception handling statements all over the place, this effectively prevents you from calling HLA Standard Library routines from the code you intend to link with C/C++⁵.

Given the rich set of language features that C/C++ supports, it should come as no surprise that the interface between the C/C++ language and assembly language is somewhat complex. Fortunately there are two facts that reduce this problem. First, HLA (v1.26 and later) supports C/C++'s calling conventions. Second, the other complex stuff you won't use very often, so you may not have to bother with it.

Note: the following sections assume you are already familiar with C/C++ programming. They make no attempt to explain C/C++ syntax or features other than as needed to explain the C/C++ assembly language interface. If you're not familiar with C/C++, you will probably want to skip this section.

Also note: although this text uses the generic term "C/C++" when describing the interface between HLA and various C/C++ compilers, the truth is that you're really interfacing HLA with the C language. There is a fairly standardized interface between C and assembly language that most vendors follow. No such standard exists for the C++ language and every vendor, if they even support an interface between C++ and assembly, uses a different scheme. In this text we will stick to interfacing HLA with the C language. Fortunately, all popular C++ compilers support the C interface to assembly, so this isn't much of a problem.

The examples in this text will use the GNU C++ compiler. There may be some minor adjustments you need to make if you're using some other C/C++ compiler; please see the vendor's documentation for more details.

12.4.1 Linking HLA Modules With C/C++ Programs

One big advantage of C/C++ over Delphi is that (most) C/C++ compiler vendors' products emit standard object files. So, working with object files and a true linker is much nicer than having to deal with Delphi's built-in linker. As nice as the Delphi system is, interfacing with assembly language is much easier in C/C++ than in Delphi.

The difference between a C and a C++ compilation occurs in the external declarations for the functions you intend to write in assembly language. For example, in a C source file you would simply write:

5. Note that the HLA Standard Library source code is available; feel free to modify the routines you want to use and remove any exception handling statements contained therein.


```
extern char* RetHW( void );
```

However, in a C++ environment, you would need the following external declaration:

```
extern "C"
{
    extern char* RetHW( void );
};
```

The 'extern "C"' clause tells the compiler to use standard C linkage even though the compiler is processing a C++ source file (C++ linkage is different than C and definitely far more complex; this text will not consider pure C++ linkage since it varies so much from vendor to vendor).

The following sample program demonstrates this external linkage mechanism by writing a short HLA program that returns the address of a string ("Hello World") in the EAX register (like Delphi, C/C++ expects functions to return their results in EAX). The main C/C++ program then prints this string to the console device.

```
#include <stdlib.h>
#include "ratc.h"

extern "C"
{
    extern char* ReturnHW( void );
};

int main()
    _begin( main )

        printf( "%s\n", ReturnHW() );
        _return 0;

    _end( main )
```

Program 12.17 Cex1 - A Simple Example of a Call to an Assembly Function from C++

```
unit ReturnHWUnit;

    procedure ReturnHW; external( "_ReturnHW" );
    procedure ReturnHW; nodisplay; noframe; noalignstk;
    begin ReturnHW;

        lea( eax, "Hello World" );
        ret();

    end ReturnHW;

end ReturnHWUnit;
```

Program 12.18 RetHW.hla - Assembly Code that Cex1 Calls

There are several new things in both the C/C++ and HLA code that might confuse you at first glance, so let's discuss these things real quick here.

The first strange thing you will notice in the C++ code is the `#include "ratc.h"` statement. RatC is a C/C++ macro library that adds several new features to the C++ language. RatC adds several interesting features and capabilities to the C/C++ language, but a primary purpose of RatC is to help make C/C++ programs a little more readable. Of course, if you've never seen RatC before, you'll probably argue that it's not as readable as pure C/C++, but even someone who has never seen RatC before can figure out 80% of RatC within a minutes. In the example above, the `_begin` and `_end` clauses clearly map to the "{" and "}" symbols (notice how the use of `_begin` and `_end` make it clear what function or statement associates with the braces; unlike the guesswork you've got in standard C). The `_return` statement is clearly equivalent to the C return statement. As you'll quickly see, all of the standard C control structures are improved slightly in RatC. You'll have no trouble recognizing them since they use the standard control structure names with an underscore prefix. This text promotes the creation of readable programs, hence the use of RatC in the examples appearing in this chapter⁶. You can find out more about RatC on Webster at <http://webster.cs.ucr.edu>.

The C/C++ program isn't the only source file to introduce something new. If you look at the HLA code you'll notice that the LEA instruction appears to be illegal. It takes the following form:

```
lea( eax, "Hello World" );
```

The LEA instruction is supposed to have a memory and a register operand. This example has a register and a constant; what is the address of a constant, anyway? Well, this is a syntactical extension that HLA provides to 80x86 assembly language. If you supply a constant instead of a memory operand to LEA, HLA will create a static (readonly) object initialized with that constant and the LEA instruction will return the address of that object. In this example, HLA will emit the string to the constants segment and then load EAX with the address of the first character of that string. Since HLA strings always have a zero terminating byte, EAX will contain the address of a zero-terminated string which is exactly what C++ wants. If you look back at the original C++ code, you will see that `RetHW` returns a `char*` object and the main C++ program displays this result on the console device.

If you haven't figured it out yet, this is a round-about version of the venerable "Hello World" program.

Microsoft VC++ users can compile this program from the command line by using the following commands⁷:

```
hla -c RetHW.hla          // Compiles and assembles RetHW.hla to RetHW.obj
cl Cex1.cpp RetHW.obj    // Compiles C++ code and links it with RetHW.obj
```

If you're a Borland C++ user, you'd use the following command sequence:

```
hla -o:omf RetHW.hla     // Compile HLA file to an OMF file.
bcc32i Cex1.cpp RetHW.obj // Compile and link C++ and assembly code.
                          // Could also use the BCC32 compiler.
```

GCC users can compile this program from the command line by using the following commands:

```
hla -o:omf RetHW.hla     // Compile HLA file to an OMF file.
bcc32i Cex1.cpp RetHW.obj // Compile and link C++ and assembly code.
                          // Could also use the BCC32 compiler.
```

6. If RatC really annoys you, just keep in mind that you've only got to look at a few RatC programs in this chapter. Then you can go back to the old-fashioned C code and hack to your heart's content!

7. This text assumes you've executed the `VCVARS32.BAT` file that sets up the system to allow the use of VC++ from the command line.

12.4.2 Register Preservation

Unlike Delphi, a single language with a single vendor, there is no single list of registers that you can freely use as scratchpad values within an assembly language function. The list changes by vendor and even changes between versions from the same vendor. However, you can safely assume that EAX is available for scratchpad use since C functions return their result in the EAX register. You should probably preserve everything else.

12.4.3 Function Results

C/C++ compilers universally seem to return ordinal and pointer function results in AL, AX, or EAX depending on the operand's size. The compilers probably return floating point results on the top of the FPU stack as well. Other than that, check your C/C++ vendor's documentation for more details on function return locations.

12.4.4 Calling Conventions

The standard C/C++ calling convention is probably the biggest area of contention between the C/C++ and HLA languages. VC++ and BCC both support multiple calling conventions. BCC even supports the Pascal calling convention that HLA uses, making it trivial to write HLA functions for BCC programs⁸. However, before we get into the details of these other calling conventions, it's probably a wise idea to first discuss the standard C/C++ calling convention.

Both VC++ and BCC *decorate* the function name when you declare an external function. For external "C" functions, the decoration consists of an underscore. If you look back at Program 12.18 you'll notice that the external name the HLA program actually uses is "_RetHW" rather than simply "RetHW". The HLA program itself, of course, uses the symbol "RetHW" to refer to the function, but the external name (as specified by the optional parameter to the EXTERNAL option) is "_RetHW". In the C/C++ program (Program 12.17) there is no explicit indication of this decoration; you simply have to read the compiler documentation to discover that the compiler automatically prepends this character to the function name⁹. Fortunately, HLA's EXTERNAL option syntax allows us to *undecorate* the name, so we can refer to the function using the same name as the C/C++ program. Name decoration is a trivial matter, easily fixed by HLA.

A big problem is the fact that C/C++ pushes parameters on the stack in the opposite direction of just about every other (non-C based) language on the planet; specifically, C/C++ pushes actual parameters on the stack from right to left instead of the more common left to right. This means that you cannot declare a C/C++ function with two or more parameters and use a simple translation of the C/C++ external declaration as your HLA procedure declaration, i.e., the following are not equivalent:

```
external void CToHLA( int p, unsigned q, double r );
procedure CToHLA( p:int32; q:uns32; r:real64 ); external;
```

Were you to call *CToHLA* from the C/C++ program, the compiler would push the *r* parameter first, the *q* parameter second, and the *p* parameter third - exactly the opposite order that the HLA code expects. As a result, the HLA code would use the L.O. double word of *r* as *p*'s value, the H.O. double word of *r* as *q*'s value, and the combination of *p* and *q*'s values as the value for *r*. Obviously, you'd most likely get an incorrect result from this calculation. Fortunately, there's an easy solution to this problem: use the @CDECL procedure option in the HLA code to tell it to reverse the parameters:

```
procedure CToHLA( p:int32; q:uns32; r:real64 ); @cdecl; external;
```

8. Microsoft used to support the Pascal calling convention, but when they stopped supporting their QuickPascal language, they dropped support for this option.

9. Most compilers provide an option to turn this off if you don't want this to occur. We will assume that this option is active in this text since that's the standard for external C names.

Now when the C/C++ code calls this procedure, it push the parameters on the stack and the HLA code will retrieve them in the proper order.

There is another big difference between the C/C++ calling convention and HLA: HLA procedures automatically clean up after themselves by removing all parameters pass to a procedure prior to returning to the caller. C/C++, on the other hand, requires the caller, not the procedure, to clean up the parameters. This has two important ramifications: (1) if you call a C/C++ function (or one that uses the C/C++ calling sequence), then your code has to remove any parameters it pushed upon return from that function; (2) your HLA code cannot automatically remove parameter data from the stack if C/C++ code calls it. The @CDECL procedure option tells HLA not to generate the code that automatically removes parameters from the stack upon return. Of course, if you use the @NOFRAME option, you must ensure that you don't remove these parameters yourself when your procedures return to their caller.

One thing HLA cannot handle automatically for you is removing parameters from the stack when you call a procedure or function that uses the @CDECL calling convention; for example, you must manually pop these parameters whenever you call a C/C++ function from your HLA code.

Removing parameters from the stack when a C/C++ function returns to your code is very easy, just execute an “add(constant, esp);” instruction where *constant* is the number of parameter bytes you've pushed on the stack. For example, the *CToHLA* function has 16 bytes of parameters (two *int32* objects and one *real64* object) so the calling sequence (in HLA) would look something like the following:

```
CToHLA( pVal, qVal, rVal ); // Assume this is the macro version.
add( 16, esp );           // Remove parameters from the stack.
```

Cleaning up after a call is easy enough. However, if you're writing the function that must leave it up to the caller to remove the parameters from the stack, then you've got a tiny problem – by default, HLA procedures always clean up after themselves. If you use the @CDECL option and don't specify the @NOFRAME option, then HLA automatically handles this for you. However, if you use the @NOFRAME option, then you've got to ensure that you leave the parameter data on the stack when returning from a function/procedure that uses the @CDECL calling convention.

If you want to leave the parameters on the stack for the caller to remove, then you must write the standard entry and exit sequences for the procedure that build and destroy the activation record (see “The Standard Entry Sequence” on page 813 and “The Standard Exit Sequence” on page 814). This means you've got to use the @NOFRAME (and @NODISPLAY) options on your procedures that C/C++ will call. Here's a sample implementation of the *CToHLA* procedure that builds and destroys the activation record:

```
procedure _CToHLA( rValue:real64; q:uns32; p:int32 ); @nodisplay; @noframe;
begin _CToHLA;

    push( ebp );           // Standard Entry Sequence
    mov( esp, ebp );
    // sub( _vars_, esp ); // Needed if you have local variables.
    .
    .                       // Code to implement the function's body.
    .
    mov( ebp, esp );       // Restore the stack pointer.
    pop( ebp );           // Restore link to previous activation record.
    ret();                 // Note that we don't remove any parameters.

end _CToHLA;
```

12.4.5 Pass by Value and Reference in C/C++

A C/C++ program can pass parameters to a procedure or function using one of two different mechanisms: pass by value and pass by reference. Since pass by reference parameters use pointers, this parameter passing mechanism is completely compatible between HLA and C/C++. The following two lines provide an

external declaration in C++ and the corresponding external (public) declaration in HLA for a pass by reference parameter using the calling convention:

```
extern void HasRefParm( int& refparm );           // C++
procedure HasRefParm( var refparm: int32 ); external; // HLA
```

Like HLA, C++ will pass the 32-bit address of whatever actual parameter you specify when calling the *HasRefParm* procedure. Don't forget, inside the HLA code, that you must dereference this pointer to access the actual parameter data. See the chapter on Intermediate Procedures for more details (see "Pass by Reference" on page 817).

Like HLA, C++ lets you pass untyped parameters by reference. The syntax to achieve this in C++ is the following:

```
extern void UntypedRefParm( void* parm1 );
```

Actually, this is not a reference parameter, but a value parameter with an untyped pointer.

In HLA, you can use the VAR keyword as the data type to specify that you want an untyped reference parameter. Here's the corresponding prototype for the *UntypedRefParm* procedure in HLA:

```
procedure UntypedRefParm( var parm1:var );
external;
```

12.4.6 Scalar Data Type Correspondence Between C/C++ and HLA

When passing parameters between C/C++ and HLA procedures and functions, it's very important that the calling code and the called code agree on the basic data types for the parameters. In this section we will draw a correspondence between the C/C++ scalar data types and the HLA (v1.x) data types.

Assembly language supports any possible data format, so HLA's data type capabilities will always be a superset of C/C++'s. Therefore, there may be some objects you can create in HLA that have no counterpart in C/C++, but the reverse is not true. Since the assembly functions and procedures you write are generally manipulating data that C/C++ provides, you don't have to worry too much about not being able to process some data passed to an HLA procedure by C/C++.

C/C++ provides a wide range of different integer data types. Unfortunately, the exact representation of these types is implementation specific. The following table lists the C/C++ types as currently implemented by Borland C++ and Microsoft VC++. This table may very well change as 64-bit compilers become available.

Table 5: C/C++ and HLA Integer Types

C/C++	HLA Equivalent	Range	
		Minimum	Maximum
int	int32	-2147483648	2147483647
unsigned	uns32	0	4294967295
signed char	int8	-128	127
short	int16	-32768	32767
long	int32	-2147483648	2147483647

Table 5: C/C++ and HLA Integer Types

C/C++	HLA Equivalent	Range	
		Minimum	Maximum
unsigned char	uns8	0	255
unsigned short	uns16	0	65535

In addition to the integer values, C/C++ supports several non-integer ordinal types. The following table provides their HLA equivalents:

Like the integer types, C/C++ supports a wide range of real numeric formats. The following table presents these types and their HLA equivalents.

Table 6: Real Types in C/C++ and HLA

C/C++	HLA	Range	
		Minimum	Maximum
double	real64	5.0 E-324	1.7 E+308
float	real32	1.5 E-45	3.4 E+38

The last scalar type of interest is the pointer type. Both HLA and C/C++ use a 32-bit address to represent pointers, so these data types are completely equivalent in both languages.

12.4.7 Passing String Data Between C/C++ and HLA Code

C/C++ uses zero terminated strings. Algorithms that manipulate zero-terminated strings are not as efficient as functions that work on length-prefixed strings; on the plus side, however, zero-terminated strings are very easy to work with. HLA's strings are downwards compatible with C/C++ strings since HLA places a zero byte at the end of each HLA string. Since you'll probably not be calling HLA Standard Library string routines, the fact that C/C++ strings are not upwards compatible with HLA strings generally won't be a problem. If you do decide to modify some of the HLA string functions so that they don't raise exceptions, you can always translate the *str.cStrToStr* function that translates zero-terminated C/C++ strings to HLA strings.

A C/C++ string variable is typically a `char*` object or an array of characters. In either case, C/C++ will pass the address of the first character of the string to an external procedure whenever you pass a string as a parameter. Within the procedure, you can treat the parameter as an indirect reference and dereference to pointer to access characters within the string.

12.4.8 Passing Record/Structure Data Between HLA and C/C++

Records in HLA are (mostly) compatible with C/C++ structs. You can easily translate a C/C++ struct to an HLA record. In this section we'll explore how to do this and learn about the incompatibilities that exist between HLA records and C/C++ structures.

For the most part, translating C/C++ records to HLA is a no brainer. Just grab the “guts” of a structure declaration and translate the declarations to HLA syntax within a RECORD..ENDRECORD block and you’re done.

Consider the following C/C++ structure type declaration:

```
typedef struct
{
    unsigned char day;
    unsigned char month;
    int year;
    unsigned char dayOfWeek;
} dateType;
```

The translation to an HLA record is, for the most part, very straight-forward. Just translate the field types accordingly and use the HLA record syntax (see “Records” on page 483) and you’re in business. The translation is the following:

```
type
    recType:
        record

            day: byte;
            month: byte;
            year: int32;
            dayOfWeek: byte;

        endrecord;
```

There is one minor problem with this example: data alignment. Depending on your compiler and whatever defaults it uses, C/C++ might not pack the data in the structure as compactly as possible. Some C/C++ compilers will attempt to align the fields on double word or other boundaries. With double word alignment of objects larger than a byte, the previous C/C++ **typedef** statement is probably better modelled by

```
type
    recType:
        record

            day: byte;
            month: byte;
            padding: byte[2]; // Align year on a four-byte boundary.
            year: int32;
            dayOfWeek: byte;
            morePadding: byte[3]; // Make record an even multiple of four bytes.

        endrecord;
```

Of course, a better solution is to use HLA’s ALIGN directive to automatically align the fields in the record:

```
type
    recType:
        record

            day: byte;
            month: byte;
            align( 4 ); // Align year on a four-byte boundary.
            year: int32;
            dayOfWeek: byte;
            align(4); // Make record an even multiple of four bytes.

        endrecord;
```

Alignment of the fields is good insofar as access to the fields is faster if they are aligned appropriately. However, aligning records in this fashion does consume extra space (five bytes in the examples above) and that can be expensive if you have a large array of records whose fields need padding for alignment.

You will need to check your compiler vendor's documentation to determine whether it packs or pads structures by default. Most compilers give you several options for packing or padding the fields on various boundaries. Padded structures might be a bit faster while packed structures (i.e., no padding) are going to be more compact. You'll have to decide which is more important to you and then adjust your HLA code accordingly.

Note that by default, C/C++ passes structures by value. A C/C++ program must explicitly take the address of a structure object and pass that address in order to simulate pass by reference. In general, if the size of a structure exceeds about 16 bytes, you should pass the structure by reference rather than by value.

12.4.9 Passing Array Data Between HLA and C/C++

Passing array data between some procedures written in C/C++ and HLA is little different than passing array data between two HLA procedures. First of all, C/C++ can only pass arrays by reference, never by value. Therefore, you must always use pass by reference inside the HLA code. The following code fragments provide a simple example:

```
int CArray[128][4];

extern void PassedArray( int array[128][4] );
```

Corresponding HLA code:

```
type
  CArray: int32[ 128, 4];

procedure PassedArray( var ary: CArray ); external;
```

As the above examples demonstrate, C/C++'s array declarations are similar to HLA's insofar as you specify the bounds of each dimension in the array.

C/C++ uses row-major ordering for arrays. So if you're accessing elements of a C/C++ multi-dimensional array in HLA code, be sure to use the row-major order computation (see "Row Major Ordering" on page 469).

12.5 Putting It All Together

Most real-world assembly code that is written consists of small modules that programmers link to programs written in other languages. Most languages provide some scheme for interfacing that language with assembly (HLA) code. Unfortunately, the number of interface mechanisms is sufficiently close to the number of language implementations to make a complete exposition of this subject impossible. In general, you will have to refer to the documentation for your particular compiler in order to learn sufficient details to successfully interface assembly with that language.

Fortunately, nagging details aside, most high level languages do share some common traits with respect to assembly language interface. Parameter passing conventions, stack clean up, register preservation, and several other important topics often apply from one language to the next. Therefore, once you learn how to interface a couple of languages to assembly, you'll quickly be able to figure out how to interface to others (given the documentation for the new language).

This chapter discusses the interface between the Delphi and C/C++ languages and assembly language. Although there are more popular languages out there (e.g., Visual Basic), Delphi and C/C++ introduce most of the concepts you'll need to know in order to interface a high level language with assembly language.

Beyond that point, all you need is the documentation for your specific compiler and you'll be interfacing assembly with that language in no time.

Volume Five: Advanced Procedures

This volume begins the discussion of some specialized information that is of interest to those wanting to become advanced assembly language programmers. Many of the techniques appearing in this volume rarely find their way into typical assembly language programs; this is not because this material is unimportant, but, rather, because most assembly language programs are not written by advanced assembly language programmers. This volume is also of interest to those intending to write compilers or other language translators. This volume discusses the run-time environments that may high level languages use.

- Chapter One: Thunks
This chapter describes a special type of indirect procedure call known as the *thunk*. Thunks are useful for deferring the execution of sequences of code in the program.
- Chapter Two: Low Level Iterator Implementation
This chapter discusses how to implement iterators in "pure" assembly language (a topic too advanced to present in earlier volumes of this text). This chapter also discusses alternative implementations of iterators in an assembly language program.
- Chapter Three: Coroutines
This chapter describes a special type of program unit known as the coroutine. Coroutines are excellent structures to use when several pieces of code "take turns" executing, such as when players take turns in a game.
- Chapter Four: Low Level Parameter Implementation
This chapter discusses several new ways to pass parameters and how to implement parameter passing in low-level ("pure") assembly language.
- Chapter Five: Lexical Nesting
This chapter discusses the concept of block structured programming languages and how to implement local and non-local automatic variable access in a program.
- Chapter Six: Questions and Exercises
This chapter provides questions, programming projects, and laboratory exercises for this volume.

Volume Five:

Advanced Procedures

1.1 Chapter Overview

This chapter discusses thunks which are special types of procedures and procedure calls you can use to defer the execution of some procedure call. Although the use of thunks is not commonplace in standard assembly code, their semantics are quite useful in AI (artificial intelligence) and other programs. The proper use of thunks can dramatically improve the performance of certain programs. Perhaps a reason thunks do not find extensive use in assembly code is because most assembly language programmers are unaware of their capabilities. This chapter will solve that problem by presenting the definition of thunks and describe how to use them in your assembly programs.

1.2 First Class Objects

The actual low-level implementation of a thunk, and the invocation of a thunk, is rather simple. However, to understand why you would want to use a thunk in an assembly language program we need to jump to a higher level of abstraction and discuss the concept of *First Class Objects*.

A first class object is one you can treat like a normal scalar data variable. You can pass it as a parameter (using any arbitrary parameter passing mechanism), you can return it as a function result, you can change the object's value via certain legal operations, you can retrieve its value, and you can assign one instance of a first class object to another. An *int32* variable is a good example of a first class object.

Now consider an array. In many languages, arrays are not first class objects. Oh, you can pass them as parameters and operate on them, but you can't assign one array to another nor can you return an array as a function result in many languages. In other languages, however, all these operations are permissible on arrays so they are first class objects (in such languages).

A statement sequence (especially one involving procedure calls) is generally not a first class object in many programming languages. For example, in C/C++ or Pascal/Delphi you cannot pass a sequence of statements as a parameter, assign them to a variable, return them as a function result, or otherwise operate on them as though they were data. You cannot create arbitrary arrays of statements nor can you ask a sequence of statements to execute themselves except at their point of declaration.

If you've never used a language that allows you to treat executable statements as data, you're probably wondering why anyone would ever want to do this. There are, however, some very good reasons for wanting to treat statements as data and execute them on demand. If you're familiar with the C/C++ programming language, consider the C/C++ "?" operator:

```
expr ? Texpr : Fexpr
```

For those who are unfamiliar with the "?" operator, it evaluates the first expression (*expr*) and then returns the value of *Texpr* if *expr* is true, it evaluates and returns *Fexpr* if *expr* evaluates false. Note that this code does not evaluate *Fexpr* if *expr* is true; likewise, it does not evaluate *Texpr* if *expr* is false. Contrast this with the following C/C++ function:

```
int ifexpr( int x, int t, int f )
{
    if( x ) return t;
    return f;
}
```

A function call of the form "ifexpr(expr, Texpr, Fexpr);" is not semantically equivalent to "expr ? Texpr : Fexpr". The *ifexpr* call always evaluates all three parameters while the conditional expression operator ("?") does not. If either *Texpr* or *Fexpr* produces a side-effect, then the call to *ifexpr* may produce a different result than the conditional operator, e.g.,

```
i = (x==y) ? a++ : b--;
j = ifexpr( x==y, c++, d-- );
```

In this example either *a* is incremented or *b* is decremented, but not both because the conditional operator only evaluates one of the two expressions based on the values of *x* and *y*. In the second statement, however, the code both increments *c* and decrements *d* because C/C++ always evaluates all value parameters before calling the function; that is, C/C++ eagerly evaluates function parameter expressions (while the conditional operator uses deferred evaluation).

Supposing that we wanted to defer the execution of the statements "c++" and "d--" until inside the function's body, this presents a classic case where it would be nice to treat a pair of statements as first class objects. Rather than pass the value of "c++" or "d--" to the function, we pass the actual statements and expand these statements inside the function wherever the format parameter occurs. While this is not possible in C/C++, it is possible in certain languages that support the use of statements as first class objects. Naturally, if it can be done in any particular language, then it can be done in assembly language.

Of course, at the machine code level a statement sequence is really nothing more than a sequence of bytes. Therefore, we could treat those statements as data by directly manipulating the object code associated with that statement sequence. Indeed, in some cases this is the best solution. However, in most cases it will prove too cumbersome to manipulate a statement sequence by directly manipulating its object code. A better solution is to use a pointer to the statement sequence and CALL that sequence indirectly whenever we want to execute it. Using a pointer in this manner is usually far more efficient than manipulating the code directly, especially since you rarely change the instruction sequence itself. All you really want to do is defer the execution of that code. Of course, to properly return from such a sequence, the sequence must end with a RET instruction. Consider the following HLA implementation of the "ifexpr" function given earlier:

```
procedure ifexpr( expr:boolean; trueStmts:dword; falseStmts:dword );
    returns( "eax" );

begin ifexpr;

    if( expr ) then

        call( trueStmts );

    else

        call( falseStmts );

    endif;

end ifexpr;
.
.
.
    jmp overStmt1;
stmt1: mov( c, eax );
        inc( c );
        ret();

overStmt1:
    jmp overStmt2
stmt2: mov( d, eax );
        dec( d );
        ret();

overStmt2:
ifexpr( exprVal, &stmt1, &stmt2 );
```

(for reasons you'll see shortly, this code assumes that the *c* and *d* variables are global, static, objects.)

Notice how the code above passes the addresses of the *stmt1* and *stmt2* labels to the *ifexpr* procedure. Also note how the code sequence above jumps over the statement sequences so that the code only executes them in the body of the *ifexpr* procedure.

As you can see, the example above creates two mini-procedures in the main body of the code. Within the *ifexpr* procedure the program calls one of these mini-procedures (*stmt1* or *stmt2*). Unlike standard HLA procedures, these mini-procedures do not set up a proper activation record. There are no parameters, there are no local variables, and the code in these mini-procedures does not execute the standard entry or exit sequence. In fact, the only part of the activation record present in this case is the return address.

Because these mini-procedures do not manipulate EBP's value, EBP is still pointing at the activation record for the *ifexpr* procedure. For this reason, the *c* and *d* variables must be global, static objects; you must not declare them in a VAR section. For if you do, the mini-procedures will attempt to access these objects in *ifexpr*'s activation record, not and the caller's activation record. This, of course, would return the wrong value.

Fortunately, there is a way around this problem. HLA provides a special data type, known as a thunk, that eliminates this problem. To learn about thunks, keep reading...

1.3 Thunks

A *thunk* is an object with two components: a pointer containing the address of some code and a pointer to an execution environment (e.g., an activation record). Thunks, therefore, are an eight-byte (64-bit) data type, though (unlike a *qword*) the two pieces of a thunk are independent. You can declare thunk variables in an HLA program, assign one thunk to another, pass thunks as parameters, return them as function results, and, in general, do just about anything that is possible with a 64-bit data type containing two double word pointers.

To declare a thunk in HLA you use the *thunk* data type, e.g.,

```
static
  myThunk: thunk;
```

Like other 64-bit data types HLA does not provide a mechanism for initializing thunks you declare in a static section. However, you'll soon see that it is easy to initialize a thunk within the body of your procedures.

A *thunk* variable holds two pointers. The first pointer, in the L.O. double word of the thunk, points at some execution environment, that is, an activation record. The second pointer, in the H.O. double word of the thunk, points at the code to execute for the thunk.

To "call" a thunk, you simply apply the *()* suffix to the thunk's name. For example, the following "calls" *myThunk* in the procedure where you've declared *myThunk*:

```
myThunk( );
```

Thunks never have parameters, so the parameter list must be empty.

A thunk invocation is a bit more involved than a simple procedure call. First of all, a thunk invocation will modify the value in EBP (the pointer to the current procedure's activation record), so the thunk invocation must begin by preserving EBP's value on the stack. Next, the thunk invocation must load EBP with the address of the thunk's execution environment; that is, the code must load the L.O. double word of the thunk value into EBP. Next, the thunk must call the code at the address specified by the H.O. double word of the thunk variable. Finally, upon returning from this code, the thunk invocation must restore the original activation record pointer from the stack. Here's the exact sequence HLA emits to a statement like "myThunk()":

```
push( (type dword myThunk) );      // Pass execution environment as parm.
call( (type dword myThunk[4]) );  // Call the thunk
```

The body of a thunk, that is, the code at the address found in the H.O. double word of the thunk variable, is not a standard HLA procedure. In particular, the body of a thunk does not execute the standard entry or exit sequences for a standard procedure. The calling code passes the pointer to the execution environment

(i.e., an activation record) on the stack.. It is the thunk's responsibility to preserve the current value of EBP and load EBP with this value appearing on the stack. After the thunk loads EBP appropriately, it can execute the statements in the body of the thunk, after which it must restore EBP's original value.

Because a thunk variable contains a pointer to an activation record to use during the execution of the thunk's code, it is perfectly reasonable to access local variables and other local objects in the activation record active when you define the thunk's body. Consider the following code:

```

procedure SomeProc;
var
  c: int32;
  d: int32;
  t: thunk;
begin SomeProc;

  mov( ebp, (type dword t));
  mov( &thunk1, (type dword t[4]));
  jmp OverThunk1;
thunk1:
  push( EBP );           // Preserve old EBP value.
  mov( [esp+8], ebp );   // Get pointer to original thunk environment.
  mov( d, eax );
  add( c, eax );
  pop( ebp );           // Restore caller's environment.
  ret( 4 );             // Remove EBP value passed as parameter.
OverThunk1:
  .
  .
  .
  t(); // Computes the sum of c and d into EAX.

```

This example initializes the *t* variable with the value of *SomeProc*'s activation record pointer (EBP) and the address of the code sequence starting at label *thunk1*. At some later point in the code the program invokes the thunk which begins by pushing the pointer to *SomeProc*'s activation record. Then the thunk executes the PUSH/MOV/MOV/ADD/POP/RET sequence starting at address *thunk1*. Since this code loads EBP with the address of the activation record containing *c* and *d*, this code sequence properly adds these variables together and leaves their sum in EAX. Perhaps this example is not particularly exciting since the invocation of *t* occurs while EBP is still pointing at *SomeProc*'s activation record. However, you'll soon see that this isn't always the case.

1.4 Initializing Thunks

In the previous section you saw how to manually initialize a thunk variable with the environment pointer and the address of an in-line code sequence. While this is a perfectly legitimate way to initialize a thunk variable, HLA provides an easier solution: the THUNK statement.

The HLA THUNK statement uses the following syntax:

```
thunk thunkVar := #{ code sequence }#;
```

thunkVar is the name of a thunk variable and *code_sequence* is a sequence of HLA statements (note that the sequence does not need to contain the thunk entry and exit sequences. Specifically, it doesn't need the "push(ebp);" and "mov([esp+8]);" instructions at the beginning of the code, nor does it need to end with the "pop(ebp);" and "ret(4);" instructions. HLA will automatically supply the thunk's entry and exit sequences.

Here's the example from the previous section rewritten to use the THUNK statement:

```

procedure SomeProc;
var
  c: int32;
  d: int32;

```



```

t: thunk;
begin SomeProc;

    thunk t :=
        #{
            mov( d, eax );
            add( c, eax );
        }#;
    .
    .
    .
    t(); // Computes the sum of c and d into EAX.

```

Note how much clearer and easier to read this code sequence becomes when using the THUNK statement. You don't have to stick in statements to initialize *t*, you don't have to jump over the thunk body, you don't have to include the thunk entry/exit sequences, and you don't wind up with a bunch of statement labels in the code. Of course, HLA emits the same code sequence as found in the previous section, but this form is much easier to read and work with.

1.5 Manipulating Thunks

Since a thunk is a 64-bit variable, you can do anything with a thunk that you can do, in general, with any other qword data object. You can assign one thunk to another, compare thunks, pass thunks a parameters, return thunks as function results, and so on. That is to say, thunks are first class objects. Since a thunk is a representation of a sequence of statements, those statements are effectively first class objects. In this section we'll explore the various ways we can manipulate thunks and the statements associated with them.

1.5.1 Assigning Thunks

To assign one thunk to another you simply move the two double words from the source thunk to the destination thunk. After the assignment, both thunks specify the same sequence of statements and the same execution environment; that is, the thunks are now aliases of one another. The order of assignment (H.O. double word first or L.O. double word first) is irrelevant as long as you assign both double words before using the thunk's value. By convention, most programmers assign the L.O. double word first. Here's an example of a thunk assignment:

```

mov( (type dword srcThunk), eax );
mov( eax, (type dword destThunk));
mov( (type dword srcThunk[4]), eax );
mov( eax, (type dword destThunk[4]));

```

If you find yourself assigning one thunk to another on a regular basis, you might consider using a macro to accomplish this task:

```

#macro movThunk( src, dest );

    mov( (type dword src), eax );
    mov( eax, (type dword dest));
    mov( (type dword src[4]), eax );
    mov( eax, (type dword dest[4]));

#endmacro;

```

If the fact that this macro's side effect of disturbing the value in EAX is a concern to you, you can always copy the data using a PUSH/POP sequence (e.g., the HLA extended syntax MOV instruction):

```
#macro movThunk( src, dest );

    mov( (type dword src), (type dword dest));
    mov( (type dword src[4]), (type dword dest[4]));

#endmacro;
```

If you don't plan on executing any floating point code in the near future, or you're already using the MMX instruction set, you can also use the MMX MOVQ instruction to copy these 64 bits with only two instructions:

```
movq( src, mm0 );
movq( mm0, dest );
```

Don't forget, however, to execute the EMMS instruction before calling any routines that might use the FPU after this sequence.

1.5.2 Comparing Thunks

You can compare two thunks for equality or inequality using the standard 64-bit comparisons (see "Extended Precision Comparisons" on page 857). If two thunks are equal then they refer to the same code sequence with the same execution environment; if they are not equal, then they could have different code sequences or different execution environments (or both) associated with them. Note that it doesn't make any sense to compare one thunk against another for less than or greater than. They're either equal or not equal.

Of course, it's quite easy to have two thunks with the same environment pointer and different code pointers. This occurs when you initialize two thunk variables with separate code sequences in the same procedure, e.g.,

```
thunk t1 :=
    #{
        mov( 0, eax );
        mov( i, ebx );
    }#;

thunk t2 :=
    #{
        mov( 4, eax );
        mov( j, ebx );
    }#;

// At this point, t1 and t2 will have the same environment pointer
// (EBP's value) but they will have different code pointers.
```

Note that it is quite possible for two thunks to refer to the same statement sequence yet have different execution environments. This can occur when you have a recursive function that initializes a pair of thunk variables with the same instruction sequence on different recursive calls of the function. Since each recursive invocation of the function will have its own activation record, the environment pointers for the two thunks will be different even though the pointers to the code sequence are the same. However, if the code that initializes a specific thunk is not recursive, you can sometimes compare two thunks by simply comparing their code pointers (the H.O. double words of the thunks) if you're careful about never using thunks once their execution environment goes away (i.e., the procedure in which you originally assigned the thunk value returns to its caller).

1.5.3 Passing Thunks as Parameters

Since the thunk data type is effectively equivalent to a qword type, there is little you can do with a qword object that you can't also do with a thunk object. In particular, since you can pass qwords as parameters you can certainly pass thunks as parameters to procedures.

To pass a thunk by value to a procedure is very easy, simply declare a formal parameter using the thunk data type:

```

procedure HasThunkParm( t:thunk );
var
    i:integer;
begin HasThunkParm;

    mov( 1, i );
    t();           // Invoke the thunk passed as a parameter.
    mov( i, eax ); // Note that t does not affect our environment.

end HasThunkParm;

.
.
.
thunk  thunkParm :=
    #{
        mov( 0, i ); // Not the same "i" as in HasThunkParm!
    }#;

HasThunkParm( thunkParm );

```

Although a thunk is a pointer (a pair of pointers, actually), you can still pass thunks by value. Passing a thunk by value passes the values of those two pointer objects to the procedure. The fact that these values are the addresses of something else is not relevant, you're passing the data by value.

HLA automatically pushes the value of a thunk on the stack when passing a thunk by value. Since thunks are 64-bit objects, you can only pass them on the stack, you cannot pass them in a register¹. When HLA passes a thunk, it pushes the H.O. double word (the code pointer) of the thunk first followed by the L.O. double word (the environment pointer). This way, the two pointers are situated on the stack in the same order they normally appear in memory (the environment pointer at the lowest address and the code pointer at the highest address).

If you decide to manually pass a thunk on the stack yourself, you must push the two halves of the thunk on the stack in the same order as HLA, i.e., you must push the H.O. double word first and the L.O. double word second. Here's the call to *HasThunkParm* using manual parameter passing:

```

push( (type dword thunkParm[4]) );
push( (type dword thunkParm) );
call HasThunkParm;

```

You can also pass thunks by reference to a procedure using the standard pass by reference syntax. Here's a typical procedure prototype with a pass by reference thunk parameter:

```

procedure refThunkParm( var t:thunk ); forward;

```

When you pass a thunk by reference, you're passing a pointer to the thunk itself, not the pointers to the thunk's execution environment or code sequence. To invoke such a thunk you must manually dereference the pointer to the thunk, push the pointer to the thunk's execution environment, and indirectly call the code sequence. Here's an example implementation of the *refThunkParm* prototype above:

1. Technically, you could pass a thunk in two 32-bit registers. However, you will have to do this manually; HLA will not automatically move the two pointers into two separate registers for you.

```

procedure refThunkParm( var t:thunk );
begin refThunkParm;

    push( eax );
    .
    .
    .
    mov( t, eax );           // Get pointer to thunk object.
    push( [eax] );          // Push pointer to thunk's environment.
    call( (type dword [eax+4]) ); // Call the code sequence.
    .
    .
    .
    pop( eax );

end refThunkParm;

```

Of course, one of the main reasons for passing an object by reference is so you can assign a value to the actual parameter value. Passing a thunk by reference provides this same capability – you can assign a new code sequence address and execution environment pointer to a thunk when you pass it by reference. However, always be careful when assigning values to thunk reference parameters within a procedure that you specify an execution environment that will still be valid when the code actually invokes the thunk. We'll explore this very problem in a later section of this chapter (see “Activation Record Lifetimes and Thunks” on page 1288).

Although we haven't yet covered this, HLA does support several other parameter passing mechanisms beyond pass by value and pass by reference. You can certainly pass thunks using these other mechanisms. Indeed, thunks are the basis for two of HLA's parameter passing mechanisms: pass by name and pass by evaluation. However, this is getting a little ahead of ourselves; we'll return to this subject in a later chapter in this volume.

1.5.4 Returning Thunks as Function Results

Like any other first class data object, we can also return thunks as the result of some function. The only complication is the fact that a thunk is a 64-bit object and we normally return function results in a register. To return a full thunk as a function result, we're going to need to use two registers or a memory location to hold the result.

To return a 64-bit (non-floating point) value from a function there are about three or four different locations where we can return the value: in a register pair, in an MMX register, on the stack, or in a memory location. We'll immediately discount the use of the MMX registers since their use is not general (i.e., you can't use them simultaneously with floating point operations). A global memory location is another possible location for a function return result, but the use of global variables has some well-known deficiencies, especially in a multi-threaded/multi-tasking environment. Therefore, we'll avoid this solution as well. That leaves using a register pair or using the stack to return a thunk as a function result. Both of these schemes have their advantages and disadvantages, we'll discuss these two schemes in this section.

Returning thunk function results in registers is probably the most convenient way to return the function result. The big drawback is obvious – it takes two registers to return a 64-bit thunk value. By convention, most programmers return 64-bit values in the EDX:EAX register pair. Since this convention is very popular, we will adopt it in this section. Keep in mind, however, that you may use almost any register pair you like to return this 64-bit value (though ESP and EBP are probably off limits).

When using EDX:EAX, EAX should contain the pointer to the execution environment and EDX should contain the pointer to the code sequence. Upon return from the function, you should store these two registers into an appropriate thunk variable for future use.

To return a thunk on the stack, you must make room on the stack for the 64-bit thunk value prior to pushing any of the function's parameters onto the stack. Then, just before the function returns, you store the

think result into these locations on the stack. When the function returns it cleans up the parameters it pushed on the stack but it does not free up the thunk object. This leaves the 64-bit thunk value sitting on the top of the stack after the function returns.

The following code manually creates and destroys the function's activation record so that it can specify the thunk result as the first two parameters of the function's parameter list:

```

procedure RtnThunkResult
(
  ThunkCode:dword; // H.O. dword of return result goes here.
  ThunkEnv:dword; // L.O. dword of return result goes here.
  selection:boolean; // First actual parameter.
  tTrue: thunk; // Return this thunk if selection is true.
  tFalse:thunk // Return this thunk if selection is false.
); @nodisplay; @noframe;
begin RtnThunkResult;

  push( ebp ); // Set up the activation record.
  mov( esp, ebp );
  push( eax );

  if( selection ) then

    mov( (type dword tTrue), eax );
    mov( eax, ThunkEnv );
    mov( (type dword tTrue[4]), eax );
    mov( eax, ThunkCode );

  else

    mov( (type dword tFalse), eax );
    mov( eax, ThunkEnv );
    mov( (type dword tFalse[4]), eax );
    mov( eax, ThunkCode );

  endif;

  // Clean up the activation record, but leave the eight
  // bytes of the thunk return result on the stack when this
  // function returns.

  pop( eax );
  pop( ebp );
  ret( _parms_ - 8 ); // _parms_ is total # of bytes of parameters (28).

end RtnThunkResult;
.
.
.
// Example of call to RtnThunkResult and storage of return result.
// (Note passing zeros as the thunk values to reserve storage for the
// thunk return result on the stack):

RtnThunkResult( 0, 0, ChooseOne, t1, t2 );
pop( (type dword SomeThunkVar) );
pop( (type dword SomeThunkVar[4]) );

```

If you prefer not to list the thunk parameter as a couple of pseudo-parameters in the function's parameter list, you can always manually allocate storage for the parameters prior to the call and refer to them using the "[ESP+disp]" or "[EBP+disp]" addressing mode within the function's body.

1.6 Activation Record Lifetimes and Thunks

There is a problem that can occur when using thunks in your applications: it's quite possible to invoke a thunk long after the associated execution environment (activation record) is no longer valid. Consider the following HLA code that demonstrates this problem:

```
static
  BadThunk: thunk;

  procedure proc1;
  var
    i:int32;
  begin proc1;

    thunk BadThunk :=
      #{
        stdout.put( "i = ", i, nl );
      };
    mov( 25, i );

  end proc1;

  procedure proc2;
  var
    j:int32;
  begin proc2;

    mov( 123, j );
    BadThunk();

  end proc2;
  .
  .
  .
```

If the main program in this code fragment calls *proc1* and then immediately calls *proc2*, this code will probably print "i = 123" although there is no guarantee this will happen (the actual result depends on a couple of factors, although "i = 123" is the most likely output).

The problem with this code example is that *proc1* initializes *BadThunk* with the address of an execution environment that is no longer "live" when the program actually executes the thunk's code. The *proc1* procedure constructs its own activation record and initializes the variable *i* in this activation record with the value 25. This procedure also initializes *BadThunk* with the address of the code sequence containing the *stdout.put* statement and it initializes *BadThunk*'s execution environment pointer with the address of *proc1*'s activation record. Then *proc1* returns to its caller. Unfortunately upon returning to its caller, *proc1* also obliterates its activation record even though *BadThunk* still contains a pointer into this area of memory. Later, when the main program calls *proc2*, *proc2* builds its own activation record (most likely over the top of *proc1*'s old activation record). When *proc2* invokes *BadThunk*, *BadThunk* uses the original pointer to *proc1*'s activation record (which is now invalid and probably points at *proc2*'s activation record) from which to fetch *i*'s value. If nothing extra was pushed or popped between the *proc1* invocation and the *proc2* invocation, then *j*'s value in *proc2* is probably at the same memory location as *i* was in *proc1*'s invocation. Hence, the *stdout.put* statement in the thunk's code will print *j*'s value.

This rather trivial example demonstrates an important point about using thunks – you must always ensure that a thunk's execution environment is still valid whenever you invoke a thunk. In particular, if you use HLA's THUNK statement to automatically initialize a thunk variable with the address of a code

sequence and the execution environment of the current procedure, you must not invoke that thunk once the procedure returns to its caller; for at that point the thunk's execution environment pointer will not be valid.

This discussion does not suggest that you can only use a thunk within the procedure in which you assign a value to that thunk. You may continue to invoke a thunk, even from outside the procedure whose activation record the thunk references, until that procedure returns to its caller. So if that procedure calls some other procedure (or even itself, recursively) then it is legal to call the thunk associated with that procedure.

1.7 Comparing Thunks and Objects

Thunks are very similar to objects insofar as you can easily implement an abstract data type with a thunk. Remember, an abstract data type is a piece of data and the operation(s) on that data. In the case of a thunk, the execution environment pointer can point at the data while the code address can point at the code that operates on the data. Since you can also use objects to implement abstract data types, one might wonder how objects and thunks compare to one another.

Thunks are somewhat less "structured" than objects. An object contains a set of data values and operations (methods) on those data values. You cannot change the data values an object operates upon without fundamentally changing the object (i.e., selecting a different object in memory). It is possible, however, to change the execution environment pointer in a thunk and have that thunk operate on fundamentally different data. Although such a course of action is fraught with difficulty and very error-prone, there are some times when changing the execution environment pointer of a thunk will produce some interesting results. This text will leave it up to you to discover how you could abuse thunks in this fashion.

1.8 An Example of a Thunk Using the Fibonacci Function

By now, you're probably thinking "thunks may be interesting, but what good are they?" The code associated with creating and invoking thunks is not spectacularly efficient (compared, say, to a straight procedure call). Surely using thunks must negatively impact the execution time of your code, eh? Well, like so many other programming constructs, the misuse and abuse of thunks can have a negative impact on the execution time of your programs. However, in an appropriate situation thunks can dramatically improve the performance of your programs. In this section we'll explore one situation where the use of thunks produces an amazing performance boost: the calculation of a Fibonacci number.

Earlier in this text there was an example of a Fibonacci number generation program (see "Fibonacci Number Generation" on page 50). As you may recall, the Fibonacci function `fib(n)` is defined recursively for $n \geq 1$ as follows:

```
fib(1) = 1;
fib(2) = 1;
fib( n ) = fib( n-1 ) + fib( n-2 )
```

One problem with this recursive definition for *fib* is that it is extremely inefficient to compute. The number of clock cycles this particular implementation requires to execute is some exponential factor of n . Effectively, as n increases by one this algorithm takes twice as long to execute.

The big problem with this recursive definition is that it computes the same values over and over again. Consider the statement "`fib(n) = fib(n-1) + fib(n-2)`". Note that the computation of `fib(n-1)` also computes `fib(n-2)` (since `fib(n-1) = fib(n-2) + fib(n-3)` for all $n \geq 4$). Although the computation of `fib(n-1)` computes the value of `fib(n-2)` as part of its calculation, this simple recursive definition doesn't save that result, so it must recompute `fib(n-2)` upon returning from `fib(n-1)` in order to complete the calculation of `fib(n)`.

Since the calculation of `Fib(n-1)` generally computes `Fib(n-2)` as well, what would be nice is to have this function return both results simultaneously; that is, not only should `Fib(n-1)` return the Fibonacci number for $n-1$, it should also return the Fibonacci number for $n-2$ as well. In this example, we will use a thunk to store the result of `Fib(n-2)` into a local variable in the Fibonacci function.

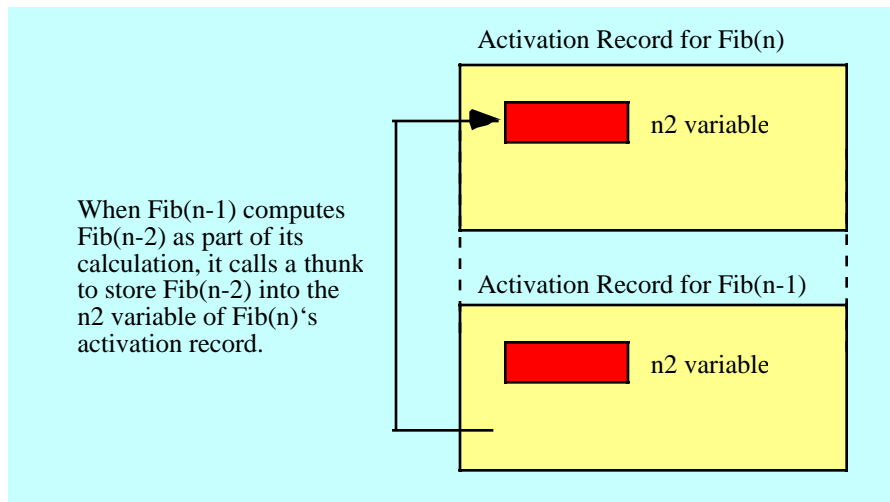


Figure 1.1 Using a Thunk to Set the Fib(n-2) Value in a Different Activation Record

The following program provides two versions of the Fibonacci function: one that uses thunks to pass the Fib(n-2) value back to a previous invocation of the function. Another version of this function computes the Fibonacci number using the traditional recursive definition. The program computes the running time of both implementations and displays the results. Without further ado, here's the code:

```

program fibThunk;
#include( "stdlib.hhf" )

// Fibonacci function using a thunk to calculate fib(n-2)
// without making a recursive call.

procedure fib( n:uns32; nm2:thunk ); nodisplay; returns( "eax" );
var
  n2: uns32;      // A recursive call to fib stores fib(n-2) here.
  t:  thunk;     // This thunk actually stores fib(n-2) in n2.

begin fib;

  // Special case for n = 1, 2. Just return 1 as the
  // function result and store 1 into the fib(n-2) result.

  if( n <= 2 ) then

    mov( 1, eax ); // Return as n-1 value.
    nm2();        // Store into caller as n-2 value.

  else

    // Create a thunk that will store the fib(n-2) value
    // into our local n2 variable.

    thunk  t :=
           #{
           mov( eax, n2 );
           }#;

```



```

    mov( n, eax );
    dec( eax );
    fib( eax, t ); // Compute fib(n-1).

    // Pass back fib(n-1) as the fib(n-2) value to a previous caller.

    nm2();

    // Compute fib(n) = fib(n-1) [in eax] + fib(n-2) [in n2]:
    add( n2, eax );

endif;

end fib;

// Standard fibonacci function using the slow recursive implementation.

procedure slowfib( n:uns32 ); nodisplay; returns( "eax" );
begin slowfib;

    // For n= 1,2 just return 1.

    if( n <= 2 ) then

        mov( 1, eax );

    else

        // Return slowfib(n-1) + slowfib(n-2) as the function result:

        dec( n );
        slowfib( n ); // compute fib(n-1)
        push( eax ); // Save fib(n-1);

        dec( n ); // compute fib(n-2);
        slowfib( n );

        add( [esp], eax ); // Compute fib(n-1) [on stack] + fib(n-2) [in eax].
        add( 4, esp ); // Remove old value from stack.

    endif;

end slowfib;

var
    prevTime:dword[2]; // Used to hold 64-bit result from RDTSC instr.
    qw: qword; // Used to compute difference in timing.
    dummy:thunk; // Used in original calls to fib.

begin fibThunk;

    // "Do nothing" thunk used by the initial call to fib.
    // This thunk simply returns to its caller without doing
    // anything.

    thunk dummy := #{ }#;

```

```

// Call the fibonacci routines to "prime" the cache:

fib( 1, dummy );
slowfib( 1 );

// Okay, compute the times for the two fibonacci routines for
// values of n from 1 to 32:

for( mov( 1, ebx ); ebx < 32; inc( ebx ) ) do

    // Read the time stamp counter before calling fib:
    rdtsc();
    mov( eax, prevTime );
    mov( edx, prevTime[4] );

    fib( ebx, dummy );
    mov( eax, ecx );

    // Read the timestamp counter and compute the approximate running
    // time of the current call to fib:

    rdtsc();
    sub( prevTime, eax );
    sbb( prevTime[4], edx );
    mov( eax, (type dword qw));
    mov( edx, (type dword qw[4]));

    // Display the results and timing from the call to fib:

    stdout.put
    (
        "n=",
        (type uns32 ebx):2,
        " fib(n) = ",
        (type uns32 ecx):7,
        " time="
    );
    stdout.putu64size( qw, 5, ' ' );

    // Okay, repeat the above for the slowfib implementation:

    rdtsc();
    mov( eax, prevTime );
    mov( edx, prevTime[4] );

    slowfib( ebx );
    mov( eax, ecx );
    rdtsc();
    sub( prevTime, eax );
    sbb( prevTime[4], edx );
    mov( eax, (type dword qw));
    mov( edx, (type dword qw[4]));

    stdout.put( " slowfib(n) = ", (type uns32 ecx ):7, " time = " );
    stdout.putu64size( qw, 8, ' ' );
    stdout.newln();

endfor;

```

```
end fibThunk;
```

Program 1.1 Fibonacci Number Generation Using Thunks

This (relatively) simple modification to the Fibonacci function produces a dramatic difference in the run-time of the code. The run-time of the thunk implementation is now well within reason. The following table lists the same run-times of the two functions (thunk implementation vs. standard recursive implementation). As you can see, this small change to the program has made a very significant difference.

Table 1: Running Time of the FIB and SlowFib Functions

n	Fib Execution Time (Thunk Implementation) ^a	SlowFib Execution Time (Recursive Implementation)
1	60	97
2	152	100
3	226	166
4	270	197
5	302	286
6	334	414
7	369	594
8	397	948
9	432	1513
10	473	2421
11	431	3719
12	430	6010
13	467	9763
14	494	15758
15	535	25522
16	564	41288
17	614	66822
18	660	108099
19	745	174920

Table 1: Running Time of the FIB and SlowFib Functions

n	Fib Execution Time (Thunk Implementation) ^a	SlowFib Execution Time (Recursive Implementation)
20	735	283001
21	791	457918
22	886	740894
23	943	1198802
24	919	1941077
25	966	3138466
26	1015	5094734
27	1094	8217396
28	1101	13297000
29	1158	21592819
30	3576 ^b	34927400
31	1315	56370705

a. All times are in CPU cycles as measured via RDTSC on a Pentium II processor.

b. This value was not recorded properly because of OS overhead.

Note that a thunk implementation of the Fibonacci function is not the only way to improve the performance of this function. One could have just as easily (more easily, in fact) passed the address of the local $n2$ variable by reference and had the recursive call store the $\text{Fib}(n-2)$ value directly into the $n2$ variable. For that matter, one could have written an interactive (rather than recursive) solution to this problem that computes the Fibonacci number very efficiently. However, alternate solutions to Fibonacci aside, this example does clearly demonstrate that the use of a thunk in certain situations can dramatically improve the performance of an algorithm.

1.9 Thunks and Artificial Intelligence Code

Although the use of thunks to compute the Fibonacci number in the previous section produced a dramatic performance boost, thunks clearly were not necessary for this operation (indeed, not too many people really need to compute Fibonacci numbers, for that matter). Although this example demonstrates that thunks can improve performance in various situations, it does not demonstrate the need for thunks. After all, there are even more efficient implementations of the Fibonacci function (though nothing quite so dramatic as the difference between Fib and SlowFib, which went from exponential to linear execution time) that do not involve the use of thunks. So although the use of thunks can increase the performance of the Fibonacci function (over the execution time of the standard recursive implementation), the example in the previous section

does not demonstrate the need for thunks since there are better implementations of the Fibonacci function that do not use thunks. In this section we will explore some types of calculations for which thunks present a very good, if not the best, solution.

In the field of Artificial Intelligence (AI from this point forward) researchers commonly use interpreted programming languages such as LISP or Prolog to implement various algorithms. Although you could write an AI program in just about any language, these interpreted languages like LISP and Prolog have a couple of benefits that help make writing AI programs much less difficult. Among a long list of other features, two important features stand out: (1) statements in these languages are first class objects, (2) these languages can defer the evaluation of function parameters until the parameters' values are actually needed (lazy evaluation). Since thunks provide exactly these two features in an HLA program, it should come as no surprise that you can use thunks to implement various AI algorithm in assembly language.

Before going too much farther, you should realize that AI programs usually take advantage of many other features in LISP and Prolog besides the two noted above. Automatic dynamic memory allocation and garbage collection are two big features that many AI programs use, for example. Also, the run-time interpretation of language statements is another big feature (i.e., the user of a program can input a string containing a LISP or Prolog statement and the program can execute this string as part of the program). Although it is certainly possible to achieve all this in an assembly language program, such support built into languages like LISP and Prolog may require (lots of) additional coding in assembly language. So please don't allow this section to trivialize the effort needed to write AI programs in assembly language; writing (good) AI programs is difficult and tools like LISP and Prolog can reduce that effort.

Of course, a major problem with languages like LISP and Prolog is that programs written in these (interpreted) languages tend to run very slow. Some people may argue that there isn't a sufficient difference in performance between programs written in C/C++ and assembly to warrant the extra effort of writing the code in assembly; such a claim is difficult to make about LISP or Prolog programs versus an assembly equivalent. Whereas a well-written assembly program may be only a couple of times faster than a well-written C/C++ program, a well-written assembly program will probably be tens, if not hundreds or thousands, of times faster than the equivalent LISP or Prolog code. Therefore, reworking at least a portion of an AI program written in one of these interpreted languages can produce a very big boost in the performance of the AI application.

Traditionally, one of the big problem areas AI programmers have had when translating their applications to a lower-level language has been the issue of "function calls (statements) as first class objects and the need for lazy evaluation." Traditional third generation programming languages like C/C++ and Pascal simply do not provide these facilities. AI applications that make use of these facilities in languages like LISP or Prolog often have to be rewritten in order to avoid the use of these features in the lower-level languages. Assembly language doesn't suffer from this problem. Oh, it may be difficult to implement some feature in assembly language, but if it can be done, it can be done in assembly language. So you'll never run into the problem of assembly language being unable to implement some feature from LISP, Prolog, or some other language.

Thunks are a great vehicle for deferring the execution of some code until a later time (or, possibly, forever). One application area where deferred execution is invaluable is in a game. Consider a situation in which the current state of a game suggests one of several possible moves a piece could make based on a decision made by an adversary (e.g., a particular chess piece could make one of several different moves depending on future moves by the other color). You could represent these moves as a list of thunks and executing the move by selecting and executing one of the thunks from the list at some future time. You could base the selection of the actual thunk to execute on adversarial moves that occur at some later time.

Thunks are also useful for passing results of various calculations back to several different points in your code. For example, in a multi-player strategy game the activities of one player could be broadcast to a list of interested players by having those other players "register" a thunk with the player. Then, whenever the player does something of interest to those other players, the program could execute the list of thunks and pass whatever data is important to those other players via the thunks.

1.10 Thunks as Triggers

Thunks are also useful as *triggers*. A trigger is a device that fires whenever a certain condition is met. Probably the most common example of a trigger is a database trigger that executes some code whenever some condition occurs in the database (e.g., the entry of an invalid data or the update of some field). Triggers are useful insofar as they allow the use of *declarative programming* in your assembly code. Declarative programming consists of some declarations that automatically execute when a certain condition exists. Such declarations do not execute sequentially or in response to some sort of call. They are simply part of the programming environment and automatically execute whenever appropriate. At the machine level, of course, some sort of call or jump must be made to such a code sequence, but at a higher level of abstraction the code seems to "fire" (execute) all on its own.

To implement declarative programming using triggers you must get in the habit of writing code that always calls a "trigger routine" (i.e., a thunk) at any given point in the code where you would want to handle some event. By default, the trigger code would be an empty thunk, e.g.:

```
procedure DefaultTriggerProc; @nodisplay; @noframe;
begin DefaultTriggerProc;

    // Immediately return to the caller and pop off the environment
    // pointer passed to us (probably a NULL pointer).

    ret(4);

end DefaultTriggerProc;

static
    DefaultTrigger:    thunk: @nostorage;
                     dword 0, &DefaultTriggerProc;
```

The code above consists of two parts: a procedure that corresponds to the default thunk code to execute and a declaration of a default trigger thunk object. The procedure body consists of nothing more than a return that pops the EBP value the thunk invocation pushes and then returns back to the thunk invocation. The default trigger thunk variable contains NULL (zero) for the EBP value and the address of the *DefaultTriggerProc* code as the code pointer. Note that the value we pass as the environment pointer (EBP value) is irrelevant since *DefaultTriggerProc* ignores this value.

To use a trigger, you simply declare a thunk variable like *DefaultTrigger* above and initialize it with the address of the *DefaultTriggerProc* procedure. You will need a separate thunk variable for each trigger event you wish to process; however, you will only need one default trigger procedure (you can initialize all trigger thunks with the address of this same procedure). Generally, these trigger thunks will be global variables so you can access the thunk values throughout your program. Yes, using global variables is often a no-no from a structured point of view, but triggers tend to be global objects that several different procedures share, so using global objects is appropriate here. If using global variables for these thunks offends you, then bury them in a class and provide appropriate accessor methods to these thunks.

Once you have a thunk you want to use as a trigger, you invoke that thunk from the appropriate point in your code. As a concrete example, suppose you have a database function that updates a record in the database. It is common (in database programs) to trigger an event after the update and, possibly, before the update. Therefore, a typical database update procedure might invoke two thunks – one before and one after the body of the update procedure's code. The following code fragment demonstrates how you code do this:

```
static
    preUpdate:        thunk: @nostorage;
                     dword 0, &DefaultTriggerProc;

    postUpdate:       thunk: @nostorage;
                     dword 0, &DefaultTriggerProc;
```

```

procedure databaseUpdate( << appropriate parameters >> );
  << declarations >>
begin databaseUpdate;

  preUpdate(); // Trigger the pre-update event.

  << Body of update procedure >>

  postUpdate(); // Trigger the post-update event.

end databaseUpdate;

```

As written, of course, these triggers don't do much. They call the default trigger procedure that immediately returns. Thus far, the triggers are really nothing more than a waste of time and space in the program. However, since the *preUpdate* and *postUpdate* thunks are variables, we can change their values under program control and redirect the trigger events to different code.

When changing a trigger's value, it's usually a good idea to first preserve the existing thunk data. There isn't any guarantee that the thunk points at the default trigger procedure. Therefore, you should save the value so you can restore it when you're done handling the trigger event (assuming you are writing an event handler that shuts down before the program terminates). If you're setting and restoring a trigger value in a procedure, you can copy the global thunk's value into a local variable prior to setting the thunk and you can restore the thunk from this local variable prior to returning from the procedure:

```

procedure DemoRestoreTrigger;
var
  RestoreTrigger: dword[2];
begin DemoRestoreTrigger;

  // The following three statements "register" a thunk as the
  // "GlobalEvent" trigger:

  mov( (type dword GlobalEvent[0]), RestoreTrigger[0] );
  mov( (type dword GlobalEvent[4]), RestoreTrigger[4] );
  thunk GlobalEvent := #{ <<thunk body >> }#;

  << Body of DemoRestoreTrigger procedure >>

  // Restore the original thunk as the trigger event:

  mov( RestoreTrigger[0], (type dword GlobalEvent[0]) );
  mov( RestoreTrigger[4], (type dword GlobalEvent[4]) );

end DemoRestoreTrigger;

```

Note that this code works properly even if *DemoRestoreTrigger* is recursive since the *RestoreTrigger* variable is an automatic variable. You should always use automatic (VAR) objects to hold the saved values since static objects have only a single instance (which would fail in a multi-threaded environment or if *DemoRestoreTrigger* is recursive).

One problem with the code in the example above is that it replaces the current trigger with a new trigger. While this is sometimes desirable, more often you'll probably want to *chain* the trigger events. That is, rather than having a trigger call the most recent thunk, which returns to the original code upon completion, you'll probably want to call the original thunk you replaced before or after the current thunk executes. This way, if several procedures register a trigger on the same global event, they will all "fire" when the event occurs. The following code fragment shows the minor modifications to the code fragment above needed to pull this off:

```

procedure DemoRestoreTrigger;

```

```

var
  PrevTrigger: thunk;
begin DemoRestoreTrigger;

  // The following three statements "register" a thunk as the
  // "GlobalEvent" trigger:

  mov( (type dword GlobalEvent[0]), (type dword PrevTrigger[0]) );
  mov( (type dword GlobalEvent[4]), (type dword PrevTrigger[4]) );
  thunk GlobalEvent :=
    #{
      PrevThunk();
      <<thunk body >>
    }#;

  << Body of DemoRestoreTrigger procedure >>

  // Restore the original thunk as the trigger event:

  mov( (type dword PrevTrigger[0]), (type dword GlobalEvent[0]) );
  mov( (type dword PrevTrigger[4]), (type dword GlobalEvent[4]) );

end DemoRestoreTrigger;

```

The principal differences between this version and the last is that *PrevTrigger* (a thunk) replaces the *RestoreTrigger* (two double words) variable and the thunk code invokes *PrevTrigger* before executing its own code. This means that the thunk's body will execute *after* all the previous thunks in the chain. If you would prefer to execute the thunks body before all the previous thunks in the chain, then simply invoke the thunk *after* the thunk's body, e.g.,

```

thunk GlobalEvent :=
  #{
    <<thunk body >>
    PrevThunk();
  }#;

```

In practice, most programs set a trigger event once and let a single, global, trigger handler process events from that point forward. However, if you're writing more sophisticated code that enables and disables trigger events throughout, you might want to write a macro that helps automate saving, setting, and restore thunk objects. Consider the following HLA macro:

```

#macro EventHandler( Event, LocalThunk );

  mov( (type dword Event[0]), (type dword LocalThunk[0]) );
  mov( (type dword Event[4]), (type dword LocalThunk[4]) );
  thunk Event :=

#terminator EndEventHandler;

  mov( (type dword LocalThunk[0]), (type dword Event[0]) );
  mov( (type dword LocalThunk[4]), (type dword Event[4]) );

#endmacro;

```

This macro lets you write code like the following:

```

procedure DemoRestoreTrigger;
var
  PrevTrigger: thunk;
begin DemoRestoreTrigger;

```



```

EventHandler( GlobalEvent, PrevTrigger ) // Note: no semicolon here!

    #{
        PrevThunk();
        <<thunk body >>
    }#;

    << Body of DemoRestoreTrigger procedure >>

EndEventHandler;

end DemoRestoreTrigger;

```

Especially note the comment stating that no semicolon follows the *EventHandler* macro invocation. If you study the *EventHandler* macro carefully, you'll notice that macro ends with the first half of a THUNK statement. The body of the *EventHandler.EndEventHandler* statement (macro invocation) must begin with a thunk body declaration that completes the THUNK statement begun in *EventHandler*. If you put a semicolon at the end of the *EventHandler* statement, this will insert the semicolon into the middle of the THUNK statement, resulting in a syntax error. If these syntactical gymnastics bother you, you can always remove the THUNK statement from the macro and require the end user to type the full THUNK statement at the beginning of the macro body. However, saving this extra type is what macros are all about; most users would probably rather deal with remembering not to put a semicolon at the end of the *EventHandler* statement rather than do this extra typing.

1.11 Jumping Out of a Thunk

Because a thunk is a procedure nested within another procedure's body, there are some interesting situations that can arise during program execution. One such situation is jumping out of a thunk and into the surrounding code during the execution of that thunk. Although it is possible to do this, you must exercise great caution when doing so. This section will discuss the precautions you must take when leaving a thunk other than via a RET instruction.

Perhaps the best place to start is with a couple of examples that demonstrate various ways to abnormally exit a thunk. The first thunk in the example below demonstrates a simple JMP instruction while the second thunk in this example demonstrates leaving a thunk via a BREAK statement.

```

procedure ExitThunks;
var
    jmpFrom:thunk;
    breakFrom:thunk;
begin ExitThunks;

    thunk jmpFrom :=
        #{
            // Just jump out of this thunk and back into
            // the ExitThunks procedure:

            jmp XTlabel;
        }#;

    // Execute the thunk above (which winds up jumping to the
    // XTlabel label below:

    jmpFrom();

    XTlabel:

```

```

// Create a loop inside the ExitThunks procedure and
// define a thunk within this loop. Use a BREAK statement
// within the thunk to exit the thunk (and loop).

forever

    thunk breakFrom :=
        #{
            // Break out of this thunk and the surrounding
            // loop via the following BREAK statement:

            break;
        }#;

    // Invoke the thunk (which causes use to exit from the
    // surrounding loop):

    breakFrom();

endfor;

end ExitThunks;

```

Obviously, you should avoid constructs like these in your thunks. The control flow in the procedure above is very unusual, to say the least, and others reading this code will have a difficult time fully comprehending what is going on. Of course, like other structured programming techniques that make programs easier to read, you may discover the need to write code like this under special circumstances. Just don't make a habit of doing this gratuitously.

There is a problem with breaking out of the thunks as was done in the code above: this scheme leaves a bunch of data on the stack (specifically, the thunk's parameter, the return address, and the saved EBP value in this particular example). Had *ExitThunks* pushed some registers on the stack that it needed to preserve, ESP would not be properly pointing at those register upon reaching the end of the function. Therefore, popping these registers off the stack would load garbage into the registers. Fortunately, the HLA standard exit sequence reloads ESP from EBP prior to popping EBP's value and the return address off the stack; this resynchronizes ESP prior to returning from the procedure. However, anything you push on the stack after the standard entry sequence will not be on the top of stack if you prematurely bail out of a thunk as was done in the previous example.

The only reasonable solution is to save a copy of the stack pointer's value in a local variable after you push any important data on the stack. Then restore ESP from this local (automatic) variable before attempting to pop any of that data off the stack. The following implementation of *ExitThunks* demonstrates this principle in action:

```

procedure ExitThunks;
var
    jmpFrom:    thunk;
    breakFrom:  thunk;
    ESPsave:    dword;

begin ExitThunks;

    push( eax );           // Registers we wish to preserve.
    push( ebx );
    push( ecx );
    push( edx );
    mov( esp, ESPsave ); // Preserve ESP's value for return.

    thunk jmpFrom :=
        #{
            << Code, as appropriate, for this thunk >>

```

```

        // Just jump out of this thunk and back into
        // the ExitThunks procedure:

        jmp XLabel;
    }#;

// Execute the thunk above (which winds up jumping to the
// XLabel label below:

jmpFrom();

XLabel:

// Create a loop inside the ExitThunks procedure and
// define a thunk within this loop. Use a BREAK statement
// within the thunk to exit the thunk (and loop).

forever

    thunk breakFrom :=
        #{
            << Code, as appropriate, for this thunk >>

            // Break out of this thunk and the surrounding
            // loop via the following BREAK statement:

            break;
        }#;

    // Invoke the thunk (which causes use to exit from the
    // surrounding loop):

    breakFrom();

endfor;

<< Any other code required by the procedure >>

// Restore ESP's value from ESPsave in case one of the thunks (or both)
// above have prematurely exited, leaving garbage on the stack.

mov( ESPsave, esp );

// Restore the registers and leave:

pop( edx );
pop( ecx );
pop( ebx );
pop( eax );

end ExitThunks;

```

This scheme will work properly because the thunks always set up EBP to point at *ExitThunks*' activation record (this is true even if the program calls these thunks from some other procedures). The *ESPsave* variable must be an automatic (VAR) variable if this code is to work properly in all cases.

1.12 Handling Exceptions with Thunks

Thunks are also useful for passing exception information back to some code in the calling tree when the HLA exception handling code would be inappropriate (e.g., if you don't want to immediately abort the operation of the current code, you just want to pass data back to some previous code in the current call chain). Before discussing how to implement some exception handler with a thunk, perhaps we should discuss why we would want to do this. After all, HLA has an excellent exception handling mechanism – the TRY..ENDTRY and RAISE statements; why not use those instead of processing exceptions manually with thunks? There are two reasons for using thunks to handle exceptions – you might want to bypass the normal exception handling code (i.e., skip over TRY..ENDTRY blocks for a certain event and pass control directly to some fixed routine) or you might want to resume execution after an exception occurs. We'll look at these two mechanisms in this section.

One of the uses for thunks in exception handling code is to bypass any intermediate TRY..ENDTRY statements between the point of the exception and the handler you'd like to use for the exception. For example, suppose you have the following call chain in your program:

```
HasExceptionHandler->MayHaveOne->MayHaveAnother->CausesTheException
```

In this sequence the procedure *CausesTheException* encounters some exceptional condition. Were you to write the code using the standard RAISE and TRY..ENDTRY statements, then the last TRY..ENDTRY statement (that handles the specific exception) would execute its EXCEPT clause and deal with this exception. In the current example, that means that *MayHaveOne* or *MayHaveAnother* could trap and attempt to handle this exception. Using the standard exception handling mechanism, it is very difficult to ensure that *HasExceptionHandler* is the only procedure that responds to this exception.

One way to avoid this problem is to use a thunk to transfer control to *HasExceptionHandler* rather than the RAISE statement. By declaring a global thunk and initializing it within *HasExceptionHandler* to execute the exception handler, you can bypass any intermediate procedures in the call chain and jump directly to *HasExceptionHandler* from the offending code. Don't forget to save ESP's value and restore it if you bail out of the exception handler code inside the thunk and jump directly into the *HasExceptionHandler* code (see "Jumping Out of a Thunk" on page 1299).

Granted, needing to skip over exception handlers is a bit of a synthetic problem that you won't encounter very often in real-life programs. However, the second feature raised above, resuming the original code after handling an exception, is something you may need to do from time to time. HLA's exceptions do not allow you to resume the code that raised the exception, so if you need this capability thunks provide a good solution. To resume the interrupted code when using a thunk, all you have to do is return from the thunk in the normal fashion. If you don't want to resume the original code, then you can jump out of the thunk and into the surrounding procedure code (don't forget to save and restore ESP in that surrounding code, see "Jumping Out of a Thunk" on page 1299 for details). The nice thing about a thunk is that you don't have to decide whether you're going to bail out of the thunk or resume the execution of the original code while writing your program. You can write some code within the thunk to make this decision at run-time.

1.13 Using Thunks in an Appropriate Manner

This chapter presents all sorts of novel uses for thunks. Thunks are really neat and you'll find all kinds of great uses for them if you just think about them for a bit. However, it's also easy to get carried away and use thunks in an inappropriate fashion. Remember, thunks are not only a pointer to a procedure but a pointer to an execution environment as well. In many circumstances you don't need the execution environment pointer (i.e., the pointer to the activation record). In those cases you should remember that you can use a simple procedure pointer rather than a thunk to indirectly call the "thunk" code. A simple indirect call is a bit more efficient than a thunk invocation, so unless you really need all the features of the thunk, just use a procedure pointer instead.

1.14 Putting It All Together

Although thunks are quite useful, you don't see them used in many programs. There are two reasons for this – most high level languages don't support thunks and, therefore, few programmers have sufficient experience using thunks to know how to use this appropriately. Most people learning assembly language, for example, come from a standard imperative programming language background (C/C++, Pascal, BASIC, FORTRAN, etc.) and have never seen this type of programming construct before. Those who are used to programming in languages where thunks are available (or a similar construct is available) tend not to be the ones who learn assembly language.

If you happen to lack the prerequisite knowledge of thunks, you should not write off this chapter as unimportant. Thunks are definitely a programming tool you should be aware of, like recursion, that's really handy in lots of situations. You should watch out for situations where thunks are applicable and use them as appropriate.

We'll see additional uses for thunks in the next chapter on iterators and in the chapter on advanced parameter passing techniques, later in this volume.

Iterators

Chapter Two

2.1 Chapter Overview

This chapter discusses the low-level implementation of iterators. Earlier, this text briefly discussed iterators and the FOREACH loop in the chapter on intermediate procedures (see “Iterators and the FOREACH Loop” on page 843). This chapter will review that information, discuss some uses for iterators, and then present the low-level implementation of this interesting control structure.

2.2 Review of Iterators

An iterator is a cross between a control structure and a function. Although common high level languages do not support iterators, they are present in some very high level languages¹. Iterators provide a combination state machine/function call mechanism that lets a function pick up where it last left off on each new call. Iterators are also part of a loop control structure, with the iterator providing the value of the loop control variable on each iteration.

To understand what an iterator is, consider the following for loop from Pascal:

```
for I := 1 to 10 do <some statement>;
```

When learning Pascal you were probably taught that this statement initializes *i* with one, compares *i* with 10, and executes the statement if *i* is less than or equal to 10. After executing the statement, the *FOR* statement increments *i* and compares it with 10 again, repeating the process over and over again until *i* is greater than 10.

While this description is semantically correct, and indeed, it’s the way that most Pascal compilers implement the FOR loop, this is not the only point of view that describes how the for loop operates. Suppose, instead, that you were to treat the TO reserved word as an operator. An operator that expects two parameters (one and ten in this case) and returns the range of values on each successive execution. That is, on the first call the TO operator would return one, on the second call it would return two, etc. After the tenth call, the TO operator would fail which would terminate the loop. This is exactly the description of an iterator.

In general, an iterator controls a loop. Different languages use different names for iterator controlled loops, this text will just use the name FOREACH as follows:

```
foreach iterator() do
  statements;
endfor;
```

An iterator returns two values: a boolean success or failure value and a function result. As long as the iterator returns success, the FOREACH statement executes the statements comprising the loop body. If the iterator returns failure, the FOREACH loop terminates and executes the next sequential statement following the FOREACH loop’s body.

Iterators are considerably more complex than normal functions. A typical function call involves two basic operations: a call and a return. Iterator invocations involve four basic operations:

- 1) Initial iterator call
- 2) Yielding a value
- 3) Resumption of an iterator

¹. Ada and PL/I support very limited forms of iterators, though they do not support the type of iterators found in CLU, SETL, Icon, and other languages.

4) Termination of an iterator.

To understand how an iterator operates, consider the following short example:

```
iterator range( start:int32; stop:int32 );
begin range;

    forever

        mov( start, eax );
        breakif( eax > stop );
        yield();
        inc( start );

    endfor;

end range;
```

In HLA, iterator calls may only appear in the FOREACH statement. With the exception of the "yield();" statement above, anyone familiar with HLA should be able to figure out the basic logic of this iterator.

An iterator in HLA may return to its caller using one of two separate mechanisms, it can return to the caller by exiting through the "end Range;" statement or it may yield a value by executing the "yield();" statement. An iterator succeeds if it executes the "yield();" statement, it fails if it simply returns to the caller. Therefore, the FOREACH statement will only execute its corresponding statement if you exit an iterator with a "yield();". The FOREACH statement terminates if you simply return from the iterator. In the example above, the iterator returns the values *start..stop* via a "yield();" statement and then the iterator terminates. The loop

```
foreach Range(1,10) do

    stdout.put( (type uns32 eax ), nl );

endfor;
```

is comparable to the code:

```
for( mov( 1, eax ); eax <= 10; inc( eax ) ) do

    stdout.put( (type uns32 eax ), nl );

endfor;
```

When an HLA program first executes the FOREACH statement, it makes an initial call to the iterator. The iterator runs until it executes a "yield();" or it returns. If it executes the "yield();" statement, it returns the value in EAX as the iterator result and it succeeds. If it simply returns, the iterator returns failure and no iterator result. In the current example, the initial call to the iterator returns success and the value one.

Assuming a successful return (as in the current example), the FOREACH statement returns the current result in EAX and executes the FOREACH loop body. After executing the loop body, the FOREACH statement calls the iterator again. However, this time the FOREACH statement resumes the iterator rather than making an initial call. An iterator resumption continues with the first statement following the last "yield();" it executes. In the *range* example, a resumption would continue execution at the "inc(start);" statement. On the first resumption, the *range* iterator would add one to *start*, producing the value two. Two is less than ten (*stop*'s value) so the FOREACH loop would repeat and the iterator would yield the value two. This process would repeat over and over again until the iterator yields ten. Upon resuming after yielding ten, the iterator would increment start to eleven and then return, rather than yield, since this new value is not less than or equal to ten. When the *Range* iterator returns (fails), the FOREACH loop terminates.

2.2.1 Implementing Iterators Using In-Line Expansion

The implementation of an iterator is rather complex. To begin with, consider a first attempt at an assembly implementation of the FOREACH statement above:

```

    push( 1 );    // Manually pass 1 and 10 as parameters.
    push( 10 );
    call Range_initial;
    jc Failure;
ForLp: stdout.put( (type uns32 eax), nl );
    call Range_Resume;
    jnc ForLp;
Failure:

```

Although this looks like a straight-forward implementation project, there are several issues to consider. First, the call to *Range_Resume* above looks simple enough, but there is no fixed address that corresponds to the resume address. While it is certainly true that this *Range* example has only one resume address, in general you can have as many "yield();" statements as you like in an iterator. For example, the following iterator returns the values 1, 2, 3, and 4:

```

iterator OneToFour;
begin OneToFour;

    mov( 1, eax ); yield();
    mov( 2, eax ); yield();
    mov( 3, eax ); yield();
    mov( 4, eax ); yield();

end OneToFour;

```

The initial call would execute the "mov(1, eax);" and "yield();" statements. The first resumption would execute the "mov(2, eax);" and "yield();" statements, the second resumption would execute "mov(3, eax);" and "yield();", etc. Obviously there is no single resume address the calling code can count on.

There are a couple of additional details left to consider. First, an iterator is free to call procedures and functions. If such a procedure or function executes the "yield();" statement then resumption by the FOREACH statement continues execution within the procedure or function that executed the "yield();" ². Second, the semantics of an iterator require all local variables and parameters to maintain their values until the iterator terminates. That is, yielding does not deallocate local variables and parameters. Likewise, any return addresses left on the stack (e.g., the call to a procedure or function that executes the "yield();" statement) must not be lost when a piece of code yields and the corresponding FOREACH statement resumes the iterator. In general, this means you cannot use the standard call and return sequence to yield from or resume to an iterator because you have to preserve the contents of the stack.

While there are several ways to implement iterators in assembly language, perhaps the most practical method is to have the iterator call the loop controlled by the iterator and have the loop return back to the iterator function. Of course, this is counter-intuitive. Normally, one thinks of the iterator as the function that the loop calls on each iteration, not the other way around. However, given the structure of the stack during the execution of an iterator, the counter-intuitive approach turns out to be easier to implement.

Some high level languages support iterators in exactly this fashion. For example, Metaware's Professional Pascal Compiler for the PC supports iterators ³. Were you to create a Professional Pascal code sequence as follows:

```

iterator OneToFour:integer;
begin
    yield 1;

```

2. This requires the use of nested procedures, a subject we will discuss in a later chapter.

3. Obviously, this is a non-standard extension to the Pascal programming language provided in Professional Pascal.

```

    yield 2;
    yield 3;
    yield 4;
end;
```

and call it in the main program as follows:

```
for i in OneToFour do writeln(i);
```

Professional Pascal would completely rearrange your code. Instead of turning the iterator into an assembly language function and calling this function from within the FOR loop body, this code would turn the FOR loop body into a function, expand the iterator in-line (much like a macro) and call the FOR loop body function on each yield. That is, Professional Pascal would probably produce assembly language that looks something like the following:

```

// The following procedure corresponds to the for loop body
// with a single parameter (I) corresponding to the loop
// control variable:

procedure ForLoopCode( i:int32 ); nodisplay;
begin ForLoopCode;

    mov( i, eax );
    stdout.put( i, nl );

end ForLoopCode;

// The follow code would be emitted in-line upon encountering the
// for loop in the main program, it corresponds to an in-line
// expansion of the iterator as though it were a macro,
// substituting a call for the yield instructions:

    ForLoopCode( 1 );
    ForLoopCode( 2 );
    ForLoopCode( 3 );
    ForLoopCode( 4 );
```

This method for implementing iterators is convenient and produces relatively efficient (fast) code. It does, however, suffer from a couple drawbacks. First, since you must expand the iterator in-line wherever you call it, much like a macro, your program could grow large if the iterator is not short and you use it often. Second, this method of implementing the iterator completely hides the underlying logic of the code and makes your assembly language programs difficult to read and understand.

2.2.2 Implementing Iterators with Resume Frames

In-line expansion is not the only way to implement iterators. There is another method that preserves the structure of your program at the expense of a slightly more complex implementation. Several high level languages, including Icon and CLU, use this implementation.

To start with, you will need another stack frame: the *resume frame*. A resume frame contains two entries: a yield return address (that is, the address of the next instruction after the yield statement) and a *dynamic link*, that is a pointer to the iterator's activation record. Typically the dynamic link is just the value in the EBP register at the time you execute the yield statement. This version implements the four parts of an iterator as follows:

- 1) A CALL instruction for the initial iterator call,
- 2) A CALL instruction for the YIELD statement,
- 3) A RET instruction for the resume operation, and

- 4) A RET instruction to terminate the iterator.

To begin with, an iterator will require two return addresses rather than the single return address you would normally expect. The first return address appearing on the stack is the termination return address. The second return address is where the subroutine transfers control on a *yield* operation. The calling code must push these two return addresses upon initial invocation of the iterator. The stack, upon initial entry into the iterator, should look something like Figure 2.1.

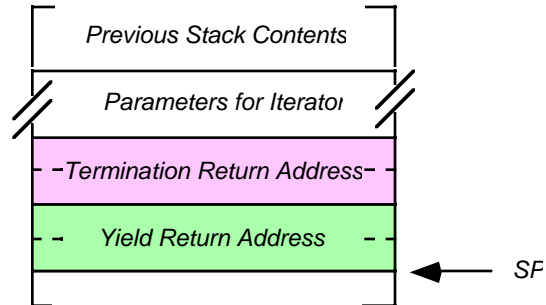


Figure 2.1 Iterator Activation Record

As an example, consider the *Range* iterator presented earlier. This iterator requires two parameters, a starting value and an ending value:

```
foreach range(1,10) do
    stdout.put( i, nl );
endfor;
```

The code to make the initial call to the *range* iterator, producing a stack like the one above, could be the following:

```
push( 1 );          // Push start parameter value.
push( 10 );         // Push stop parameter value.
push( &ForDone );  // Push termination address.
call range;         // Call the iterator.
.
.
.
ForDone:
```

fordone is the first statement immediately following the FOREACH loop, that is, the instruction to execute when the iterator returns failure. The FOREACH loop body must begin with the first instruction following the call to *range*. At the end of the FOREACH loop, rather than jumping back to the start of the loop, or calling the iterator again, this code should just execute a RET instruction. The reason will become clear in a moment. So the implementation of the above FOREACH statement could be the following:

```
push( 1 );          // Push start parameter value.
push( 10 );         // Push stop parameter value.
push( &ForDone );  // Push termination address.
call range;         // Call the iterator.
push( ebp );        // Preserve iterator's ebp value.
mov( [esp+8], ebp ); // Get original EBP value passed to us by range.
stdout.put( i, nl ); // Display i's value.
```

```

    pop( ebp );           // Restore iterator's EBP value.
    ret(4 );             // Return and clean EBP value off stack.
ForDone:

```

Granted, this doesn't look anything at all like a loop. However, by playing some major tricks with the stack, you'll see that this code really does iterate the loop body (*stdout.put*) as intended.

Now consider the *range* iterator itself, here's the (mostly) low-level code to do the job:

```

iterator range( start:int32; stop:int32 ); @nodisplay; @noframe;
begin range;

    push( ebp );        // Standard Entry Sequence
    mov( esp, ebp );

ForEverLbl:

    mov( start, eax );
    cmp( eax, stop );
    jng ForDone;

    yield();
    inc( start );
    jmp ForEverLbl;

ForDone:
    pop( ebp );
    add( 4, esp );
    ret( 8 );

end range;

```

Although this routine is rather short, don't let its size deceive you; it's quite complex. The best way to describe how this iterator operates is to take it a few instructions at a time. The first two instructions are the standard entry sequence for a procedure. Upon execution of these two instructions, the stack looks like that in Figure 2.2.

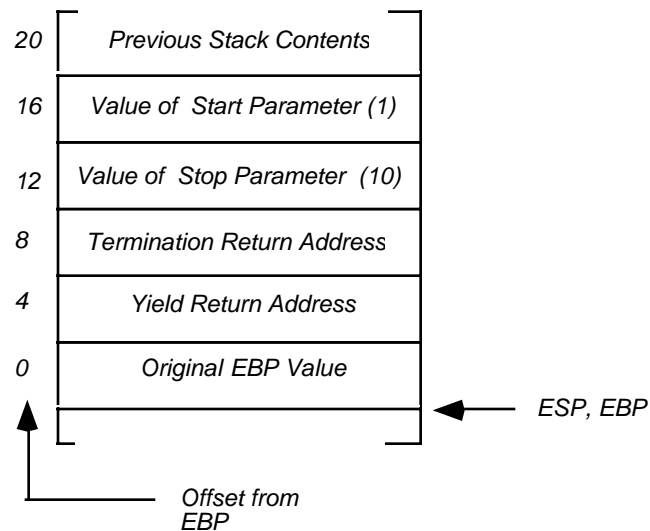


Figure 2.2 Range Activation Record

The next three statements in the *range* iterator, at label *ForEverLbl*, implement the termination test of the loop. When the *start* parameter contains a value greater than the *stop* parameter, control transfers to the *ForDone* label at which point the code pops the value of EBP off the stack, pops the success return address off the stack (since this code will not return back to the body of the iterator loop) and then returns via the termination return address that is immediately above the success return address on the stack. The return instruction also pops the two parameters (eight bytes) off the stack.

The real work of the iterator occurs in the body of the loop. The main question here is "what is this *yield* procedure and what is it doing?". To understand what *yield* is, we must consider what it is that *yield* does. Whenever the iterator executes the *yield* statement, it calls the body of the FOREACH loop that invoked the iterator. Since the body of the FOREACH loop is the first statement following the call to the iterator, it turns out that the iterator's return address points at that body of code. Therefore, the *yield* statement does the unusual operation of calling a subroutine pointed at by the iterator's (success) return address.

Simply calling the body of the FOREACH loop is not all the *yield* call must do. The body of the FOREACH loop is (probably) in a different procedure with its own activation record. That FOREACH loop body may very well access variables that are local to the procedure containing that loop body; therefore, the *yield* statement must also pass the original procedure's EBP value as a parameter so that the loop body can restore EBP to point at the FOREACH loop body's activation record (while, of course, preserving EBP's value within the iterator itself). The caller's EBP value (also known as the dynamic link) was the value the iterator pushes on the stack in the standard entry sequence. Therefore, [EBP+0] will point at this dynamic link value. To properly implement the yield operation, the iterator must emit the following code:

```
push( ebp );           // Save iterator's activation record pointer.
call((type dword [ebp+4])); // Call the return address.
```

The PUSH and CALL instructions build the resume frame and then return control to the body of the FOREACH loop. The CALL instruction is not calling a subroutine. What it is really doing here is finishing off the resume frame (by storing the *yield* resume address into the resume frame) and then it returns control back to the body of the FOREACH loop by jumping indirect through the success return address pushed on the stack by the initial call to the iterator. After the execution of this call, the stack frame looks like that in Figure 2.3.

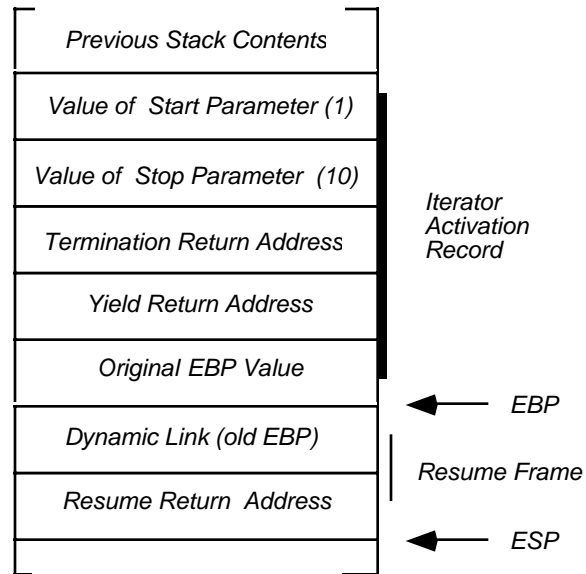


Figure 2.3 Range Resume Record

By convention, the EAX register contains the return value for the iterator. As with functions, EAX is a good place to return the iterator return result.

Immediately after yielding back to the FOREACH loop, the code must reload EBP with the original value prior to the iterator invocation. This allows the calling code to correctly access parameters and local variables in its own activation record rather than the activation record of the iterator. The FOREACH loop body begins by preserving EBP's value (the pointer to iterator's activation record) and then loading EBP with the value pushed on the stack by the *yield* statement. Of course, in this example reloading EBP isn't necessary because the body of the FOREACH loop does not reference any memory locations off the EBP register, but in general you will need to save EBP's value and load EBP with the value pushed on the stack by the *yield* statement.

At the end of the FOREACH loop body the "pop(ebp);" and "ret(4);" instructions restore EBP's value, cleans up the environment pointer passed as a parameter, and resumes the iterator. The RET instruction pops the return address off the stack which returns control back to the iterator immediately after the call emitted by the *yield* statement.

Of course, this is a lot of work to create a piece of code that simply repeats a loop ten times. A simple FOR loop would have been much easier and quite a bit more efficient than the FOREACH implementation described in this section. This section used the *range* iterator because it was easy to show how iterators work using *range*, not because actually implementing *range* as an iterator is a good idea.

Note that HLA does not provide an actual *yield* statement. If you look carefully at this code, and you think back to the last chapter, you will notice that the *yield* "statement" generates exactly the same code as a *think* invocation. Indeed, were you to dump the HLA symbol table when compiling a program containing the *range* iterator, you'd discover that *yield* is actually a local variable of type *think* within the *range* iterator code. The offset of this *think* is zero (from EBP) in the iterator's activation record (see Figure 2.4):

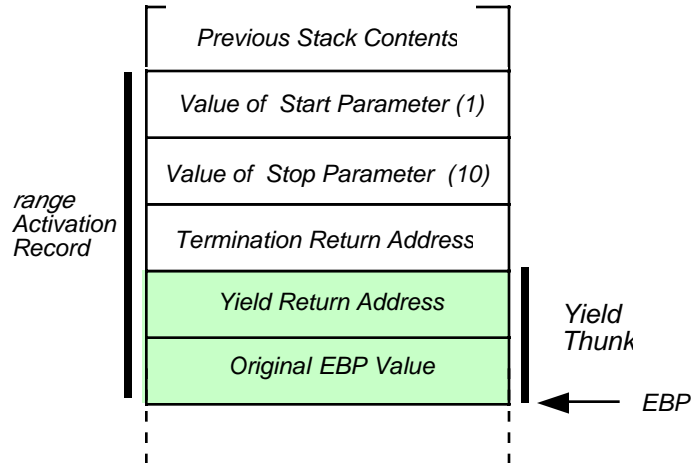


Figure 2.4 Yield Thunk in the *range* Iterator's Activation Record

This thunk value is somewhat unusual. If you look closely, you'll realize that this thunk's value is actually the return address the original call to *range* pushes along with the old EBP value that the *range* iterator code pushes as part of the standard entry sequence. In other words, the call to *range* and the standard entry sequence automatically initializes the *yield* thunk! How's that for elegance? All the HLA compiler has to do to create the *yield* thunk is to automatically create this "yield" symbol and associate *yield* with the address of the old EBP value that the *range* standard entry sequence pushes on the stack.

As you may recall from the chapter on Thunks, the calling sequence for a thunk is the following:

```
push( << Thunk's Environment Pointer >> );
call( << Thunk's Code Pointer >> );
```

In the case of the *yield* thunk in an iterator, the calling sequence looks like this:

```
pushd( [ebp] ); // Pass iterator caller's EBP as parameter.
call( (type dword [ebp+4] )); // Call the FOREACH loop's body.
```

The body of the FOREACH loop, like any thunk, must preserve EBP's value and load EBP with the value pushed on the stack in the code sequence above. Prior to returning (resuming) back to the iterator, the FOREACH loop body must restore the iterator's EBP value and it must remove the environment pointer parameter from the stack upon return. The code given earlier for the FOREACH loop does this:

```
// "FOREACH range( 1, 10) do" statement:

push( 1 ); // Push start parameter value.
push( 10 ); // Push stop parameter value.
push( &ForDone ); // Push termination address.
call range; // Call the iterator.

// FOREACH loop body (a thunk):

push( ebp ); // Preserve iterator's ebp value.
mov( [esp+8], ebp ); // Get original EBP value passed to us by range.
stdout.put( i, nl ); // Display i's value.
pop( ebp ); // Restore iterator's EBP value.
ret(4 ); // Return and clean EBP value off stack.
```

```

    // endfor;

ForDone:

```

Of course, HLA does not make you write this low-level code. You can actually use a FOREACH statement in your program. The above code is the low-level implementation of the following high-level HLA code:

```

foreach range( 1, 10 ) do

    stdout.put( i, nl );

endforeach;

```

The HLA compiler automatically emits the code to preserve and set up EBP at the beginning of the FOREACH loop's body; HLA also automatically emits the code to restore EBP and return to the iterator (removing the environment pointer parameter from the stack).

2.3 Other Possible Iterator Implementations

Thus far, this text has given two different implementations for iterators and the FOREACH loop. Although the resume frame/thunk implementation of the previous section is probably the most common implementation in HLA programs (since the HLA compiler automatically generates this type of code for FOREACH loops and iterators), don't get the impression that this is the only, or best, way to implement iterators and the FOREACH loop. Other possible implementations certainly exist and in some specialized situations some other implementation may offer some advantages. In this section we'll look at a couple of ways to implement iterators and the FOREACH loop.

The standard HLA implementation of iterators uses two separate return addresses for an iterator call: a success/yield address and a failure address. This organization is elegant given the thunk implementation of HLA's FOREACH statement, but there are other ways to return success/failure from an iterator. For example, you could use the value of the carry flag upon return from the iterator call to denote success or failure. Then a call to the iterator might take the following form:

```

ForEachLoopLbl:
    << Push any Necessary Parameters >>
    call iter;
    jc iterFails;

    << Code for the body of the iterator >>

    jmp ForEachLoopLbl;

iterFails:

```

One problem with this approach is that the code reenters the iterator on each iteration of the loop. This means it always passes the same parameters and reconstructs the activation record on each call of the iterator. Clearly, you cannot use this scheme if the iterator needs to maintain state information in its activation record between calls to the iterator. Furthermore, if the iterator yields from different points, then transferring control to the first statement after each yield statement will be a problem. There is a trick you can pull to tell the iterator whether this is the first invocation or some other invocation - pass a special parameter to indicate the first call of an iterator. You can do this as follows:

```

    pushd( 0 ); // Zero indicates the first call to iter.
ForEachLoopLbl:
    << Push Any Other Necessary Parameters >>
    call iter; // iter is a standard HLA procedure, not an iterator.
    jc iterFails;

```



```
<< Code for the body of the iterator >>
```

```
    pushd( 1 ); // One indicates a re-entry into the iterator.
    jmp ForEachLoopLbl;
```

```
iterFails:
```

Notice how this code pushes a zero as the first parameter on the first call to *iter* and it pushes a one on each invocation thereafter.

What if the iterator needs to maintain state information (i.e., local variable values) between calls? Well, the easiest way to handle this using the current scheme is to pass extra parameters by value and use those parameters as the local variables. When the iterator returns success, it should not clean up the parameters on the stack, instead, it will leave them there for the next iteration of the "FOREACH" loop. E.g., consider the following implementation and invocation of the *ArithRange* iterator (this iterator returns the sum of all the values between the *start* and *stop* parameters):

```
// Note: we have to use a procedure, not an iterator, here because we don't
// want HLA generating funny code for us.
//
// "sum" is actually a local variable in which we maintain state information.
// It must contain zero on the first entry to denote the initial entry into
// this code.
```

```
procedure ArithRange( start:uns32; stop:uns32; sum:uns32 );
    @nodisplay; @noframe;
begin ArithRange;

    push( ebp );           // Standard entry sequence.
    mov( esp, ebp );

    mov( start, eax );
    if( eax <= stop ) then

        add( sum, eax ); // Compute arithmetic sum of values.
        mov( eax, sum ); // Save for the next time through and return in EAX.
        pop( ebp );     // Restore pointer to caller's environment.
        cld;            // Indicate success on return.
        ret();          // Note that we don't pop parameters!

    endif;
    pop( ebp );        // Restore pointer to caller's environment.
    stc;               // Indicate return on failure.
    ret( 12 );         // On failure, remove the parameters from the stack.
```

```
end ArithRange;
```

```
.
.
.
```

```
    pushd( 1 );           // Pass the start parameter value.
    pushd( 10 );         // Pass the stop parameter value.
    pushd( 0 );          // Must pass zero in for sum value.
```

```
ForEachLoop:
```

```
    call ArithRange;     // Call our "iterator".
    jc ForEachDone;     // Quit on failure, fall through on success.
```

```
<< Foreach loop body code >>
```

```
    jmp ForEachLoop;
```

ForEachDone:

Notice how this code pushes the parameters on the stack for the first invocation of the *ArithRange* iterator, but it does not push the parameters on the stack on successive iterations of the loop. That's because the code does not remove these parameter values until it fails. Therefore, on each iteration of the loop, the parameter values left on the stack by the previous invocation of *ArithRange* are still on the stack for the next invocation. When the iterator fails, it pops the parameters off the stack so the stack is clean on exit from the "FOREACH" loop above.

If you can spare a register, there is a slightly more efficient way to implement iterators and the FOREACH loop (HLA doesn't use this scheme by default because it promise not to monkey with your register set in the code generation for the high level control structures). Consider the following code that HLA emits for a *yield* call:

```
push( [ebp] );           // Pass FOREACH loop's EBP value as a parameter.
call( [ebp+4] );        // Call the success address.
```

The FOREACH loop body code looks like the following:

```
push( ebp );           // Save iterator's EBP value.
mov( [esp+8], ebp );  // Fetch our EBP value pushed by the iterator.
<< loop body >>
pop( ebp );           // Restore iterator's EBP value.
ret( 4 );             // Resume the iterator and remove EBP value.
```

This code can be improved slightly by preserving and setting the EBP value within the iterator. Consider the following *yield* and FOREACH loop body code:

```
push( ebp );           // Save iterator's EBP value.
mov( [ebp+4], edx );  // Put success address into an available register.
mov( [ebp], ebp );   // Set up FOREACH loop's EBP value.
call edx;             // Call the success address.
pop( ebp );           // Restore our EBP value.
```

Here's the corresponding FOREACH loop body:

```
<< Loop Body >>
ret();
```

These scheme isn't amazingly better than the standard resume frame approach (indeed, it is about the same). But in some situations the fact that the loop body code doesn't have to mess with the stack may be important.

2.4 Breaking Out of a FOREACH Loop

The BREAK, BREAKIF, CONTINUE, and CONTINUEIF statements are active within a FOREACH loop, but there are some problems you must consider if you attempt to break out of a FOREACH loop with a BREAK or BREAKIF statement. In this section we'll look at the problems of prematurely leaving a FOREACH loop.

Keep in mind that an iterator leaves some information on the stack during the execution of the FOREACH loop body. Remember, the iterator doesn't return back to the FOREACH loop in order to execute the loop body; it actually calls the FOREACH loop body. That call leaves a resume frame plus all parameters, local variables, and other information in the iterator's activation record on the stack when it calls the FOREACH loop body. If you attempt to bail out of the FOREACH loop using BREAK, BREAKIF, or (worse still) a conditional or unconditional jump, ESP does not automatically revert back to the value prior to the execution of the FOREACH statement. The HLA generated code can only clean up the stack properly if the iterator returns via the failure address.

Although HLA cannot clean up the stack for you, it is quite possible for you to clean up the stack yourself. The easiest way to do this is to store the value in ESP to a local variable immediately prior to the exe-

cution of the FOREACH statement. Then simply reload ESP from this value prior to prematurely leaving the FOREACH loop. Here's some code that demonstrates how to do this:

```

mov( esp, espSave );
foreach range( 1, 10 ) do

    << foreach loop body >>

    if( some_condition ) then

        mov( espSave, esp );
        break;

    endif;

    << more loop body code >>

endfor;

```

By restoring ESP from the *espSave* variable, this code removes all the activation record information from the stack prior to leaving the FOREACH loop body. Notice the MOV instruction immediately before the FOREACH statement that saves the stack position prior to calling the *range* iterator.

2.5 An Iterator Implementation of the Fibonacci Number Generator

Consider for a moment the Fibonacci number generator from the chapter on Thunks (this is the slow, non-thunk, implementation):

```

// Standard fibonacci function using the slow recursive implementation.

procedure slowfib( n:uns32 ); nodisplay; returns( "eax" );
begin slowfib;

    // For n= 1,2 just return 1.

    if( n <= 2 ) then

        mov( 1, eax );

    else

        // Return slowfib(n-1) + slowfib(n-2) as the function result:

        dec( n );
        slowfib( n ); // compute fib(n-1)
        push( eax ); // Save fib(n-1);

        dec( n ); // compute fib(n-2);
        slowfib( n );

        add( [esp], eax ); // Compute fib(n-1) [on stack] + fib(n-2) [in eax].
        add( 4, esp ); // Remove old value from stack.

    endif;

```

```
end slowfib;
```

Program 2.1 Recursive Implementation of the Fibonacci Number Generator

This particular function generates the n^{th} Fibonacci number by computing the first through the n^{th} the Fibonacci number. If you wanted to generate a sequence of Fibonacci numbers, you could use a FOR loop as follows:

```
for( mov( 1, ebx ); ebx < n; inc( ebx ) ) do

    slowfib( ebx );
    stdout.put( "Fib(", (type uns32 ebx), ") = ", (type uns32 eax), nl );

endfor;
```

True to its name, this implementation of *slowfib* runs quite slowly as n gets larger. The reason this function takes so much time (as pointed out in the chapter on Thunks) is that each recursive call recomputes all the previous Fibonacci numbers. Given the $(n-1)^{\text{st}}$ and $(n-2)^{\text{nd}}$ Fibonacci numbers, computing the n^{th} Fibonacci number is trivial and very efficient – you simply add the previous two values together. The efficiency loss occurs when the recursive implementation computes the same value over and over again. The chapter on Thunks described how to eliminate this recomputation by passing the computed values to other invocations of the functions. This allows the Fibonacci function to compute the n^{th} Fibonacci number in n units of time rather than 2^n units of time, a dramatic improvement. However, since each call to the (efficient) Fibonacci generator requires n units of time to compute its result, the loop above (which repeats n times) requires approximately n^2 units of time to run. This does not seem reasonable since it clearly takes only a few instructions to compute a new Fibonacci number given the previous two values; that is, we should be able to compute the n^{th} Fibonacci number in n units of time. Iterators provide a trivial way to implement a Fibonacci number generator that generates a sequence of n Fibonacci numbers in n units of time. The following program demonstrates how to do this.

```
program fibIter;
#include( "stdlib.hhf" )

// Fibonacci function using a thunk to calculate fib(n-2)
// without making a recursive call.

procedure fib( n:uns32; nm2:thunk ); nodisplay; returns( "eax" );
var
    n2: uns32;      // A recursive call to fib stores fib(n-2) here.
    t:  thunk;     // This thunk actually stores fib(n-2) in n2.

begin fib;

    // Special case for n = 1, 2. Just return 1 as the
    // function result and store 1 into the fib(n-2) result.

    if( n <= 2 ) then

        mov( 1, eax ); // Return as n-1 value.
        nm2();        // Store into caller as n-2 value.

    else

        // Create a thunk that will store the fib(n-2) value
        // into our local n2 variable.
```

```

thunk  t :=
      #{
        mov( eax, n2 );
      }#;

mov( n, eax );
dec( eax );
fib( eax, t ); // Compute fib(n-1).

// Pass back fib(n-1) as the fib(n-2) value to a previous caller.

nm2();

// Compute fib(n) = fib(n-1) [in eax] + fib(n-2) [in n2]:
add( n2, eax );

endif;

end fib;

// Standard fibonacci function using the slow recursive implementation.
procedure slowfib( n:uns32 ); nodisplay; returns( "eax" );
begin slowfib;

  // For n= 1,2 just return 1.

  if( n <= 2 ) then

    mov( 1, eax );

  else

    // Return slowfib(n-1) + slowfib(n-2) as the function result:

    dec( n );
    slowfib( n ); // compute fib(n-1)
    push( eax ); // Save fib(n-1);

    dec( n ); // compute fib(n-2);
    slowfib( n );

    add( [esp], eax ); // Compute fib(n-1) [on stack] + fib(n-2) [in eax].
    add( 4, esp ); // Remove old value from stack.

  endif;

end slowfib;

// FibNum-
//
// Iterator that generates all the fibonacci numbers between 1 and n.

iterator FibNum( n:uns32 ); nodisplay;
var
  Fibn_1: uns32; // Holds Fib(n-1) for a given n.

```

```

Fibn_2: uns32;      // Holds Fib(n-2) for a given n.
CurFib: uns32;    // Current index into fib sequence.

begin FibNum;

  mov( 1, Fibn_1 ); // Initialize these guys upon initial entry.
  mov( 1, Fibn_2 );
  mov( 1, eax );    // Fib(0) = 1
  yield();
  mov( 1, eax );    // Fib(1) = 1;
  yield();
  mov( 2, CurFib );
  forever

    mov( CurFib, eax ); // Compute sequence up to the nth #.
    breakif( eax > n );
    mov( Fibn_2, eax ); // Compute this result.
    add( Fibn_1, eax );

    // Recompute the Fibn_1 and Fibn_2 values:

    mov( Fibn_1, Fibn_2 );
    mov( eax, Fibn_1 );

    // Return current value:

    yield();

    // Next value in sequence:

    inc( CurFib );

  endfor;

end FibNum;

var
  prevTime:dword[2]; // Used to hold 64-bit result from RDTSC instr.
  qw: qword;        // Used to compute difference in timing.
  dummy:thunk;      // Used in original calls to fib.

begin fibIter;

  // "Do nothing" thunk used by the initial call to fib.
  // This thunk simply returns to its caller without doing
  // anything.

  thunk dummy := #{ }#;

  // Call the fibonacci routines to "prime" the cache:

  fib( 1, dummy );
  slowfib( 1 );
  foreach FibNum( 1 ) do
  endfor;

```

```

// Okay, compute the running times for the three fibonacci routines to
// generate a sequence of n fibonacci numbers where n ranges from
// 1 to 32:

for( mov( 1, ebx ); ebx < 32; inc( ebx ) ) do

    // Emit the index:

    stdout.put( (type uns32 ebx):2, stdio.tab );

    // Compute the # of cycles needed to compute the Fib via iterator:

    rdtsc();
    mov( eax, prevTime );
    mov( edx, prevTime[4] );

    foreach FibNum( ebx ) do

    endfor;

    rdtsc();
    sub( prevTime, eax );
    sbb( prevTime[4], edx );
    mov( eax, (type dword qw));
    mov( edx, (type dword qw[4]));

    stdout.putu64Size( qw, 4, ' ' );
    stdout.putc( stdio.tab );

    // Read the time stamp counter before calling fib:

    rdtsc();
    mov( eax, prevTime );
    mov( edx, prevTime[4] );

    for( mov( 1, ecx ); ecx <= ebx; inc( ecx ) ) do

        fib( ecx, dummy );

    endfor;

    // Read the timestamp counter and compute the approximate running
    // time of the current call to fib:

    rdtsc();
    sub( prevTime, eax );
    sbb( prevTime[4], edx );
    mov( eax, (type dword qw));
    mov( edx, (type dword qw[4]));

    // Display the results and timing from the call to fib:

    stdout.putu64Size( qw, 10, ' ' );
    stdout.putc( stdio.tab );

    // Okay, repeat the above for the slowfib implementation:

```

```

rdtsc();
mov( eax, prevTime );
mov( edx, prevTime[4] );

for( mov( 1, ecx ); ecx <= ebx; inc( ecx ) ) do

    slowfib( ebx );

endfor;

rdtsc();
sub( prevTime, eax );
sbb( prevTime[4], edx );
mov( eax, (type dword qw));
mov( edx, (type dword qw[4]));

stdout.putu64Size( qw, 10, ' ' );
stdout.newln();

endfor;

end fibIter;

```

Program 2.2 Fibonacci Iterator Example Program

The important concept here is that the *FibNum* iterator maintains its state across calls. In particular, it keeps track of the current iteration and the previous two Fibonacci values. Therefore, the iterator takes very little time to compute the result of each number in the sequence. This is far more efficient than either Fibonacci number generator from the chapter on Thunks, as the following table attests.

Table 1: CPU Cycle Times for Various Fibonacci Implementations

n	Iterator Implementation	Thunk Implementation	Recursive Implementation
1	156	233	98
2	148	98	77
3	178	221	271
4	193	376	399
5	213	509	879
6	213	712	1758
7	233	919	3493
8	252	1166	6531
9	271	1460	12568

Table 1: CPU Cycle Times for Various Fibonacci Implementations

n	Iterator Implementation	Thunk Implementation	Recursive Implementation
10	290	1759	22871
11	308	2139	40727
12	331	2503	71986
13	349	2938	126443
14	372	3341	220673
15	380	3877	382364
16	402	4326	660506
17	417	4977	1160660
18	452	5528	1954253
19	487	6222	3322819
20	479	6840	5685066
21	524	7691	9621772
22	545	8313	16339720
23	576	9292	27709571
24	599	10029	47036825
25	616	11274	80102556
26	650	12348	132583731
27	653	13172	222580780
28	683	14339	374788752
29	694	15394	627559062
30	722	16363	1054201515
31	732	17727	1756744511

2.6 Iterators and Recursion

It is completely possible, and sometimes very useful, to recursively call an iterator. In this section we'll explore the syntax for this and present a couple of useful recursive iterators.

Although there is nothing stopping you from manually calling an iterator with the CALL instruction, the only valid (high level syntax) invocation of an iterator is via the FOREACH statement. Therefore, to recursively call an iterator, that iterator must contain a FOREACH loop to recursively call itself.

Iterators are especially useful for traversing tree and graph data structures. Some of the best (and most efficient) examples of recursive iterators are those that traverse such structures. Unfortunately, this text does not assume the prerequisite knowledge of such data structures, so it cannot use such examples to demonstrate recursive iterators. Nevertheless, it's worth mentioning this fact here because if you are familiar with graph traversal algorithms (or will be learning them in the future) you should consider using iterators for this purpose.

One useful iterator that doesn't require a tremendous amount of prerequisite knowledge is the traversal of a binary search tree implemented within an array. We won't go into the details of what a binary search tree is or why you would use it here other than to describe some properties of that tree. A binary search tree, implemented as an array, is a data structure that allows one to quickly search for some value within the structure. The values are arranged in the tree such that after each comparison you can eliminate half of the possible values with a single comparison. As a result, if the array contains n items, you can locate a particular item of interest in $\log_2 n$ units of time. To achieve this efficient search time, you have to arrange the data in the array in a particular fashion and then use a specific algorithm when searching through the tree. For the sake of our example, we'll assume that the data in the array is sorted (that is, $a[0] < a[1] < a[2] < \dots < a[n-1]$ for some definition of "less than"). The binary search algorithm then takes the following form:

1. Set $i = n$
2. Set $j = i / 2$ (integer/truncating division)
3. Quit if $i=j$ (failed to find value in the search tree).
4. Compare the *key* value (the one you're searching for) against $a[i]$.
5. If $\text{key} > a[i]$ then set $j = (i - j + 1)/2 + j$
else set $i = j$ and then $j = i/2$
6. Go to step 3.

How this works and what it does is irrelevant here. What is important to this section is the arrangement of the data in the array that forms the binary search tree (specifically, the sorted nature of the data). Of course, if we wanted to generate a list of numbers in the sorted order, that would be especially trivial, all we would have to do is step through the array one element at a time. Suppose, however, that we wanted to generate a list of median values in this array. That is, the first value to generate would be the median of all the values, the second and third values would be the two median values of the array "slice" on either side of the original median. The next four values would be the medians of the array slices around the previous two medians and so on. If you're wondering what good such a sequence could be, well, were we to store this sequence into successive elements of an array, we could develop a binary search algorithm that is a little bit faster than the algorithm above (faster by some multiplicative constant). Hence, by running this iterator over the sorted data, we can come up with a slightly faster searching algorithm.

The following is the iterator that generates this particular sequence:

```

program RecIter;
#include( "stdlib.hhf" )

const
  MaxData := 17;      // # of data items to process.

type
  AryType: uns32[ MaxData ];

static

  // Here is the sorted data we'll process.  For simplicity, we'll just

```

```

// fill the array with 1..MaxData at indices 0..MaxData-1.

SortedData: AryType :=
[
  // Fill this table with the values 1..MaxData:

  ?i := 1;
  #while( i < MaxData )

      i,
      ?i := i + 1;

  #endwhile
  i
];

/*****
/*
/* MedianVal iterator-
/*
/* Given a sorted array, this iterator yields the median value,
/* then it recursively yields a list of median values for the
/* array slice consisting of the array elements whose index is
/* less than the median value. Finally, it yields the list of
/* median values for the array slice built from the array elements
/* whose indices are greater than the median element.
/*
/* Inputs:
/* Ary-
/*     The array whose elements we are to process.
/*
/* start-
/*     Starting index for the array slice.
/*
/* last-
/*     Ending index (plus one) for the array slice.
/*
/* Yields:
/* A list of median values in EAX (one median value on
/* each iteration of the corresponding FOREACH loop).
/*
/* Notes:
/* This iterator wipes out EBX.
/*
*****/

iterator MedianVal( var Ary: AryType; start:uns32; last:uns32 ); nodisplay;
var
  median: uns32;
begin MedianVal;

  mov( last, eax );
  sub( start, eax );
  if( @a ) then

    shr( 1, eax );           // Compute half the size.
    add( start, eax );      // Compute median index.
    mov( eax, median );    // Save for later.

```

```

mov( Ary, ebx );           // Compute address of median element.
mov( [ebx][eax*4], eax ); // Get median element.
yield();

// Recursively yield the medians for the elements at indices below
// the element we just yielded. Do this by recursively calling
// the iterator to generate the list and then yield whatever value
// the iterator returns.

foreach MedianVal( Ary, start, median ) do

    yield();

endfor;

// Recursively yield the medians for the array slice whose indices
// are greater than the current array element. Note that we don't
// include the median value itself in this list as we already
// returned that value above.

mov( median, eax );
inc( eax );
foreach MedianVal( Ary, eax, last ) do

    yield();

endfor;

endif;

end MedianVal;

// Main program that tests the functionality of this iterator.

begin RecIter;

    foreach MedianVal( SortedData, 0, MaxData ) do

        stdout.put( "Value = ", (type uns32 eax), nl );

    endfor;

end RecIter;

```

Program 2.3 Recursive Iterator to Rearrange Data for a Binary Search

The program above produces the following output:

```

Value = 9
Value = 5
Value = 3
Value = 2
Value = 1
Value = 4
Value = 7
Value = 6
Value = 8

```

```
Value = 14  
Value = 12  
Value = 11  
Value = 10  
Value = 13  
Value = 16  
Value = 15  
Value = 17
```

2.7 Calling Other Procedures Within an Iterator

It is perfectly legal to call other procedures from an iterator. However, unless that procedure is nested within the iterator (see “Lexical Nesting” on page 1375), you cannot yield from that procedure using the iterator’s *yield* thunk unless you pass the thunk as a parameter to the other procedure. Other than the fact that you must remember that there is additional information on the stack, calling a procedure from an iterator is really no different than calling an iterator from any other procedure.

2.8 Iterators Within Classes

You can declare an iterator within a class. Iterators are called via the class’ virtual method table, just like methods. This means that you can override an iterator at run-time. See the chapter on classes for more details.

2.9 Putting It Altogether

This chapter provides a brief introduction to the low-level implementation of iterators and then discusses several different ways that you may use iterators in your programs. Although iterators are not as familiar as other program units, they are quite useful in many important situations. You should try to use iterators in your programs wherever appropriate, even if you’re not familiar with iterators from your high level language experiences.

Coroutines and Generators

Chapter Three

3.1 Chapter Overview

This chapter discusses two special types of program units known as coroutines and generators. A coroutine is similar to a procedure and a generator is similar to a function. The principle difference between these program units and procedures/functions is that the call/return mechanism is different. Coroutines are especially useful for multiplayer games and other program flow where sections of code "take turns" executing. Generators, as their name implies, are useful for generating a sequence of values; in many respects generators are quite similar to HLA's iterators without all the restrictions of the iterators (i.e., you can only use iterators within a FOREACH loop).

3.2 Coroutines

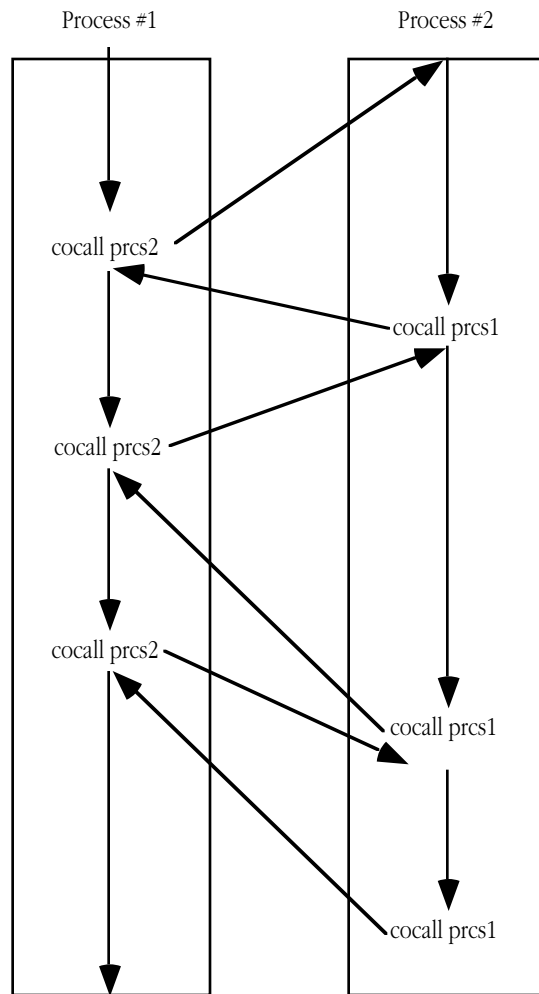
A common programming paradigm is for two sections of code to swap control of the CPU back and forth while executing. There are two ways to achieve this: preemptive and cooperative. In a preemptive system, two or more *processes* or *threads* take turns executing with the *task switch* occurring independently of the executing code. A later volume in this text will consider preemptive multitasking, where the Operating System takes responsibility for interrupting one task and transferring control to some other task. In this chapter we'll take a look at coroutines that explicitly transfer control to another section of the code¹

When discussing coroutines, it is instructive to review how HLA's iterators work, since there is a strong correspondence between iterators and coroutines. Like iterators, there are four types of entries and returns associated with a coroutine:

- Initial entry. During the initial entry, the coroutine's caller sets up the stack and otherwise initializes the coroutine.
- Cocal to another coroutine / coreturn to the previous coroutine.
- Coreturn from another coroutine / cocal to the current coroutine.
- Final return from the coroutine (future calls require reinitialization).

A *cocal* operation transfers control between two coroutines. A cocal is effectively a call and a return instruction all rolled into one operation. From the point of view of the process executing the cocal, the cocal operation is equivalent to a procedure call; from the point of view of the processing being called, the cocal operation is equivalent to a return operation. When the second process cocal the first, control resumes *not at the beginning of the first process*, but immediately after the last cocal operation from that coroutine (this is similar to returning from a FOREACH loop after a *yield* operation). If two processes execute a sequence of mutual cocal, control will transfer between the two processes in the following fashion:

1. The term "cooperative" in this chapter doesn't not imply the use of that oxymoronic term "cooperative multitasking" that Microsoft and Apple used before they got their operating system acts together. Cooperative in this chapter means that two blocks of code explicitly pass control between one another. In a multiprogramming system (the proper technical term for "cooperative multitasking" the operating system still decides which program unit executes after some other thread of execution voluntarily gives up the CPU.

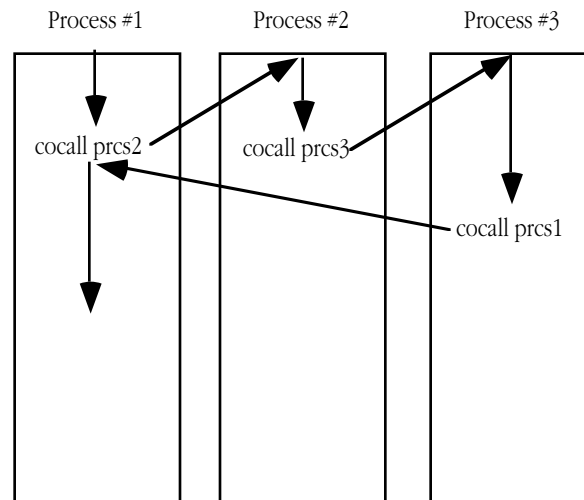


Cocall Sequence Between Two Processes

Figure 3.1 Cocall Sequence

Cocalls are quite useful for games where the “players” take turns, following different strategies. The first player executes some code to make its first move, then cocalls the second player and allows it to make a move. After the second player makes its move, it cocalls the first process and gives the first player its second move, picking up immediately after its cocall. This transfer of control bounces back and forth until one player wins.

Note, by the way, that a program may contain more than two coroutines. If coroutine one cocalls coroutine two, and coroutine two cocalls coroutine three, and then coroutine three cocalls coroutine one, coroutine one picks up immediately in coroutine one after the cocall it made to coroutine two.



Cocalls Between Three Processes

Figure 3.2 Cocalls Between Three Processes

Since a `cocall` effectively *returns* to the target coroutine, you might wonder what happens on the *first* `cocall` to any process. After all, if that process has not executed any code, there is no “return address” where you can resume execution. This is an easy problem to solve, we need only initialize the return address of such a process to the address of the first instruction to execute in that process.

A similar problem exists for the stack. When a program begins execution, the main program (coroutine one) takes control and uses the stack associated with the entire program. Since each process must have its own stack, where do the other coroutines get their stacks? There is also the question of “how much space should one reserve for each stack?” This, of course, varies with the application. If you have a simple application that doesn’t use recursion or allocate any local variables on the stack, you could get by with as little as 256 bytes of stack space for a coroutine. On the other hand, if you have recursive routines or allocate storage on the stack, you will need considerably more space. But this is getting a little ahead of ourselves, how do we create and call coroutines in the first place?

HLA does not provide a special syntax for coroutines. Instead, the HLA Standard Library provides a class with set of procedures and methods (in the `coroutines` library module) that lets you turn any procedure (or set of procedures) into a coroutine. The name of this class is `coroutine` and whenever you want to create a coroutine object, you need to declare a variable of type `coroutine` to maintain important state information about that coroutine’s thread of execution. Here are a couple of typical declarations that might appear within the VAR section of your main program:

```
var
  FirstPlayer: pointer to coroutine;
  OtherPlayer: coroutine;
```

Note that a coroutine variable is not the coroutine itself. Instead, the coroutine variable keeps track of the machine state when you switch between the declared coroutine and some other coroutine in the program (including the main program, which is a special case of a coroutine). The coroutine’s “body” is a procedure that you write independently of the coroutine variable and associate with that coroutine object.

The `coroutine` class contains a constructor that uses the conventional name `coroutine.create`. This constructor requires two parameters and has the following prototype:

```
procedure coroutine.create( stacksize:dword; body:procedure );
```

The first parameter specifies the size (in bytes) of the stack to allocate for this coroutine. The construction will allocate storage for the new coroutine in the system's heap using dynamic allocation (i.e., *malloc*). As a general rule, you should allocate at least 256 bytes of storage for the stack, more if your coroutine requires it (for local variables and return addressees). Remember, the system allocates all local variables in the coroutine (and in the procedures that the coroutine calls) on this stack; so you need to reserve sufficient space to accommodate these needs. Also note that if there are any recursive procedures in the coroutine's thread of execution, you will need some additional stack space to handle the recursive calls.

The second parameter is the address of the procedure where execution begins on the first *cocall* to this coroutine. Execution begins with the first executable statement of this procedure. If the procedure has some local variables, the procedure must build a stack frame (i.e., you shouldn't specify the *NOFRAME* procedure option). Procedures you execute via a *cocall* should never have any parameters (the calling code will not properly set up those parameters and references to the parameter list may crash the machine).

This constructor is a conventional HLA class constructor. On entry, if *ESI* contains a non-null value, the constructor assumes that it points at a coroutine class object and the constructor initializes that object. On the other hand, if *ESI* contains *NULL* upon entry into the constructor, then the constructor allocates new storage for a coroutine object on the heap, initializes that object, and returns a pointer to the object in the *ESI* register. The examples in this chapter will always assume dynamic allocation of the coroutine object (i.e., we'll use pointers).

To transfer control from one coroutine (including the main program) to another, you use the *coroutine.cocall* method. This method, which has no parameters, switches the thread of execution from the current coroutine to the coroutine object associated with the method. For example, if you're in the main program, the following two *cocalls* transfer control to the *FirstPlayer* and then the *OtherPlayer* coroutines:

```
FirstPlayer.cocall();
OtherPlayer.cocall();
```

There are two important things to note here. First, the syntax is not quite the same as a procedure call. You don't use *cocall* along with some operand that specifies which coroutine to transfer to (as you would if this were a *CALL* instruction); instead, you specify the coroutine object and invoke the *cocall* method for that object. The second thing to keep in mind is that these are coroutine transfers, not subroutine calls; therefore, the *FirstPlayer* coroutine doesn't necessarily return back to this code sequence. *FirstPlayer* could transfer control directly to *OtherPlayer* or some other coroutine once it finishes whatever tasks it's working on. Some coroutine has to explicitly transfer control to the thread of execution above in order for it to (directly) transfer control to *OtherPlayer*.

Although coroutines are more general than procedures and don't have to use the call/return semantics, it is perfectly possible to simulate and call and return exchange between two coroutines. If one coroutine calls another and that second coroutine *cocalls* the first, you get semantics that are very similar to a call/return (of course, on the next call to the second coroutine control resumes after the *cocall*, not at the start of the coroutine, but we will ignore that difference here). The only problem with this approach is that it is not general: both coroutines have to be aware of the other. Consider a cooperating pair of coroutines, *master* and *slave*. The *master* coroutine corresponds to the main program and the *slave* coroutine corresponds to a procedure that the main program calls. Unfortunately, *slave* is not general purpose like a standard procedure because it explicitly calls the *master* coroutine (at least, using the techniques we've seen thus far). Therefore, you cannot call it from an arbitrary coroutine and expect control to transfer back to that coroutine. If *slave* contains a *cocall* to the *master* coroutine it will transfer control there rather than back to the "calling" coroutine. Although these are the semantics we expect of coroutines, it would sometimes be nice if a coroutine could return to whomever invoked it without explicitly knowing who invoked it. While it is possible to set up some coroutine variables and pass this information between coroutines, the HLA Standard Library Coroutines Module provides a better solution: the *coret* procedure.

On each call to a coroutine, the coroutine run-time support code remembers the last coroutine that made a *cocall*. The *coret* procedure uses this information to transfer control back to the last coroutine that made a *cocall*. Therefore, the *slave* coroutine above can execute the *coret* procedure to transfer control back to whomever called it without knowing who that was.

Note that the *coret* procedure is not a member of the *coroutine* class. Therefore, you do not preface the call with "coroutine." You invoke it directly:

```
coret();
```

Another important issue to keep in mind with *coret* is that it only keeps track of the last cocall. It does not maintain a stack of coroutine "return addresses" (there are several different stacks in use by coroutines, on which one does it keep this information?). Therefore, you cannot make two cocalls in a row and then execute two corets to return control back to the original coroutine. If you need this facility, you're going to need to create and maintain your own stack of coroutine calls. Fortunately, the need for something like this is fairly rare. You generally don't use coroutines as though they were procedures and in the few instances where this is convenient, a single level of return address is usually sufficient.

By default, every HLA main program is a coroutine. Whenever you compile an HLA program (as opposed to a UNIT), the HLA compiler automatically inserts two pieces of extra code at the beginning of the main program. The first piece of extra code initializes the HLA exception handling system, the second sets up a coroutine variable for the main program. The purpose of this variable is to allow other coroutines to transfer control back to the main program; after all, if you transfer control from one coroutine to another using a statement like *VarName.cocall*, you're going to need a coroutine variable associated with the main program in order to cocall the main program. HLA automatically creates and initializes this variable when execution of the main program begins. So the only question is, "how do you gain access to this variable?"

The answer is simply, really. Whenever you include the "coroutines.hhf" header file (or "stdlib.hhf" which automatically includes "coroutines.hhf") HLA declares a static coroutine variable for you that is associated with the main program's coroutine object. That declaration looks something like the following²:

```
static MainPgm:coroutine; external( "<<external name for MainPgm>>" );
```

Therefore, to transfer control from one coroutine to the main program's coroutine, you'd use a cocall like the following:

```
MainPgm.cocall();
```

The last method of interest to us in the *coroutine* class is the *coroutine.cofree* method. This is the destructor for the *coroutine* class. Calling this method frees up the stack storage associated with the coroutine and cleans up other state information associated with that coroutine. A typical call might look like the following:

```
OtherPlayer.cofree();
```

Warning: do not call the *cofree* method from within the coroutine you are freeing up. There is no guarantee that the stack and coroutine state variables remain valid for a given coroutine after you call *cofree*. Generally, it is a good idea to call the *cofree* method in the same code that originally created the coroutine via the *coroutine.create* call. It goes without saying that you must not call a coroutine after you've destroyed it via the *cofree* method call.

Remember that the *coret* procedure call is really a special form of *cocall*. Therefore, you need to be careful about executing *coret* after calling the *cofree* method as you may wind up "returning" to the coroutine you just destroyed.

After you call the *cofree* method, it is perfectly reasonable to create a new coroutine using that same coroutine variable by once again calling the *coroutine.create* procedure. However, you should always ensure that you call the *cofree* method prior to calling *coroutine.create* or the stack space allocated in the original call will be lost in the system (i.e., you'll create a memory leak).

This is very important: you never "return" from a coroutine using a RET instruction (e.g., by "falling off the end of the procedure) or via the HLA EXIT/EXITIF statement. The only legal ways to "return" from a coroutine are via the *cocall* and *coret* operations. If you RETURN or EXIT from a coroutine, that coroutine enters a special mode that rejects any future cocalls and immediately returns control back to whomever

2. The external name doesn't appear here because it is subject to change. See the coroutines.hhf header file if you need to know the actual external name for some reason.

cocalled it in the first place. Most coroutines contain an infinite loop that transfers control back to the start of the coroutine to repeat whatever function they perform once they complete all the code in the coroutine. You will probably want to implement this functionality in your coroutines as well.

3.3 Parameters and Register Values in Coroutine Calls

As you've probably noticed, coroutine calls via `cocall` don't support generic parameters for transferring data between two coroutines. There are a couple of reasons for this. First of all, passing parameters to a coroutine is difficult because we typically use the stack to pass parameters and coroutines all use different stacks. Therefore, the parameters one coroutine passes to another won't be on the correct stack (and, therefore, inaccessible) when the second coroutine continues execution. Another problem with passing parameters between coroutines is that a typical coroutine has several entry points (immediately after each `cocall` in that coroutine). Nevertheless, it is often important to communicate information between coroutines. We will explore ways to do that in this section.

If we can pass parameters on the stack, that basically leaves registers and global memory locations³. Registers are the easy and obvious solution. However, keep in mind that you have a limited set of registers available so you can't pass much data between coroutines in the registers. On the other hand, you can pass a pointer to a block of data in a register (see "Passing Parameters via a Parameter Block" on page 1353 for details).

Another place to pass parameters between coroutines is in global memory locations. Keep in mind that coroutines each have their own stack. Therefore, one coroutine does not have access to another coroutine's automatic variables appearing in a VAR section (this is true even if those VAR objects appear in the main program). Always use `STATIC`, `STORAGE`, or `READONLY` objects when communicating data between coroutines using global variables. If you must communicate an automatic or dynamic object, pass the address of that object in a register or some static global variable.

A bigger problem than where we pass parameters to a coroutine is "How do we deal with parameters we pass to a coroutine?" Parameters work well in procedures because we always enter the procedure at the same point and the state of the procedure is usually the same upon entry (with the possible exception of static variable values). This is not true for coroutines. Consider the following code that executes as a coroutine:

```
procedure IsACoroutine; nodisplay; noframe;
begin IsACoroutine;

    /*
    << Do something upon initial entry >>

    coret();    // Invoke previous coroutine
    /*

    << Do some more stuff upon return >>

    forever

        OtherCoroutine.cocall(); // Invoke a third coroutine.

        << Do some more stuff here >>

        MainPgm.cocall();        // Transfer control back to the main program.
        /*

        << do some stuff >>
```

3. The chapter on low-level parameter implementation in this volume discusses different places you can pass parameters between procedures. Check out that chapter for more details on this subject.

```

    coret();                // Return to whomever cocalled us.
    /*

    << do some more stuff >>

endfor;

end IsACoroutine;

```

In this code you'll find several comments of the form `"/**"`. These comments mark the point at which some other coroutine can reenter this code. Note that, in general, the "calling" coroutine has no idea which entry point it will invoke and, likewise, this coroutine has no idea who invoked it at any given point. Passing in meaningful parameters and properly processing them under these conditions is difficult, at best. The only reasonable solution is to make every invocation pass exactly the same type of data in the same location and then write your coroutines to handle this data appropriately upon each entry. Even though this solution is more reasonable than the other possibilities, maintaining code like this is very difficult.

If you're really dead set on passing parameters to a coroutine, the best solution is to have a single entry point into the code so you've only got to handle the parameter data in one spot. Consider the following procedure that other threads invoke as a coroutine:

```

procedure HasAParm; nodisplay;
begin HasAParm;

    << Initialization code goes here, assume no parameter >>
    forever

        coret(); // Or cocall some other coroutine.

        << deal with parameter data passed into this coroutine >>

    endfor;

end HasAParm;

```

Note that there are two entry points into this code: the first occurs on the initial entry. The other occurs whenever the `coret()` procedure returns via a `cocall` to this coroutine. Immediately after the `coret` statement, the code above can process whatever parameter data the calling code has set up. After processing that data, this code returns to the invoking coroutine and that coroutine (directly or indirectly) can invoke this code again with more data.

3.4 Recursion, Reentrancy, and Variables

The fact that each coroutine has its own stack impacts access to variables from within coroutines. In particular, the only automatic (VAR) objects you can normally access are those declared within the coroutine itself and any procedures it calls. In a later chapter of this volume we'll take a look at nested procedures and how one could access local variables outside of the current procedure. For the most part, that discussion does not apply to procedures that are coroutines.

If you wish to share information between two or more coroutines, the best place to put such information is in a static object (STATIC, READONLY, and STORAGE variables). Such data does not appear on a stack and is accessible to multiple coroutines (and other procedures) simultaneously.

Whenever you execute a `cocall` instruction, the system suspends the current thread of execution and switches to a different coroutine, effectively by returning to that other coroutine and picking up where it left off (via a `coret` or `cocall` operation). This means that it isn't really possible to recursively `cocall` some coroutine. Consider the following (vain) attempt to achieve this:

```

procedure IsACoroutine; nodisplay;

```

```

begin IsACoroutine;

    << Do some initial stuff >>

    IAC.cocall(); // Note: IAC is initialized with the address of IsACoroutine.

    << Do some more stuff >>

end IsACoroutine;

```

This code assumes that *IAC* is a coroutine variable initialized with the address of the *IsACoroutine* procedure. The *IAC.cocall* statement, therefore, is an attempt to recursively call this coroutine. However, all that happens is that this call leaves the *IsACoroutine* code and then the coroutine system passes control to where *IAC* last left off. That just happens to be the *IAC.cocall* statement that just left the *IsACoroutine* procedure. Therefore this code immediately returns back to itself.

Although the idea of a recursive coroutine doesn't make sense, it is certainly possible to call procedures from within a coroutine and those procedures can be recursive. You can even make cocalls from those (recursive) procedures and the coroutine system will automatically return back into the recursive calls when control transfers back to the coroutine.

Although coroutines are not recursive, it is quite possible for the coroutine run-time system to reenter a procedure that is a coroutine. This situation can occur when you have two coroutine variables, initialize them both with the address of the same procedure, and then execute a cocall to each of the coroutine objects. Consider the following simple example:

```

procedure Reentered; nodisplay;
begin Reentered;

    << do some initialization or other work >>

    coret();

    << do some other stuff >>

end Reentered;

.
.
.
CV1.create( 256, &Reentered ); // Initialize two coroutine variables with
CV2.create( 256, &Reentered ); // the address of the same procedure.
CV1.cocall(); // Start the first coroutine.
CV2.cocall(); // Start the second coroutine.

<< At this point, both CV1 and CV2 are suspended within Reentered >>

```

Notice at the end of this code sequence, both the CV1 and CV2 coroutines are executing inside the *Reentered* procedure. That is, if you cocall either one of them, the cocalled routine will continue execution after the *coret* statement. This is not an example of a recursive coroutine call, but it certainly demonstrates that you can reenter some procedure while some other coroutine is currently executing the code within that procedure.

Reentering some code (whether you do this via a coroutine call or some other mechanism) is a perfectly reasonable thing to do. Indeed, recursion is one example of reentrancy. However, there are some special considerations you must be aware of when it is possible to reenter some code.

The principal issue is the use of variables in reentrant code. Suppose the *Reentered* procedure above had the following declarations:

```

var
    i: int32;
    j: uns32;

```

One concern you might have is that the two different coroutines executing in the same procedure would share these variables. However, keep in mind that HLA allocates automatic variables on the stack. Since each coroutine has its own stack, they're going to get their own private copies of the variables. Therefore, if *CV1* stores a value into *i* and *j*, *CV2* will not see these values. While this may seem to be a problem, this is actually what you want. You generally don't want one coroutine's thread of execution affecting the calculation in a different thread of execution.

The discussion above applies only to automatic variables. HLA does not allocate static objects (those you declare in the *STATIC*, *READONLY*, and *STORAGE* sections) on the stack. Therefore, such variables are not associated with a specific coroutine; instead, all coroutines that are executing in the same procedure share the same variables. Therefore, you shouldn't use static variables within a procedure that serves as a coroutine (especially reentrant coroutines) unless you explicitly want to share that data among other coroutines or other procedures.

Coroutines have their own stack and maintain that stack between cocalls to other coroutines. Therefore, like iterators, coroutines maintain their state (including the value of automatic variables) across cocalls. This is true even if you leave a coroutine via some other procedure than the main procedure for the coroutine. For example, suppose coroutine *A* calls some procedure *B* and then within procedure *B* there is a cocall to some other coroutine *C*. Whenever coroutine *C* executes *coret* or some coroutine (including *C*) cocalls *A*, control transfers back into procedure *B* and procedure *B*'s state is maintained (including the values of all local variables *B* initialize prior to the cocall). When *B* executes a return instruction, it will return back to procedure *A* who originally called *B*.

In theory it's even possible to call a procedure as well as cocall that procedure (it's hard to imagine why you would want to do this and it's probably quite difficult to pull it off correctly, but it's certainly possible). This is such a bizarre situation that we won't consider it any farther here.

3.5 Generators

A generator is to a function what a coroutine is to a procedure. That is, the whole purpose of a generator is to return a value like a function result. As far as HLA and the Coroutines Library Module is concerned, there is absolutely no difference between a generator and a coroutine (anymore than there is a syntactical difference between a function and a procedure to HLA). Clearly, there are some semantic differences; this section will describe the semantics of a generator and propose a convention for generator implementation.

The best way to describe a generator is to begin with the discussion of a special-purpose generator object that HLA does support – the iterator. An iterator is a special form of a generator that does not require its own stack. Iterators share the same stack as the calling code (i.e., the code containing the *FOREACH* loop that invokes the iterator). Because of the semantics of the *FOREACH* loop, iterators can leave their activation records on the stack and, therefore, maintain their local state between exits to the *FOREACH* loop body. The disadvantage to this scheme is that the calling semantics of an iterator are very rigidly defined; you cannot call an iterator from an arbitrary point in a program and the iterator's state is preserved only for the execution of the *FOREACH* loop.

By using its own stack, a generator removes these restrictions. You can call a generator from any point in the program (except, of course, within the generator itself – remember, recursive coroutines are not possible). Also, the state (i.e., the activation record) of a generator is not tied to the execution of some syntactical item like a *FOREACH* loop. The generator maintains its local state from the point of its first call to the point you call *cofree* on that generator.

One major difference between iterators and generators is the fact that generators don't use the *yield* statement (thunk) to return results back to the calling code. To send a value back to whomever invokes the generator, the generator must cocall the original coroutine. Since one can call a generator from different points in the code, and in particular, from different coroutines, the typical way to "return" a value back to the "caller" is to use the *coret* procedure call after loading the return result into a register.

A typical generator does not use the *cocall* operation. The *cocall* method transfers control to some other (explicitly defined) coroutine. As a general rule, generators (like functions) return control to whomever

called them. They do not explicitly pass control through to some other coroutine. Keep in mind that the *coret* procedure only returns control to the last coroutine. Therefore, if some generator passes control to another coroutine, there is no way to anonymously return back to whomever called the generator in the first place. That information is lost at the point of the second *cocall* operator. Since the main purpose of a generator is to return a value to whomever cocalled it, finding cocalls in a generator would be unusual indeed.

One problem with generators is that, like the coroutines upon which they are based, you cannot pass parameters to a generator via the stack. In most cases this is perfectly acceptable. As their name implies, generators typically generate an independent stream of data once they begin execution. After initialization, a generator generally doesn't require any additional information in order to generate its data sequence. Although this is the typical case, it is not the only case; sometimes you may need to write generators that need parameter data on each call. So what's the best way to handle this?

In a previous section (see "Parameters and Register Values in Coroutine Calls" on page 1334) we discussed a couple of ways to pass parameters to coroutines. While those techniques apply here as well, they are not particularly convenient (certainly not as convenient as passing parameters to a standard HLA procedure). Because of their function-like nature, it is more common to have to pass parameters to a generator (versus a generic coroutine) and you'll probably make more calls to generators that require parameters (versus similar calls to coroutines). Therefore, it would be nice if there were a "high-level" way of passing parameters to generators. Well, with a couple of tricks we can easily accomplish this⁴.

Remember that we cannot pass parameters to a coroutine (or generator) on the stack. The most convenient place to pass coroutine parameters is in registers or in static, global, memory locations. Unfortunately, writing a sequence of instructions to load up registers with parameter values (or, worse yet, copy the parameter data to global variables) prior to invoking a generator is a real pain. Fortunately, HLA's macros come to the rescue here; we can easily write a macro that lets us invoke a generator using a high level syntax and the macro can take care of the dirty work of loading registers (or global memory locations) with the necessary values. As an example, consider a generator, *_MyGen*, that expects two parameters in the EAX and EBX registers. Here's a macro, *MyGen*, that sets up the registers and invokes this generator:

```
#macro MyGen( ParmForEAX, ParmForEBX );

    mov( ParmForEAX, eax );
    mov( ParmForEBX, ebx );
    _MyGen.cocall();

#endmacro;

.
.
.
MyGen( 5, i );
```

You could, with just a tiny bit more effort, pass the parameters in global memory locations using this same technique.

In a few situations, you'll really need to pass parameters to a generator on the stack. We'll not go into the reasons or details here, but there are some rare circumstances where this is necessary. In many other circumstances, it may not be necessary but it's certainly more convenient to pass the parameters on the stack because you get to take advantage of HLA's high level parameter passing syntax when you use this scheme (i.e., you get to choose the parameter passing mechanism and HLA will automatically handle a lot of the gory details behind parameter passing for you when you use the stack). The best solution in this situation is to write a wrapper procedure. A wrapper procedure is a short procedure that reorganizes a parameter list before calling some other procedure. The macro above is a simple example of a wrapper – it takes two the (text) parameters and moves their run-time data into EAX and EBX prior to cocalling *_MyGen*. We could have just as easily written a procedure to accomplish this same task:

```
procedure MyGen( ParmForEAX:dword; ParmForEBX:dword ); nodisplay;
begin MyGen;
```

4. By the way, generators are coroutines, so these tricks apply to generic coroutines as well.


```

mov( ParmForEAX, eax );
mov( ParmForEBX, ebx );
_MyGen.cocall();

end MyGen;

```

Beyond the obvious time/space trade-offs between macros and procedures, there is one other big difference between these two schemes: the procedure variation allows you to specify a parameter passing mechanism like pass by reference, pass by value/result, pass by name, etc⁵. Once HLA knows the parameter passing mechanism, it can automatically emit code to process the actual parameters for you. Suppose, for example, you needed pass by value/result semantics. Using the macro invocation, you'd have to explicitly write a lot of code to pull this off. In the procedure above, about the only change you'd need is to add some code to store any returned results back into *ParmForEAX* or *ParmForEBX* (whichever uses the pass by value/result mechanism).

Since coroutines and generators share the same memory address space as the calling coroutine, it is not correct to say that a coroutine or generator does not have access to the stack of the calling code. The stack is in the same address space as the coroutine/generator; the only problem is that the coroutine doesn't know exactly where any parameters may be sitting in that memory space. This is because procedures use the value in ESP⁶ to indirectly reference the parameters passed on the stack and, unfortunately, the *cocall* method changes the value of the ESP register upon entry into the coroutine/generator. However, were we to pass the original value of ESP (or some other pointer into an activation record) through to a generator, then it would have direct access to those values on the stack. Consider the following modification to the *MyGen* procedure above:

```

procedure MyGen( ParmForEAX:dword; ParmForEBX:dword ); nodisplay;
begin MyGen;

    mov( ebp, ebx );
    _MyGen.cocall();

end MyGen;

```

Notice that this code does not directly copy the two parameters into some locations that are directly accessible in the generator. Instead, this procedure simply copies the base address of *MyGen*'s activation record (the value in EBP) into the EBX register for use in the generator. To gain access to those parameters, the generator need only index off of EBX using appropriate offsets for the two parameters in the activation record. Perhaps the easiest way to do this is by declaring an explicit record declaration that corresponds to *MyGen*'s activation record:

```

type
  MyGenAR:
    record
      OldEBP:    dword;
      RtnAdrs:   dword;
      ParmForEAX: dword;
      ParmForEBX: dword;
    endrecord;

```

Now, within the *_MyGen* generator code, you can access the parameters on the stack using code like the following:

```

mov( (type MyGenAR [ebx]).ParmForEAX, eax );
mov( (type MyGenAR [ebx]).ParmForEBX, edx );

```

5. For a discussion of pass by value/result and pass by name parameter passing mechanisms, see the chapter on low-level parameter implementation in this volume.

6. Okay, a procedure typically uses the value in EBP, but that same procedure also loads EBP with the value of ESP in the standard entry sequence.

This scheme works great for pass by value and pass by reference parameters (those we've seen up to this point). There are other parameter passing mechanisms, this scheme doesn't work as well for those other parameter passing mechanisms. Fortunately, you won't use those other parameter passing methods anywhere near as often as pass by value and pass by name.

3.6 Exceptions and Coroutines

Exceptions represent a special problem in coroutines. The HLA TRY..ENDTRY statement typically surrounds a block of statements one wishes to protect from errant code. The TRY..ENDTRY statement is a dynamic control structure insofar as this statement also protects any procedures you call from within the TRY..ENDTRY block. So the obvious question is "does the TRY..ENDTRY statement protect a coroutine you call from within such a block as well?" The short answer is "no, it does not."

Actually, in the first implementation of the HLA coroutines module, the exception handling system did pass control from one coroutine to another whenever an exception occurred. However, it became immediately obvious that this behavior causes non-intuitive results. Coroutines tend to be independent entities and to have one coroutine transfer control to another without an explicit ccall creates some problems. Therefore, the HLA compiler and the coroutines module now treat each coroutine as a standalone entity as far as exceptions are concerned. If an exception occurs within some coroutine and there isn't an outstanding TRY..ENDTRY block active for that coroutine, then the system behaves as though there were no active TRY..ENDTRY at all, *even if there is an active TRY..ENDTRY block in another coroutine*; in other words, the program aborts execution. Keep this in mind when using exception handling in your coroutines. For more details on exception handling, see the chapter on Exception Handling in this volume.

3.7 Putting It All Together

This chapter discusses a novel program unit – the coroutine. Coroutines have many special properties that make them especially valuable in certain situations. Coroutines are not built into the HLA language. Rather, HLA implements them via the HLA Coroutines Module in the HLA Standard Library. This chapter began by discussing the methods found in that library module. Next, this chapter discusses the use of variables, recursion, reentrancy and machine state in a coroutine. This chapter also discusses how to create generators using coroutines and pass parameters to a generator in a function-like fashion. Finally, this chapter briefly discussed the use of the TRY..ENDTRY statement in coroutines.

Coroutines and generators are like iterators insofar as they are control structures that few high level languages implement. Therefore, most programmers are unfamiliar with the concept of a coroutine. This, unfortunately, leads to the lack of consideration of coroutines in a program, even where a coroutine is the most suitable control structure to use. You should avoid this trap and learn how to use coroutines and generators properly so that you'll know when to use them when the need arises.

Advanced Parameter Implementation Chapter Four

4.1 Chapter Overview

This chapter discusses advanced parameter passing techniques in assembly language. Both low-level and high-level syntax appears in this chapter. This chapter discusses the more advanced pass by value/result, pass by result, pass by name, and pass by lazy evaluation parameter passing mechanisms. This chapter also discusses how to pass parameters in a low-level manner and describes where you can pass such parameters.

4.2 Parameters

Although there is a large class of procedures that are totally self-contained, most procedures require some input data and return some data to the caller. Parameters are values that you pass to and from a procedure. There are many facets to parameters. Questions concerning parameters include:

- where is the data coming from?
- how do you pass and return data?
- what is the amount of data to pass?

Previous chapters have touched on some of these concepts (see the chapters on beginning and intermediate procedures as well as the chapter on Mixed Language Programming). This chapter will consider parameters in greater detail and describe their low-level implementation.

4.3 Where You Can Pass Parameters

Up to this point we've mainly used the 80x86 hardware stack to pass parameters. In a few examples we've used machine registers to pass parameters to a procedure. In this section we explore several different places where we can pass parameters. Common places are

- in registers,
- in FPU or MMX registers,
- in global memory locations,
- on the stack,
- in the code stream, or
- in a parameter block referenced via a pointer.

Finally, the amount of data has a direct bearing on where and how to pass it. For example, it's generally a bad idea to pass large arrays or other large data structures by value because the procedure has to copy that data onto the stack when calling the procedure (when passing parameters on the stack). This can be rather slow. Worse, you cannot pass large parameters in certain locations; for example, it is not possible to pass a 16-element int32 array in a register.

Some might argue that the only locations you need for parameters are the register and the stack. Since these are the locations that high level languages use, surely they should be sufficient for assembly language programmers. However, one advantage to assembly language programming is that you're not as constrained as a high level language; this is one of the major reasons why assembly language programs can be more efficient than compiled high level language code. Therefore, it's a good idea to explore different places where we can pass parameters in assembly language.

This section discusses six different locations where you can pass parameters. While this is a fair number of different places, undoubtedly there are many other places where one can pass parameters. So don't let this section prejudice you into thinking that this is the only way to pass parameters.

4.3.1 Passing Parameters in (Integer) Registers

Where you pass parameters depends, to a great extent, on the size and number of those parameters. If you are passing a small number of bytes to a procedure, then the registers are an excellent place to pass parameters. The registers are an ideal place to pass value parameters to a procedure. If you are passing a single parameter to a procedure you should use the following registers for the accompanying data types:

Data Size	Pass in this Register
Byte:	al
Word:	ax
Double Word:	eax
Quad Word:	edx:eax

This is, by no means, a hard and fast rule. If you find it more convenient to pass 32 bit values in the ESI or EBX register, by all means do so. However, most programmers use the registers above to pass parameters.

If you are passing several parameters to a procedure in the 80x86's registers, you should probably use up the registers in the following order:

First	Last
eax, edx, esi, edi, ebx, ecx	

In general, you should avoid using EBP register. If you need more than six parameters, perhaps you should pass your values elsewhere.

HLA provides a special high level syntax that lets you tell HLA to pass parameters in one or more of the 80x86 integer registers. Consider the following syntax for an HLA parameter declaration:

```
varname : typename in register
```

In this example, *varname* represents the parameter's name, *typename* is the type of the parameter, and *register* is one of the 80x86's eight-, 16-, or 32-bit integer registers. The size of the data type must be the same as the size of the register (e.g., "int32" is compatible with a 32-bit register). The following is a concrete example of a procedure that passes a character value in a register:

```
procedure swapcase( chToSwap: char in al ); nodisplay; noframe;
begin swapcase;

    if( chToSwap in 'a'..'z' ) then

        and( $5f, chToSwap ); // Convert lower case to upper case.

    elseif( chToSwap in 'A'..'Z' ) then

        or( $20, chToSwap );

    endif;
    ret();
end swapcase;
```

There are a couple of important issues to note here. First, within the procedure's body, the parameter's name is an alias for the corresponding register if you pass the parameter in a register. In other words, *chToSwap* in the previous code is equivalent to "al" (indeed, within the procedure HLA actually defines *chToSwap* as a TEXT constant initialized with the string "al"). Also, since the parameter was passed in a register rather than on the stack, there is no need to build a stack frame for this procedure; hence the absence of the standard entry and exit sequences in the code above. Note that the code above is exactly equivalent to the following code:

```

// Actually, the following parameter list is irrelevant and
// you could remove it. It does, however, help document the
// fact that this procedure has a single character parameter.

procedure swapcase( chToSwap: char in al ); nodisplay; noframe;
begin swapcase;

    if( al in 'a'..'z' ) then

        and( $5f, al ); // Convert lower case to upper case.

    elseif( al in 'A'..'Z' ) then

        or( $20, al );

    endif;
    ret();

end swapcase;

```

Whenever you call the *swapcase* procedure with some actual (byte sized) parameter, HLA will generate the appropriate code to move that character value into the AL register prior to the call (assuming you don't specify AL as the parameter, in which case HLA doesn't generate any extra code at all). Consider the following calls that the corresponding code that HLA generates:

```

// swapcase( 'a' );

    mov( 'a', al );
    call swapcase;

// swapcase( charVar );

    mov( charVar, al );
    call swapcase;

// swapcase( (type char [ebx]) );

    mov( [ebx], al );
    call swapcase;

// swapcase( ah );

    mov( ah, al );
    call swapcase;

// swapcase( al );

    call swapcase; // al's value is already in al!

```

The examples above all use the pass by value parameter passing mechanism. When using pass by value to pass parameters in registers, the size of the actual parameter (and formal parameter) must be exactly the same size as the register. Therefore, you are limited to passing eight, sixteen, or thirty-two bit values in the registers by value. Furthermore, these object must be scalar objects. That is, you cannot pass composite (array or record) objects in registers even if such objects are eight, sixteen, or thirty-two bits long.

You can also pass reference parameters in registers. Since pass by reference parameters are four-byte addresses, you must always specify a thirty-two bit register for pass by reference parameters. For example, consider the following *memfill* function that copies a character parameter (passed in AL) throughout some number of memory locations (specified in ECX), at the memory location specified by the value in EDI:

```

// memfill- This procedure stores <ECX> copies of the byte in AL starting

```

```

// at the memory location specified by EDI:

procedure memfill
(
    charVal: char in al;
    count: uns32 in ecx;
    var    dest: byte in edi    // dest is passed by reference
);
    nodisplay; noframe;

begin memfill;

    pushfd();    // Save D flag;
    push( ecx ); // Preserve other registers.
    push( edi );

    cld();    // increment EDI on string operation.
    rep.stosb(); // Store ECX copies of AL starting at EDI.

    pop( edi );
    pop( ecx );
    popfd();
    ret();    // Note that there are no parameters on the stack!

end memfill;

```

It is perfectly possible to pass some parameters in registers and other parameters on the stack to an HLA procedure. Consider the following implementation of *memfill* that passes the *dest* parameter on the stack:

```

procedure memfill
(
    charVal: char in al;
    count: uns32 in ecx;
    var    dest: var
);
    nodisplay;

begin memfill;

    pushfd();    // Save D flag;
    push( ecx ); // Preserve other registers.
    push( edi );

    cld();    // increment EDI on string operation.
    mov( dest, edi ); // get dest address into EDI for STOSB.
    rep.stosb(); // Store ECX copies of AL starting at EDI.

    pop( edi );
    pop( ecx );
    popfd();

end memfill;

```

Of course, you don't have to use the HLA high level procedure calling syntax when passing parameters in the registers. You can manually load the values into registers prior to calling a procedure (with the `CALL` instruction) and you can refer directly to those values via registers within the procedure. The disadvantage to this scheme, of course, is that the code will be a little more difficult to write, read, and modify. The advantage of the scheme is that you have more control and can pass any eight, sixteen, or thirty-two bit value between the procedure and its callers (e.g., you can load a four-byte array or record into a 32-bit register and call the procedure with that value in a single register, something you cannot do when using the high level language syntax for procedure calls). Fortunately, HLA gives you the choice of whichever parameter pass-

ing scheme is most appropriate, so you can use the manual passing mechanism when it's necessary and use the high level syntax whenever it's not necessary.

There are other parameter passing mechanism beyond pass by value and pass by reference that we will explore in this chapter. We will take a look at ways of passing parameters in registers using those parameter passing mechanisms as we encounter them.

4.3.2 Passing Parameters in FPU and MMX Registers

Since the 80x86's FPU and MMX registers are also registers, it makes perfect sense to pass parameters in these locations if appropriate. Although using the FPU and MMX registers is a little bit more work than using the integer registers, it's generally more efficient than passing the parameters in memory (e.g., on the stack). In this section we'll discuss the techniques and problems associated with passing parameters in these registers.

The first thing to keep in mind is that the MMX and FPU register sets are not independent. These two register sets overlap, much like the eight, sixteen, and thirty-two bit integer registers. Therefore, you cannot pass some parameters in FPU registers and other parameters in MMX registers to a given procedure. For more details on this issue, please see the chapter on the MMX Instruction Set. Also keep in mind that you must execute the EMMS instruction after using the MMX instructions before executing any FPU instructions. Therefore, it's best to partition your code into sections that use the FPU registers and sections that use the MMX registers (or better yet, use only one register set throughout your program).

The FPU represents a fairly special case. First of all, it only makes sense to pass real values through the FPU registers. While it is technically possible to pass other values through the FPU registers, efficiency and accuracy restrictions severely limit what you can do in this regard. This text will not consider passing anything other than real values in the floating point registers, but keep in mind that it is possible to pass generic groups of bits in the FPU registers if you're really careful. Do keep in mind, though, that you need a very detailed knowledge of the FPU if you're going to attempt this (exceptions, rounding, and other issues can cause the FPU to incorrectly manipulate your data under certain circumstances). Needless to say, you can only pass objects by value through the FPU registers; pass by reference isn't applicable here.

Assuming you're willing to pass only real values through the FPU registers, some problems still remain. In particular, the FPU's register architecture does not allow you to load the FPU registers in an arbitrary fashion. Remember, the FPU register set is a stack; so you have to push values onto this stack in the reverse order you wish the values to appear in the register file. For example, if you wish to pass the real variables *r*, *s*, and *t* in FPU registers ST0, ST1, and ST2, you must execute the following code sequence (or something similar):

```
fld( t );    // t -> ST0, but ultimately winds up in ST2.
fld( s );    // s -> ST0, but ultimately winds up in ST1.
fld( r );    // r -> ST0.
```

You cannot load some floating point value into an arbitrary FPU register without a bit of work. Furthermore, once inside the procedure that uses real parameters found on the FPU stack, you cannot easily access arbitrary values in these registers. Remember, FPU arithmetic operations automatically "renumber" the FPU registers as the operations push and pop data on the FPU stack. Therefore, some care and thought must go into the use of FPU registers as parameter locations since those locations are dynamic and change as you manipulate items on the FPU stack.

By far, the most common use of the FPU registers to pass value parameters to a function is to pass a single value parameter in the register so the procedure can operate directly on that parameter via FPU operations. A classic example might be a SIN function that expects its angle in degrees (rather than radians, and the FSIN instruction expects). The function could convert the degree to radians and then execute the FSIN instruction to complete the calculation.

Keep in mind the limited size of the FPU stack. This effectively eliminates the possibility of passing real parameter values through the FPU registers in a recursive procedure. Also keep in mind that it is rather difficult to preserve FPU register values across a procedure call, so be careful about using the FPU registers

to pass parameters since such operations could disturb the values already on the FPU stack (e.g., cause an FPU stack overflow).

The MMX register set, although it shares the same physical silicon as the FPU, does not suffer from the all same problems as the FPU register set when it comes to passing parameters. First of all, the MMX registers are true registers that are individually accessible (i.e., they do not use a stack implementation). You may pass data in any MMX register and you do not have to use the registers in a specific order. Of course, if you pass parameter data in an MMX register, the procedure you're calling must not execute any FPU instructions before you're done with the data or you will lose the value(s) in the MMX register(s).

In theory, you can pass any 64-bit data to a procedure in an MMX register. However, you'll find the use of the MMX register set most convenient if you're actually operating on the data in those registers using MMX instructions.

4.3.3 Passing Parameters in Global Variables

Once you run out of registers, the only other (reasonable) alternative you have is main memory. One of the easiest places to pass parameters is in global variables in the data segment. The following code provides an example:

```
// ThisProc-
//
// Global variable "Ref1Proc1" contains the address of a pass by reference
// parameter. Global variable "Value1Proc1" contains the value of some
// pass by value parameter. This procedure stores the value of the
// "Value1Proc1" parameter into the actual parameter pointed at by
// "Ref1Proc1".

procedure ThisProc; @nodisplay; @noframe;
begin ThisProc;

    mov( Ref1Proc1, ebx );      // Get address of reference parameter.
    mov( Value1Proc, eax );    // Get Value parameter.
    mov( eax, [ebx] );         // Copy value to actual ref parameter.
    ret();

end ThisProc;
.
.
.

// Sample call to the procedure (includes setting up parameters )

    mov( xxx, eax );           // Pass this parameter by value
    mov( eax, Value1Proc1 );
    lea( eax, yyyy );         // Pass this parameter by reference
    mov( eax, Ref1Proc1 );
    call ThisProc;
```

Passing parameters in global locations is inelegant and inefficient. Furthermore, if you use global variables in this fashion to pass parameters, the subroutines you write cannot use recursion. Fortunately, there are better parameter passing schemes for passing data in memory so you do not need to seriously consider this scheme.

4.3.4 Passing Parameters on the Stack

Most high level languages use the stack to pass parameters because this method is fairly efficient. Indeed, in most of the examples found in this text up to this chapter, passing parameters on the stack has been the standard solution. To pass parameters on the stack, push them immediately before calling the subroutine. The subroutine then reads this data from the stack memory and operates on it appropriately. Consider the following HLA procedure declaration and call:

```
procedure CallProc( a:dword; b:dword; c:dword );
.
.
.

CallProc( i, j, k+4);
```

By default, HLA pushes its parameters onto the stack in the order that they appear in the parameter list. Therefore, the 80x86 code you would typically write for this subroutine call is¹

```
push( i );
push( j );
mov( k, eax );
add( 4, eax );
push( eax );
call CallProc;
```

Upon entry into *CallProc*, the 80x86's stack looks like that shown in Figure 4.1

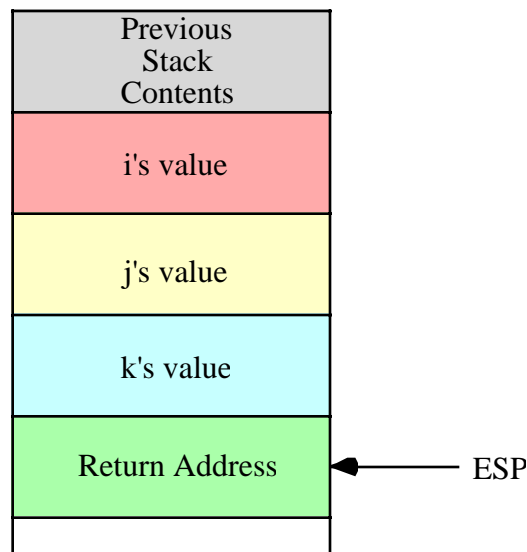


Figure 4.1 Activation Record for CallProc Invocation

Since the chapter on intermediate procedures discusses how to access these parameters, we will not repeat that discussion here. Instead, this section will attempt to tie together material from the previous chapters on procedures and the chapter on Mixed Language Programming.

1. Actually, you'd probably use the HLA high level calling syntax in the typical case, but we'll assume the use of the low-level syntax for the examples appearing in this chapter.

As noted in the chapter on intermediate procedures, the HLA compiler automatically associates some (positive) offset from EBP with each (non-register) parameter you declare in the formal parameter list. Keeping in mind that the base pointer for the activation record (EBP) points at the saved value of EBP and the return address is immediately above that, the first double word of parameter data starts at offset +8 from EBP in the activation record (see Figure 4.2 for one possible arrangement).

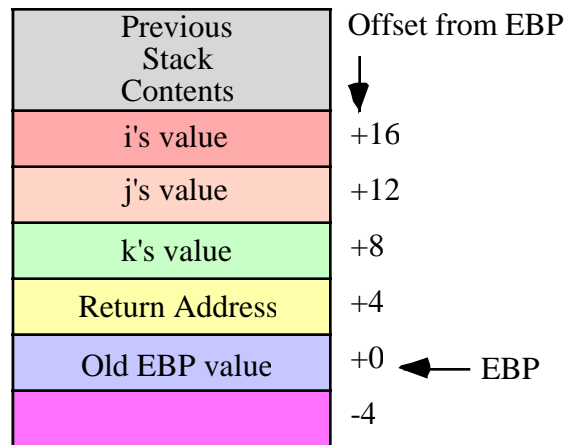


Figure 4.2 Offsets into CallProc's Activation Record

The parameter layout in Figure 4.2 assumes that the caller (as in the previous example) pushes the parameters in the order (left to right) that they appear in the formal parameter list; that is, this arrangement assumes that the code pushes *i* first, *j* second, and *k*+4 last. Because this is convenient and easy to do, most high level languages (and HLA, by default) push their parameters in this order. The only problem with this approach is that it winds up locating the first parameter at the highest address in memory and the last parameter at the lowest address in memory. This non-intuitive organization isn't much of a problem because you normally refer to these parameters by their name, not by their offset into the activation record. Hence, whether *i* is at offset +16 or +8 is usually irrelevant to you. Of course, you could refer to these parameters using memory references like "[ebp+16]" or "[ebp+8]" but, in general, that would be exceedingly poor programming style.

In some rare cases, you may actually need to refer to the parameters' values using an addressing mode of the form "[ebp+*disp*]" (where *disp* represents the offset of the parameter into the activation record). One possible reason for doing this is because you've written a macro and that macro always emits a memory operand using this addressing mode. However, even in this case you shouldn't use literal constants like "8" and "16" in the address expression. Instead, you should use the @OFFSET compile-time function to have HLA calculate this offset value for you. I.e., use an address expression of the form:

```
[ebp + @offset( a )]
```

There are two reasons you should specify the addressing mode in this fashion: (1) it's a little more readable this way, and, more importantly, (2) it is easier to maintain. For example, suppose you decide to add a parameter to the end of the parameter list. This causes all the offsets in *CallProc* to change. If you've used address expressions like "[ebp+16]" in your code, you've got to go locate each instance and manually change it. On the other hand, if you use the @OFFSET operator to calculate the offset of the variable in the activation record, then HLA will automatically recompute the current offset of a variable each time you recompile the program; hence you can make changes to the parameter list and not worry about having to manually change the address expressions in your programs.

Although pushing the actual parameters on the stack in the order of the formal parameters' declarations is very common (and the default case that HLA uses), this is not the only order a program can use. Some high level languages (most notably, C, C++, Java, and other C-derived languages) push their parameters in the reverse order, that is, from right to left. The primary reason they do this is to allow variable parameter

lists, a subject we will discuss a little later in this chapter (see “Variable Parameter Lists” on page 1368). Because it is very common for programmers to interface HLA programs with programs written in C, C++, Java, and other such languages, HLA provides a mechanism that allows it to process parameters in this order.

The `@CDECL` and `@STDCALL` procedure options tell HLA to reverse the order of the parameters in the activation record. Consider the previous declaration of `CallProc` using the `@CDECL` procedure option:

```
procedure CallProc( a:dword; b:dword; c:dword ); @cdecl;
.
.
.
    CallProc( i, j, k+4 );
```

To implement the call above you would write the following code:

```
mov( k, eax );
add( 4, eax );
push( eax );
push( j );
push( i );
call CallProc;
```

Compare this with the previous version and note that we’ve pushed the parameter values in the opposite order. As a general rule, if you’re not passing parameters between routines written in assembly and C/C++ or you’re not using variable parameter lists, you should use the default parameter passing order (left-to-right). However, if it’s more convenient to do so, don’t be afraid of using the `@CDECL` or `@STDCALL` options to reverse the order of the parameters.

Note that using the `@CDECL` or `@STDCALL` procedure option immediately changes the offsets of all parameters in a parameter list that has two or more parameters. This is yet another reason for using the `@OFFSET` operator to calculate the offset of an object rather than manually calculating this. If, for some reason, you need to switch between the two parameter passing schemes, the `@OFFSET` operator automatically recalculates the offsets.

One common use of assembly language is to write procedures and functions that a high level language program can call. Since different high level languages support different calling mechanisms, you might initially be tempted to write separate procedures for those languages (e.g., Pascal) that push their parameters in a left-to-right order and a separate version of the procedure for those languages (e.g., C) that push their parameters in a right-to-left order. Strictly speaking, this isn’t necessary. You may use HLA’s conditional compilation directives to create a single procedure that you can compile for other high level language. Consider the following procedure declaration fragment:

```
procedure CallProc( a:dword; b:dword; c:dword );
#if( @defined( CLanguage ))
    @cdecl;
#endif
```

With this code, you can compile the procedure for the C language (and similar languages) by simply defining the constant `CLanguage` at the beginning of your code. To compile for Pascal (and similar languages) you would leave the `CLanguage` symbol undefined.

Another issue concerning the use of parameters on the stack is “who takes the responsibility for cleaning these parameters off the stack?” As you saw in the chapter on Mixed Language Programming, various languages assign this responsibility differently. For example, in languages like Pascal, it is the procedure’s responsibility to clean up parameters off the stack before returning control to the caller. In languages like C/C++, it is the caller’s responsibility to clean up parameters on the stack after the procedure returns. By default, HLA procedures use the Pascal calling convention, and therefore the procedures themselves take responsibility for cleaning up the stack. However, if you specify the `@CDECL` procedure option for a given procedure, then HLA does not emit the code to remove the parameters from the stack when a procedure

returns. Instead, HLA leaves it up to the caller to remove those parameters. Therefore, the call above to `CallProc` (the one with the `@CDECL` option) isn't completely correct. Immediately after the call the code should remove the 12 bytes of parameters it has pushed on the stack. It could accomplish this using code like the following:

```
mov( k, eax );
add( 4, eax );
push( eax );
push( j );
push( i );
call CallProc;
add( 12, esp );    // Remove parameters from the stack.
```

Many C compilers don't emit an `ADD` instruction after each call that has parameters. If there are two or more procedures in a row, and the previous contents of the stack is not needed between the calls, the C compilers may perform a slight optimization and remove the parameter only after the last call in the sequence. E.g., consider the following:

```
pushd( 5 );
call Proc1Parm

push( i );
push( eax );
call Proc2Parms;

add( 12, esp );    // Remove parameters for Proc1Parm and Proc2Parms.
```

The `@STDCALL` procedure option is a combination of the `@CDECL` and `@PASCAL` calling conventions. `@STDCALL` passes its parameters in the right-to-left order (like C/C++) but requires the procedure to remove the parameters from the stack (like `@PASCAL`). It's also possible to pass parameters in the left-to-right order (like `@PASCAL`) and require the caller to remove the parameters from the stack (like C), but HLA does not provide a specific syntax for this. If you want to use this calling convention, you will need to manually build and destroy the activation record, e.g.,

```
procedure CallerPopsParms( i:int32; j:uns32; r:real64 ); nodisplay; noframe;
begin CallerPopsParms;

    push( ebp );
    mov( esp, ebp );
    .
    .
    .
    mov( ebp, esp );
    pop( ebp );
    ret();    // Don't remove any parameters from the stack.

end CallerPopsParms;

.
.
.
pushd( 5 );
pushd( 6 );
pushd( (type dword r[4])); // Assume r is an eight-byte real.
pushd( (type dword r));
call CallerPopsParms;
add( 16, esp );    // Remove 16 bytes of parameters from stack.
```

Notice how this procedure uses the Pascal calling convention (to get parameters in the left-to-right order) but manually builds and destroys the activation record so that HLA doesn't automatically remove the parameters from the stack. Although the need to operate this way is nearly non-existent, it's interesting to note that it's still possible to do this in assembly language.

4.3.5 Passing Parameters in the Code Stream

The chapter on Intermediate Procedures introduced the mechanism for passing parameters in the code stream with a simple example of a *Print* subroutine. The *Print* routine is a very space-efficient way to print literal string constants to the standard output. A typical call to *Print* takes the following form:

```
call Print
byte "Hello World", 0 // Strings after Print must end with a zero!
```

As you may recall, the *Print* routine pops the return address off the stack and uses this as a pointer to a zero terminated string, printing each character it finds until it encounters a zero byte. Upon finding a zero byte, the *Print* routine pushes the address of the byte following the zero back onto the stack for use as the new return address (so control returns to the instruction following the zero byte). For more information on the *Print* subroutine, see the section on Code Stream Parameters in the chapter on Intermediate Procedures.

The *Print* example demonstrates two important concepts with code stream parameters: passing simple string constants by value and passing a variable length parameter. Contrast this call to *Print* with an equivalent call to the HLA Standard Library *stdout.puts* routine:

```
stdout.puts( "Hello World" );
```

It may look like the call to *stdout.puts* is simpler and more efficient. However, looks can be deceiving and they certainly are in this case. The statement above actually compiles into code similar to the following:

```
push( HWString );
call stdout.puts;
.
.
.
// In the CONSTs segment:

        dword 11 // Maximum string length
        dword 11 // Current string length
HWS     byte "Hello World", 0
HWString dword HWS
```

As you can see, the *stdout.puts* version is a little larger because it has three extra dword declarations plus an extra PUSH instruction. (It turns out that *stdout.puts* is faster because it prints the whole string at once rather than a character at a time, but the output operation is so slow anyway that the performance difference is not significant here.) This demonstrates that if you're attempting to save space, passing parameters in the code stream can help.

Note that the *stdout.puts* procedure is more flexible than *Print*. The *Print* procedure only prints string literal constants; you cannot use it to print string variables (as *stdout.puts* can). While it is possible to print string variables with a variant of the *Print* procedure (passing the variable's address in the code stream), this still isn't as flexible as *stdout.puts* because *stdout.puts* can easily print static and local (automatic) variables whereas this variant of *Print* cannot easily do this. This is why the HLA Standard Library uses the stack to pass the string variable rather than the code stream. Still, it's instructive to look at how you would write such a version of *Print*, so we'll do that in just a few moments.

One problem with passing parameters in the code stream is that the code stream is read-only². Therefore, any parameter you pass in the code stream must, necessarily, be a constant. While one can easily dream up some functions to whom you always pass constant values in the parameter lists, most procedures work best if you can pass different values (through variables) on each call to a procedure. Unfortunately, this is not possible when passing parameters by value to a procedure through the code stream. Fortunately, we can also pass data by reference through the code stream.

2. Technically, it is possible to make the code segment writable, but we will not consider that possibility here.

When passing reference parameters in the code stream, we must specify the address of the parameter(s) following the CALL instruction in the source file. Since we can only pass constant data (whose value is known at compile time) in the code stream, this means that HLA must know the address of the objects you pass by reference as parameters when it encounters the instruction. This, in turn, means that you will usually pass the address of static objects (STATIC, READONLY, and STORAGE) variables in the code stream. In particular, HLA does not know the address of an automatic (VAR) object at compile time, so you cannot pass the address of a VAR object in the code stream³.

To pass the address of some static object in the code stream, you would typically use the dword directive and list the object's name in the dword's operand field. Consider the following code that expects three parameters by reference:

Calling sequence:

```
static
  I:uns32;
  J:uns32;
  K:uns32;
  .
  .
  .
  call  AddEm;
  dword I,J,K;
```

Whenever you specify the name of a STATIC object in the operand field of the dword directive, HLA automatically substitutes the four-byte address of that static object for the operand. Therefore, the object code for the instruction above consists of the call to the *AddEm* procedure followed by 12 bytes containing the static addresses of *I*, *J*, and *K*. Assuming that the purpose of this code is to add the values in *J* and *K* together and store the sum into *I*, the following procedure will accomplish this task:

```
procedure AddEm; @nodisplay;
begin AddEm;

  push( eax );           // Preserve the registers we use.
  push( ebx );
  push( ecx );
  mov( [ebp+4], ebx ); // Get the return address.
  mov( [ebx+4], ecx ); // Get J's address.
  mov( [ecx], eax );   // Get J's value.
  mov( [ebx+8], ecx ); // Get K's address.
  add( [ecx], eax );   // Add in K's value.
  mov( [ebx], ecx );   // Get I's address.
  mov( eax, [ecx] );   // Store sum into I.
  add( 12, ebx );      // Skip over addresses in code stream.
  mov( ebx, [ebp+4] ); // Save as new return address.
  pop( ecx );
  pop( ebx );
  pop( eax );

end AddEm;
```

This subroutine adds *J* and *K* together and stores the result into *I*. Note that this code uses 32 bit constant pointers to pass the addresses of *I*, *J*, and *K* to *AddEm*. Therefore, *I*, *J*, and *K* must be in a static data segment. Note at the end of this procedure how the code advances the return address beyond these three pointers in the code stream so that the procedure returns beyond the address of *K* in the code stream.

The important thing to keep in mind when passing parameters in the code stream is that you must always advance the procedure's return address beyond any such parameters before returning from that pro-

3. You may, however, pass the offset of that variable in some activation record. However, implementing the code to access such an object is an exercise that is left to the reader.

cedure. If you fail to do this, the procedure will return into the parameter list and attempt to execute that data as machine instructions. The result is almost always catastrophic. Since HLA does not provide a high level syntax that automatically passes parameters in the code stream for you, you have to manually pass these parameters in your code. This means that you need to be extra careful. For even if you've written your procedure correctly, it's quite possible to create a problem if the calls aren't correct. For example, if you leave off a parameter in the call to *AddEm* or insert an extra parameter on some particular call, the code that adjusts the return address will not be correct and the program will probably not function correctly. So take care when using this parameter passing mechanism.

4.3.6 Passing Parameters via a Parameter Block

Another way to pass parameters in memory is through a *parameter block*. A parameter block is a set of contiguous memory locations containing the parameters. Generally, you would use a record object to hold the parameters. To access such parameters, you would pass the subroutine a pointer to the parameter block. Consider the subroutine from the previous section that adds *J* and *K* together, storing the result in *I*; the code that passes these parameters through a parameter block might be

Calling sequence:

```

type
  AddEmParmBlock:
    record
      i: pointer to uns32;
      j: uns32;
      k: uns32;
    endrecord;

static
  a: uns32;
  ParmBlock: AddEmParmBlock := AddEmParmBlock: [ &a, 2, 3 ];

procedure AddEm( var pb:AddEmParmBlock in esi ); nodisplay;
begin AddEm;

  push( eax );
  push( ebx );
  mov( (type AddEmParmBlock [esi]).j, eax );
  add( (type AddEmParmBlock [esi]).k, eax );
  mov( (type AddEmParmBlock [esi]).i, ebx );
  mov( eax, [ebx] );
  pop( ebx );
  pop( eax );

end AddEm;

```

This form of parameter passing works well when passing several static variables by reference or constant parameters by value, because you can directly initialize the parameter block as was done above.

Note that the pointer to the parameter block is itself a parameter. The examples in this section pass this pointer in a register. However, you can pass this pointer anywhere you would pass any other reference parameter – in registers, in global variables, on the stack, in the code stream, even in another parameter block! Such variations on the theme, however, will be left to your own imagination. As with any parameter, the best place to pass a pointer to a parameter block is in the registers. This text will generally adopt that policy.

Parameter blocks are especially useful when you make several different calls to a procedure and in each instance you pass constant values. Parameter blocks are less useful when you pass variables to procedures, because you will need to copy the current variable's value into the parameter block before the call (this is

roughly equivalent to passing the parameter in a global variable. However, if each particular call to a procedure has a fixed parameter list, and that parameter list contains constants (static addresses or constant values), then using parameter blocks can be a useful mechanism.

Also note that class fields are also an excellent place to pass parameters. Because class fields are very similar to records, we'll not create a separate category for these, but lump class fields together with parameter blocks.

4.4 How You Can Pass Parameters

There are six major mechanisms for passing data to and from a procedure, they are

- pass by value,
- pass by reference,
- pass by value/returned,
- pass by result,
- pass by name, and
- pass by lazy evaluation

Actually, it's quite easy to invent some additional ways to pass parameters beyond these six ways, but this text will concentrate on these particular mechanisms and leave other approaches to the reader to discover.

Since this text has already spent considerable time discussing pass by value and pass by reference, the following subsections will concentrate mainly on the last four ways to pass parameters.

4.4.1 Pass by Value-Result

Pass by value-result (also known as value-returned) combines features from both the pass by value and pass by reference mechanisms. You pass a value-result parameter by address, just like pass by reference parameters. However, upon entry, the procedure makes a temporary copy of this parameter and uses the copy while the procedure is executing. When the procedure finishes, it copies the temporary copy back to the original parameter.

This copy-in and copy-out process takes time and requires extra memory (for the copy of the data as well as the code that copies the data). Therefore, for simple parameter use, pass by value-result may be less efficient than pass by reference. Of course, if the program semantics require pass by value-result, you have no choice but to pay the price for its use.

In some instances, pass by value-returned is more efficient than pass by reference. If a procedure only references the parameter a couple of times, copying the parameter's data is expensive. On the other hand, if the procedure uses this parameter value often, the procedure amortizes the fixed cost of copying the data over many inexpensive accesses to the local copy (versus expensive indirect reference using the pointer to access the data).

HLA supports the use of value/result parameters via the VALRES keyword. If you prefix a parameter declaration with VALRES, HLA will assume you want to pass the parameter by value/result. Whenever you call the procedure, HLA treats the parameter like a pass by reference parameter and generates code to pass the address of the actual parameter to the procedure. Within the procedure, HLA emits code to copy the data referenced by this point to a local copy of the variable⁴. In the body of the procedure, you access the parameter as though it were a pass by value parameter. Finally, before the procedure returns, HLA emits code to copy the local data back to the actual parameter. Here's the syntax for a typical procedure that uses pass by value result:

4. This statement assumes that you're not using the @NOFRAME procedure option.


```

procedure AddandZero( valres p1:uns32; valres p2:uns32 ); @nodisplay;
begin AddandZero;

    mov( p2, eax );
    add( eax, p1 );
    mov( 0, p2 );

end AddandZero;

```

A typical call to this function might look like the following:

```
AddandZero( j, k );
```

This call computes "j := j+k;" and "k := 0;" simultaneously.

Note that HLA automatically emits the code within the *AddandZero* procedure to copy the data from *p1* and *p2*'s actual parameters into the local variables associated with these parameters. Likewise, HLA emits the code, just before returning, to copy the local parameter data back to the actual parameter. HLA also allocates storage for the local copies of these parameters within the activation record. Indeed, the names *p1* and *p2* in this example are actually associated with these local variables, not the formal parameters themselves. Here's some code similar to that which HLA emits for the *AddandZero* procedure earlier:

```

procedure AddandZero( var p1_ref: uns32; var p2_ref:uns32 );
    @nodisplay;
    @noframe;
var
    p1: uns32;
    p2: uns32;
begin AddandZero;

    push( ebp );
    sub( _vars_, esp ); // Note: _vars_ is "8" in this example.
    push( eax );
    mov( p1_ref, eax );
    mov( [eax], eax );
    mov( eax, p1 );
    mov( p2_ref, eax );
    mov( [eax], eax );
    mov( eax, p2 );
    pop( eax );

    // Actual procedure body begins here:

    mov( p2, eax );
    add( eax, p1 );
    mov( 0, p2 );

    // Clean up code associated with the procedure's return:

    push( eax );
    push( ebx );
    mov( p1_ref, ebx );
    mov( p1, eax );
    mov( eax, [ebx] );
    mov( p2_ref, ebx );
    mov( p2, eax );
    mov( eax, [ebx] );
    pop( ebx );
    pop( eax );
    ret( 8 );

end AddandZero;

```

As you can see from this example, pass by value/result has considerable overhead associated with it in order to copy the data into and out of the procedure's activation record. If efficiency is a concern to you, you should avoid using pass by value/result for parameters you don't reference numerous times within the procedure's body.

If you pass an array, record, or other large data structure via pass by value/result, HLA will emit code that uses a `MOVS` instruction to copy the data into and out of the procedure's activation record. Although this copy operation will be slow for larger objects, you needn't worry about the compiler emitting a ton of individual `MOV` instructions to copy a large data structure via value/result.

If you specify the `@NOFRAME` option when you actually declare a procedure with value/result parameters, HLA does not emit the code to automatically allocate the local storage and copy the actual parameter data into the local storage. Furthermore, since there is no local storage, the formal parameter names refer to the address passed as a parameter rather than to the local storage. For all intents and purposes, specifying `@NOFRAME` tells HLA to treat the pass by value/result parameters as pass by reference. The calling code passes in the address and it is your responsibility to dereference that address and copy the local data into and out of the procedure. Therefore, it's quite unusual to see an HLA procedure use pass by value/result parameters along with the `@NOFRAME` option (since using pass by reference achieves the same thing).

This is not to say that you shouldn't use `@NOFRAME` when you want pass by value/result semantics. The code that HLA generates to copy parameters into and out of a procedure isn't always the most efficient because it always preserves all registers. By using `@NOFRAME` with pass by value/result parameters, you can supply slightly better code in some instances; however, you could also achieve the same effect (with identical code) by using pass by reference.

When calling a procedure with pass by value/result parameters, HLA pushes the address of the actual parameter on the stack in a manner identical to that for pass by reference parameters. Indeed, when looking at the code HLA generates for a pass by reference or pass by value/result parameter, you will not be able to tell the difference. This means that if you manually want to pass a parameter by value/result to a procedure, you use the same code you would use for a pass by reference parameter; specifically, you would compute the address of the object and push that address onto the stack. Here's code equivalent to what HLA generates for the previous call to `AddandZero`⁵:

```
// AddandZero( k, j );

    lea( eax, k );
    push( eax );
    lea( eax, j );
    push( eax );
    call AddandZero;
```

Obviously, pass by value/result will modify the value of the actual parameter. Since pass by reference also modifies the value of the actual parameter to a procedure, you may be wondering if there are any semantic differences between these two parameter passing mechanisms. The answer is yes – in some special cases their behavior is different. Consider the following code that is similar to the Pascal code appearing in the chapter on intermediate procedures:

```
procedure uhoh( var i:int32; var j:int32 ); @nodisplay;
begin uhoh;

    mov( i, ebx );
    mov( 4, (type int32 [ebx]) );
    mov( j, ecx );
    mov( [ebx], eax );
    add( [ecx], eax );
    stdout.put( "i+j=", (type int32 eax), nl );
```

5. Actually, this code is a little more efficient since it doesn't worry about preserving EAX's value; this example assumes the presence of the "@use eax;" procedure option.

```

end uhoh;
.
.
.
var
  k: int32;
.
.
.
  mov( 5, k );
  uhoh( k, k );
.
.
.

```

As you may recall from the chapter on Intermediate Procedures, the call to *uhoh* above prints "8" rather than the expected value of "9". The reason is because *i* and *j* are aliases of one another when you call *uhoh* and pass the same variable in both parameter positions.

If we switch the parameter passing mechanism above to value/result, then *i* and *j* are not exactly aliases of one another so this procedure exhibits different semantics when you pass the same variable in both parameter positions. Consider the following implementation:

```

procedure uhoh( valres i:int32; valres j:int32 ); nodisplay;
begin uhoh;

  mov( 4, i );
  mov( i, eax );
  add( j, eax );
  stdout.put( "i+j=", (type int32 eax), nl );

end uhoh;
.
.
.
var
  k: int32;
.
.
.
  mov( 5, k );
  uhoh( k, k );
.
.
.

```

In this particular implementation the output value is "9" as you would intuitively expect. The reason this version produces a different result is because *i* and *j* are not aliases of one another within the procedure. These names refer to separate local objects that the procedure happens to initialize with the value of the same variable upon initial entry. However, when the body of the procedure executes, *i* and *j* are distinct so storing four into *i* does not overwrite the value in *j*. Hence, when this code adds the values of *i* and *j* together, *j* still contains the value 5, so this procedure displays the value nine.

Note that there is a question of what value *k* will have when *uhoh* returns to its caller. Since pass by value/result stores the value of the formal parameter back into the actual parameter, the value of *k* could either be four or five (since *k* is the formal parameter associated with both *i* and *j*). Obviously, *k* may only contain one or the other of these values. HLA does not make any guarantees about which value *k* will hold other than it will be one or the other of these two possible values. Obviously, you can figure this out by writing a simple program, but keep in mind that future versions of HLA may not respect the current ordering; worse, it's quite possible that within the same version of HLA, for some calls it could store *i*'s value into *k* and for other calls it could store *j*'s value into *k* (not likely, but the HLA language allows this). The order by

which HLA copies value/result parameters into and out of a procedure is completely implementation dependent. If you need to guarantee the copying order, then you should use the `@NOFRAME` option (or use pass by reference) and copy the data yourself.

Of course, this ambiguity exists only if you pass the same actual parameter in two value/result parameter positions on the same call. If you pass different actual variables, this problem does not exist. Since it is very rare for a program to pass the same variable in two parameter slots, particularly two pass by value/result slots, it is unlikely you will ever encounter this problem.

HLA implements pass by value/result via pass by reference and copying. It is also possible to implement pass by value/result using pass by value and copying. When using the pass by reference mechanism to support pass by value/result, it is the procedure's responsibility to copy the data from the actual parameter into the local copy; when using the pass by value form, it is the caller's responsibility to copy the data to and from the local object. Consider the following implementation that (manually) copies the data on the call and return from the procedure:

```

procedure DisplayAndClear( val i:int32 ); @nodisplay; @noframe;
begin DisplayAndClear;

    push( ebp );           // NOFRAME, so we have to do this manually.
    mov( esp, ebp );

    stdout.put( "I = ", i, nl );
    mov( 0, i );

    pop( ebp );
    ret();                 // Note that we don't clean up the parameters.

end DisplayAndClear;
.
.
.
push( m );
call DisplayAndClear;
pop( m );
stdout.put( "m = ", m, nl );
.
.
.
```

The sequence above displays "I = 5" and "m = 0" when this code sequence runs. Note how this code passes the value in on the stack and then returns the result back on the stack (and the caller copies the data back to the actual parameter).

In the example above, the procedure uses the `@NOFRAME` option in order to prevent HLA from automatically removing the parameter data from the stack. Another way to achieve this effect is to use the `@CDECL` procedure option (that tells HLA to use the C calling convention, which also leaves the parameters on the stack for the caller to clean up). Using this option, we could rewrite the code sequence above as follows:

```

procedure DisplayAndClear( val i:int32 ); @nodisplay; @cdecl;
begin DisplayAndClear;

    stdout.put( "I = ", i, nl );
    mov( 0, i );

end DisplayAndClear;
.
.
.
DisplayAndClear( m );
pop( m );
```

```

stdout.put( "m = ", m, nl );
.
.
.

```

The advantage to this scheme is that HLA automatically emits the procedure's entry and exit sequences so you don't have to manually supply this information. Keep in mind, however, that the @CDECL calling sequence pushes the parameters on the stack in the reverse order of the standard HLA calling sequence. Generally, this won't make a difference to your code unless you explicitly assume the order of parameters in memory. Obviously, this won't make a difference at all when you've only got a single parameter.

The examples in this section have all assumed that we've passed the value/result parameters on the stack. Indeed, HLA only supports this location if you want to use a high level calling syntax for value/result parameters. On the other hand, if you're willing to manually pass the parameters in and out of a procedure, then you may pass the value/result parameters in other locations including the registers, in the code stream, in global variables, or in parameter blocks.

Passing parameters by value/result in registers is probably the easiest way to go. All you've got to do is load an appropriate register with the desired value before calling the procedure and then leave the return value in that register upon return. When the procedure returns, it can use the register's value however it sees fit. If you prefer to pass the value/result parameter by reference rather than by value, you can always pass in the address of the actual object in a 32-bit register and do the necessary copying within the procedure's body.

Of course, there are a couple of drawbacks to passing value/result parameters in the registers; first, the registers can only hold small, scalar, objects (though you can pass the address of a large object in a register). Second, there are a limited number of registers. But if you can live with these drawbacks, registers provide a very efficient place to pass value/result parameters.

It is possible to pass certain value/result parameters in the code stream. However, you'll always pass such parameters by their address (rather than by value) to the procedure since the code stream is in read-only memory (and you can't write a value back to the code stream). When passing the actual parameters via value/result, you must pass in the address of the object in the code stream, so the objects must be static variables so HLA can compute their addresses at compile-time. The actual implementation of value/result parameters in the code stream is left as an exercise for the end of this volume.

There is one advantage to value/result parameters in the HLA/assembly programming environment. You get semantics very similar to pass by reference without having to worry about constant dereferencing of the parameter throughout the code. That is, you get the ability to modify the actual parameter you pass into a procedure, yet within the procedure you get to access the parameter like a local variable or value parameter. This simplification makes it easier to write code and can be a real time saver if you're willing to (sometimes) trade off a minor amount of performance for easier to read-and-write code.

4.4.2 Pass by Result

Pass by result is almost identical to pass by value-result. You pass in a pointer to the desired object and the procedure uses a local copy of the variable and then stores the result through the pointer when returning. The only difference between pass by value-result and pass by result is that when passing parameters by result you do not copy the data upon entering the procedure. Pass by result parameters are for returning values, not passing data to the procedure. Therefore, pass by result is slightly more efficient than pass by value-result since you save the cost of copying the data into the local variable.

HLA supports pass by result parameters using the RESULT keyword prior to a formal parameter declaration. Consider the following procedure declaration:

```

procedure HasResParm( result r:uns32 ); nodisplay;
begin HasResParm;

    mov( 5, r );

end HasResParm;

```

Like pass by value/result, modification of the pass by result parameter results (ultimately) in the modification of the actual parameter. The difference between the two parameter passing mechanisms is that pass by result parameters do not have a known initial value upon entry into the code (i.e., the HLA compiler does not emit code to copy any data into the parameter upon entry to the procedure).

Also like pass by value/result, you may pass result parameters in locations other than on the stack. HLA does not support anything other than the stack when using the high level calling syntax, but you may certainly pass result parameters manually in registers, in the code stream, in global variables, and in parameter blocks.

4.4.3 Pass by Name

Some high level languages, like ALGOL-68 and Panacea, support pass by name parameters. Pass by name produces semantics that are similar (though not identical) to textual substitution (e.g., like macro parameters). However, implementing pass by name using textual substitution in a compiled language (like ALGOL-68) is very difficult and inefficient. Basically, you would have to recompile a function every time you call it. So compiled languages that support pass by name parameters generally use a different technique to pass those parameters. Consider the following Panacea procedure (Panacea's syntax is sufficiently similar to HLA's that you should be able to figure out what's going on):

```
PassByName: procedure(name item:integer; var index:integer);
begin PassByName;

    foreach index in 0..10 do

        item := 0;

    endfor;

end PassByName;
```

Assume you call this routine with the statement "PassByName(A[i], i);" where A is an array of integers having (at least) the elements *A[0]..A[10]*. Were you to substitute (textually) the pass by name parameter *item* you would obtain the following code:

```
begin PassByName;

    foreach I in 0..10 do

        A[I] := 0;

    endfor;

end PassByName;
```

This code zeros out elements 0..10 of array A.

High level languages like ALGOL-68 and Panacea compile pass by name parameters into *functions* that return the address of a given parameter. So in one respect, pass by name parameters are similar to pass by reference parameters insofar as you pass the address of an object. The major difference is that with pass by reference you compute the address of an object before calling a subroutine; with pass by name the subroutine itself calls some function to compute the address of the parameter whenever the function references that parameter.

So what difference does this make? Well, reconsider the code above. Had you passed *A[I]* by reference rather than by name, the calling code would compute the address of *A[I]* just before the call and passed in this address. Inside the *PassByName* procedure the variable *item* would have always referred to a single address, not an address that changes along with *I*. With pass by name parameters, *item* is really a function

that computes the address of the parameter into which the procedure stores the value zero. Such a function might look like the following:

```
procedure ItemThunk; @nodisplay; @noframe;
begin ItemThunk;

    mov( i, eax );
    lea( eax, A[eax*4] );
    ret();

end ItemThunk;
```

The compiled code inside the *PassByName* procedure might look something like the following:

```
; item := 0;

call ItemThunk;
mov( 0, (type dword [eax]));
```

Thunk is the historical term for these functions that compute the address of a pass by name parameter. It is worth noting that most HLLs supporting pass by name parameters do not call thunks directly (like the call above). Generally, the caller passes the address of a thunk and the subroutine calls the thunk indirectly. This allows the same sequence of instructions to call several different thunks (corresponding to different calls to the subroutine). In HLA, of course, we will use HLA thunk variables for this purpose. Indeed, when you declare a procedure with a pass by name parameter, HLA associates the thunk type with that parameter. The only difference between a parameter whose type is thunk and a pass by name parameter is that HLA requires a thunk constant for the pass by name parameter (whereas a parameter whose type is thunk can be either a thunk constant or a thunk variable). Here's a typical procedure prototype using a pass by name variable (note the use of the NAME keyword to specify pass by name):

```
procedure HasNameParm( name nameVar:uns32 );
```

Since *nameVar* is a thunk, you call this object rather than treat it as data or as a pointer. Although HLA doesn't enforce this, the convention is that a pass by name parameter returns the address of the object whenever you invoke the thunk. The procedure then dereferences this address to access the actual data. The following code is the HLA equivalent of the Panacea procedure given earlier:

```
procedure passByName( name ary:int32; var ip:int32 ); @nodisplay;
const i:text := "(type int32 [ebx])";
begin passByName;

    mov( ip, ebx );
    mov( 0, i );
    while( i <= 10 ) do

        ary(); // Get address of "ary[i]" into eax.
        mov( i, ecx );
        mov( ecx, (type int32 [eax]) );
        inc( i );

    endwhile;

end passByName;
```

Notice how this code assumes that the *ary* thunk returns a pointer in the EAX register.

Whenever you call a procedure with a pass by name parameter, you must supply a thunk that computes some address and returns that address in the EAX register (or wherever you expect the address to be sitting upon return from the thunk; convention dictates the EAX register). Here is some code that demonstrates how to pass a thunk constant for a pass by name parameter to the procedure above:

```
var
```

```

index:uns32;
array: uns32[ 11 ]; // Has elements 0..10.
.
.
.
passByName
(
    thunk
    #{
        push( ebx );
        mov( index, ebx );
        lea( eax, array[ebx*4] );
        pop( ebx );
    }#,
    index
);

```

The "thunk #{...}#" sequence specifies a literal thunk that HLA compiles into the code stream. For the environment pointer, HLA pushes the current value for EBP, for the procedure pointer, HLA passes in the address of the code in the " #{...}#" braces. Whenever the *passByName* procedure actually calls this thunk, the run-time system restores EBP with the pointer to the current procedure's activation record and executes the code in these braces. If you look carefully at the code above, you'll see that this code loads the EAX register with the address of the *array[index]* variable. Therefore, the *passByName* procedure will store the next value into this element of *array*.

Pass by name parameter passing has garnered a bad name because it is a notoriously slow mechanism. Instead of directly or indirectly accessing an object, you have to first make a procedure call (which is expensive compared to an access) and then dereference a pointer. However, because pass by name parameters defer their evaluation until you actually access an object, pass by name effectively gives you a deferred pass by reference parameter passing mechanism (deferring the calculation of the address of the parameter until you actually access that parameter). This can be very important in certain situations. As you've seen in the chapter on thunks, the proper use of deferred evaluation can actually improve program performance. Most of the complaints about pass by name are because someone misused this parameter passing mechanism when some other mechanism would have been more appropriate. There are times, however, when pass by name is the best approach.

It is possible to transmit pass by name parameters in some location other than the stack. However, we don't call them pass by name parameters anymore; they're just thunks (that happen to return an address in EAX) at that point. So if you wish to pass a pass by name parameter in some other location than the stack, simply create a thunk object and pass your parameter as the thunk.

4.4.4 Pass by Lazy-Evaluation

Pass by name is similar to pass by reference insofar as the procedure accesses the parameter using the address of the parameter. The primary difference between the two is that a caller directly passes the address on the stack when passing by reference, it passes the address of a function that computes the parameter's address when passing a parameter by name. The pass by lazy evaluation mechanism shares this same relationship with pass by value parameters – the caller passes the address of a function that computes the parameter's value if the first access to that parameter is a read operation.

Pass by lazy evaluation is a useful parameter passing technique if the cost of computing the parameter value is very high and the procedure may not use the value. Consider the following HLA procedure header:

```

procedure PassByEval( lazy a:int32; lazy b:int32; lazy c:int32 );

```

Consider the *PassByEval* procedure above. Suppose it takes several minutes to compute the values for the *a*, *b*, and *c* parameters (these could be, for example, three different possible paths in a Chess game). Perhaps the *PassByEval* procedure only uses the value of one of these parameters. Without pass by lazy evaluation, the calling code would have to spend the time to compute all three parameters even though the

procedure will only use one of the values. With pass by lazy evaluation, however, the procedure will only spend the time computing the value of the one parameter it needs. Lazy evaluation is a common technique artificial intelligence (AI) and operating systems use to improve performance since it provides deferred parameter evaluation capability.

HLA's implementation of pass by lazy evaluation parameters is (currently) identical to the implementation of pass by name parameters. Specifically, pass by lazy evaluation parameters are thunks that you must call within the body of the procedure and that you must write whenever you call the procedure. The difference between pass by name and pass by lazy evaluation is the convention surrounding what the thunks return. By convention, pass by name parameters return a pointer in the EAX register. Pass by lazy evaluation parameters, on the other hand, return a value, not an address. Where the pass by lazy evaluation thunk returns its value depends upon the size of the value. However, by convention most programmers return eight-, 16-, 32-, and 64-bit values in the AL, AX, EAX, and EDX:EAX registers, respectively. The exceptions are floating point values (the convention is to use the ST0 register) and MMX values (the convention is to use the MM0 register for MMX values).

Like pass by name, you only pass by lazy evaluation parameters on the stack. Use thunks if you want to pass lazy evaluation parameters in a different location.

Of course, nothing is stopping you from returning a value via a pass by name thunk or an address via a pass by lazy evaluation thunk, but to do so is exceedingly poor programming style. Use these parameter pass mechanisms as they were intended.

4.5 Passing Parameters as Parameters to Another Procedure

When a procedure passes one of its own parameters as a parameter to another procedure, certain problems develop that do not exist when passing variables as parameters. Indeed, in some (rare) cases it is not logically possible to pass some parameter types to some other procedure. This section deals with the problems of passing one procedure's parameters to another procedure.

Pass by value parameters are essentially no different than local variables. All the techniques in the previous sections apply to pass by value parameters. The following sections deal with the cases where the calling procedure is passing a parameter passed to it by reference, value-result, result, name, and lazy evaluation.

4.5.1 Passing Reference Parameters to Other Procedures

Passing a reference parameter though to another procedure is where the complexity begins. Consider the following HLA procedure skeleton:

```

procedure ToProc(??? parm:dword);
begin ToProc;
    .
    .
    .
end ToProc;

procedure HasRef(var refparm:dword);

begin HasRef;
    .
    .
    .
    ToProc(refParm);
    .
    .

```

```
end HasRef;
```

The “???” in the *ToProc* parameter list indicates that we will fill in the appropriate parameter passing mechanism as the discussion warrants.

If *ToProc* expects a pass by value parameter (i.e., ??? is just an empty string), then *HasRef* needs to fetch the value of the *refparm* parameter and pass this value to *ToProc*. The following code accomplishes this⁶:

```
mov( refparm, ebx ); // Fetch address of actual refparm value.
pushd( [ebx] );      // Pass value of refparm variable on the stack.
call ToProc;
```

To pass a reference parameter by reference, value-result, or result parameter is easy – just copy the caller’s parameter as-is onto the stack. That is, if the *parm* parameter in *ToProc* above is a reference parameter, a value-result parameter, or a result parameter, you would use the following calling sequence:

```
push( refparm );
call ToProc;
```

We get away with passing the value of *refparm* on the stack because *refparm* currently contains the address of the actual object that *ToProc* will reference. Therefore, we need only copy this value (which is an address).

To pass a reference parameter by name is fairly easy. Just write a thunk that grabs the reference parameter’s address and returns this value. In the example above, the call to *ToProc* might look like the following:

```
ToProc
(
    thunk
    #{
        mov( refparm, eax );
    }#
);
```

To pass a reference parameter by lazy evaluation is very similar to passing it by name. The only difference (in *ToProc*’s calling sequence) is that the thunk must return the value of the variable rather than its address. You can easily accomplish this with the following thunk:

```
ToProc
(
    thunk
    #{
        mov( refparm, eax ); // Get the address of the actual parameter
        mov( [eax], eax );  // Get the value of the actual parameter.
    }#
);
```

Note that HLA’s high level procedure calling syntax automatically handles passing reference parameters as value, reference, value/result, and result parameters. That is, when using the high level procedure call syntax and *ToProc*’s parameter is pass by value, pass by reference, pass by value/result, or pass by result, you’d use the following syntax to call *ToProc*:

```
ToProc( refparm );
```

HLA will automatically figure out what data it needs to push on the stack for this procedure call.

6. The examples in this section all assume the use of a display. If you are using static links, be sure to adjust all the offsets and the code to allow for the static link that the caller must push immediately before a call.

Unfortunately, HLA does not automatically handle the case where you pass a reference parameter to a procedure via pass by name or pass by lazy evaluation. You must explicitly write this think yourself.

4.5.2 Passing Value-Result and Result Parameters as Parameters

If you have a pass by value/result or pass by result parameter that you want to pass to another procedure, and you use the standard HLA mechanism for pass by value/result or pass by result parameters, passing those parameters on to another procedure is trivial because HLA creates local variables using the parameters' names. Therefore, there is no difference between a local variable and a pass by value/result or pass by result parameter in this particular case. Once HLA has made a local copy of the value-result or result parameter or allocates storage for it, you can treat that variable just like a value parameter or a local variable with respect to passing it on to other procedures. In particular, if you're using the HLA high level calling syntax in your code, HLA will automatically pass that procedure by value, reference, value/result, or by result to another procedure. If you're passing the parameter by name or by lazy evaluation to another procedure, you must manually write the think that computes the address of the variable (pass by name) or obtains the value of the variable (pass by lazy evaluation) prior to calling the other procedure.

Of course, it doesn't make sense to use the value of a result parameter until you've stored a value into that parameter's local storage. Therefore, take care when passing result parameters to other procedures that you've initialized a result parameter before using its value.

If you're manually passing pass by value/result or pass by result parameters to a procedure and then you need to pass those parameters on to another procedure, HLA cannot automatically generate the appropriate code to pass those parameters on. This is especially true if you've got the parameter sitting in a register or in some location other than a local variable. Likewise, if the procedure you're calling expects you to pass a value/result or result parameter using some mechanism other than passing the address on the stack, you will also have manually write the code to pass the parameter on through. Since such situations are specific to a given situation, the only advice this text can offer is to suggest that you carefully think through what you're doing. Remember, too, that if you use the @NOFRAME procedure option, HLA does not make local copies, so you will have to compute and pass the addresses of such parameters manually.

4.5.3 Passing Name Parameters to Other Procedures

Since a pass by name parameter's think returns the address of a parameter, passing a name parameter to another procedure is very similar to passing a reference parameter to another procedure. The primary differences occur when passing the parameter on as a name parameter.

Unfortunately, HLA's high level calling syntax doesn't automatically deal with pass by name parameters. The reason is because HLA doesn't assume that thinks return an address in the EAX register (this is a convention, not a requirement). A programmer who writes the think could return the address somewhere else, or could even create a think that doesn't return an address at all! Therefore, it is up to you to handle passing pass by name parameters to other procedures.

When passing a name parameter as a value parameter to another procedure, you first call the think, dereference the address the think returns, and then pass the value to the new procedure. The following code demonstrates such a call when the think returns the variable's address in EAX:

```

CallThink();      // Call the think which returns an address in EAX.
pushd( [eax] );  // Push the value of the object as a parameter.
call ToProc;     // Call the procedure that expects a value parameter.
.
.
.

```

Passing a name parameter to another procedure by reference is very easy. All you have to do is push the address the think returns onto the stack. The following code, that is very similar to the code above, accomplishes this:

```

CallThunk(); // Call the thunk which returns an address in EAX.
push( eax ); // Push the address of the object as a parameter.
call ToProc; // Call the procedure that expects a value parameter.
.
.
.

```

Passing a name parameter to another procedure as a pass by name parameter is very easy; all you need to do is pass the thunk on to the new procedure. The following code accomplishes this:

```

push( (type dword CallThunk));
push( (type dword CallThunk[4]));
call ToProc;

```

To pass a name parameter to another procedure by lazy evaluation, you need to create a thunk for the lazy-evaluation parameter that calls the pass by name parameter's thunk, dereferences the pointer, and then returns this value. The implementation is left as a programming project.

4.5.4 Passing Lazy Evaluation Parameters as Parameters

Lazy evaluation are very similar to name parameters except they typically return a value in EAX (or some other register) rather than an address. This means that you may only pass lazy evaluation parameters by value or by lazy evaluation to another procedure (since they don't have an address associated with them).

4.5.5 Parameter Passing Summary

The following table describes how to pass parameters from one procedure as parameters to another procedure. The rows specify the "input" parameter passing mechanism (how the parameter was passed into the current procedure) and the rows specify the "output" parameter passing mechanism (how the procedure passing the parameter on to another procedure as a parameter).

Table 1: Passing Parameters as Parameters to Another Procedure

	Pass as Value	Pass as Reference	Pass as Value-Result	Pass as Result	Pass as Name	Pass as Lazy Evaluation
Value	Pass the value	Pass address of the value parameter	Pass address of the value parameter	Pass address of the value parameter	Create a thunk that returns the address of the value parameter	Create a thunk that returns the value
Reference	Dereference parameter and pass the value it points at	Pass the address (value of the reference parameter)	Pass the address (value of the reference parameter)	Pass the address (value of the reference parameter)	Create a thunk that passes the address (value of the reference parameter)	Create a thunk that dereferences the reference parameter and returns its value

Table 1: Passing Parameters as Parameters to Another Procedure

	Pass as Value	Pass as Reference	Pass as Value-Result	Pass as Result	Pass as Name	Pass as Lazy Evaluation
Value-Result	Pass the local value as the value parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Create a thunk that returns the address of the local value of the value-result parameter	Create a thunk that returns the value in the local value of the value-result parameter
Result	Pass the local value as the value parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Create a thunk that returns the address of the local value of the result parameter	Create a thunk that returns the value in the local value of the result parameter
Name	Call the thunk, dereference the pointer, and pass the value at the address the thunk returns	Call the thunk and pass the address it returns as the parameter	Call the thunk and pass the address it returns as the parameter	Call the thunk and pass the address it returns as the parameter	Pass the address of the thunk and any other values associated with the name parameter	Write a thunk that calls the name parameter's thunk, dereferences the address it returns, and then returns the value at that address
Lazy Evaluation	If necessary, call the thunk to obtain the Lazy Eval parameter's value. Pass the local value as the value parameter	Not possible. Lazy Eval parameters return a value which does not have an address.	Not possible. Lazy Eval parameters return a value which does not have an address.	Not possible. Lazy Eval parameters return a value which does not have an address.	Not possible. Lazy Eval parameters return a value which does not have an address.	Create a thunk that calls the caller's Lazy Eval parameter. This new thunk returns that result as its result.

4.6 Variable Parameter Lists

On occasion you may need the ability to pass a varying number of parameters to a given procedure. The *stdout.put* routine in the HLA Standard Library provides a good example of where having a variable number of parameters is useful. There are two ways to accomplish this: (1) Fake it and use a macro rather than a procedure (this is what the HLA Standard Library does, for example, with the *stdout.put* invocation – *stdout.put* is a macro not a procedure), and (2) Pass in some information to the procedure that describes how many parameters in must process and where it can find those parameters. We'll take a look at both of these mechanisms in this section.

HLA's macro facility allows a varying number of parameters by specifying an empty array parameter as the last formal parameter in the macro list, e.g.,

```
#macro VariableNumOfParms( a, b, c[] );
    .
    .
    .
#endmacro;
```

Whenever HLA processes a macro declaration like the one above, it associates the first two actual parameters with the formal parameters *a* and *b*; any remaining actual parameters becomes strings in the constant string array *c*. By using the @ELEMENTS compile-time function, you can determine how many additional parameters appear in the parameter list (which can be zero or more).

Of course, a macro is not a procedure. So the fact that we have a list of text constants and a string array that represents our actual parameter list does not satisfy the requirements for a varying parameter list at run-time. However, we can write some compile-time code that parses the parameter list and calls an appropriate set of procedures to handle each and every parameter passed to the macro. For example, the *stdout.put* macro splits up the parameter list and calls a sequence of routines (like *stdout.puts* and *stdout.puti32*) to handle each parameter individually.

Breaking up a macro's variable parameter list into a sequence of procedure calls with only one parameter per call may not solve a need you have for varying parameter lists. That being the case, it may still be possible to use macros to implement varying parameters for a procedure. If the number of parameters is within some range, then you can use the function overloading trick discussed in the chapter on macros to call one of several different procedures, each with a different number of parameters. Please see the chapter on macros for additional details.

Although macros provide a convenient way to implement a varying parameter list, they do suffer from some limitations that make them unsuitable for all applications. In particular, if you need to call a single procedure and pass it an indeterminate number of parameters (no limits), then the tricks with macros employed above won't work well for you. In this situation you will need to push the parameters on the stack (or pass them somewhere else) and include some information that tells the procedure how many parameters you're passing (and, perhaps, their size). The most common way to do this is to push the parameters onto the stack and then, as the last parameter, push the parameter count (or size) onto the stack.

Most procedures that push a varying number of parameters on the stack use the C/C++ calling convention. There are two reasons for this: (1) the parameters appear in memory in a natural order (the number of parameters followed by the first parameter, followed by the second parameter, etc.), (2) the caller will need to remove the parameters from the stack since each call can have a different number of parameters and the 80x86 RET instruction can only remove a fixed (constant) number of parameter bytes.

One drawback to using the C/C++ calling convention to pass a variable number of parameters on the stack is that you must manually push the parameters and issue a CALL instruction; HLA does not provide a high-level language syntax for declaring and calling procedures with a varying number of parameters.

Consider, as an example, a simple *MaxUns32* procedure that computes the maximum of *n* uns32 values passed to it. The caller begins by pushing *n* uns32 values and then, finally, it also pushes *n*. Upon return, the caller removes the *n+1* uns32 values (including *n*) from the stack. The function, presumably, returns the maximum value in the EAX register. Here's a typical call to *MaxUns32*:

```

push( i );
push( j );
push( k );
pushd( 10 );

push( 4 );          // n=4, number of parameters passed to this code.
call MaxUns32;     // Compute the maximum of the above.
add( 20, esp );    // Remove the parameters from the stack.

```

The *MaxUns32* procedure itself must first fetch the value for *n* from a known location (in this case, it will be just above the return address on the stack). The procedure can then use this value to step through each of the other parameters found on the stack above the value for *n*. Here's some sample code that accomplishes this:

```

procedure MaxUns32; nodisplay; noframe;
const n:text := "(type uns32 [ebp+8])";
const first:text := "(type uns32 [ebp+12])";
begin MaxUns32;

    push( ebp );
    mov( esp, ebp );
    push( ebx );
    push( ecx );

    mov( n, ecx );
    if( ecx > 0 ) then

        lea( ebx, first );
        mov( first, eax );    // Use this as the starting Max value.
        repeat

            if( eax < [ebx] ) then

                mov( [ebx], eax );

            endif;
            add( 4, ebx );
            dec( ecx );

        until( ecx = 0 );

    else

        // There were no parameter values to try, so just return zero.

        xor( eax, eax );

    endif;
    pop( ecx );
    pop( ebx );
    pop( ebp );
    ret();    // Can't remove the parameters!

end MaxUns32;

```

This code assumes that n is at location `[ebp+8]` (which it will be if n is the last parameter pushed onto the stack) and that n `uns32` values appear on the stack above this point. It steps through each of these values searching for the maximum, which the function returns in `EAX`. If n contains zero upon entry, this function simply returns zero in `EAX`.

Passing a single parameter count, as above, works fine if all the parameters are the same type and size. If the size and/or type of each parameter varies, you will need to pass information about each individual parameter on the stack. There are many ways to do this, a typical mechanism is to simply preface each parameter on the stack with a double word containing its size in bytes. Another solution is that employed by the `printf` function in the C standard library - pass an array of data (a string in the case of `printf`) that contains type information that the procedure can interpret at run-time to determine the type and size of the parameters. For example, the C `printf` function uses format strings like `"%4d"` to determine the size (and count, via the number of formatting options that appear within the string) of the parameters.

4.7 Function Results

Functions return a result, which is nothing more than a result parameter. In assembly language, there are very few differences between a procedure and a function. That is why there isn't a "function" directive. Functions and procedures are usually different in high level languages, function calls appear only in expressions, subroutine calls as statements⁷. Assembly language doesn't distinguish between them.

You can return function results in the same places you pass and return parameters. Typically, however, a function returns only a single value (or single data structure) as the function result. The methods and locations used to return function results is the subject of the next four sections.

4.7.1 Returning Function Results in a Register

Like parameters, the 80x86's registers are the best place to return function results. The `getc` routine in the HLA Standard Library is a good example of a function that returns a value in one of the CPU's registers. It reads a character from the keyboard and returns the ASCII code for that character in the `AL` register. By convention, most programmers return function results in the following registers:

Use	First	Last
Bytes:	al, ah, dl, dh, cl, ch, bl, bh	
Words:	ax, dx, cx, si, di, bx	
Double words:	eax, edx, ecx, esi, edi, ebx	
Quad words:	edx:eax	
Real Values:	ST0	
MMX Values:	MM0	

Once again, this table represents general guidelines. If you're so inclined, you could return a double word value in (`CL`, `DH`, `AL`, `BH`). If you're returning a function result in some registers, you shouldn't save and restore those registers. Doing so would defeat the whole purpose of the function.

7. "C" is an exception to this rule. C's procedures and functions are all called functions. PL/I is another exception. In PL/I, they're all called procedures.

4.7.2 Returning Function Results on the Stack

Another good place where you can return function results is on the stack. The idea here is to push some dummy values onto the stack to create space for the function result. The function, before leaving, stores its result into this location. When the function returns to the caller, it pops everything off the stack except this function result. Many HLLs use this technique (although most HLLs on the IBM PC return function results in the registers). The following code sequences show how values can be returned on the stack:

```

procedure RtnOnStack( RtnResult: dword; parm1: uns32; parm2:uns32 );
  @nodisplay;
  @noframe;
  var
    LocalVar: uns32;
  begin RtnOnStack;

    push( ebp );           // The standard entry sequence
    mov( esp, ebp );
    sub( _vars_, esp );

    << code that leaves a value in RtnResult >>

    mov( ebp, esp );      // Not quite standard exit sequence.
    pop( ebp );
    ret( __parms_-4 );    // Don't pop RtnResult off stack on return!

  end RtnOnStack;

```

Calling sequence:

```

RtnOnStack( 0, p1, p2 ); // "0" is a dummy value to reserve space.
pop( eax );             // Retrieve return result from stack.

```

Although the caller pushed 12 bytes of data onto the stack, *RtnOnStack* only removes eight bytes. The first “parameter” on the stack is the function result. The function must leave this value on the stack when it returns.

4.7.3 Returning Function Results in Memory Locations

Another reasonable place to return function results is in a known memory location. You can return function values in global variables or you can return a pointer (presumably in a register or a register pair) to a parameter block. This process is virtually identical to passing parameters to a procedure or function in global variables or via a parameter block.

Returning parameters via a pointer to a parameter block is an excellent way to return large data structures as function results. If a function returns an entire array, the best way to return this array is to allocate some storage, store the data into this area, and leave it up to the calling routine to deallocate the storage. Most high level languages that allow you to return large data structures as function results use this technique.

Of course, there is very little difference between returning a function result in memory and the pass by result parameter passing mechanism. See “Pass by Result” on page 1359 for more details.

4.7.4 Returning Large Function Results

Returning small scalar values in the registers or on the stack makes a lot of sense. However, mechanism for returning function results does not scale very well to large data structures. The registers are too small to

return large records or arrays and returning such data on the stack is a lot of work (not to mention that you've got to copy the data from the stack to its final resting spot upon return from the function). In this section we'll take a look at a couple of methods for returning large objects from a function.

The traditional way to return a large function result is to pass the location where one is to store the result as a pass by reference parameter. The advantage to this scheme is that it is relatively efficient (speed-wise) and doesn't require any extra space; the procedure uses the final destination location as scratch pad memory while it is building up the result. The disadvantage to this scheme is that it is very common to pass the destination variable as an input parameter (thus creating an alias). Since, in a high level language, you don't have the problems of aliases with function return results, this is a non-intuitive semantic result that can create some unexpected problems.

A second solution, though a little bit less efficient, is to use a pass by result parameter to return the function result. Pass by result parameters get their own local copy of the data that the system copies back over the destination location once the function is complete (thus avoiding the problem with aliases). The drawback to using pass by result, especially with large return values, is the fact that the program must copy the data from the local storage to the destination variable when the function completes. This data copy operation can take a significant amount of time for really large objects.

Another solution for returning large objects, that is relatively efficient, is to allocate storage for the object in the function, place whatever data you wish to return in the allocated storage, and then return a pointer to this storage. If the calling code references this data indirectly rather than copying the data to a different location upon return, this mechanism and run significantly faster than pass by result. Of course, it is not as general as using pass by result parameters, but with a little planning it is easy to arrange you code so that it works with pointers to large objects. String functions are probably the best example of this function result return mechanism in practice. It is very common for a function to allocate storage for a string result on the heap and then return a "string variable" in EAX (remember that strings in HLA are pointers).

4.8 Putting It All Together

This chapter discusses how and where you can pass parameters in an assembly language program. It continues the discussion of parameter passing that appears in earlier chapters in this text. This chapter discusses, in greater detail, several of the different places that a program can pass parameters to a procedure including registers, FPU/MMX register, on the stack, in the code stream, in global variables, and in parameters blocks. While this is not an all-inclusive list, it does cover the more common places where programs pass parameters.

In addition to where, this chapter discusses how programs can pass parameters. Possible ways include pass by value, pass by reference, pass by value/result, pass by result, pass by name, and pass by lazy evaluation. Again, these don't represent all the possible ways one could think of, but it does cover (by far) the most common ways programs pass parameters between procedures.

Another parameter-related issue this chapter discusses is how to pass parameters passed into one procedure as parameters to another procedure. Although HLA's high level calling syntax can take care of the grungy details for you, it's important to know how to pass these parameters manually since there are many instances where you will be forced to write the code that passes these parameters (not to mention, it's a good idea to know how this works, just on general principles).

This chapter also touches on passing a variable number of parameters between procedures and how to return function results from a procedure.

This chapter will not be the last word on parameters. We'll take another look at parameters in the very next chapter when discussing lexical scope.

Lexical Nesting

Chapter Five

5.1 Chapter Overview

This chapter discusses nested procedures and the issues associated with calling such procedure and accessing local variables in nested procedures. Nesting procedures offers HLA users a modicum of built-in information hiding support. Therefore, the material in this chapter is very important for those wanting to write highly structured code. This information is also important to those who want to understand how block structured high level languages like Pascal and Ada operate.

5.2 Lexical Nesting, Static Links, and Displays

In block structured languages like Pascal¹ it is possible to nest procedures and functions. Nesting one procedure within another limits the access to the nested procedure; you cannot access the nested procedure from outside the enclosing procedure. Likewise, variables you declare within a procedure are visible inside that procedure and to all procedures nested within that procedure². This is the standard block structured language notion of scope that should be quite familiar to anyone who has written Pascal or Ada programs.

There is a good deal of complexity hidden behind the concept of scope, or lexical nesting, in a block structured language. While accessing a local variable in the current activation record is efficient, accessing global variables in a block structured language can be very inefficient. This section will describe how a high level language like Pascal deals with non-local identifiers and how to access global variables and call non-local procedures and functions.

5.2.1 Scope

Scope in most high level languages is a static, or compile-time concept³. Scope is the notion of when a name is visible, or accessible, within a program. This ability to hide names is useful in a program because it is often convenient to reuse certain (non-descriptive) names. The *i* variable used to control most FOR loops in high level languages is a perfect example.

The scope of a name limits its visibility within a program. That is, a program has access to a variable name only within that name's scope. Outside the scope, the program cannot access that name. Many programming languages, like Pascal and C++, allow you to reuse identifiers if the scopes of those multiple uses do not overlap. As you've seen, HLA provides scoping features for its variables. There is, however, another issue related to scope: *address binding* and *variable lifetime*. Address binding is the process of associating a memory address with a variable name. Variable lifetime is that portion of a program's execution during which a memory location is bound to a variable. Consider the following Pascal procedures:

```
procedure One(Entry:integer);
var
  i,j:integer;

  procedure Two(Parm:integer);
  var j:integer;
  begin
    for j:= 0 to 5 do writeln(i+j);
```

-
1. Note that C and C++ are not block structured languages. Other block structured languages include Algol, Ada, and Modula-2.
 2. Subject, of course, to the limitation that you not reuse the identifier within the nested procedure.
 3. There are languages that support dynamic, or run-time, scope; this text will not consider such languages.

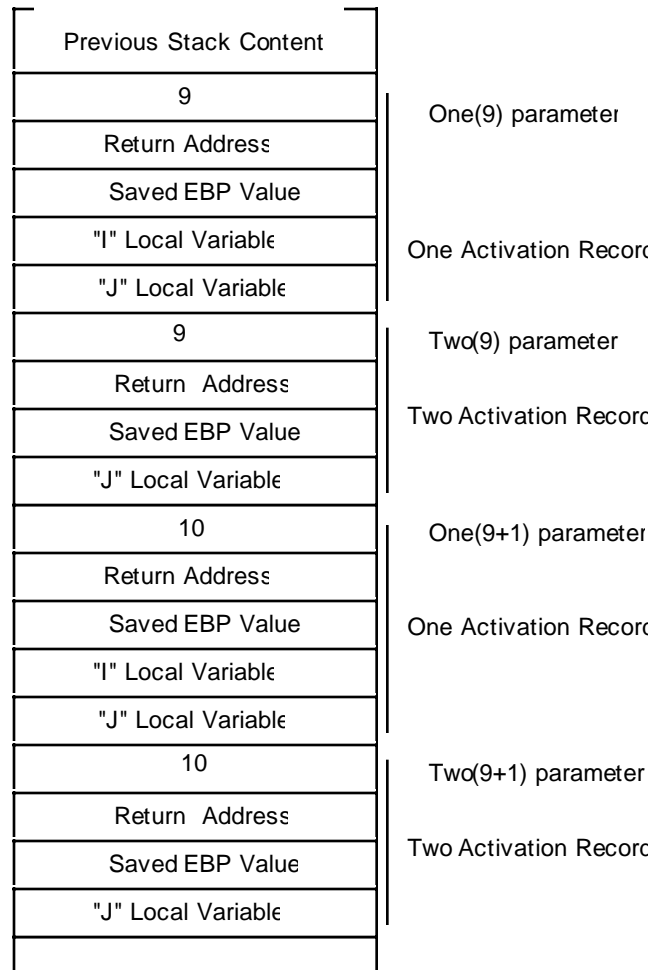


Figure 5.2 Activation Records for a Series of Recursive Calls of One and Two

As you can see, there are several copies of *I* and *J* on the stack at this point. Procedure *Two* (the currently executing routine) would access *J* in the most recent activation record that is at the bottom of Figure 5.2. The previous instance of *Two* will only access the variable *J* in its activation record when the current instance returns to *One* and then back to *Two*.

The lifetime of a variable's instance is from the point of activation record creation to the point of activation record destruction. Note that the first instance of *J* above (the one at the top of the diagram above) has the longest lifetime and that the lifetimes of all instances of *J* overlap.

5.2.3 Static Links

Pascal will allow procedure *Two* access to *I* in procedure *One*. However, when there is the possibility of recursion there may be several instances of *I* on the stack. Pascal, of course, will only let procedure *Two* access the most recent instance of *I*. In the stack diagram in Figure 5.2, this corresponds to the value of *I* in the activation record that begins with "*One(9+1)*parameter." The only problem is *how do you know where to find the activation record containing I?*

A quick, but poorly thought out answer, is to simply index backwards into the stack. After all, you can easily see in the diagram above that *I* is at offset eight from *Two*'s activation record. Unfortunately, this is not always the case. Assume that procedure *Three* also calls procedure *Two* and the following statement appears within procedure *One*:

```
If (Entry <5) then Three(Entry*2) else Two(Entry);
```

With this statement in place, it's quite possible to have two different stack frames upon entry into procedure *Two*: one with the activation record for procedure *Three* sandwiched between *One* and *Two*'s activation records and one with the activation records for procedures *One* and *Two* adjacent to one another. Clearly a fixed offset from *Two*'s activation record will not always point at the *I* variable on *One*'s most recent activation record.

The astute reader might notice that the saved EBP value in *Two*'s activation record points at the caller's activation record. You might think you could use this as a pointer to *One*'s activation record. But this scheme fails for the same reason the fixed offset technique fails. EBP's old value, the dynamic link, points at the caller's activation record. Since the caller isn't necessarily the enclosing procedure the dynamic link might not point at the enclosing procedure's activation record.

What is really needed is a pointer to the enclosing procedure's activation record. Many compilers for block structured languages create such a pointer, the *static link*. Consider the following Pascal code:

```
procedure Parent;
var i,j:integer;

    procedure Child1;
    var j:integer;
    begin
        for j := 0 to 2 do writeln(i);
    end {Child1};

    procedure Child2;
    var i:integer;
    begin
        for i := 0 to 1 do Child1;
    end {Child2};
begin {Parent}

    Child2;
    Child1;

end;
```

Just after entering *Child1* for the first time, the stack would look like Figure 5.3. When *Child1* attempts to access the variable *i* from *Parent*, it will need a pointer, the static link, to *Parent*'s activation record. Unfortunately, there is no way for *Child1*, upon entry, to figure out on it's own where *Parent*'s activation record lies in memory. It will be necessary for the caller (*Child2* in this example) to pass the static link to *Child1*. In general, the callee can treat the static link as just another parameter; usually pushed on the stack immediately before executing the CALL instruction.

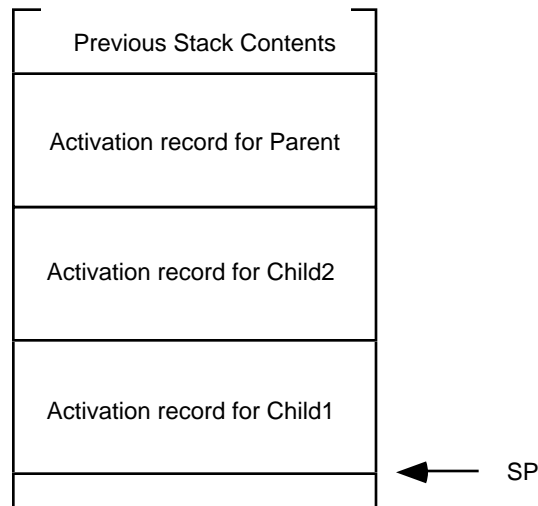


Figure 5.3 Activation Records After Several Nested Calls

To fully understand how to pass static links from call to call, you must first understand the concept of a lexical level. Lexical levels in Pascal correspond to the static nesting levels of procedures and functions. Most compiler writers specify lex level zero as the main program. That is, all symbols you declare in your main program exist at lex level zero. Procedure and function names appearing in your main program define lex level one, *no matter how many procedures or functions appear in the main program*. They all begin a new copy of lex level one. For each level of nesting, Pascal introduces a new lex level. Figure 5.4 shows this.

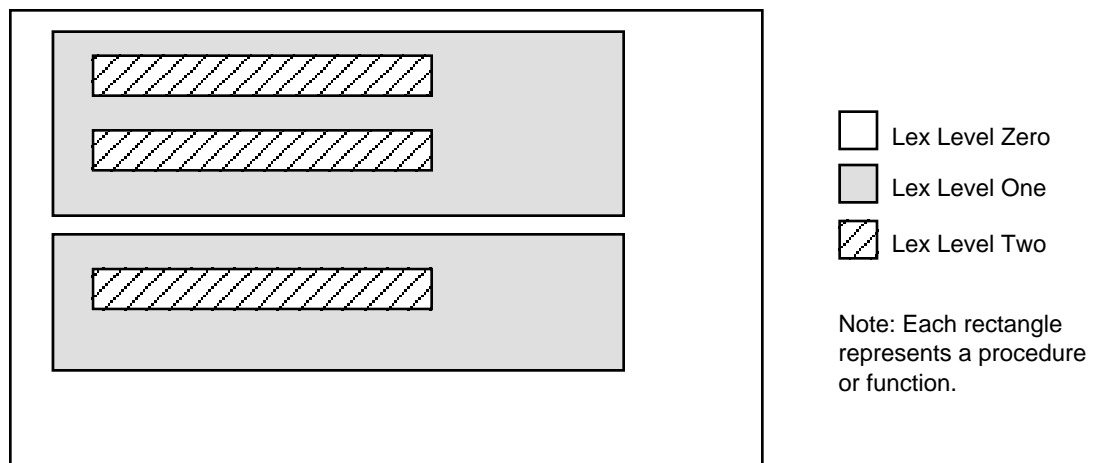


Figure 5.4 Procedure Schematic Showing Lexical Levels

During execution, a program may only access variables at a lex level less than or equal to the level of the current routine. Furthermore, only one set of values at any given lex level are accessible at any one time⁴ and those values are always in the most recent activation record at that lex level.

Before worrying about how to access non-local variables using a static link, you need to figure out how to pass the static link as a parameter. When passing the static link as a parameter to a program unit (procedure or function), there are three types of calling sequences to worry about:

- A program unit calls a child procedure or function. If the current lex level is n , then a child procedure or function is at lex level $n+1$ and is local to the current program unit. Note that most block structured languages do not allow calling procedures or functions at lex levels greater than $n+1$.
- A program unit calls a peer procedure or function. A peer procedure or function is one at the same lexical level as the current caller and a single program unit encloses both program units.
- A program unit calls an ancestor procedure or function. An ancestor unit is either the parent unit, a parent of an ancestor unit, or a peer of an ancestor unit.

Calling sequences for the first two types of calls above are very simple. For the sake of this example, assume the activation record for these procedures takes the generic form in Figure 5.5.

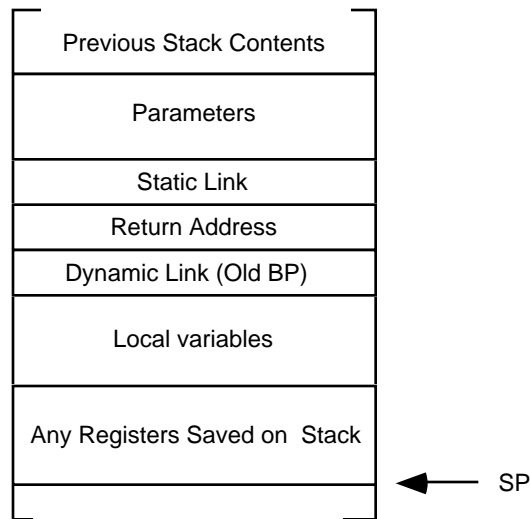


Figure 5.5 Generic Activation Record

When a parent procedure or function calls a child program unit, the static link is nothing more than the value in the EBP register immediately prior to the call. Therefore, to pass the static link to the child unit, just push EBP before executing the call instruction:

```
<Push Other Parameters onto the stack>
push( ebp );
call ChildUnit;
```

Of course the child unit can process the static link on the stack just like any other parameter. In this case, the static and dynamic links are exactly the same. In general, however, this is not true.

If a program unit calls a peer procedure or function, the current value in EBP is not the static link. It is a pointer to the caller's local variables and the peer procedure cannot access those variables. However, as peers, the caller and callee share the same parent program unit, so the caller can simply push a copy of its

4. There is one exception. If you have a *pointer* to a variable and the pointer remains accessible, you can access the data it points at even if the variable actually holding that data is inaccessible. Of course, in (standard) Pascal you cannot take the address of a local variable and put it into a pointer. However, certain dialects of Pascal (e.g., Turbo) and other block structured languages will allow this operation.

static link onto the stack before calling the peer procedure or function. The following code will do this assuming the current procedure's static link is on the stack immediately above the return address:

```
<Push Other Parameters onto the Stack>
    pushd( [ebp+8] );
    call PeerUnit;
```

Calling an ancestor is a little more complex. If you are currently at lex level n and you wish to call an ancestor at lex level m ($m < n$), you will need to traverse the list of static links to find the desired activation record. The static links form a list of activation records. By following this chain of activation records until it ends, you can step through the most recent activation records of all the enclosing procedures and functions of a particular program unit. The stack diagram in Figure 5.6 shows the static links for a sequence of procedure calls statically nested five lex levels deep.

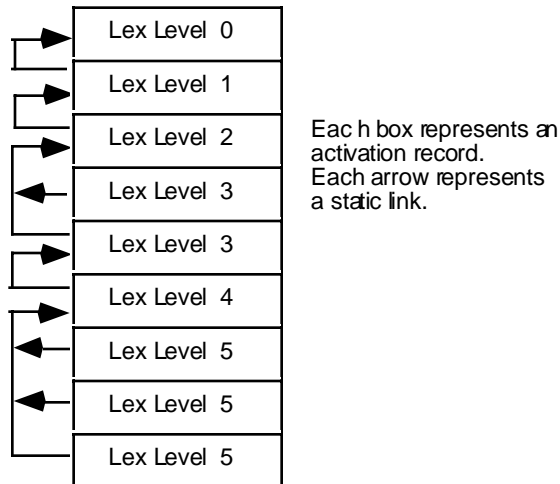


Figure 5.6 Static Links

If the program unit currently executing at lex level five wishes to call a procedure at lex level three, it must push a static link to the most recently activated program unit at lex level two. In order to find this static link you will have to *traverse* the chain of static links. If you are at lex level n and you want to call a procedure at lex level m you will have to traverse $(n-m)+1$ static links. The code to accomplish this is

```
// Current lex level is 5. This code locates the static link for,
// and then calls a procedure at lex level 2. Assume all calls are
// near:

<Push necessary parameters>

mov( [ebp+8], ebx ); // Traverse static link to LL 4.
mov( [ebx+8], ebx ); // To Lex Level 3.
mov( [ebx+8], ebx ); // To Lex Level 2.
pushd( [ebx+8] ); // Ptr to most recent LL 1 activation record.
call ProcAtLL2;
```

5.2.4 Accessing Non-Local Variables Using Static Links

In order to access a non-local variable, you must traverse the chain of static links until you get a pointer to the desired activation record. This operation is similar to locating the static link for a procedure call outlined in the previous section, except you traverse only $n-m$ static links rather than $(n-m)+1$ links to obtain a pointer to the appropriate activation record. Consider the following Pascal code:

```

procedure Outer;
var i:integer;

    procedure Middle;
    var j:integer;

        procedure Inner;
        var k:integer;
        begin
            k := 3;
            writeln(i+j+k);

        end;

    begin {middle}

        j := 2;
        writeln(i+j);
        Inner;

    end; {middle}

begin {Outer}

    i := 1;
    Middle;

end; {Outer}

```

The *Inner* procedure accesses global variables at lex level $n-1$ and $n-2$ (where n is the lex level of the *Inner* procedure). The *Middle* procedure accesses a single global variable at lex level $m-1$ (where m is the lex level of procedure *Middle*). The following HLA code could implement these three procedures:

```

procedure Inner; @nodisplay; @noframe;
var
    k:int32;
begin Inner;

    push( ebp );
    mov( esp, ebp );
    sub( _vars_, esp );           // Make room for k.

    mov( 3, k );                 // Initialize k.
    mov( [ebp+8], ebx );         // Static link to previous lex level.
    mov( [ebx-4], eax );         // Get j's value.
    add( k, eax );               // Add in k's value.
    mov( [ebx+8], ebx );         // Get static link to Outer's activation record.
    add( [ebx-4], eax );         // Add in i's value to sum.
    stdout.puti( eax );         // Display the sum.
    stdout.newln();

    mov( ebp, esp );            // Standard exit sequence.

```

```

    pop( ebp );
    ret( 4 );          // Removes the stack link from the stack.

end Inner;

procedure Middle; @nodisplay; @noframe;
var
    j:int32;
begin Middle;

    push( ebp );
    mov( esp, ebp );
    sub( _vars_, esp );    // Make room for j.

    mov( 2, j );          // Initialize j.
    mov( [ebp+8], ebx );  // Get the static link.
    mov( [ebx-4], eax );  // Get i's value.
    add( j, eax );        // Compute i+j.
    stdout.put( eax, nl ); // Display their sum.

    push( ebp );          // Static link for inner.
    call Inner;

    mov( ebp, esp );      // Standard exit sequence
    pop( ebp );
    ret( 4 );             // Removes static link from stack.
end Middle;

procedure Outer; @nodisplay; @noframe;
var
    i:int32;
begin Outer;

    push( ebp );
    mov( esp, ebp );
    sub( _vars_, esp );    // Make room for i.

    mov( 1, i );          // Give i an initial value.
    push( ebp );          // Static link for middle.
    call Middle;

    mov( ebp, esp );      // Remove local variables
    pop( ebp );
    ret( 4 );             // Removes static link.

end Outer;

```

Note that as the difference between the lex levels of the activation records increases, it becomes less and less efficient to access global variables. Accessing global variables in the previous activation record requires only one additional instruction per access, at two lex levels you need two additional instructions, etc. If you analyze a large number of Pascal programs, you will find that most of them do not nest procedures and functions and in the ones where there are nested program units, they rarely access global variables. There is one major exception, however. Although Pascal procedures and functions rarely access local variables inside other procedures and functions, they frequently access global variables declared in the main program. Since such variables appear at lex level zero, access to such variables would be as inefficient as possible when using the static links. To solve this minor problem, most 80x86 based block structured languages allocate variables at lex level zero directly in the STATIC segment and access them directly.

5.2.5 Nesting Procedures in HLA

The example in the previous treats the procedures, syntactically, as non-nested procedures and relies upon the programmer to manually handle the lexical nesting. A severe drawback to this mechanism is that it forces the programmer to manually compute the offsets of non-local variables. Although HLA does not provide automatic support for static links, HLA does allow us to nest procedures and provides some compile-time functions to help us calculate offsets into non-global activation records. Furthermore, we can treat the static link as a parameter to the procedures, so we don't have to refer to the static link using address expressions like "[ebx+8]".

Like Pascal, HLA lets you nest procedures. You may insert a procedure in the declaration section of another procedure. The Inner, Middle, and Outer procedures of the previous section could have been written in a fashion like the following:

```

procedure Outer; @nodisplay; @noframe;
var
    i:int32;

    procedure Middle; @nodisplay; @noframe;
    var
        j:int32;

        procedure Inner; @nodisplay; @noframe;
        var
            k:int32;
        begin Inner;

            << Code for the Inner procedure >>

        end Inner;

    begin Middle;

        << code for the Middle procedure >>

    end Middle;

begin Outer;

    << code for the Outer procedure >>

end Outer;

```

There are two advantages to this scheme:

1. The identifier *Inner* is local to the *Middle* procedure and is not accessible outside *Middle* (not even to *Outer*); similarly, the identifier *Middle* is local to *Outer* and is not accessible outside *Outer*. This information hiding feature lets you prevent other code from accidentally accessing these nested procedures, just as for local variables.
2. The local identifiers *i* and *j* are accessible to the nested procedures.

Before discussing how to use this feature to access non-local variables in a more reasonable fashion using static links, let's also consider the issue of the static link itself. The static link is really nothing more than a special parameter to these functions, therefore we can declare the static link as a parameter using HLA's high level procedure declaration syntax. Since the static link must always be at a fixed offset in the activation record for all procedures, the most reasonable thing to do is always make the stack link the first parameter in the list⁵; this ensures that the static link is always found at offset "+8" in the activation record. Here's the declarations above with the static links added as parameters:

```

procedure Outer( outerStaticLink:dword ); @nodisplay; @noframe;

```

```

var
  i:int32;

procedure Middle( middleStaticLink:dword ); @nodisplay; @noframe;
var
  j:int32;

  procedure Inner( innerStaticLink:dword ); @nodisplay; @noframe;
  var
    k:int32;
  begin Inner;

    << Code for the Inner procedure >>

  end Inner;

begin Middle;

  << code for the Middle procedure >>

end Middle;

begin Outer;

  << code for the Outer procedure >>

end Outer;

```

All that remains is to discuss how one references non-local (automatic) variables in this code. As you may recall from the chapter on Intermediate Procedures in Volume Four, HLA references local variables and parameters using an address expression of the form "[ebp±offset]" where offset represents the offset of the variable into the activation record (parameters typically have a positive offset, local variables have a negative offset). Indeed, we can use the HLA compile-time @offset function to access the variables without having to manually figure out the variable's offset in the activation record, e.g.,

```
mov( [ebp+@offset( i )], eax );
```

The statement above is semantically equivalent to

```
mov( i, eax );
```

assuming, of course, that *i* is a local variable in the current procedure.

Because HLA automatically associates the EBP register with local variables, HLA will not allow you to use a non-local variable reference in a procedure. For example, if you tried to use the statement "mov(i, eax);" in procedure *Inner* in the example above, HLA would complain that you cannot access non-local in this manner. The problem is that HLA associates EBP with automatic variables and outside the procedure in which you declare the local variable, EBP does not point at the activation record holding that variable. Hence, the instruction "mov(i, eax);" inside the *Inner* procedure would actually load *k* into EAX, not *i* (because *k* is at the same offset in *Inner's* activation record as *i* in *Outer's* activation record).

While it's nice that HLA prevents you from making the mistake of such an illegal reference, the fact remains that there needs to be some way of referring to non-local identifiers in a procedure. HLA uses the following syntax to reference a non-local, automatic, variable:

```
reg32::identifier
```

5. Assuming, of course, that you're using the default Pascal calling convention. If you were using the CDECL or STDCALL calling convention, you would always make the static link the last parameter in the parameter list.

`reg32` represents any of the 80x86's 32-bit general purpose registers and *identifier* is the non-local identifier you wish to access. HLA substitutes an address expression of the form "[reg₃₂+@offset(identifier)]" for this expression. Given this syntax, we can now rewrite the Inner, Middle, and Outer example in a high level fashion as follows:

```

procedure Outer( outerStaticLink:dword ); @nodisplay;
var
  i:int32;

  procedure Middle( middleStaticLink:dword ); @@nodisplay;
  var
    j:int32;

    procedure Inner( innerStaticLink:dword ); nodisplay;
    var
      k:int32;
    begin Inner;

      mov( 3, k );           // Initialize k.
      mov( innerStaticLink, ebx ); // Static link to previous lex level.
      mov( ebx::j, eax );    // Get j's value.
      add( k, eax );        // Add in k's value.

      // Get static link to Outer's activation record and
      // add in i's value:

      mov( ebx::outerStaticLink ebx );
      add( ebx::i, eax );

      // Display the results:

      stdout.puti( eax );    // Display the sum.
      stdout.newln();

    end Inner;

  begin Middle;

    mov( 2, j );           // Initialize j.
    mov( middleStaticLink, ebx ); // Get the static link.
    mov( ebx::i, eax );    // Get i's value.
    add( j, eax );        // Compute i+j.
    stdout.put( eax, nl ); // Display their sum.

    Inner( ebp );         // Inner's static link is EBP.

  end Middle;

begin Outer;

  mov( 1, i );           // Give i an initial value.
  Middle( ebp );        // Static link for middle.

end Outer;

```

This example provides only a small indication of the work needed to access variables using static links. In particular, accessing `@ebx::i` in the *Inner* procedure was simplified by the fact that EBX already contained *Middle's* static link. In the typical case, it's going to take one instruction for each lex level the code

traverses in order to access a given non-local automatic variable. While this might seem bad, in typical programs you rarely access non-local variables, so the situation doesn't arrive often enough to worry about.

HLA does not provide built-in support for static links. If you are going to use static links in your programs, then you must manually pass the static links as parameters to your procedures (i.e., HLA will not take care of this for you). While it is possible to modify HLA to automatically handle static links for you, HLA provides a different mechanism for accessing non-local variables - the display. To learn about displays, keep reading...

5.2.6 The Display

After reading the previous section you might get the idea that one should never use non-local variables, or limit non-local accesses to those variables declared at lex level zero. After all, it's often easy enough to put all shared variables at lex level zero. If you are designing a programming language, you can adopt the C language designer's philosophy and simply not provide block structure. Such compromises turn out to be unnecessary. There is a data structure, the *display*, that provides efficient access to any set of non-local variables.

A display is simply an array of pointers to activation records. *Display[0]* contains a pointer to the most recent activation record for lex level zero, *Display[1]* contains a pointer to the most recent activation record for lex level one, and so on. Assuming you've maintained the *Display* array in the current STATIC segment it only takes two instructions to access any non-local variable. Pictorially, the display works as shown in Figure 5.7.

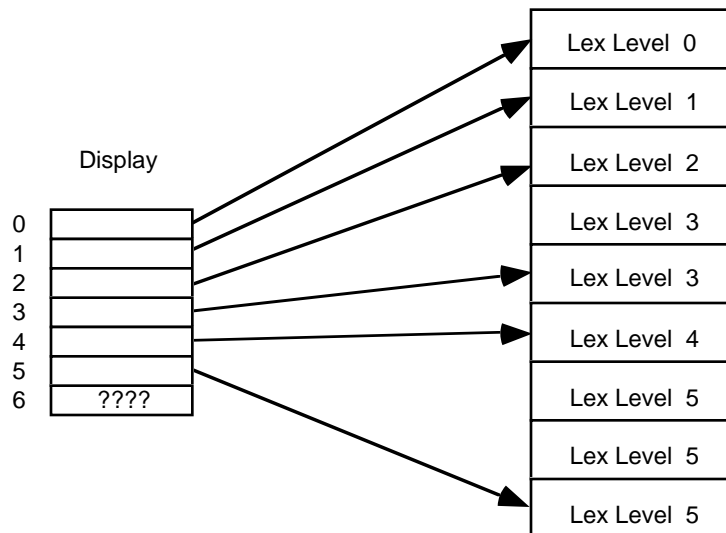


Figure 5.7 The Display

Note that the entries in the display always point at the most recent activation record for a procedure at the given lex level. If there is no active activation record for a particular lex level (e.g., lex level six above), then the entry in the display contains garbage.

The maximum lexical nesting level in your program determines how many elements there must be in the display. Most programs have only three or four nested procedures (if that many) so the display is usually quite small. Generally, you will rarely require more than 10 or so elements in the display.

Another advantage to using a display is that each individual procedure can maintain the display information itself, the caller need not get involved. When using static links the calling code has to compute and pass the appropriate static link to a procedure. Not only is this slow, but the code to do this must appear before every call. If your program uses a display, the callee, rather than the caller, maintains the display so you only need one copy of the code per procedure.

Although maintaining a single display in the STATIC segment is easy and efficient, there are a few situations where it doesn't work. In particular, when passing procedures as parameters, the single level display doesn't do the job. So for the general case, a solution other than a static array is necessary. Therefore, this chapter will not go into the details of how to maintain a static display since there are some problems with this approach.

Intel, when designing the 80286 microprocessor, studied this problem very carefully (because Pascal was popular at the time and they wanted to be able to efficiently handle Pascal constructs). They came up with a generalized solution that works for all cases. Rather than using a single display in a static segment, Intel's designers decided to have each procedure carry around its own local copy of the display. The HLA compiler automatically builds an Intel-compatible display at the beginning of each procedure, assuming you don't use the @NODISPLAY procedure option. An Intel-compatible display is part of a procedure's activation record and takes the form shown in Figure 5.8:

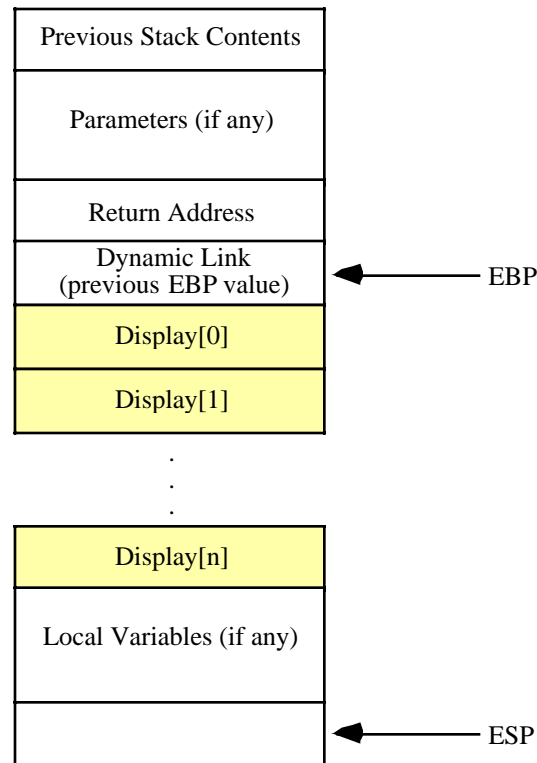


Figure 5.8 Intel-Compatible Display in an Activation Record

If we assume that the lex level of the main program is zero, then the display for a given procedure at lex level n will contain $n+1$ double word elements. *Display[0]* is a pointer to the activation record for the main program, *Display[1]* is a pointer to the activation record of the most recently activated procedure at lex level one. Etc. *Display[n]* is a pointer to the current procedure's activation record (i.e., it contains the value found in EBP while this procedure executes). Normally, the procedure would never access element n of *Display* since the procedure can index off EBP directly; However, as you'll soon see, we'll need the *Display[n]* entry to build displays for procedures at higher lex levels.

One important fact to note about the Intel-compatible display array: its elements appear backwards in memory. Remember, the stack grows downwards from high addresses to low addresses. If you study Figure 5.8 for a moment you'll discover that *Display[0]* is at the highest memory address and *Display[n]* is at the lowest memory address, exactly the opposite for standard array organization. It turns out that we'll always access the display using a constant offset, so this reversal of the array ordering is no big deal. We'll just use negative offsets from *Display[0]* (the base address of the array) rather than the usual positive offsets.

If the @NODISPLAY procedure option is not present, HLA treats the display as a predeclared local variable in the procedure and inserts the name "_display_" into the symbol table. The offset of the *_display_* variable in the activation record is the offset of the *Display[0]* entry in Figure 5.8. Therefore, you can easily access an element of this array at run-time using a statement like:

```
mov( _display_[ -lexLevel*4 ], ebx );
```

The "*4" component appears because *_display_* is an array of double words. *lexLevel* must be a constant value that specifies the lex level of the procedure whose activation record you'd like to obtain. The minus sign prefixing this expression causes HLA to index downwards in memory as appropriate for the display object.

Although it's not that difficult to figure out the lex level of a procedure manually, the HLA compile-time language provides a function that will compute the lex level of a given procedure for you – the @LEX function. This function accepts a single parameter that must be the name of an HLA procedure (that is currently in scope). The @LEX function returns an appropriate value for that function that you can use as an index into the *_display_* array. Note that @LEX returns one for the main program, two for procedures you declare in the main program, three for procedures you declare in procedures you declare in the main program, etc. If you are writing a unit, all procedures you declare in that unit exist at lex level two.

The following program is a variation of the Inner/Middle/Outer example you've seen previously in this chapter. This example uses displays and the @LEX function to access the non-local automatic variables:

```
program DisplayDemo;
#include( "stdlib.hhf" )

macro Display( proc );

    _display_[ -@lex( proc ) * 4]

endmacro;

procedure Outer;
var
    i:int32;

    procedure Middle;
    var
        j:int32;

        procedure Inner;
        var
            k:int32;
        begin Inner;

            mov( 4, k );
            mov( Display( Middle ), ebx );
            mov( ebx::j, eax );           // Get j's value.
            add( k, eax );               // Add in k's value.

            // Get static link to Outer's activation record and
```

```

        // add in i's value:

        mov( Display( Outer ), ebx );
        add( ebx::i, eax );

        // Display the results:

        stdout.puti32( eax );           // Display the sum.
        stdout.newln();

    end Inner;

begin Middle;

    mov( 2, j );                       // Initialize j.
    mov( Display( Outer ), ebx );      // Get the static link.
    mov( ebx::i, eax );                // Get i's value.
    add( j, eax );                      // Compute i+j.
    stdout.puti32( eax );              // Display their sum.
    stdout.newln();

    Inner();

end Middle;

begin Outer;

    mov( 1, i );                       // Give i an initial value.
    Middle();                           // Static link for middle.

end Outer;

begin DisplayDemo;

    Outer();

end DisplayDemo;

```

Program 5.1 Demonstration of Displays in an HLA Program

Assuming you do not attach the @NODISPLAY procedure option to a procedure you write in HLA, HLA will automatically emit the code (as part of the standard entry sequence) to build a display for that procedure. Up to this chapter, none of the programs in this text have used nested procedures⁶, therefore there has been no need for a display. For that reason, most programs appearing in this text (since the introduction of the @NODISPLAY option) have attached @NODISPLAY to the procedure. It doesn't make a program incorrect to build a display if you never use it, but it does make the procedure a tiny bit slower and a tiny bit larger, hence the use of the @NODISPLAY option up to this point.

6. Technically, this statement is not true. Every procedure you've written has been nested inside the main program. However, none of the sample programs to date have considered the possibility of accessing the main program's automatic (VAR) variables. Hence there has been no need for a display until now).

5.2.7 The 80x86 ENTER and LEAVE Instructions

When designing the 80286, Intel's CPU designers decided to add two instructions to help maintain displays. This was done because Pascal was the popular high level language at the time and Pascal was a block structured language that could benefit from having a display. Since then, C/C++ has replaced Pascal as the most common implementation language, so these two instructions have fallen into disuse since C/C++ is not a block structured language. Still, you can take advantage of these instructions when writing assembly code with nested procedures.

Unfortunately, these two instructions, ENTER and LEAVE, are quite slow. The problem with these instructions is that C/C++ became popular shortly after Intel designed these instructions, so Intel never bothered to optimize them since few high-performance compilers actually used these instructions. On today's processors, it's actually faster to execute a sequence of instructions that do the same job than it is to actually use these instructions; hence most compilers that build displays (like HLA) emit a discrete sequence of instructions to build the display. Do keep in mind that, although these two instructions are slower than their discrete counterparts, they are generally shorter. So if you're trying to save code space rather than write the fastest possible code, using ENTER and LEAVE can help.

The LEAVE instruction is very simple to understand. It performs the same operation as the two instructions:

```
mov( ebp, esp );
pop( ebp );
```

Therefore, you may use the instruction for the standard procedure exit code. On an 80386 or earlier processor, the LEAVE instruction is faster than the equivalent move and pop sequence. However, the LEAVE instruction is slower on 80486 and later processors.

The ENTER instruction takes two operands. The first is the number of bytes of local storage the current procedure requires, the second is the lex level of the current procedure. The enter instruction does the following:

```
// enter( Locals, LexLevel );

    push( ebp );           // Save dynamic link
    mov( esp, tempreg );  // Save for later.
    cmp( LexLevel, 0 );   // Done if this is lex level zero.
    je Lex0;
lp:  dec( LexLevel );
    jz Done;
    sub( 4, ebp );        // Index into display in previous activation record
    pushd( [ebp] );      // and push the element there.
    jmp lp;

Done:
    push( tempreg );     // Add entry for current lex level.
Lex0:
    mov( tempreg, ebp ); // Pointer to current activation record.
    sub( _vars_, esp );  // Allocate storage for local variables.
```

As you can see from this code, the ENTER instruction copies the display from activation record to activation record. This can get quite expensive if you nest the procedures to any depth. Most high level languages, if they use the ENTER instruction at all, always specify a nesting level of zero to avoid copying the display throughout the stack.

The ENTER instruction puts the value for the *_display[n]* entry at location EBP-(n*4). The ENTER instruction does not copy the value for *display[0]* into each stack frame. Intel assumes that you will keep the main program's global variables in the data segment. To save time and memory, they do not bother copying the *_display[0]* entry. This is why HLA uses lex level one for the main program – in HLA the main program can have automatic variables and, therefore, requires a display entry.

The ENTER instruction is very slow, particularly on 80486 and later processors. If you really want to copy the display from activation record to activation record it is probably a better idea to push the items yourself. The following code snippets show how to do this:

```
// enter( n, 0 ); (n bytes of local variables, lex level zero.)

    push( ebp );           // As you can see, "enter( n, 0 );" corresponds to
    mov( esp, ebp );      // the standard entry sequence for non-nested
    sub( n, esp );         // procedures.

// enter( n, 1 );

    push( ebp );           // Save dynamic link (current EBP value).
    pushd( [ebp-4] );      // Push display[1] entry from previous act rec.
    lea( ebp, [esp-4] );   // Point EBP at the base of new act rec.
    sub( n, esp );         // Allocate local variables.

// enter( n, 2 );

    push( ebp );           // Save dynamic link (current EBP value).
    pushd( [ebp-4] );      // Push display[1] entry from previous act rec.
    pushd( [ebp-8] );      // Push display[2] entry from previous act rec.
    lea( ebp, [esp-8] );   // Point EBP at the base of new act rec.
    sub( n, esp );         // Allocate local variables.

// enter( n, 3 );

    push( ebp );           // Save dynamic link (current EBP value).
    pushd( [ebp-4] );      // Push display[1] entry from previous act rec.
    pushd( [ebp-8] );      // Push display[2] entry from previous act rec.
    pushd( [ebp-12] );     // Push display[3] entry from previous act rec.
    lea( ebp, [esp-12] );  // Point EBP at the base of new act rec.
    sub( n, esp );         // Allocate local variables.

// enter( n, 4 );

    push( ebp );           // Save dynamic link (current EBP value).
    pushd( [ebp-4] );      // Push display[1] entry from previous act rec.
    pushd( [ebp-8] );      // Push display[2] entry from previous act rec.
    pushd( [ebp-12] );     // Push display[3] entry from previous act rec.
    pushd( [ebp-16] );     // Push display[3] entry from previous act rec.
    lea( ebp, [esp-16] );  // Point EBP at the base of new act rec.
    sub( n, esp );         // Allocate local variables.

// etc.
```

If you are willing to believe Intel's cycle timings, you'll find that the ENTER instruction is almost never faster than a straight line sequence of instructions that accomplish the same thing. If you are interested in saving space rather than writing fast code, the ENTER instruction is generally a better alternative. The same is generally true for the LEAVE instruction as well. It is only one byte long, but it is slower than the corresponding "mov(esp, ebp);" and "pop(ebp);" instructions. The following sample program demonstrates how to access non-local variables using a display. This code does not use the @LEX function in the interest of making the lex level access clear; normally you would use the @LEX function rather than the literal constants appearing in this example.

```
program EnterLeaveDemo;
#include( "stdlib.hhf" )
```

```

procedure LexLevel2;

    procedure LexLevel3a;
    begin LexLevel3a;

        stdout.put( nl "LexLevel3a:" nl );
        stdout.put( "esp = ", esp, " ebp = ", ebp, nl );
        mov( _display_[0], eax );
        stdout.put( "display[0] = ", eax, nl );
        mov( _display_[-4], eax );
        stdout.put( "display[-1] = ", eax, nl );

    end LexLevel3a;

    procedure LexLevel3b; noframe;
    begin LexLevel3b;

        enter( 0, 3 );

        stdout.put( nl "LexLevel3b:" nl );
        stdout.put( "esp = ", esp, " ebp = ", ebp, nl );
        mov( _display_[0], eax );
        stdout.put( "display[0] = ", eax, nl );
        mov( _display_[-4], eax );
        stdout.put( "display[-1] = ", eax, nl );

        leave;
        ret();

    end LexLevel3b;

begin LexLevel2;

    stdout.put( "LexLevel2: esp=", esp, " ebp = ", ebp, nl nl );
    LexLevel3a();
    LexLevel3b();

end LexLevel2;

begin EnterLeaveDemo;

    stdout.put( "main: esp = ", esp, " ebp= ", ebp, nl );
    LexLevel2();

end EnterLeaveDemo;

```

Program 5.2 Demonstration of Enter and Leave in HLA

Starting with HLA v1.32, HLA provides the option of emitting ENTER or LEAVE instructions rather than the discrete sequences for a procedure's standard entry and exit sequences. The @ENTER procedure option tells HLA to emit the ENTER instruction for a procedure, the @LEAVE procedure option tells HLA to emit the LEAVE instruction in place of the standard exit sequence. See the HLA documentation for more details.

5.3 Passing Variables at Different Lex Levels as Parameters.

Accessing variables at different lex levels in a block structured program introduces several complexities to a program. The previous section introduced you to the complexity of non-local variable access. This problem gets even worse when you try to pass such variables as parameters to another program unit. The following subsections discuss strategies for each of the major parameter passing mechanisms.

For the purposes of discussion, the following sections will assume that “local” refers to variables in the current activation record, “global” refers to static variables in a static segment, and “intermediate” refers to automatic variables in some activation record other than the current activation record (this includes automatic variables in the main program). These sections will pass all parameters on the stack. You can easily modify the details to pass these parameters elsewhere, should you choose.

5.3.1 Passing Parameters by Value

Passing value parameters to a program unit is no more difficult than accessing the corresponding variables; all you need do is push the value on the stack before calling the associated procedure.

To (manually) pass a global variable by value to another procedure, you could use code like the following:

```
push( GlobalVariable ); // Assume "GlobalVariable" is a static object.
call proc;
```

To pass a local variable by value to another procedure, you could use the following code⁷:

```
push( LocalVariable );
call proc;
```

To pass an intermediate variable as a value parameter, you must first locate that intermediate variable’s activation record and then push its value onto the stack. The exact mechanism you use depends on whether you are using static links or a display to keep track of the intermediate variable’s activation records. If using static links, you might use code like the following to pass a variable from two lex levels up from the current procedure:

```
mov( [ebp+8], ebx ); // Assume static link is at offset 8 in Act Rec.
mov( [ebx], ebx ); // Traverse the second static link.
push( ebx::IntVar ); // Push the intermediate variable’s value.
call proc;
```

Passing an intermediate variable by value when you are using a display is somewhat easier. You could use code like the following to pass an intermediate variable from lex level one:

```
mov( _display_[ -1*4 ], ebx ); // Remember each _display_ entry is 4 bytes.
push( ebx::IntVar ); // Pass the intermediate variable.
call proc;
```

It is possible to use the HLA high level procedure calling syntax when passing intermediate variables as parameters by value. The following code demonstrates this:

```
mov( _display_[ -1*4 ], ebx );
proc( ebx::IntVar );
```

This example uses a display because HLA automatically builds the display for you. If you decide to use static links, you’ll have to modify this code appropriately.

7. The non-global examples all assume the variable is at offset -2 in their activation record. Change this as appropriate in your code.

5.3.2 Passing Parameters by Reference, Result, and Value-Result

The pass by reference, result, and value-result parameter mechanisms generally pass the address of parameter on the stack⁸. In an earlier chapter, you've seen how to pass global and local parameters using these mechanisms. In this section we'll take a look at passing intermediate variables by reference, value/result, and by result.

To pass an intermediate variable by reference, value/result, or by result, you must first locate the activation record containing the variable so you can compute the effective address into the stack segment. When using static links, the code to pass the parameter's address might look like the following:

```
mov( [ebp+8], ebx ); // Assume static link is at offset 8 in Act Rec.
mov( [ebx], ebx ); // Traverse the second static link.
lea( eax, ebx::IntVar ); // Get the intermediate variable's address.
push( eax ); // Pass the address on the stack.
call proc;
```

When using a display, the calling sequence might look like the following:

```
mov( _display_[ -1*4 ], ebx ); // Remember each _display_ entry is 4 bytes.
lea( eax, ebx::IntVar ); // Pass the intermediate variable.
push( eax );
call proc;
```

It is possible to use the HLA high level procedure calling syntax when passing parameters by reference, by value/result, or by result. The following code demonstrates this:

```
mov( _display_[ -1*4 ], ebx );
proc( ebx::IntVar );
```

The nice thing about the high level syntax is that it is identical whether you're passing parameters by value, reference, value/result, or by result.

As you may recall from the chapter on Low-Level Parameter Implementation, there is a second way to pass a parameter by value/result. You can push the value onto the stack and then, when the procedure returns, pop this value off the stack and store it back into the variable from whence it came. This is just a special case of the pass by value mechanism described in the previous section.

5.3.3 Passing Parameters by Name and Lazy-Evaluation in a Block Structured Language

Since you pass a thunk when passing parameters by name or by lazy-evaluation, the presence of global, intermediate, and local variables does not affect the calling sequence to the procedure. Instead, the thunk has to deal with the differing locations of these variables. Since HLA thunks already contain the pointer to the activation record for that thunk, returning a local (to the thunk) variable's address or value is especially trivial. About the only catch is what happens if you pass an intermediate variable by name or by lazy evaluation to a procedure. However, the calculation of the ultimate address (pass by name) or retrieval of the value (pass by lazy evaluation) is nearly identical to the code in the previous two sections. Hence, this code will be left as an exercise at the end of this volume.

8. As you may recall, pass by reference, value-result, and result all use the same calling sequence. The differences lie in the procedures themselves.

5.4 Passing Procedures as Parameters

Many programming languages let you pass a procedure or function name as a parameter. This lets the caller pass along various actions to perform inside a procedure. The classic example is a plot procedure that graphs some generic math function passed as a parameter to plot.

HLA lets you pass procedures and functions by declaring them as follows:

```
procedure DoCall( x:procedure );
begin DoCall;

    x();

end DoCall;
```

The statement "DoCall(xyz);" calls *DoCall* that, in turn, calls procedure *xyz*.

Whenever you pass a procedure's address in this manner, HLA only passes the address of the procedure as the parameter value. Upon entry into procedure *x* via the *DoCall* invocation, the *x* procedure first creates its own display by copying appropriate entries from *DoCall*'s display. This gives *x* access to all intermediate variables that HLA allows *x* to access.

Keep in mind that thunks are special cases of functions that you call indirectly. However, there is a major difference between a thunk and a procedure – thunks carry around the pointer to the activation record they intend to use. Therefore, the thunk does not copy the calling procedure's display; instead, it uses the display of an existing procedure to access intermediate variables.

5.5 Faking Intermediate Variable Access

As you've probably noticed by now, accessing non-local (intermediate) variables is a bit less efficient than accessing local or global (static) variables. High level languages like Pascal that support intermediate variable access hide a lot of effort from the programmer that becomes painfully visible when attempting the same thing in assembly language. When attempting to write maintainable and readable code, you may want to break up a large procedure into a sequence of smaller procedures and make those smaller procedures local to a surrounding procedure that simply calls these smaller routines. Unfortunately, if the original procedure you're breaking up contains lots of local variables that code throughout the procedure shares, short of restructuring your code you will have to leave those variables in the outside procedure and access them as intermediate variables. Using the techniques of this chapter may make this task a bit unpleasant, especially if you access those variables a large number of times. This may dissuade you from attempting to break up the procedure into smaller units. Fortunately, under certain special circumstances, you can avoid the headaches of intermediate variable access in situations like this.

Consider the following short code sequence:

```
procedure MainProc;
var
    ALocalVar: dword;

    procedure proc; @nodisplay; @noframe;
    begin proc;

        mov( ebp::ALocalVar, eax );
        ret();

    end proc;

begin MainProc;

    mov( 5, ALocalVar );
```



```
proc();  
  
// EAX now contains five...  
  
end MainProc;
```

Notice that the *proc* procedure has the @NOFRAME option, so HLA does not emit the standard entry sequence to build an activation record. This means that upon entry to *proc*, EBP still points at *MainProc*'s activation record. Therefore, this code can access the *ALocalVar* variable by using the syntax *ebp::ALocalVar*. No other code is necessary.

The drawback to this scheme is that *proc* may not contain any parameters or local variables (which would require setting EBP to point at *proc*'s activation record). However, if you can live with this limitation, then this is a useful trick for accessing local variables one lex level up from the current procedure.

5.6 Putting It All Together

This chapter introduces the concept of lexical nesting commonly found in block structured languages like Pascal, Ada, and Modula-2. This chapter introduces the notion of scope, static procedure nesting, binding, variable lifetime, static links, the display, intermediate variables, and passing intermediate variables as parameters. Although few assembly programs use these features, they are occasionally useful, especially when writing code that interfaces with a high level language that supports static nesting.

Answers to Selected Exercises

Appendix A




To be written.


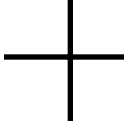
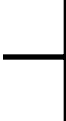
My apologies that this isn't ready yet, but other chapters and appendices in this text have a higher priority. I will get around to this appendix eventually.




In the meantime, if you have some questions about the answers to any exercises in this text, please feel free to post a question to one of the internet newsgroups like "comp.lang.asm.x86" or "alt.lang.asm". Because of the high volume of email I receive daily, I will not answer questions sent to me via email. Note that posting the message to the net is very efficient because others get to share the solution. So please post your questions there.

Console Graphic Characters




Appendix B




<p>\$DA 21 8</p> 	<p>\$C2 194</p> 	<p>\$BF 19 1</p> 
---	--	---

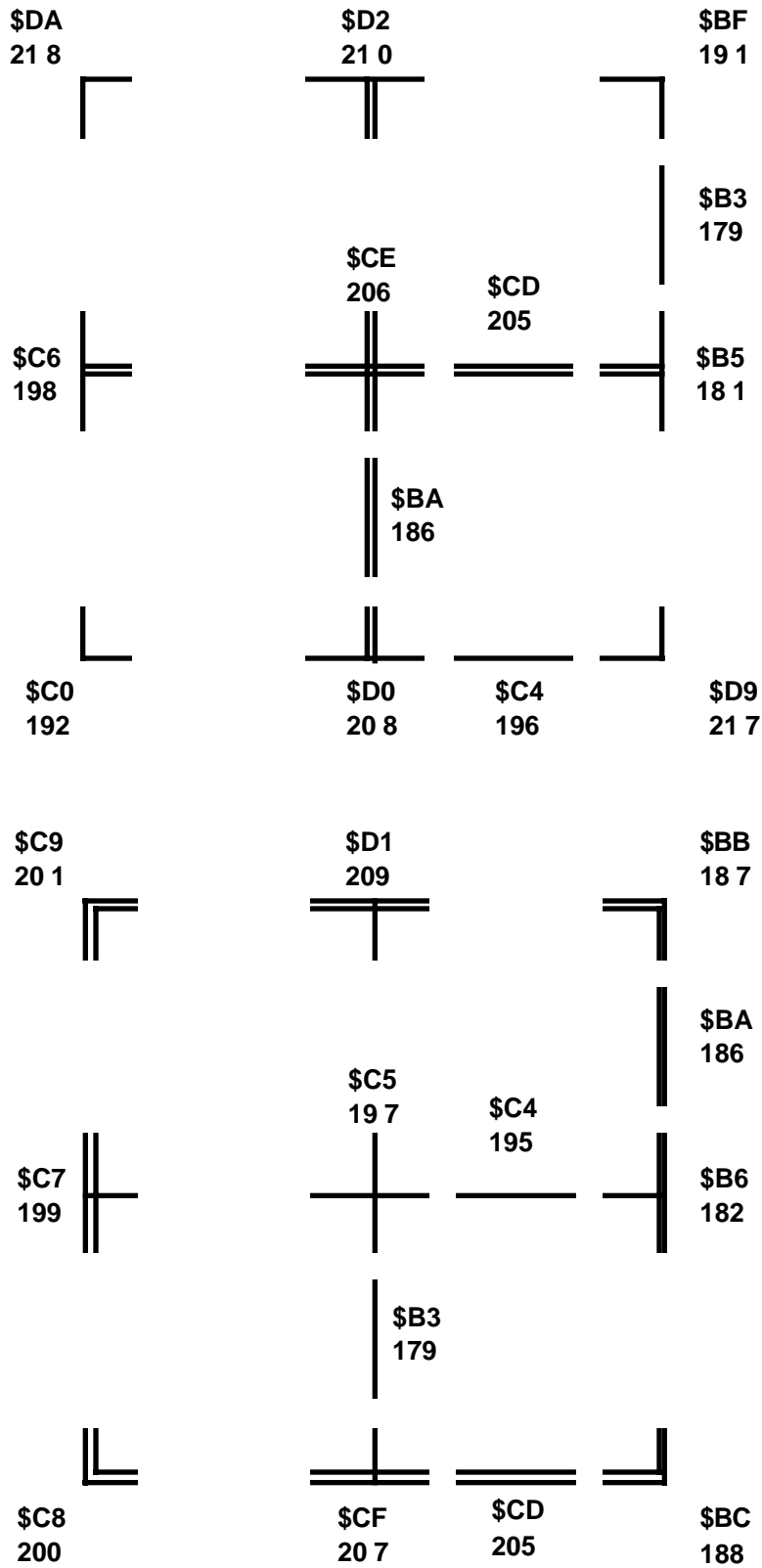
<p>\$C3 195</p> 	<p>\$C5 19 7</p> 	<p>\$C4 196</p>	<p>\$B3 179</p>	<p>\$B4 180</p> 
--	---	---	--	--






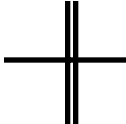








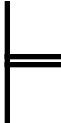
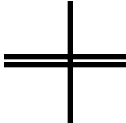





<p>\$C0 192</p> 	<p>\$C1 19 3</p> 	<p>\$D9 217</p> 
--	---	--

<p>\$C9 201</p> 	<p>\$CB 203</p> 	<p>\$BB 187</p> 
--	--	--

<p>\$CC 204</p> 	<p>\$CE 206</p> 	<p>\$CD 205</p>	<p>\$BA 186</p>	<p>\$B9 185</p> 
--	--	---	--	--

<p>\$C8 200</p> 	<p>\$CA 202</p> 	<p>\$BC 188</p> 
--	--	--



\$D6 21 4		\$D2 21 0		\$B7 18 3	
				\$BA 18 6	
\$C7 19 9		\$D7 21 5		\$C4 19 6	
				\$B6 18 2	
\$D3 21 1		\$D0 20 8		\$BD 18 9	
\$D5 21 3		\$D1 20 9		\$B8 18 4	
\$C6 19 8		\$D8 21 6		\$CD 20 5	
				\$B5 18 1	
\$D4 21 2		\$CF 20 7		\$BE 19 0	

HLA Programming Style Guidelines

Appendix C

C.1 Introduction

Most people consider assembly language programs difficult to read. While there are a multitude of reasons why people feel this way, the primary reason is that assembly language does not make it easy for programmers to write readable programs. This doesn't mean it's impossible to write readable programs, only that it takes an extra effort on the part of an assembly language programmer to produce readable code.

One of the design goals of the High Level Assembler (HLA) was to make it possible for assembly language programmers to write readable assembly language programs. Nevertheless, without discipline, pandemonium will result in any program of any decent size. Even if you adhere to a fixed set of style guidelines, others may still have trouble reading and understanding your code. Equally important to following a set of style guidelines is that you following a generally accepted set of style guidelines; guidelines that others are familiar and agree with. The purpose of this appendix, written by the designer of the HLA language, is to provide a consistent set of guidelines that HLA programmers can use consistently. Unless you can show a good reason to violate these rules, you should following them carefully when writing HLA programs; other HLA programmers will thank you for this.

C.1.1 Intended Audience

Of course, an assembly language program is going to be nearly unreadable to someone who doesn't know assembly language. This is true for almost any programming language. Other than burying a tutorial on 80x86 assembly language in a program's comments, there is no way to address this problem¹ other than to assume that the reader is familiar with assembly language programming and specifically HLA.

In view of the above, it makes sense to define an "intended audience" that we intend to have read our assembly language programs. Such a person should:

- Be a reasonably competent 80x86 assembly language/HLA programmer.
- Be reasonably familiar with the problem the assembly language program is attempting to solve.
- Fluently read English².
- Have a good grasp of high level language concepts.
- Possess appropriate knowledge for someone working in the field of Computer Science (e.g., understands standard algorithms and data structures, understands basic machine architecture, and understands basic discrete mathematics).

C.1.2 Readability Metrics

One has to ask "What is it that makes one program more readable than another?" In other words, how do we measure the "readability" of a program? The usual metric, "I know a well-written program when I see one" is inappropriate; for most people, this translates to "If your programs look like my better programs then they are readable, otherwise they are not." Obviously, such a metric is of little value since it changes with every person.

To develop a metric for measuring the readability of an assembly language program, the first thing we must ask is "Why is readability important?" This question has a simple (though somewhat flippant) answer:

-
1. Doing so (inserting an 80x86 tutorial into your comments) would wind up making the program *less* readable to those who already know assembly language since, at the very least, they'd have to skip over this material; at the worst they'd have to read it (wasting their time).
 2. Or whatever other natural language is in use at the site(s) where you develop, maintain, and use the software.

Readability is important because programs are *read* (furthermore, a line of code is typically read ten times more often than it is written). To expand on this, consider the fact that most programs are read and maintained by other programmers (Steve McConnell claims that up to ten generations of maintenance programmers work on a typical real world program before it is rewritten from scratch; furthermore, they spend up to 60% of their effort on that code simply figuring out how it works). The more readable your programs are, the less time these other people will have to spend figuring out what your program does. Instead, they can concentrate on adding features or correcting defects in the code.

For the purposes of this document, we will define a "readable" program as one that has the following trait:

- A "readable" program is one that a competent programmer (one who is familiar with the problem the program is attempting to solve) can pick up, without ever having seen the program before, and fully comprehend the entire program in a minimal amount of time.

That's a tall order! This definition doesn't sound very difficult to achieve, but few non-trivial programs ever really achieve this status. This definition suggests that an appropriate programmer (i.e., one who is familiar with the problem the program is trying to solve) can pick up a program, read it at their normal reading pace (just once), and fully comprehend the program. Anything less is not a "readable" program.

Of course, in practice, this definition is unusable since very few programs reach this goal. Part of the problem is that programs tend to be quite long and few human beings are capable of managing a large number of details in their head at one time. Furthermore, no matter how well-written a program may be, "a competent programmer" does not suggest that the programmer's IQ is so high they can read a statement a fully comprehend its meaning without expending much thought. Therefore, we must define readability, not as a boolean entity, but as a scale. Although truly unreadable programs exist, there are many "readable" programs that are less readable than other programs. Therefore, perhaps the following definition is more realistic:

- A readable program is one that consists of one or more *modules*. A competent program should be able to pick a given module in that program and achieve an 80% comprehension level by expending no more than an average of one minute for each statement in the program.

An 80% comprehension level means that the programmer can correct bugs in the program and add new features to the program without making mistakes due to a misunderstanding of the code at hand.

C.1.3 How to Achieve Readability

The "I'll know one when I see one" metric for readable programs provides a big hint concerning how one should write programs that are readable. As pointed out early, the "I'll know it when I see it" metric suggests that an individual will consider a program to be readable if it is very similar to (good) programs that this particular person has written. This suggests an important trait that readable programs must possess: consistency. If all programmers were to write programs using a consistent style, they'd find programs written by others to be similar to their own, and, therefore, easier to read. This single goal is the primary purpose of this appendix - to suggest a consistent standard that everyone will follow.

Of course, consistency by itself is not good enough. Consistently bad programs are not particularly easy to read. Therefore, one must carefully consider the guidelines to use when defining an all-encompassing standard. The purpose of this paper is to create such a standard. However, don't get the impression that the material appearing in this document appears simply because it sounded good at the time or because of some personal preferences. The material in this paper comes from several software engineering texts on the subject (including *Elements of Programming Style*, *Code Complete*, and *Writing Solid Code*), nearly 20 years of personal assembly language programming experience, and research that led to the development of a set of generic programming guidelines for industrial use.

This document assumes consistent usage by its readers. Therefore, it concentrates on a lot of mechanical and psychological issues that affect the readability of a program. For example, uppercase letters are harder to read than lower case letters (this is a well-known result from psychology research). It takes longer

for a human being to recognize uppercase characters, therefore, an average human being will take more time to read text written all in upper case. Hence, this document suggests that one should avoid the use of uppercase sequences in a program. Many of the other issues appearing in this document are in a similar vein; they suggest minor changes to the way you might write your programs that make it easier for someone to recognize some pattern in your code, thus aiding in comprehension.

C.1.4 How This Document is Organized

This document follows a top-down discussion of readability. It starts with the concept of a program. Then it discusses modules. From there it works its way down to procedures. Then it talks about individual statements. Beyond that, it talks about components that make up statements (e.g., instructions, names, and operators). Finally, this paper concludes by discussing some orthogonal issues.

Section Two discusses programs in general. It primarily discusses documentation that must accompany a program and the organization of source files. It also discusses, briefly, configuration management and source code control issues. Keep in mind that figuring out how to build a program (make, assemble, link, test, debug, etc.) is important. If your reader fully understands the "heapsort" algorithm you are using, but cannot build an executable module to run, they still do not fully understand your program.

Section Three discusses how to organize modules in your program in a logical fashion. This makes it easier for others to locate sections of code and organizes related sections of code together so someone can easily find important code and ignore unimportant or unrelated code while attempting to understand what your program does.

Section Four discusses the use of procedures within a program. This is a continuation of the theme in Section Three, although at a lower, more detailed, level.

Section Five discusses the program at the level of the statement. This (large) section provides the meat of this proposal. Most of the rules this paper presents appear in this section.

Section Six discusses comments and other documentation appearing within the source code.

Section Seven discusses those items that make up a statement (labels, names, instructions, operands, operators, etc.) This is another large section that presents a large number of rules one should follow when writing readable programs. This section discusses naming conventions, appropriateness of operators, and so on.

Section Eight discusses data types and other related topics.

C.1.5 Guidelines, Rules, Enforced Rules, and Exceptions

Not all rules are equally important. For example, a rule that you check the spelling of all the words in your comments is probably less important than suggesting that the comments all be in English³. Therefore, this paper uses three designations to keep things straight: Guidelines, Rules, and Enforced Rules.

A Guideline is a suggestion. It is a rule you should follow unless you can verbally defend why you should break the rule. As long as there is a good, defensible, reason, you should feel no apprehension violated a guideline. Guidelines exist in order to encourage consistency in areas where there are no good reasons for choosing one methodology over another. You shouldn't violate a Guideline just because you don't like it -- doing so will make your programs inconsistent with respect to other programs that do follow the Guideline (and, therefore, harder to read), however, you shouldn't lose any sleep because you violated a Guideline.

Rules are much stronger than Guidelines. You should never break a rule unless there is some external reason for doing so (e.g., making a call to a library routine forces you to use a bad naming convention). Whenever you feel you must violate a rule, you should verify that it is reasonable to do so in a peer review with at least two peers. Furthermore, you should explain in the program's comments why it was necessary

3. You may substitute the local language in your area if it is not English.

to violate the rule. Rules are just that -- rules to be followed. However, there are certain situations where it may be necessary to violate the rule in order to satisfy external requirements or even make the program more readable.

Enforced Rules are the toughest of the lot. You should *never* violate an enforced rule. If there is ever a true need to do this, then you should consider demoting the Enforced Rule to a simple Rule rather than treating the violation as a reasonable alternative.

An Exception is exactly that, a known example where one would commonly violate a Guideline, Rule, or (very rarely) Enforced Rule. Although exceptions are rare, the old adage "Every rule has its exceptions..." certainly applies to this document. The Exceptions point out some of the common violations one might expect.

Of course, the categorization of Guidelines, Rules, Enforced Rules, and Exceptions herein is one man's opinion. At some organizations, this categorization may require reworking depending on the needs of that organization.

C.1.6 Source Language Concerns

This document will assume that the entire program is written in 80x86 assembly language using the HLA assembler/compiler. Although this organization is rare in commercial applications, this assumption will, in no way, invalidate these guidelines. Other guidelines exist for various high level languages (including a set written by this paper's author). You should adopt a reasonable set of guidelines for the other languages you use and apply these guidelines to the 80x86 assembly language modules in the program.

C.2 Program Organization

A source program generally consists of one or more source, object, and library files. As a project gets larger and the number of files increases, it becomes difficult to keep track of the files in a project. This is especially true if a number of different projects share a common set of source modules. This section will address these concerns.

C.2.1 Library Functions

A library, by its very nature, suggests stability. Ignoring the possibility of software defects, one would rarely expect the number or function of routines in a library to vary from project to project. A good example is the "HLA Standard Library." One would expect "stdout.put" to behave identically in two different programs that use the Standard Library. Contrast this against two programs, each of which implement their own version of *stdout.put*. One could not reasonably assume both programs have identical implementations⁴. This leads to the following rule:

Rule:	Library functions are those routines intended for common reuse in many different assembly language programs. All assembly language (callable) libraries on a system should exist as ".lib" files and should appear in a "\lib" or "\hlalib" subdirectory.
Guideline:	"\hlalib" is probably a better choice if you're using multiple languages since those other languages may need to put files in a "\lib" directory.
Exception:	It's probably reasonable to leave the HLA Standard Library's "hlalib.lib" file in the "\hla\hlalib" directory since most people expect it there.

4. In fact, just the opposite is true. One should get concerned if both implementations *are* identical. This would suggest poor planning on the part of the program's author(s) since the same routine must now be maintained in two different programs.

The rule above ensures that the library files are all in one location so they are easy to find, modify, and review. By putting all your library modules into a single directory, you avoid configuration management problems such as having outdated versions of a library linking with one program and up-to-date versions linking with other programs.

C.2.2 Common Object Modules

This document defines a *library* as a collection of object modules that have wide application in many different programs. The HLA Standard Library is a typical example of a library. Some object modules are not so general purpose, but still find application in two or more different programs. Two major configuration management problems exist in this situation: (1) making sure the ".obj" file is up-to-date when linking it with a program; (2) Knowing which modules use the module so one can verify that changes to the module won't break existing code.

The following rules takes care of case one:

- Rule:** If two different program share an object module, then the associated source, object, and make-files for that module should appear in a subdirectory that is specific to that module (i.e., no other files in the subdirectory). The subdirectory name should be the same as the module name. If possible, you should create a set of link/alias/shortcuts to this subdirectory and place these links in the main directory of each of the projects that utilize the module. If links are not possible, you should place the module's subdirectory in a "\common" subdirectory.
- Enforced Rule:** Every subdirectory containing one or more modules should have a make file that will automatically generate the appropriate, up-to-date, ".obj" files. An individual, a batch file, or another make file should be able to automatically generate new object modules (if necessary) by simply executing the make program.
- Guideline:** Use Microsoft's nmake program. At the very least, use nmake acceptable syntax in your make-files.

The other problem, noting which projects use a given module is much more difficult. The obvious solution, commenting the source code associated with the module to tell the reader which programs use the module, is impractical. Maintaining these comments is too error-prone and the comments will quickly get out of phase and be worse than useless -- they would be incorrect. A better solution is to create alias and place this alias in the main subdirectory of each program that links the module.

- Guideline:** If a project uses a module that is not local to the project's subdirectory, create an alias to the file in the project's subdirectory. This makes locating the file very easy.

C.2.3 Local Modules

Local modules are those that a single program/project uses. Typically, the source and object code for each module appears in the same directory as the other files associated with the project. This is a reasonable arrangement until the number of files increases to the point that it is difficult to find a file in a directory listing. At that point, most programmers begin reorganizing their directory by creating subdirectories to hold many of these source modules. However, the placement, name, and contents of these new subdirectories can have a big impact on the overall readability of the program. This section will address these issues.

The first issue to consider is the contents of these new subdirectories. Since programmers rummaging through this project in the future will need to easily locate source files in a project, it is important that you organize these new subdirectories so that it is easy to find the source files you are moving into them. The best organization is to put each source module (or a small group of *strongly related* modules) into its own subdirectory. The subdirectory should bear the name of the source module minus its suffix (or the main module if there is more than one present in the subdirectory). If you place two or more source files in the same directory, ensure this set of source files forms a *cohesive* set (meaning the source files contain code that solve a single problem). A discussion of cohesiveness appears later in this document.

- Rule:** If a project directory contains too many files, try to move some of the modules to subdirectories within the project directory; give the subdirectory the same name as the source file without the suffix. This will nearly reduce the number of files in half. If this reduction is insufficient, try categorizing the source modules (e.g., FileIO, Graphics, Rendering, and Sound) and move these modules to a subdirectory bearing the name of the category.
- Enforced Rule:** Each new subdirectory you create should have its own make file that will automatically assemble all source modules within that subdirectory, as appropriate.
- Enforced Rule:** Any new subdirectories you create for these source modules should appear within the directory containing the project. The only exceptions are those modules that are, or you anticipate, sharing with other projects. See “Common Object Modules” on page 1415 for more details.

Stand-alone assembly language programs generally contain a "main" procedure – the first program unit that executes when the operating system loads the program into memory. For any programmer new to a project, this procedure is the *anchor* where one first begins reading the code and the point where the reader will continually refer. Therefore, the reader should be able to easily locate this source file. The following rule helps ensure this is the case:

- Rule:** The source module containing the *main program* should have the same name as the executable (obviously the suffix will be different). For example, if the "Simulate 886" program's executable name is "Sim886.exe" then you should find the main program in the "Sim886.hla" source file.

Finding the source file that contains the main program is one thing. Finding the main program itself can be almost as hard. Assembly language lets you give the main program any name you want. However, to make the main procedure easy to find (both in the source code and at the O/S level), you should actually name this program "main". See “Module Organization” on page 1417 for more details about the placement of the main program. An alternative is to give the main program's source file the name of the project.

- Guideline:** The name of the main procedure in an assembly language program should be "main" or the name of the entire project.

C.2.4 Program Make Files

Every project, even if it contains only a single source module, should have an associated make file. If someone wants to assemble your program, they should not have to worry about what program (e.g., HLA) to use to compile the program, what command line options to use, what library modules to use, etc. They should be able to type "nmake"⁵ and wind up with an executable program. Even if assembling the program consists of nothing more than typing the name of the assembler and the source file, you should still have a make file. Someone else may not realize that's all that is necessary.

- Enforced Rule:** The main project directory should contain a make file that will automatically generate an executable (or other expected object module) in response to a simple make/nmake command.
- Rule:** If your project uses object modules that are not in the same subdirectory as the main program's module, you should test the ".obj" files for those modules and execute the corresponding make files in their directories if the object code is out of date. You can assume that library files are up to date.
- Guideline:** Avoid using fancy "make" features. Most programmers only learn the basics about make and will not be able to understand what your make file is doing if you fully exploit the make language. Especially avoid the use of default rules since this can create havoc if someone arbitrarily adds or removes files from the directory containing the make file.

5. Or whatever make program you normally use.

C.3 Module Organization

A module is a collection of objects that are logically related. Those objects may include constants, data types, variables, and program units (e.g., functions, procedures, etc.). Note that objects in a module need not be *physically* related. For example, it is quite possible to construct a module using several different source files. Likewise, it is quite possible to have several different modules in the same source file. However, the best modules are physically related as well as logically related; that is, all the objects associated with a module exist in a single source file (or directory if the source file would be too large) and nothing else is present.

Modules contain several different objects including constants, types, variables, and program units (routines). Modules shares many of the attributes with routines (program units); this is not surprising since routines are the major component of a typical module. However, modules have some additional attributes of their own. The following sections describe the attributes of a well-written module.

Note: *Unit* and *package* are both synonyms for the term *module*.

C.3.1 Module Attributes

A module is a generic term that describes a set of program related objects (program units as well as data and type objects) that are somehow coupled. Good modules share many of the same attributes as good program units as well as the ability to hide certain details from code outside the module.

C.3.1.1 Module Cohesion

Modules exhibit the following different kinds of *cohesion* (listed from good to bad):

- Functional or logical cohesion exists if the module accomplishes exactly one (simple) task.
- Sequential or *pipelined* cohesion exists when a module does several sequential operations that must be performed in a certain order with the data from one operation being fed to the next in a “filter-like” fashion.
- Global or *communicational* cohesion exists when a module performs a set of operations that make use of a common set of data, but are otherwise unrelated.
- Temporal cohesion exists when a module performs a set of operations that need to be done at the same time (though not necessarily in the same order). A typical initialization module is an example of such code.
- Procedural cohesion exists when a module performs a sequence of operations in a specific order, but the only thing that binds them together is the order in which they must be done. Unlike sequential cohesion, the operations do not share data.
- State cohesion occurs when several different (unrelated) operations appear in the same module and a state variable (e.g., a parameter) selects the operation to execute. Typically such modules contain a case (switch) or **if..elseif..elseif...** statement.
- No cohesion exists if the operations in a module have no apparent relationship with one another.

The first three forms of cohesion above are generally acceptable in a program. The fourth (temporal) is probably okay, but you should rarely use it. The last three forms should almost never appear in a program. For some reasonable examples of module cohesion, you should consult “Code Complete”.

Guideline: Design good modules! *Good modules exhibit strong cohesion*. That is, a module should offer a (small) group of services that are logically related. For example, a “printer” module might provide all the services one would expect from a printer. The individual routines within the module would provide the individual services.

C.3.1.2 Module Coupling

Coupling refers to the way that two modules communicate with one another. There are several criteria that define the level of coupling between two modules:

- Cardinality- the number of objects communicated between two modules. The fewer objects the better (i.e., fewer parameters).
- Intimacy- how “private” is the communication? Parameter lists are the most private form; private data fields in a class or object are next level; public data fields in a class or object are next, global variables are even less intimate, and passing data in a file or database is the least intimate connection. Well-written modules exhibit a high degree of intimacy.
- Visibility- this is somewhat related to intimacy above. This refers to how visible the data is to the entire system that you pass between two modules. For example, passing data in a parameter list is direct and very visible (you always see the data the caller is passing in the call to the routine); passing data in global variables makes the transfer less visible (you could have set up the global variable long before the call to the routine). Another example is passing simple (scalar) variables rather than loading up a bunch of values into a structure/record and passing that structure/record to the callee.
- Flexibility- This refers to how easy it is to make the connection between two routines that may not have been originally intended to call one another. For example, suppose you pass a structure containing three fields into a function. If you want to call that function but you only have three data objects, not the structure, you would have to create a dummy structure, copy the three values into the field of that structure, and then call the function. On the other hand, had you simply passed the three values as separate parameters, you could still pass in structures (by specifying each field) as well as call the function with separate values. The module containing this later function is more flexible.

A module is *loosely coupled* if its functions exhibit low cardinality, high intimacy, high visibility, and high flexibility. Often, these features are in conflict with one another (e.g., increasing the flexibility by breaking out the fields from a structures [a good thing] will also increase the cardinality [a bad thing]). It is the traditional goal of any engineer to choose the appropriate compromises for each individual circumstance; therefore, you will need to carefully balance each of the four attributes above.

A module that uses loose coupling generally contains fewer errors per KLOC (thousands of lines of code). Furthermore, modules that exhibit loose coupling are easier to reuse (both in the current and future projects). For more information on coupling, see the appropriate chapter in “Code Complete”.

Guideline: Design good modules! *Good modules exhibit loose coupling.* That is, there are only a few, well-defined (visible) interfaces between the module and the outside world. Most data is private, accessible only through accessor functions (see information hiding below). Furthermore, the interface should be flexible.

Guideline: Design good modules! *Good modules exhibit information hiding.* Code outside the module should only have access to the module through a small set of public routines. All data should be private to that module. A module should implement an *abstract data type*. All interface to the module should be through a well-defined set of operations.

C.3.1.3 Physical Organization of Modules

Many languages provide direct support for modules (e.g., units in HLA, packages in Ada, modules in Modula-2, and units in Delphi/Pascal). Some languages provide only indirect support for modules (e.g., a source file in C/C++). Others, like BASIC, don’t really support modules, so you would have to simulate them by physically grouping objects together and exercising some discipline. The primary mechanism in HLA for hiding names from other modules is to implement a module as an individual source file and publish only those names that are part of the module’s interface to the outside world (i.e., EXTERNAL directives in a header file).

Rule: Each module should completely reside in a single source file. If size considerations prevent

this, then all the source files for a given module should reside in a subdirectory specifically designated for that module.

Some people have the crazy idea that modularization means putting each function in a separate source file. Such *physical modularization* generally impairs the readability of a program more than it helps. Strive instead for *logical modularization*, that is, defining a module by its actions rather than by source code syntax (e.g., separating out functions).

This document does not address the decomposition of a problem into its modular components. Presumably, you can already handle that part of the task. There are a wide variety of texts on this subject if you feel weak in this area.

C.3.1.4 Module Interface

In any language system that supports modules, there are two primary components of a module: the interface component that publicizes the module visible names and the implementation component that contains the actual code, data, and private objects. HLA (like most assemblers) uses a scheme that is very similar to the one C/C++ uses. There are directives that let you import and export names. Like C/C++, you could place these directives directly in the related source modules. However, such code is difficult to maintain (since you need to change the directives in every file whenever you modify a public name). The solution, as adopted in the HLA programming language, is to use *header files*. Header files contain all the public definitions and exports (as well as common data type definitions and constant definitions). The header file provides the *interface* to the other modules that want to use the code present in the implementation module.

The HLA EXTERNAL attribute is perfect for creating interface/header files. When you use EXTERNAL within a source module that defines a symbol, EXTERNAL behaves like a *public* directive, exporting the name to other modules. When you use EXTERNAL within a source modules that refers to an external name, EXTERNAL declares the object to be supplied in a different module. This lets you place an EXTERNAL declaration of an object in a single header file and include this file into both the modules that import and export the public names.

- Rule: Keep all module interface directives (EXTERNAL) in a single header file for a given module. Place any other common data type definitions and constant definitions in this header file as well.
- Guideline: There should only be a single header file associated with any one module (even if the module has multiple source files associated with it). If, for some reason, you feel it is necessary to have multiple header files associated with a module, you should create a single file that includes all of the other interface files. That way a program that wants to use all the header files need only include the single file.

When designing header files, make sure you can include a file more than once without ill effects (e.g., duplicate symbol errors). The traditional way to do this is to put a #IF statement like the following around all the statements in a header file:

```
; Module: MyHeader.hhf

#if( @defined( MyHeader_hhf ) )
?MyHeader_hhf:=true; // Actual type and value doesn't really matter.
    .
    .           ;Statements in this header file.
    .
#endif
```

The first time a source file includes "MyHeader.hhf" the symbol "MyHeader_hhf" is undefined. Therefore, the assembler will process all the statements in the header file. In successive include operations (during the same assembly) the symbol "MyHeader_hhf" is already defined, so the assembler ignores the body of the include file.

My would you ever include a file twice? Easy. Some header files may include other header files. By including the file "YourHeader.hhf" a module might also be including "MyHeader.hhf" (assuming "Your-

Header.hhf" contains the appropriate include directive). Your main program, that includes "YourHeader.hhf" might also need "MyHeader.hhf" so it explicitly includes this file not realizing "YourHeader.hhf" has already processed "MyHeader.hhf" thereby causing symbol redefinitions.

- Rule: Always put an appropriate #IF statement around all the definitions in a header file to allow multiple inclusion of the header file without ill effect.
- Guideline: Use the ".hhf" suffix for HLA header/interface files.
- Rule: Include files for library functions on a system should exist as ".hhf" files and should appear in the "\include" or "\hla\include" subdirectory.
- Guideline: "\hla\include" is probably a better choice if you're using multiple languages since those other languages may need to put files in a "\include" directory.
- Exception: It's probably reasonable to leave the HLA Standard Library's "stdlib.hhf" file in the "\hla\include" directory since most people expect it there.

You can also prevent multiple inclusion of a file by using the #INCLUDEONCE directive. However, it's safer to use the #IF.#ENDIF approach since that doesn't rely on the user of your include file to use the right directive.

C.4 Program Unit Organization

A program unit is any procedure, function, coroutine, iterator, subroutine, subprogram, routine, or other term that describes a section of code that abstracts a set of common operations on the computer. This text will simply use the term *procedure* or *routine* to describe these concepts.

Routines are closely related to modules, since they tend to be the major component of a module (along with data, constants, and types). Hence, many of the attributes that apply to a module also apply to routines. The following paragraphs, at the expense of being redundant, repeat the earlier definitions so you don't have to flip back to the previous sections.

C.4.1 Routine Cohesion

Routines exhibit the following kinds of *cohesion* (listed from good to bad and are mostly identical to the kinds of cohesion that modules exhibit):

- Functional or logical cohesion exists if the routine accomplishes exactly one (simple) task.
- Sequential or *pipelined* cohesion exists when a routine does several sequential operations that must be performed in a certain order with the data from one operation being fed to the next in a "filter-like" fashion.
- Global or *communicational* cohesion exists when a routine performs a set of operations that make use of a common set of data, but are otherwise unrelated.
- Temporal cohesion exists when a routine performs a set of operations that need to be done at the same time (though not necessarily in the same order). A typical initialization routine is an example of such code.
- Procedural cohesion exists when a routine performs a sequence of operations in a specific order, but the only thing that binds them together is the order in which they must be done. Unlike sequential cohesion, the operations do not share data.
- State cohesion occurs when several different (unrelated) operations appear in the same routine and a state variable (e.g., a parameter) selects the operation to execute. Typically such routines contain a case (switch) or **if..elseif..elseif...** statement.
- No cohesion exists if the operations in a routine have no apparent relationship with one another.

The first three forms of cohesion above are generally acceptable in a program. The fourth (temporal) is probably okay, but you should rarely use it. The last three forms should almost never appear in a program. For some reasonable examples of routine cohesion, you should consult "Code Complete".

Guideline: All routines should exhibit good cohesiveness. Functional cohesiveness is best, followed by sequential and global cohesiveness. Temporal cohesiveness is okay on occasion. You should avoid the other forms.

C.4.2 Routine Coupling

Coupling refers to the way that two routines communicate with one another. There are several criteria that define the level of coupling between two routines; again these are identical to the types of coupling that modules exhibit:

- Cardinality- the number of objects communicated between two routines. The fewer objects the better (i.e., fewer parameters).
- Intimacy- how “private” is the communication? Parameter lists are the most private form; private data fields in a class or object are next level; public data fields in a class or object are next, global variables are even less intimate, and passing data in a file or database is the least intimate connection. Well-written routines exhibit a high degree of intimacy.
- Visibility- this is somewhat related to intimacy above. This refers to how visible the data is to the entire system that you pass between two routines. For example, passing data in a parameter list is direct and very visible (you always see the data the caller is passing in the call to the routine); passing data in global variables makes the transfer less visible (you could have set up the global variable long before the call to the routine). Another example is passing simple (scalar) variables rather than loading up a bunch of values into a structure/record and passing that structure/record to the callee.
- Flexibility- This refers to how easy it is to make the connection between two routines that may not have been originally intended to call one another. For example, suppose you pass a structure containing three fields into a function. If you want to call that function but you only have three data objects, not the structure, you would have to create a dummy structure, copy the three values into the field of that structure, and then call the routine. On the other hand, had you simply passed the three values as separate parameters, you could still pass in structures (by specifying each field) as well as call the routine with separate values.

A function is *loosely coupled* if it exhibits low cardinality, high intimacy, high visibility, and high flexibility. Often, these features are in conflict with one another (e.g., increasing the flexibility by breaking out the fields from a structures [a good thing] will also increase the cardinality [a bad thing]). It is the traditional goal of any engineer to choose the appropriate compromises for each individual circumstance; therefore, you will need to carefully balance each of the four attributes above.

A program that uses loose coupling generally contains fewer errors per KLOC (thousands of lines of code). Furthermore, routines that exhibit loose coupling are easier to reuse (both in the current and future projects). For more information on coupling, see the appropriate chapter in “Code Complete”.

Guideline: Coupling between routines in source code should be loose.

C.4.3 Routine Size

Sometime in the 1960’s, someone decided that programmers could only look at one page in a listing at a time, therefore routines should be a maximum of one page long (66 lines, at the time). In the 1970’s, when interactive computing became popular, this was adjusted to 24 lines -- the size of a terminal screen. In fact, there is very little empirical evidence to suggest that small routine size is a good attribute. In fact, several studies on code containing artificial constraints on routine size indicate just the opposite -- shorter routines often contain more bugs per KLOC⁶.

6. This happens because shorter functions invariably have stronger coupling, leading to integration errors.

A routine that exhibits functional cohesiveness is the right size, almost regardless of the number of lines of code it contains. You shouldn't artificially break up a routine into two or more subroutines (e.g., sub_partI and sub_partII) just because you feel a routine is getting to be too long. First, verify that your routine exhibits strong cohesion and loose coupling. If this is the case, the routine is not too long. Do keep in mind, however, that a long routine is probably a good indication that it is performing several actions and, therefore, does not exhibit strong cohesion.

Of course, you can take this too far. Most studies on the subject indicate that routines in excess of 150-200 lines of code tend to contain more bugs and are more costly to fix than shorter routines. Note, by the way, that you do not count blank lines or lines containing only comments when counting the lines of code in a program.

Also note that most studies involving routine size deal with HLLs. A comparable HLA routine will contain more lines of code than the corresponding HLL routine. Therefore, you can expect your routines in assembly language to be a little longer.

Guideline:	Do not let artificial constraints affect the size of your routines. If a routine exceeds about 200-250 lines of code, make sure the routine exhibits functional or sequential cohesion. Also look to see if there aren't some generic subsequences in your code that you can turn into stand alone routines.
Rule:	Never shorten a routine by dividing it into n parts that you would always call in the appropriate sequence as a way of shortening the original routine.

C.5 Statement Organization

In an assembly language program, the author must work extra hard to make a program readable. By following a large number of rules, you can produce a program that is readable. However, by breaking a single rule *no matter how many other rules you've followed*, you can render a program unreadable. Nowhere is this more true than how you organize the statements within your program.

C.5.1 Writing "Pure" Assembly Code

Consider the following example taken from "The Art of Assembly Language Programming/DOS Edition" and converted to HLA:

The Microsoft Macro Assembler is a free form assembler. The various fields of an assembly language statement may appear in any column (as long as they appear in the proper order).

Any number of spaces or tabs can separate the various fields in the statement. To the assembler, the following two code sequences are identical:

```

mov( 0, ax );
mov( ax, bx );
add( dx, ax );
mov( ax, cx );

```

```

mov( 0,
      mov( ax,
            add( ad,
                  mov(
                    ax, cx );

```

The first code sequence is much easier to read than the second (if you don't think so, perhaps you should go see a doctor!). With respect to readability, the judicious use of spacing within your pro-

gram can make all the difference in the world.

While this is an extreme example, do note that it only takes a few mistakes to have a large impact on the readability of a program.

HLA is a free-form assembler insofar as it does not place stringent formatting requirements on its statements. For example, you can put multiple statements on a single line as well as spread a single statement across multiple lines. However, the freedom to arrange these statements in any manner is one of the primary contributors to hard to read assembly language programs. Although HLA lets you enter your programs in free-form, there is absolutely no reason you cannot adopt a fixed format. Doing so generally helps make an assembly language program much easier to read. Here are the rules you should use:

- Guideline: Only place one statement per source line.
- Rule: Within a given block of code, all mnemonics should start in the same column.
- Exception: See the indentation rules appearing later in this documentation.
- Guideline: Try to always start the comment fields on adjacent source lines in the same column (note that it is impractical to always start the comment field in the same column throughout a program).

Most people learn a high level language prior to learning assembly language. They have been firmly taught that readable (HLL) programs have their control structures properly indented to show the structure of the program. Indentation works great when you have a *block structured* language. In old-fashioned assembly language this scheme doesn't work; one of the principle benefits to HLA is that it lets you continue to use the indentation schemes you're familiar with in HLLs like C/C++ and Pascal. However, this assumes that you're using the HLA high level control structures. If you choose to work in "pure" assembly language, then these rules don't apply. The following discussion assumes the use of "pure" assembly language code; we'll address HLA's high level control statements later.

If you need to set off a sequence of statements from surrounding code, the best thing you can do is use blank lines in your source code. For a small amount of detachment, to separate one computation from another for example, a single blank line is sufficient. To really show that one section of code is special, use two, three, or even four blank lines to separate one block of statements from the surrounding code. To separate two totally unrelated sections of code, you might use several blank lines and a row of dashes or asterisks to separate the statements. E.g.,

```

mov( FileSpec, eax );
mov( 0, cl );
call MyFunction;
jc Error;

//*****

mov( &fileRecords, edi );
mov( &files, ebx );
sub( 2, ebx );

```

- Guideline: Use blank lines to separate special blocks of code from the surrounding code. Use an aesthetic looking row of asterisks or dashes if you need a stronger separation between two blocks of code (do not overdo this, however).

If two sequences of assembly language statements correspond to roughly two HLL statements, it's generally a good idea to put a blank line between the two sequences. This helps clarify the two segments of code in the reader's mind. Of course, it is easy to get carried away and insert too much white space in a program, so use some common sense here.

- Guideline: If two sequences of code in assembly language correspond to two adjacent statements in a HLL, then use a blank line to separate those two assembly sequences (assuming the sequences

are real short).

A common problem in any language (not just assembly language) is a line containing a comment that is adjacent to one or two lines containing code. Such a program is very difficult read because it is hard to determine where the code ends and the comment begins (or vice-versa). This is especially true when the comments contain sample code. It is often quite difficult to determine if what you're looking at is code or comments; hence the following enforced rule:

Enforced Rule: Always put at least one blank line between code and comments (assuming, of course, the comment is sitting only a line by itself; that is, it is not an endline comment⁷).

C.5.2 Using HLA's High Level Control Statements

Since HLA's high level control statements are so similar to high level language control statements, it's not surprising to discover that you'll use the same formatting for HLA's statements as you would with those other HLLs. Most of these statements compile to very efficient machine code (usually matching what you'd write yourself if you were writing "pure" assembly code). Since their use can make your programs more readable, you should use them whenever practical.

Guideline: Use the HLA high level control structures when they are appropriate in your programs.

There are two problems advanced assembly programmers have with high level control structures: (1) the compiler for such statements (e.g., HLA) doesn't always generate the best code, and (2) the use of such statements encourages inefficient coding on the programmer's part.

HLA's control structures are relatively limited, so point (1) above isn't as big a problem as you might expect. Nevertheless, there will certainly be situations where HLA does not generate the same exact instruction sequence you would for a given control construct. Therefore, it's a good idea to become familiar with the low-level code that HLA emits for each of the control structures so that you can intelligently choose whether to use a high level or low level control structure in a given situation. A later appendix explains how HLA generates code for the high level control structures; you should study this material. Also note that HLA emits MASM compatible assembly code, so you can certainly study HLA's output if you've got any questions about the code HLA generates.

Point (2) above is something that HLA has no control over. It is quite true that if you write "C code with MOV instructions" in HLA, the code probably isn't going to be as efficient as pure assembly code. However, with a little discipline you can prevent this problem from occurring.

One of the benefits to using the high level control structures HLA provides is that you can now use indentation of your statements to better show the structure of the program. Since HLA's high level control structures are very similar to those found in traditional high level languages, you can use well-established programming conventions when indenting statements in your HLA programs. Here are some suggestions:

Rule: Indent statements within a high-level control block four space. The ENDxxxx clause that matches the statement should begin in the same column as the statement that starts a block.

```
// Example of nesting an IF..THEN..ENDIF statement:

    if( eax = 0 ) then

        << Indent these statements four spaces >>

    endif; // endif should be at the same level as the if statement.
```

Guideline: Avoid putting multiple statements on the same line.

7. See the next section concerning comments for more information.

The HLA programming language contains eight flow-of-control statements: two conditional selection statements (IF..THEN..ELSEIF..ELSE and SWITCH..CASE..DEFAULT..ENDSWITCH), five loops (WHILE..ENDWHILE, REPEAT..UNTIL, FOR..ENDFOR, FOREACH..ENDFOR, and FOREVER..ENDFOR), a program unit invocation (i.e., procedure call), and the statement sequence.

Rule: If your code contains a chain of if..elseif..elseif.....elseif..... statements, do not use the final else clause to handle a remaining case. Only use the final else to catch an error condition. If you need to test for some value in an if..elseif..elseif.... chain, always test the value in an if or elseif statement.

The HLA Standard Library implements the multi-way selection statements (SWITCH) using a jump table. This means that the order of the cases within the selection statement is usually irrelevant. Placing the statements in a particular order rarely improves performance. Since the order is usually irrelevant to the compiler, you should organize the cases so that they are easy to read. There are two common organizations that make sense: sorted (numerically or alphabetically) or by frequency (the most common cases first). Either organization is readable; one drawback to this approach is that it is often difficult to predict which cases the program will execute most often.

Guideline: When using multi-way selection statements (case/switch) sort the cases numerically (alphabetically) or by frequency of expected occurrence.

There are three general categories of looping constructs available in common high-level languages- loops that test for termination at the beginning of the loop (e.g., WHILE), loops that test for loop termination at the bottom of the loop (e.g., REPEAT..UNTIL), and those that test for loop termination in the middle of the loop (e.g., FOREVER..ENDFOR). It is possible simulate any one of these loops using any of the others. This is particularly trivial with the FOREVER..ENDFOR construct:

```
/* Test for loop termination at beginning of FOREVER..ENDFOR */
```

```

forever
    breakif( ax = y );
    .
    .
    .
endfor;
```

```
/* Test for loop termination in the middle of FOREVER..ENDFOR */
```

```

forever
    .
    .
    .
    breakif( ax = y );
    .
    .
    .
endfor;
```

```
/* Test for loop termination at the end of FOREVER..ENDFOR */
```

```

forever
    .
    .
    .
    breakif( x = y );
endfor;
```

Given the flexibility of the FOREVER..ENDFOR control structure, you might question why one would even burden a compiler with the other loop statements. However, using the appropriate looping structure makes a program far more readable, therefore, you should never use one type of loop when the situation demands another. If someone reading your code sees a FOREVER..ENDFOR construct, they may think it's okay to insert statements before or after the exit statement in the loop. If your algorithm truly depends on WHILE..ENDWHILE or REPEAT..UNTIL semantics, the program may now malfunction.

Rule: Always use the most appropriate type of loop (categorized by termination test position). Never force one type of loop to behave like another.

Many languages provide a special case of the while loop that executes some number of times specified upon first encountering the loop (a *definite* loop rather than an *indefinite* loop). This is the “for” loop in most languages. The vast majority of the time a for loop sequences through a fixed range of value incrementing or decrementing the loop control variable by one. Therefore, most programmers automatically assume this is the way a for loop will operate until they take a closer look at the code. Since most programmers immediately expect this behavior, it makes sense to limit FOR loops to these semantics. If some other looping mechanism is desirable, you should use a WHILE loop to implement it (since the for loop is just a special case of the while loop). There are other reasons behind this decision as well.

Rule: “FOR” loops should always use an ordinal loop control variable (e.g., integer, char, boolean, enumerated type) and should always increment or decrement the loop control variable by one.

Most people expect the execution of a loop to begin with the first statement at the top of the loop, therefore,

Rule: All loops should have one entry point. The program should enter the loop with the instruction at the top of the loop.

Likewise, most people expect a loop to have a single exit point, especially if it's a WHILE or REPEAT..UNTIL loop. They will rarely look closely inside a loop body to determine if there are “break” statements within the loop once they find one exit point. Therefore,

Guideline: Loops with a single exit point are more easily understood.

Whenever a programmer sees an empty loop, the first thought is that something is missing. Therefore,

Guideline: Avoid empty loops. If testing the loop termination condition produces some side effect that is the whole purpose of the loop, move that side effect into the body of the loop. If a loop truly has an empty body, place a comment like `/* nothing */` within your code.

Even if the loop body is not empty, you should avoid side effects in a loop termination expression. When someone else reads your code and sees a loop body, they may skim right over the loop termination expression and start reading the code in the body of the loop. If the (correct) execution of the loop body depends upon the side effect, the reader may become confused since s/he did not notice the side effect earlier. The presence of side effects (that is, having the loop termination expression compute some other value beyond whether the loop should terminate or repeat) indicates that you're probably using the wrong control structure. Consider the following WHILE loop in HLA that is easily corrected:

```
while( mov( stdin.geti32(), ecx ) != 0 ) do
    << statements >>
endwhile;
```

A better implementation of this code fragment would be to use a FOREVER..ENDFOR construct:

```
forever
```



```

    stdin.geti32();
    mov( eax, ecx );
    breakif( eax = 0 );
    .
    .
    .
endfor;

```

Rule: Avoid side-effects in the computation of the loop termination expression (others may not be expecting such side effects). Also see the guideline about empty loops.

Like functions, loops should exhibit functional cohesion. That is, the loop should accomplish exactly one thing. It's very tempting to initialize two separate arrays in the same loop. You have to ask yourself, though, "what do you really accomplish by this?" You save about four machine instructions on each loop iteration, that's what. That rarely accounts for much. Furthermore, now the operations on those two arrays are tied together, you cannot change the size of one without changing the size of the other. Finally, someone reading your code has to remember two things the loop is doing rather than one.

Guideline: Make each loop perform only one function.

Programs are much easier to read if you read them from left to right, top to bottom (beginning to end). Programs that jump around quite a bit are much harder to read. Of course, the *jmp (goto)* statement is well-known for its ability to scramble the logical flow of a program, but you can produce equally hard to read code using other, structured, statements in a language. For example, a deeply nested set of if statements, some with and some without ELSE clauses, can be very difficult to follow because of the number of possible places the code can transfer depending upon the result of several different boolean expressions.

Rule: Code, as much as possible, should read from top to bottom.

Rule: Related statements should be grouped together and separated from unrelated statements with whitespace or comments.

In theory, a line of source code can be arbitrarily long. In practice, there are several practical limitations on source code lines. Paramount is the amount of text that will fit on a given terminal display device (we don't all have 21" high resolution monitors!) and what can be printed on a typical sheet of paper. Even with small fonts and wide carriage printers, keep in mind that many people like to print listings two-up or three-up in order to save paper. If this isn't enough to suggest an 80 character limit on source lines, McConnell suggests that longer lines are harder to read (remember, people tend to look at only the left side of the page while skimming through a listing).

Enforced Rule: Source code lines will not exceed 80 characters in length.

If a statement approaches the maximum limit of 80 characters, it should be broken up at a reasonable point and split across two lines. If the line is a control statement that involves a particularly long logical expression, the expression should be broken up at a logical point (e.g., at the point of a low-precedence operator outside any parentheses) and the remainder of the expression placed underneath the first part of the expression. E.g., (note that the following involves constant expressions, run-time expressions generally aren't very long):

```

#if
(
    ( ( x + y * z ) < ( ComputeProfits(1980,1990) / 1.0775 ) )
    && ( ValueOfStock[ ThisYear ] >= ValueOfStock[ LastYear ] )
)

```

```

    << statements >>

#endif

```

Many statements (e.g., IF, WHILE, FOR, and function or procedure calls) contain a keyword followed by a parenthesis. If the expression appearing between the parentheses is too long to fit on one line, consider putting the opening and closing parentheses in the same column as the first character of the start of the statement and indenting the remaining expression elements. The example above demonstrates this for the "IF" statement. The following examples demonstrate this technique for other statements:

```

while
(
    SomeFunctionReturningAValueInEAX( with, lots, of, parameters )
    <= AFunctionReturningAValueInEBX( also, has, lots, of, parameters )
) do

    << Statements to execute >>

endwhile;

fileio.put
(
    outputFileHandle,
    "Error in module ",
    ModuleName,
    " at line #",
    LineNumber,
    ", encountered illegal value",
    nl
);

```

Guideline: For statements that are too long to fit on one physical 80-column line, you should break the statement into two (or more) lines at points in the statement that will have the least impact on the readability of the statement. This situation usually occurs immediately after low-precedence operators or after commas.

If a procedure, function, or other program unit has a particularly long actual or formal parameter list, each parameter should be placed on a separate line. The following examples demonstrate a procedure declaration and call using this technique:

```

procedure MyFunction
(
    NumberOfDataPoints: int32,
    X1Root: real32,
    X2Root: real32,
    var YIntercept: real32
);

MyFunction
(
    GetNumberOfPoints(RootArray), // Assume "RETURNS" value is EAX.
    RootArray[ EBX*4 ],
    RootArray[ ECX*4 ],
    Solution
);

```

Rule: If an actual or formal parameter list is too long to fit a function call or definition on a single line, then place each parameter on a separate line and align them so they are easy to read.

Guideline: If a boolean expression exceeds the length of the source line (usually 80 characters), then break the source line into pieces and align the parentheses associated with the statement underneath the start of the statement.

This usually isn't a problem in HLA since expressions are very limited. However, if you call a function with a long parameter list you could run into this problem. One area where this problem does occur is when you're using HLA's hybrid control structures. For such sequences you should always place the statements associated with the boolean expression on separate lines and align the braces with the high level control structure, e.g.,

```

if
{
    cmp( ax, bx );
    jne true;
    cmp( ax, 5 );
    jl false;
    cmp( bx, 0 );
    je false;
}

    << statements to execute on TRUE >>

endif;

```

Rule: Always put a blank line between a high level control statement and the nested statements associated with that statement. Likewise, put a blank line between the end of the nested statements and the corresponding ENDxxx clause of the statement. E.g.,

```

if( ax = 0 ) then
                                <-- Blank line.
    << Nested Statements >>
                                <-- Blank line.
endif;

```

HLA provides special symbols like “@c” and “@s” to denote flag bits within boolean expressions. Using statements like “if(@c) then ... endif;” is semantically equivalent to using a conditional jump (JNC in this case) to jump around the THEN code. Not only is this statement semantically equivalent, it is exactly equivalent since it simply generates the JNC (or whatever) instruction to transfer control to the statement following the ENDIF. The difference between the two, from a readability point of view, is that JNC requires a statement label. As it turns out, the large number of statement labels that appear in an assembly language program contribute to the lack of readability. Hence, anything you can do to legitimately reduce the number of statement labels will improve the readability of your program. So,

Guideline: Try to use statements like “if(@c) then...endif;” rather than “jnc label; ... label:” in your programs to reduce the number of statement labels in the code. Combined with indentation, this will make your programs easier to read since the user doesn't have to search for a specific label associated with the branch (searching for the end of indentation is a much easier task).

C.6 Comments

Comments in an assembly language program generally come in two forms: *endline* comments and *standalone* comments⁸. As their names suggest, endline line comments always occur at the end of a source statement and standalone comments sit on a line by themselves⁹. These two types of comments have distinct purposes, this section will explore their use and describe the attributes of a well-commented program.

C.6.1 What is a Bad Comment?

It is amazing how many programmers claim their code is well-commented. Were you to count characters between (or after) the comment delimiters, they might have a point. Consider, however, the following comment:

```
mov( 0, ax );    //Set AX to zero.
```

Quite frankly, this comment is worse than no comment at all. It doesn't tell the reader anything the instruction itself doesn't tell and it requires the reader to take some of his or her precious time to figure out that the comment is worthless. If someone cannot tell that this instruction is setting AX to zero, they have no business reading an assembly language program. This brings up the first guideline of this section:

Guideline: Choose an intended audience for your source code and write the comments to that audience. For HLA source code, you can usually assume that the target audience are those who know a reasonable amount of HLA and assembly language.

Don't explain the actions of an assembly language instruction in your code unless that instruction is doing something that isn't obvious (and most of the time you should consider changing the code sequence if it isn't obvious what is going on). Instead, explain how that instruction is helping to solve the problem at hand. The following is a much better comment for the instruction above:

```
mov( 0, ax );    //AX is the resulting sum. Initialize it.
```

Note that the comment does not say "Initialize it to zero." Although there would be nothing intrinsically wrong with saying this, the phrase "Initialize it" remains true no matter what value you assign to AX. This makes maintaining the code (and comment) much easier since you don't have to change the comment whenever you change the constant associated with the instruction.

Guideline: Write your comments in such a way that minor changes to the instruction do not require that you change the corresponding comment.

Note: Although a trivial comment is bad (indeed, worse than no comment at all), the worst comment a program can have is one that is wrong. Consider the following statement:

```
mov( 1, ax );    //Set AX to zero.
```

It is amazing how long a typical person will look at this code trying to figure out how on earth the program sets AX to zero when it's obvious it does not do this. *People will always believe comments over code.* If there is some ambiguity between the comments and the code, they will assume that the code is tricky and that the comments are correct. Only after exhausting all possible options is the average person likely to concede that the comment must be incorrect.

Enforced Rule: Never allow incorrect comments in your program.

This is another reason not to put trivial comments like "Set AX to zero" in your code. As you modify the program, these are the comments most likely to become incorrect as you change the code and fail to keep the comments in sync. However, even some non-trivial comments can become incorrect via changes to the code. Therefore, always follow this rule:

8. This document will simply use the term *comments* when referring to standalone comments.

9. Since the label, mnemonic, and operand fields are all optional, it is legal to have a comment on a line by itself.

Enforced Rule: Always update *all* comments affected by a code change immediately after making the code change.

Undoubtedly you've heard the phrase "make sure you comment your code as though someone else wrote it for you; otherwise in six months you'll wish you had." This statement encompasses two concepts. First, don't ever think that your understanding of the current code will last. While working on a given section of a program you're probably investing considerable thought and study to figure out what's going on. Six months down the road, however, you will have forgotten much of what you figured out and the comments can go a long way to getting you back up to speed quickly. The second point this code makes is the implication that others read and write code too. You will have to read someone else's code, they will have to read yours. If you write the comments the way you would expect others to write it for you, chances are pretty good that your comments will work for them as well.

Rule: Never use racist, sexist, obscene, or other exceptionally politically incorrect language in your comments. Undoubtedly such language in your comments will come back to embarrass you in the future. Furthermore, it's doubtful that such language would help someone better understand the program.

It's much easier to give examples of bad comments than it is to discuss good comments. The following list describes some of the worst possible comments you can put in a program (from worst up to barely tolerable):

- The absolute worst comment you can put into a program is an incorrect comment. Consider the following assembly statement:

```
mov( 10, ax ); // Set AX to 11
```

 It is amazing how many programmers will automatically assume the comment is correct and try to figure out how this code manages to set the variable "A" to the value 11 when the code so obviously sets it to 10.
- The second worst comment you can place in a program is a comment that explains what a statement is doing. The typical example is something like "mov(10, ax); // Set 'A' to 10". Unlike the previous example, this comment is correct. But it is still worse than no comment at all because it is redundant and forces the reader to spend additional time reading the code (reading time is directly proportional to reading difficulty). This also makes it harder to maintain since slight changes to the code (e.g., "mov(9, ax);") requires modifications to the comment that would not otherwise be required.
- The third worst comment in a program is an irrelevant one. Telling a joke, for example, may seem cute, but it does little to improve the readability of a program; indeed, it offers a distraction that breaks concentration.
- The fourth worst comment is no comment at all.
- The fifth worst comment is a comment that is obsolete or out of date (though not incorrect). For example, comments at the beginning of the file may describe the current version of a module and who last worked on it. If the last programmer to modify the file did not update the comments, the comments are now out of date.

C.6.2 What is a Good Comment?

Steve McConnell provides a long list of suggestions for high-quality code. These suggestions include:

- **Use commenting styles that don't break down or discourage modification.** Essentially, he's saying pick a commenting style that isn't so much work people refuse to use it. He gives an example of a block of comments surrounded by asterisks as being hard to maintain. This is a poor example since modern text editors will automatically "outline" the comments for you. Nevertheless, the basic idea is sound.
- **Comment as you go along.** If you put commenting off until the last moment, then it seems like another task in the software development process always comes along and management is likely to discourage the completion of the commenting task in hopes of meeting new deadlines.

- **Avoid self-indulgent comments.** Also, you should avoid sexist, profane, or other insulting remarks in your comments. Always remember, someone else will eventually read your code.
- **Avoid putting comments on the same physical line as the statement they describe.** Such comments are very hard to maintain since there is very little room. McConnell suggests that endline comments are okay for variable declarations. For some this might be true but many variable declarations may require considerable explanation that simply won't fit at the end of a line. One exception to this rule is "maintenance notes." Comments that refer to a defect tracking entry in the defect database are okay (note that the CodeWright text editor provides a much better solution for this -- buttons that can bring up an external file). Of course, endline comments are marginally more useful in assembly language than in the HLLs that McConnell addresses, but the basic idea is sound.
- **Write comments that describe blocks of statements rather than individual statements.** Comments covering single statements tend to discuss the mechanics of that statement rather than discussing what the program is doing.
- **Focus paragraph comments on the *why* rather than the *how*.** Code should explain what the program is doing and why the programmer chose to do it that way rather than explain what each individual statement is doing.
- **Use comments to prepare the reader for what is to follow.** Someone reading the comments should be able to have a good idea of what the following code does without actually looking at the code. Note that this rule also suggests that comments should always precede the code to which they apply.
- **Make every comment count.** If the reader wastes time reading a comment of little value, the program is harder to read; period.
- **Document surprises and tricky code.** Of course, the best solution is not to have any tricky code. In practice, you can't always achieve this goal. When you do need to restore to some tricky code, make sure you fully document what you've done.
- **Avoid abbreviations.** While there may be an argument for abbreviating identifiers that appear in a program, no way does this apply to comments.
- **Keep comments close to the code they describe.** The prologue to a program unit should give its name, describe the parameters, and provide a short description of the program. It should not go into details about the operation of the module itself. Internal comments should to that.
- **Comments should explain the parameters to a function,** assertions about these parameters, whether they are input, output, or in/out parameters.
- **Comments should describe a routine's limitations, assumptions, and any side effects.**

Rule: All comments will be high-quality comments that describe the actions of the surrounding code in a concise manner

C.6.3 Endline vs. Standalone Comments

Guideline: Adjacent lines of comments should not have any interspersed blank lines. At least lead off the comment with the HLA comment character sequence (e.g., `/**`).

The guideline above suggests that your code should look like this:

```
// This is a comment with a blank line between it and the next comment.
//
// This is another line with a comment on it.
```

Rather than like this:

```
// This is a comment with a blank line between it and the next comment.

// This is another line with a comment on it.
```

The “//” appearing between the two statements suggest continuity that is not present when you remove the “//”. If two blocks of comments are truly separate and whitespace between them is appropriate, you should consider separating them by a large number of blank lines to completely eliminate any possible association between the two.

Standalone comments are great for describing the actions of the code that immediately follows. So what are endline comments useful for? Endline comments can explain how a sequence of instructions are implementing the algorithm described in a previous set of standalone comments. Consider the following code:

```
// Compute the transpose of a matrix using the algorithm:
//
//     for i := 0 to 3 do
//         for j := 0 to 3 do
//             swap( a[i][j], b[j][i] );

for( mov( 0, i); i < 3; inc( i ) ) do

    for( mov( 0, j ); j < 3; inc( j ) ) do

        mov( i, ebx );           // Compute address of a[i][j] using
        shl( 2, ebx );           // row major ordering (i*4 + j)*4.
        add( j, ebx );
        lea( ebx, a[ebx*4] );
        push( ebx );             // Push address of a[i][j] onto stack.

        mov( j, ebx );           // Compute address of b[j][i] using
        shl( 2, ebx );           // row major ordering (j*4 + i)*4.
        add( i, ebx );
        lea( ebx, b[ebx*4] );
        push( ebx );             // Push address of b[j][i] onto stack.
        call swap;               // Swap objects pointed at by [esp] and [esp+4].

    endfor;

endfor;
```

Note that the block comments before this sequence explain, in high level terms, what the code is doing. The endline comments explain how the statement sequence implements the general algorithm. Note, however, that the endline comments do not explain what each statement is doing (at least at the machine level). Rather than claiming "lea(ebx, b[ebx*4])" also multiplies the quantity in EBX by four, this code assumes the reader can figure that out for themselves (any reasonable assembly programmer would know this). Once again, keep in mind your audience and write your comments for them.

C.6.4 Unfinished Code

Often it is the case that a programmer will write a section of code that (partially) accomplishes some task but needs further work to complete a feature set, make it more robust, or remove some known defect in the code. It is common for such programmers to place comments into the code like "This needs more work," "Kludge ahead," etc. The problem with these comments is that they are often forgotten. It isn't until the code fails in the field that the section of code associated with these comments is found and their problems corrected.

Ideally, one should never have to put such code into a program. Of course, ideally, programs never have any defects in them, either. Since such code inevitably finds its way into a program, it's best to have a policy in place to deal with it, hence this section.

Unfinished code comes in five general categories: non-functional code, partially functioning code, suspect code, code in need of enhancement, and code documentation. Non-functional code might be a stub or driver that needs to be replaced in the future with actual code or some code that has severe enough defects

that it is useless except for some small special cases. This code is really bad, fortunately its severity prevents you from ignoring it. It is unlikely anyone would miss such a poorly constructed piece of code in early testing prior to release.

Partially functioning code is, perhaps, the biggest problem. This code works well enough to pass some simple tests yet contains serious defects that should be corrected. Moreover, these defects are known. Software often contains a large number of unknown defects; it's a shame to let some (prior) known defects ship with the product simply because a programmer forgot about a defect or couldn't find the defect later.

Suspect code is exactly that- code that is suspicious. The programmer may not be aware of a quantifiable problem but may suspect that a problem exists. Such code will need a later review in order to verify whether it is correct.

The fourth category, code in need of enhancement, is the least serious. For example, to expedite a release, a programmer might choose to use a simple algorithm rather than a complex, faster algorithm. S/he could make a comment in the code like "This linear search should be replaced by a hash table lookup in a future version of the software." Although it might not be absolutely necessary to correct such a problem, it would be nice to know about such problems so they can be dealt with in the future.

The fifth category, documentation, refers to changes made to software that will affect the corresponding documentation (user guide, design document, etc.). The documentation department can search for these defects to bring existing documentation in line with the current code.

This standard defines a mechanism for dealing with these five classes of problems. Any occurrence of unfinished code will be preceded by a comment that takes one of the following forms (where "_" denotes a single space):

```
//_#defect#severe_//
//_#defect#functional_//
//_#defect#suspect_//
//_#defect#enhancement_//
//_#defect#documentation_//
```

It is important to use all lower case and verify the correct spelling so it is easy to find these comments using a text editor search or a tool like grep. Obviously, a separate comment explaining the situation must follow these comments in the source code.

Examples:

```
// #defect#suspect //
// #defect#enhancement //
// #defect#documentation //
```

Notice the use of comment delimiters (the "//") on both sides even though HLA doesn't require them.

Enforced Rule: If a module contains some defects that cannot be immediately removed because of time or other constraints, the program will insert a standardized comment before the code so that it is easy to locate such problems in the future. The five standardized comments are `“//_#defect#severe_//”`, `“//_#defect#functional_//”`, `“//_#defect#suspect_//”`, `“//_#defect#enhancement_//”`, and `“//_#defect#documentation_//”` where “_” denotes a single space. The spelling and spacing should be exact so it is easy to search for these strings in the source tree.

C.6.5 Cross References in Code to Other Documents

In many instances a section of code might be intrinsically tied to some other document. For example, you might refer the reader to the user document or the design document within your comments in a program. This document proposes a standard way to do this so that it is relatively easy to locate cross references

appearing in source code. The technique is similar to that for defect reporting, except the comments take the form:

```
// text #link#location text //
```

"Text" is optional and represents arbitrary text (although it is really intended for embedding html commands to provide hyperlinks to the specified document). "Location" describes the document and section where the associated information can be found.

Examples:

```
// #link#User's Guide Section 3.1 //
// #link#Program Design Document, Page 5 //
// #link#Funcs.pas module, "xyz" function //
// <A HREF="DesignDoc.html#xyzfunc"> #link#xyzfunc </a> //
```

Guideline: If a module contains some cross references to other documents, there should be a comment that takes the form `// text #link#location text //` that provides the reference to that other document. In this comment "text" represents some optional text (typically reserved for html tags) and "location" is some descriptive text that describes the document (and a position in that document) related to the current section of code in the program.

C.7 Names, Instructions, Operators, and Operands

Although program features like good comments, proper spacing of statements, and good modularization can help yield programs that are more readable; ultimately, a programmer must read the instructions in a program to understand what it does. Therefore, do not underestimate the importance of making your statements as readable as possible. This section deals with this issue.

C.7.1 Names

According to studies done at IBM, the use of high-quality identifiers in a program contributes more to the readability of that program than any other single factor, including high-quality comments. The quality of your identifiers can make or break your program; program with high-quality identifiers can be very easy to read, programs with poor quality identifiers will be very difficult to read. There are very few "tricks" to developing high-quality names; most of the rules are nothing more than plain old-fashion common sense. Unfortunately, programmers (especially C/C++ programmers) have developed many arcane naming conventions that ignore common sense. The biggest obstacle most programmers have to learning how to create good names is an unwillingness to abandon existing conventions. Yet their only defense when quizzed on why they adhere to (existing) bad conventions seems to be "because that's the way I've always done it and that's the way everybody else does it."

The aforementioned researchers at IBM developed several programs with the following set of attributes:

- Bad comments, bad names
- Bad comments, good names
- Good comments, bad names
- Good comments, good names

As should be obvious, the programs that had bad comments and names were the hardest to read; likewise, those programs with good comments and names were the easiest to read. The surprising results concerned the other two cases. Most people assume good comments are more important than good names in a program. Not only did IBM find this to be false, they found it to be *really* false.

As it turns out, good names are even more important than good comments in a program. This is not to say that comments are unimportant, they are extremely important; however, it is worth pointing out that if you spend the time to write good comments and then choose poor names for your program's identifiers, you've damaged the readability of your program despite the work you've put into your comments. Quickly read over the following code:

```
mov( SignedValue, ax );
cwd();
add( -1, ax );
rcl( 1, dx );
mov( dx, AbsoluteValue );
```

Question: What does this code compute and store in the AbsoluteValue variable?

- The sign extension of SignedValue.
- The negation of SignedValue.
- The absolute value of SignedValue.
- A boolean value indicating that the result is positive or negative.
- Signum(SignedValue) (-1, 0, +1 if neg, zero, pos).
- Ceil(SignedValue)
- Floor(SignedValue)

The obvious answer is the absolute value of SignedValue. This is also incorrect. The correct answer is signum:

```
mov( SignedValue, ax ); // Get value to check.
cwd();                  // DX = FFFF if neg, 0000 otherwise.
add( $ffff, ax );      // Carry=0 if ax is zero, one otherwise.
rcl( 1, dx );          // DX = FFFF if AX is neg, 0 if ax=0,
mov( dx, Signum );     // 1 if ax>0.
```

Granted, this is a tricky piece of code¹⁰. Nonetheless, even without the comments you can probably figure out what the code sequence does even if you can't figure out how it does it:

```
mov( SignedValue, ax );
cwd();
add( $ffff, ax );
rcl( 1, dx );
mov( dx, Signum );
```

Based on the names alone you can probably figure out that this code computes the signum function (even if understanding *how* it does it remains a mystery). This is the "understanding 80% of the code" referred to earlier. Note that you don't need misleading names to make this code unfathomable. Consider the following code that doesn't trick you by using misleading names:

```
mov( x, ax );
cwd();
add( $ffff, ax );
rcl( 1, dx );
mov( dx, y );
```

This is a very simple example. Now imagine a large program that has many names. As the number of names increase in a program, it becomes harder to keep track of them all. If the names themselves do not provide a good clue to the meaning of the name, understanding the program becomes very difficult.

Enforced Rule: All identifiers appearing in an assembly language program must be descriptive names whose meaning and use are clear.

¹⁰. It could be worse, you should see what the "superoptimizer" outputs for the signum function. It's even shorter and harder to understand than this code.

Since labels (i.e., identifiers) are the target of jump and call instructions, a typical assembly language program may have a large number of identifiers, especially if you write in “pure” assembly and forego the HLA high level control structures. Therefore, it is tempting to begin using names like “label1, label2, label3, ...” Avoid this temptation! There is always a reason you are jumping to some spot in your code. Try to describe that reason and use that description for your label name.

Rule: Never use names like “Lbl0, Lbl1, Lbl2, ...” in your program. Always use meaningful names!

C.7.1.1 Naming Conventions

Naming conventions represent one area in Computer Science where there are far too many divergent views (program layout is the other principle area). The primary purpose of an object’s name in a programming language is to describe the use and/or contents of that object. A secondary consideration may be to describe the type of the object. Programmers use different mechanisms to handle these objectives. Unfortunately, there are far too many “conventions” in place, it would be asking too much to expect any one programmer to follow several different standards. Therefore, this standard will apply across all languages as much as possible.

The vast majority of programmers know only one language - English. Some programmers know English as a second language and may not be familiar with a common non-English phrase that is not in their own language (e.g., rendezvous). Since English is the common language of most programmers, all identifiers should use easily recognizable English words and phrases.

Rule: All identifiers that represent words or phrases must be English words or phrases.

C.7.1.2 Alphabetic Case Considerations

A case-neutral identifier will work properly whether you compile it with a compiler that has case sensitive identifiers or case insensitive identifiers. In practice, this means that all uses of the identifiers must be spelled exactly the same way (including case) *and* that no other identifier exists whose only difference is the case of the letters in the identifier. For example, if you declare an identifier “ProfitsThisYear” in Pascal (a case-insensitive language), you could legally refer to this variable as “profitsThisYear” and “PROFITSTHISYEAR”. However, this is not a case-neutral usage since a case sensitive language would treat these three identifiers as different names. Conversely, in case-sensitive languages like C/C++, it is possible to create two different identifiers with names like “PROFITS” and “profits” in the program. This is not case-neutral since attempting to use these two identifiers in a case insensitive language (like Pascal) would produce an error since the case-insensitive language would think they were the same name.

Enforced Rule: All identifiers must be “case-neutral.”

Fortunately, HLA enforces case neutrality in its identifiers; so HLA doesn’t allow you to violate this rule. However, if you are linking assembly and high level language code together, it’s a good idea to follow this rule in the HLL code to prevent problems when linking with the HLA code.

Different programmers (especially in different languages) use alphabetic case to denote different objects. For example, a common C/C++ coding convention is to use all upper case to denote a constant, macro, or type definition and to use all lower case to denote variable names or reserved words. Prolog programmers use an initial lower case alphabetic to denote a variable. Other comparable coding conventions exist. Unfortunately, there are so many different conventions that make use of alphabetic case, they are nearly worthless, hence the following rule:

Rule: You should never use alphabetic case to denote the type, classification, or any other program-related attribute of an identifier.

There are going to be some obvious exceptions to the above rule, this document will cover those exceptions a little later. Alphabetic case does have one very useful purpose in identifiers - it is useful for separating words in a multi-word identifier; more on that subject in a moment.

To produce readable identifiers often requires a multi-word phrase. Natural languages typically use spaces to separate words; we can not, however, use this technique in identifiers.

Unfortunately writing multiword identifiers makes them almost impossible to read if you do not do something to distinguish the individual words (Unfortunately writing multiword identifiers makes them almost impossible to read if you do not do something to distinguish the individual words).

There are a couple of good conventions in place to solve this problem. This standard's convention is to capitalize the first alphabetic character of each word in the middle of an identifier.

Rule: Capitalize the first letter of interior words in all multi-word identifiers.

Note that the rule above does not specify whether the first letter of an identifier is upper or lower case. Subject to the other rules governing case, you can elect to use upper or lower case for the first symbol, although you should be consistent throughout your program. The second convention is to use an underscore to separate words in a multi-word document. This is also acceptable, though the capitalization rule probably produces identifiers that are easier to read and write.

Lower case characters are easier to read than upper case. Identifiers written completely in upper case take almost twice as long to recognize and, therefore, impair the readability of a program. Yes, all upper case does make an identifier stand out. Such emphasis is rarely necessary in real programs. Yes, common C/C++ coding conventions dictate the use of all upper case identifiers. Forget them. They not only make your programs harder to read, they also violate the first rule above.

Rule: Avoid using all upper case characters in an identifier.

Some programmers prefer to begin all identifiers with a lower case letter. Others prefer to begin them with an upper case alphabetic character. Either scheme is fine as long as you apply it consistently throughout your program. Under no circumstances should you use the presence of an upper or lower case character to denote different things in your code (see the earlier rule about this).

C.7.1.3 Abbreviations

The primary purpose of an identifier is to describe the use of, or value associated with, that identifier. The best way to create an identifier for an object is to describe that object in English and then create a variable name from that description. Variable names should be meaningful, concise, and non-ambiguous to an average programmer fluent in the English language. Avoid short names. Some research has shown that programs using identifiers whose average length is 10-20 characters are generally easier to debug than programs with substantially shorter or longer identifiers.

Avoid abbreviations as much as possible. What may seem like a perfectly reasonable abbreviation to you may totally confound someone else. Consider the following variable names that have actually appeared in commercial software:

NoEmployees, NoAccounts, pend

The "NoEmployees" and "NoAccounts" variables seem to be boolean variables indicating the presence or absence of employees and accounts. In fact, this particular programmer was using the (perfectly reasonable in the real world) abbreviation of "number" to indicate the number of employees and the number of accounts. The "pend" name referred to a procedure's end rather than any pending operation.

Programmers often use abbreviations in two situations: they're poor typists and they want to reduce the typing effort, or a good descriptive name for an object is simply too long. The former case is an unacceptable reason for using abbreviations. The second case, especially if care is taken, may warrant the occasional use of an abbreviation.

Guideline: Avoid all identifier abbreviations in your programs. When necessary, use standardized abbre-

viations or ask someone to review your abbreviations. Whenever you use abbreviations in your programs, create a “data dictionary” in the comments near the names’ definition that provides a full name and description for your abbreviation.

The variable names you create should be pronounceable. “NumFiles” is a much better identifier than “NmFls”. The first can be spoken, the second you must generally spell out. Avoid homonyms and long names that are identical except for a few syllables. If you choose good names for your identifiers, you should be able to read a program listing over the telephone to a peer without overly confusing that person.

Rule: All identifiers should be pronounceable (in English) without having to spell out more than one letter.

C.7.1.4 The Position of Components Within an Identifier

When scanning through a listing, most programmers only read the first few characters of an identifier. It is important, therefore, to place the most important information (that defines and makes this identifier unique) in the first few characters of the identifier. So, you should avoid creating several identifiers that all begin with the same phrase or sequence of characters since this will force the programmer to mentally process additional characters in the identifier while reading the listing. Since this slows the reader down, it makes the program harder to read.

Guideline: Try to make most identifiers unique in the first few character positions of the identifier. This makes the program easier to read.

Corollary: Never use a numeric suffix to differentiate two names.

Many C/C++ Programmers, especially Microsoft Windows programmers, have adopted a formal naming convention known as “Hungarian Notation.” To quote Steve McConnell from Code Complete: “The term ‘Hungarian’ refers both to the fact that names that follow the convention look like words in a foreign language and to the fact that the creator of the convention, Charles Simonyi, is originally from Hungary.” One of the first rules given concerning identifiers stated that all identifiers are to be English names. Do we really want to create “artificially foreign” identifiers? Hungarian notation actually violates another rule as well: names using the Hungarian notation generally have very common prefixes, thus making them harder to read.

Hungarian notation does have a few minor advantages, but the disadvantages far outweigh the advantages. The following list from Code Complete and other sources describes what’s *wrong* with Hungarian notation:

- Hungarian notation generally defines objects in terms of basic machine types rather than in terms of abstract data types.
- Hungarian notation combines *meaning* with *representation*. One of the primary purposes of high level language is to abstract representation away. For example, if you declare a variable to be of type *integer*, you shouldn’t have to change the variable’s name just because you changed its type to *real*.
- Hungarian notation encourages lazy, uninformative variable names. Indeed, it is common to find variable names in Windows programs that contain *only* type prefix characters, without a descriptive name attached.
- Hungarian notation prefixes the descriptive name with some type information, thus making it harder for the programming to find the descriptive portion of the name.

Guideline: Avoid using Hungarian notation and any other formal naming convention that attaches low-level type information to the identifier.

Although attaching *machine* type information to an identifier is generally a bad idea, a well thought-out

name can successfully associate some high-level type information with the identifier, especially if the name implies the type or the type information appears as a suffix. For example, names like “PencilCount” and “BytesAvailable” suggest integer values. Likewise, names like “IsReady” and “Busy” indicate boolean values. “KeyCode” and “MiddleInitial” suggest character variables. A name like “StopWatchTime” probably indicates a real value. Likewise, “CustomerName” is probably a string variable. Unfortunately, it isn’t always possible to choose a great name that describes both the content and type of an object; this is particularly true when the object is an instance (or definition of) some abstract data type. In such instances, some additional text can improve the identifier. Hungarian notation is a raw attempt at this that, unfortunately, fails for a variety of reasons.

A better solution is to use a *suffix phrase* to denote the type or class of an identifier. A common UNIX/C convention, for example, is to apply a “_t” suffix to denote a type name (e.g., size_t, key_t, etc.). This convention succeeds over Hungarian notation for several reasons including (1) the “type phrase” is a suffix and doesn’t interfere with reading the name, (2) this particular convention specifies the *class* of the object (const, var, type, function, etc.) rather than a low level *type*, and (3) It certainly makes sense to change the identifier if it’s classification changes.

Guideline: If you *want* to differentiate identifiers that are constants, type definitions, and variable names, use the suffixes “_c”, “_t”, and “_v”, respectively (generally, the lack of a suffix denotes a variable).

Rule: The classification suffix should not be the only component that differentiates two identifiers.

Can we apply this suffix idea to variables and avoid the pitfalls? Sometimes. Consider a high level data type “button” corresponding to a button on a Visual BASIC or Delphi form. A variable name like “CancelButton” makes perfect sense. Likewise, labels appearing on a form could use names like “ETWWLabel” and “EditPageLabel”. Note that these suffixes still suffer from the fact that a change in type will require that you change the variable’s name. However, changes in high level types are far less common than changes in low-level types, so this shouldn’t present a big problem.

HLA provides a special operator, “`” (grave accent) that separates an identifier name from an attached comment. For example, the legal HLA identifier “Hello`world” is really just “Hello”. The characters following the grave accent (to the end of the identifier) are treated as a comment by the compiler. So if you want to attach a comment concerning the variable’s type or use to the identifier, you can use this feature in HLA.

Guideline: If you must attach low level type information to an identifier, use the HLA identifier comment (“`”, grave accent) and append the information to the end of the identifier.

C.7.1.5 Names to Avoid

Avoid using symbols in an identifier that are easily mistaken for other symbols. This includes the sets {“1” (one), “I” (upper case “I”), and “l” (lower case “L”)}, {“0” (zero) and “O” (upper case “O”)}, {“2” (two) and “Z” (upper case “Z”)}, {“5” (five) and “S” (upper case “S”)}, and {“6” (six) and “G” (upper case “G”)}

Guideline: Avoid using symbols in identifiers that are easily mistaken for other symbols (see the list above).

Avoid misleading abbreviations and names. For example, FALSE shouldn’t be an identifier that stands for “Failed As a Legitimate Software Engineer.” Likewise, you shouldn’t compute the amount of free memory available to a program and stuff it into the variable “Profits”.

Rule: Avoid misleading abbreviations and names.

You should avoid names with similar meanings. For example, if you have two variables “InputLine” and “InputLn” that you use for two separate purposes, you will undoubtedly confuse the two when writing

or reading the code. If you can swap the names of the two objects and the program still makes sense, you should rename those identifiers. Note that the names do not have to be similar, only their meanings. “InputLine” and “LineBuffer” are obviously different but you can still easily confuse them in a program.

Rule: Do not use names with similar meanings for different objects in your programs.

In a similar vein, you should avoid using two or more variables that have different meanings but similar names. For example, if you are writing a teacher’s grading program you probably wouldn’t want to use the name “NumStudents” to indicate the number of students in the class along with the variable “StudentNum” to hold an individual student’s ID number. “NumStudents” and “StudentNum” are too similar.

Rule: Do not use similar names that have different meanings.

Avoid names that sound similar when read aloud, especially out of context. This would include names like “hard” and “heart”, “Knew” and “new”, etc. Remember the discussion in the section above on abbreviations, you should be able to discuss your problem listing over the telephone with a peer. Names that sound alike make such discussions difficult.

Guideline: Avoid homonyms in identifiers.

Avoid misspelled words in names and avoid names that are commonly misspelled. Most programmers are notoriously bad spellers (look at some of the comments in our own code!). Spelling words correctly is hard enough, remembering how to spell an identifier *incorrectly* is even more difficult. Likewise, if a word is often spelled incorrectly, requiring a programmer to spell it correctly on each use is probably asking too much.

Guideline: Avoid misspelled words and names that are often misspelled in identifiers.

If you redefine the name of some library routine in your code, another program will surely confuse your name with the library’s version. This is especially true when dealing with standard library routines and APIs.

Enforced Rule: Do not reuse existing standard library routine names in your program unless you are specifically replacing that routine with one that has similar semantics (i.e., don’t reuse the name for a different purpose).

Corollary: Use Namespaces to prevent name space pollution!

C.7.1.6 Special Identifiers

By convention, HLA programmers use certain identifiers for special purposes. Any identifier beginning with an underscore falls into this category. HLA defines five such conventions. The HLA compiler does not enforce these conventions, but if you violate them you may run into problems. The following paragraphs describe each of these conventions.

HLA reserves for its own use (and the use of the HLA Standard Library) all identifiers that begin and end with a single underscore. You should never define any identifiers in your programs that take this form since your identifiers may conflict with HLA’s use. These reservation effectively reserves an “HLA namespace” of identifiers that the compiler and Standard Library can draw from without fear of breaking any existing code (that follows this convention).

Identifiers that begin and end with two underscores are reserved for use as local symbols in user-defined macros. To avoid conflicts with such symbols (especially in context-free/multi-part macros) you should never use symbols that begin and end with two underscores outside of a macro. Within a macro, you should use the convention for all symbols that are local to that macro.

By convention, HLA programmers reserve all identifiers beginning with two underscores for defining private data in classes, records, and other declaration sections. If you're using a class (or other structure) and some of its identifiers begin with two underscores, this is your hint that these fields are private to that class and subject to change. You should never directly access such fields. When you're defining your own classes, you should employ this convention to warn others when you're defining private data to that class.

Identifiers that begin with a single underscore have two uses. First, some languages and calling conventions (most notably, C) prepend an underscore to all external names. Therefore, it is common to use reserve symbols that begin with a single underscore for external linkage. The second use is closely related to the first – HLA programmers conventionally use identifiers beginning with a single underscore as a *shadow name*. Consider the following linkage to an external procedure written in C:

```
procedure _externalCFunc( parm2:int32; parm1:int32 ); external;
#macro externalCFunc( p1, p2 );

    _externalCFunc( p2, p1 );

#endmacro;
```

The C compiler exports the name “_externalCFunc” for the C function that is actually named “externalCFunc” inside the C code. Of course, inside our HLA code we would like to use the C name, not the exported name. We could easily achieve this using the following external definition:

```
procedure externalCFunc( parm2:int32; parm1:int32 ); external("_externalCFunc");
```

The only catch is that we'd have to always remember to put the parameters in the reverse order (to match C's calling convention). Savvy HLA programmers use a macro to swap the parameters as in the previous example. They use the exported C name as a shadow name of the function and then write the macro that swaps the function's parameters as the real name.

You can use shadow names for all sorts of different purposes, not just for linkage to C functions. For example, the chapter on macros in this text has given examples of function overloading that uses a macro with the overloaded function name and shadow names for the actual functions that implement each of the overloaded calls. The convention is to use a leading underscore on all the shadow function (and other object) names.

C.7.2 Instructions, Directives, and Pseudo-Opcodes

Your choice of assembly language sequences, the instructions themselves, and your choice of directives and pseudo-opcodes can have a big impact on the readability of your programs. The following subsections discuss these problems.

C.7.2.1 Choosing the Best Instruction Sequence

Like any language, you can solve a given problem using a wide variety of solutions involving different instruction sequences. As a continuing example, consider (again) the following code sequence:

```
mov( SignedValue, ax ); // Get value to check.
cwd();                 // DX = FFFF if neg, 0000 otherwise.
add( $ffff, ax );     // Carry=0 if ax is zero, one otherwise.
rcl( 1, dx );         // DX = FFFF if AX is neg, 0 if ax=0,
mov( dx, Signum );    // 1 if ax>0.
```

Now consider the following code sequence that also computes the signum function:

```
mov( SignedValue, ax ); // Get value to check.
cmp( ax, 0 );           // Check the sign.
je GotSigum;           // We're done if it's zero
mov( 1, ax );          // Assume it's positive.
```



```

                jns GotSignum;           // We're done if it was positive.
                neg( ax );              // 1 -> -1, we've got a negative value.
GotSignum:     mov( ax, Signum );

```

Yes, the second version is longer and slower. However, an average person can read the instruction sequence and figure out what it's doing; hence the second version is much easier to read than the first. Which sequence is best? Unless speed or space is an extremely critical factor and you can show that this routine is in the critical execution path, then the second version is obviously better. There is a time and a place for tricky assembly code; however, it's rare that you would need to pull tricks like this throughout your code.

So how does one choose appropriate instruction sequences when there are many possible ways to accomplish the same task? The best way is to ensure that you have a choice. Although there are many different ways to accomplish an operation, few people bother to consider any instruction sequence other than the first one that comes to their mind. Unfortunately, the "best" instruction sequence is rarely the first instruction sequence that comes to most people's minds¹¹. In order to make a choice, you have to have a choice to make. That means you should create at least two different code sequences for a given operation if there is ever a question concerning the readability of your code. Once you have at least two versions, you can choose between them based on your needs at hand. While it is impractical to "write your program twice" so that you'll have a choice for every sequence of instructions in the program, you should apply this technique to particularly bothersome code sequences.

Guideline: For particularly difficult to understand sections of code, try solving the problem several different ways. Then choose the most easily understood solution for actual incorporation into your program.

One problem with the above suggestion is that you're often too close to your own work to make decisions like "this code isn't too hard to understand, I don't have to worry about it." It is often a good idea to have someone else review your code and point out those sections they find hard to understand¹².

Guideline: Take advantage of reviews to determine those sections of code in your program that may need to be rewritten to make them easier to understand.

C.7.2.2 Control Structures

Ralph Griswold¹³ once said (roughly) the following about C, Pascal, and Icon: "C makes it easy to write hard to read programs¹⁴, Pascal makes it hard to write hard to read programs, and Icon makes it easy to write easy to read programs." Assembly language can be summed up like this: "Assembly language makes it hard to write easy to read programs and easy to write hard to read programs." It takes considerable discipline to write readable assembly language programs; *but it can be done*. Sadly, most assembly code you find today is extremely poorly written. Indeed, that state of affairs is the whole reason for this document. Once you get past issues like comments and naming conventions, issues like program control flow and data structure design have among the largest impacts on program readability. One need look no farther than the public domain code on the Internet, or at Microsoft's sample code for that matter¹⁵, to see abundant examples of poorly written assembly language code.

Fortunately, with a little discipline it is possible to write readable assembly language programs. Particularly in HLA which was designed from the beginning to allow the easy creation of readable code. How you design your control structures can have a big impact on the readability of your programs. The best way to do this can be summed up in two words: avoid spaghetti.

11. This is true regardless of what metric you use to determine the "best" code sequence.

12. Of course, if the program is a *class assignment*, you may want to check your instructor's cheating policy before showing your work to your classmates!

13. The designer of the SNOBOL4 and Icon programming languages.

14. Note that this does not infer that it is hard to write easy to read C programs. Only that if one is sloppy, one can easily write something that is near impossible to understand.

15. Okay, this is a cheap shot. In fact, most of the assembly code on this planet is poorly written.

Spaghetti code is the name given to a program that has a large number of intertwined branches and branch targets within a code sequence. Consider the following example:

```

                jmp L1;
L1:             mov( 0, ax );
                jmp L2;
L3:             mov( 1, ax );
                jmp L2;
L4:             mov( -1, ax );
                jmp L2;
L0:             mov( x, ax );
                cmp( ax, 0 );
                je L1;
                jns L3;
                jmp L4;
L2:             mov( ax, y );

```

This code sequence, by the way, is our good friend the Signum function. It takes a few moments to figure this out because as you manually trace through the code you find yourself spending more time following jumps around than you do looking at code that computes useful results. Now this is a rather extreme example, but it is also fairly short. A longer code sequence code become just as obfuscated with even fewer branches all over the place.

Spaghetti code is given this name because it resembles a bowl of spaghetti. That is, if we consider a control path in the program a spaghetti noodle, spaghetti code contains lots of intertwined branches into and out of different sections of the program. Needless to say, most spaghetti programs are difficult to understand, generally contain lots of bugs, and are often inefficient (don't forget that branches are among the slowest executing instructions on most modern processors).

So how do we resolve this? Easy by physically adopting structured programming techniques in assembly language code. Of course, "pure" 80x86 assembly language doesn't provide IF..THEN..ELSE..ENDIF, WHILE..ENDWHILE, REPEAT..UNTIL, and other such statements, but we can certainly simulate them if you insist on writing "pure" assembly code¹⁶. Consider the following high level sequence:

```

if(expression) then

    << statements to execute if expression is true >>

else

    << statements to execute if expression is false >>

endif;

```

Almost any high level language programmer can figure out what this type of statement will do. Assembly language programmers should leverage this knowledge by attempting to organize their code so it takes this same form. Specifically, the assembly language version should look something like the following:

```

    << Assembly code to compute value of expression >>

JNxx    ElsePart ;xx is the opposite condition we want to check.

    << Assembly code corresponding to the then portion >>

    jmp    AroundElsePart

ElsePart:

    << Assembly code corresponding to the else portion >>

AroundElsePart:

```

16. We'll consider the HLA high level control statements elsewhere in this appendix.

For an concrete example, consider the following:

```

if( ax = y ) then
    write( 'ax = y' );
else
    write( 'ax <> y' );
endif;

; Corresponding Assembly Code:

mov( x, ax );
cmp( ax, y );
jne ElsePart;

stdout.put( "x = y",nl );
jmp IfDone;

ElsePart:    stdout.put( "x<>y",nl );
IfDone:

```

While this may seem like the obvious way to organize an IF.THEN.ELSE..ENDIF statement, it is surprising how many people would naturally assume they've got to place the ELSE part somewhere else in the program as follows:

```

mov( x, ax );
cmp( ax, y );
jne ElsePart;

stdout.put( "x = y", nl );

IfDone:
.
.
.
ElsePart:    stdout.put( "x <> y", nl );
jmp IfDone;

```

This code organization makes the program more difficult to follow. Most programmers have a HLL background and despite a current assignment, they still work mostly in HLLs. Assembly language programs will be more readable if they mimic the HLL control constructs¹⁷.

For similar reasons, you should attempt to organize your assembly code that simulates WHILE loops, REPEAT..UNTIL loops, FOR loops, etc., so that the code resembles the HLL code (for example, a WHILE loop should physically test the condition at the beginning of the loop with a jump at the bottom of the loop).

Rule: Attempt to design your programs using HLL control structures. The organization of the assembly code that you write should physically resemble the organization of some corresponding HLL program.

Assembly language offers you the flexibility to design arbitrary control structures. This flexibility is one of the reasons good assembly language programmers can write better code than that produced by a compiler (that can only work with high level control structures). However, keep in mind that a fast program

17. Sometimes, for performance reasons, the code sequence above is justified since straight-line code executes faster than code with jumps. If the program rarely executes the ELSE portion of an if statement, always having to jump over it could be a waste of time. But if you're optimizing for speed, you will often need to sacrifice readability.

doesn't have to contain the tightest possible code in every sequence. Execution speed is nearly irrelevant in most parts of the program. Sacrificing readability for speed isn't a big win in most of the program.

Guideline: Avoid control structures that don't easily map to well-known high level language control structures in your assembly language programs. Deviant control structures should only appear in small sections of code when efficiency demands their use.

C.7.2.3 Instruction Synonyms

HLA defines several synonyms for common instructions. This is especially true for the conditional jump and "set on condition code" instructions. For example, JA and JNBE are synonyms for one another. Logically, one could use either instruction in the same context. However, the choice of synonym can have an impact on the readability of a code sequence. To see why, consider the following:

```

        if( x <= y ) then
            << true statements>>
        else
            << false statements>>
        endif

// Assembly code:

        mov( x, ax );
        cmp( ax, y );
        ja ElsePart;

        << true code >>

        jmp IfDone;

ElsePart:    << false code >>
IfDone:
```

When someone reads this program, the "JA" statement skips over the true portion. Unfortunately, the "JA" instruction gives the illusion we're checking to see if something is greater than something else; in actuality, we're testing to see if some condition is less than or equal, not greater than. As such, this code sequence hides some of the original intent of high level algorithm. One solution is to swap the false and true portions of the code:

```

        mov( x, ax );
        cmp( ax, y );
        jbe ThenPart;

        << false code >>

        jmp IfDone;

ThenPart:    << true code >>
IfDone:
```

This code sequence uses the conditional jump that matches the high level algorithm's test (less than or equal). However, this code is now organized in a non-standard fashion (it's an IF.ELSE.THEN.ENDIF statement). This hurts the readability more than using the proper jump helped it. Now consider the following solution:

```

        mov( x, ax );
        cmp( ax, y );
        jnbe ElsePart;

        << true code >>
```

```

        jmp IfDone;

ElsePart:    << false code >>
IfDone:

```

This code is organized in the traditional IF..THEN..ELSE..ENDIF fashion. Instead of using JA to skip over the then portion, it uses JNBE to do so. This helps indicate, in a more readable fashion, that the code falls through on below or equal and branches if it is not below or equal. Since the instruction (JNBE) is easier to relate to the original test (<=) than JA, this makes this section of code a little more readable.

Rule: When skipping over some code because some condition has failed (e.g., you fall into the code because the condition is successful), always use a conditional jump of the form "JNxx" to skip over the code section. For example, to fall through to a section of code if one value is less than another, use the JNL or JNB instruction to skip over the code. Of course, if you are testing a negative condition (e.g., testing for equality) then use an instruction of the form Jx to skip over the code.

C.8 Data Types

Prior to the arrival of MASM from Microsoft for the 80x86, most assemblers provided very little capability for declaring and allocated complex data types. Generally, you could allocate bytes, words, and other primitive machine structures. You could also set aside a block of bytes. As high level languages improved their ability to declare and use abstract data types, assembly language fell farther and farther behind. Then MASM came along and changed all that¹⁸. HLA expands the ability to declare abstract data types even farther than MASM. Unfortunately, many new assembly language programmers don't bother learning and using these data typing facilities because they're already overwhelmed by assembly language and want to minimize the number of things they've got to learn. This is really a shame because HLA's data typing is one of the biggest improvements to assembly language since using mnemonics rather than binary opcodes for machine level programming.

Note that HLA is a "high-level" assembler. It does things assemblers for other chips won't do like checking the types of operands and reporting errors if there are mismatches. Some people, who are used to assemblers on other machines find this annoying. However, it's a great idea in assembly language for the same reason it's a great idea in HLLs¹⁹. These features have one other beneficial side-effect: they help other understand what you're trying to do in your programs. It should come as no surprise, then, that this style guide will encourage the use of these features in your assembly language programs.

C.8.1 Declaring Structures in Assembly Language

HLA provides an excellent facility for declaring and using records and unions; for some reason, many assembly language programmers ignore them and manually compute offsets to fields within structures in their code. Not only does this produce hard to read code, the result is nearly unmaintainable as well.

Rule: When a structure data type is appropriate in an assembly language program, declare the corresponding structure in the program and use it. Do not compute the offsets to fields in the structure manually, use the standard structure "dot-notation" to access fields of the structure.

One problem with using structures occurs when you access structure fields indirectly (i.e., through a pointer). Indirect access always occurs through a register. Once you load a pointer value into a register, the program doesn't readily indicate what pointer you are using. This is especially true if you use the indirect

18. Okay, MASM wasn't the first, but such techniques were not popularized until MASM appeared.

19. Of course, MASM gives you the ability to override this behavior when necessary. Therefore, the complaints from "old-hand" assembly language programmers that this is insane are groundless.

access several times in a section of code without reloading the register(s). One solution is to use a text constant to create a special symbol that expands as appropriate. Consider the following code:

```

type
  s:record

    a: int32;
    b: int32;

  endrecord;
  .
  .
  .
static
  r: s;
  ptr2r: pointer to s;
  .
  .
  .
  mov( ptr2r, edi );
  mov( (type s [edi]).a, eax ); // No indication this is ptr2r!
  .
  .
  .
  mov( ebx, (type s [edi]).b ); // Still no indication.

```

Now consider the following:

```

type
  s:record

    a: int32;
    b: int32;

  endrecord;

  sptr : pointer to s;
  .
  .
  .
static
  r: s;
  ptr2r: sptr := q;
  ?_r:text := (type s [edi])";
  .
  .
  .
  mov( ptr2r, edi );
  mov( _r.a, eax ); // Now it's a lot more clear that we're using r.
  .
  .
  .
  mov( ebx, _r.b ); // It's still clear that we're using r!

```

Note that the "_" symbol is a legal identifier character to HLA, hence "_r" is just another symbol. Of course, you must always make sure to load the pointer into EDI when using the text constant above. If you use several different registers to access the data that "r" points at, this trick may not make the code anymore readable since you will need several text constants that all mean the same thing.

The 80x86 Instruction Set

Appendix D

The following three tables discuss the integer/control, floating point, and MMX instruction sets. This document uses the following abbreviations:

- imm- A constant value, must be appropriate for the operand size.
- imm8- An eight-bit immediate constant. Some instructions limit the range of this value to less than 0..255.
- immL- A 16- or 32-bit immediate constant.
- immH- A 16- or 32-bit immediate constant.
- reg- A general purpose integer register.
- reg8- A general purpose eight-bit register
- reg16- A general purpose 16-bit register.
- reg32- A general purpose 32-bit register.
- mem- An arbitrary memory location using any of the available addressing modes.
- mem16- A word variable using any legal addressing mode.
- mem32- A dword variable using any legal addressing mode.
- mem64- A qword variable using any legal addressing mode.
- label- A statement label in the program.
- ProcedureName-The name of a procedure in the program.

Instructions that have two source operands typically use the first operand as a source operand and the second operand as a destination operand. For exceptions and other formats, please see the description for the individual instruction.

Note that this appendix only lists those instructions that are generally useful for application programming. HLA actually supports some additional instructions that are useful for OS kernel developers; please see the HLA documentation for more details on those instructions.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
aaa()	ASCII Adjust after Addition. Adjusts value in AL after a decimal addition operation.
aad()	ASCII Adjust before Division. Adjusts two unpacked values in AX prior to a decimal division.
aam()	ASCII Adjust AX after Multiplication. Adjusts the result in AX for a decimal multiply.
aas()	ASCII Adjust AL after Subtraction. Adjusts the result in AL for a decimal subtraction.
adc(imm, reg); adc(imm, mem); adc(reg, reg); adc(reg, mem); adc(mem, reg);	Add with carry. Adds the source operand plus the carry flag to the destination operand.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
add(imm, reg); add(imm, mem); add(reg, reg); add(reg, mem); add(mem, reg);	Add. Adds the source operand to the destination operand.
and(imm, reg); and(imm, mem); and(reg, reg); and(reg, mem); and(mem, reg);	Bitwise AND. Logically ANDs the source operand into the destination operand. Clears the carry and overflow flags and sets the sign and zero flags according to the result.
bound(reg, mem); bound(reg, immL, immH);	<p>Bounds check. Reg and memory operands must be the same size and they must be 16 or 32-bit values. This instruction compares the register operand against the value at the specified memory location and raises an exception if the register's value is less than the value in the memory location. If greater or equal, then this instruction compares the register to the next word or dword in memory and raises an exception if the register's value is greater.</p> <p>The second form of this instruction is an HLA extended syntax instruction. HLA encodes the constants as two memory locations and then emits the first form of this instruction using these newly created memory locations.</p> <p>For the second form, the constant values must not exceed the 16-bit or 32-bit register size.</p>
bsf(reg, reg); bsr(mem, reg);	Bit Scan Forward. The two operands must be the same size and they must be 16-bit or 32-bit operands. This instruction locates the first set bit in the source operand and stores the bit number into the destination operand and clears the zero flag. If the source operand does not have any set bits, then this instruction sets the zero flag and the dest register value is undefined.
bsr(reg, reg); bsr(mem, reg);	Bit Scan Reverse. The two operands must be the same size and they must be 16-bit or 32-bit operands. This instruction locates the last set bit in the source operand and stores the bit number into the destination operand and clears the zero flag. If the source operand does not have any set bits, then this instruction sets the zero flag and the dest register value is undefined.
bswap(reg32);	Byte Swap. This instruction reverses the order of the bytes in a 32-bit register. It swaps bytes zero and three and it swaps bytes one and two. This effectively converts data between the little endian (used by Intel) and big endian (used by some other CPUs) formats.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
bt(reg, mem); bt(reg, reg); bt(imm8, reg); bt(imm8, mem);	Register and memory operands must be 16- or 32-bit values. Eight bit immediate values must be in the range 0..15 for 16-bit registers, 0..31 for 32-bit registers, and 0..255 for memory operands. Source register must be in the range 0..15 or 0..31 for registers. Any value is legal for the source register if the destination operand is a memory location. This instruction copies the bit in the second operand, whose bit position the first operand specifies, into the carry flag.
btc(reg, mem); btc(reg, reg); btc(imm8, reg); btc(imm8, mem);	Bit test and complement. As above, except this instruction also complements the value of the specified bit in the second operand. Note that this instruction first copies the bit to the carry flag, then complements it. To support atomic operations, the memory-based forms of this instruction are always “memory locked” and they always directly access main memory; the CPU does not use the cache for this result. Hence, this instruction always operates at memory speeds (i.e., slow).
btr(reg, mem); btr(reg, reg); btr(imm8, reg); btr(imm8, mem);	Bit test and reset. Same as BTC except this instruction tests and resets (clears) the bit.
bts(reg, mem); bts(reg, reg); bts(imm8, reg); bts(imm8, mem);	Bit test and set. Same as BTC except this instructions tests and sets the bit.
call label; call(label); call(reg32); call(mem32);	Pushes a return address onto the stack and calls the subroutine at the address specified. Note that the first two forms are the same instruction. The other two forms provide indirect calls via a register or a pointer in memory.
cbw();	Convert Byte to Word. Sign extends AL into AX.
cdq();	Convert double word to quadword. Sign extends EAX into EDX:EAX.
clc();	Clear Carry.
cld();	Clear direction flag. When the direction flag is clear the string instructions increment ESI and/or EDI after each operation.
cli();	Clear the interrupt enable flag.
cmc();	Complement (invert) Carry.
cmova(mem, reg); cmova(reg, reg); cmova(reg, mem);	Conditional Move (if above). Copies the source operand to the destination operand if the previous comparison found the left operand to be greater than (unsigned) the right operand (c=0, z=0). Register and memory operands must be 16-bit or 32-bit values, eight-bit operands are illegal. Does not affect the destination operand if the condition is false.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
cmovae(mem, reg); cmovae(reg, reg); cmovae(reg, mem);	Conditional move if above or equal (see cmova for details).
cmovb(mem, reg); cmovb(reg, reg); cmovb(reg, mem);	Conditional move if below (see cmova for details).
cmovbe(mem, reg); cmovbe(reg, reg); cmovbe(reg, mem);	Conditional move if below or equal (see cmova for details).
cmovc(mem, reg); cmovc(reg, reg); cmovc(reg, mem);	Conditional move if carry set (see cmova for details).
cmove(mem, reg); cmove(reg, reg); cmove(reg, mem);	Conditional move if equal (see cmova for details).
cmovg(mem, reg); cmovg(reg, reg); cmovg(reg, mem);	Conditional move if (signed) greater (see cmova for details).
cmovge(mem, reg); cmovge(reg, reg); cmovge(reg, mem);	Conditional move if (signed) greater or equal (see cmova for details).
cmovl(mem, reg); cmovl(reg, reg); cmovl(reg, mem);	Conditional move if (signed) less than (see cmova for details).
cmovle(mem, reg); cmovle(reg, reg); cmovle(reg, mem);	Conditional move if (signed) less than or equal (see cmova for details).
cmovna(mem, reg); cmovna(reg, reg); cmovna(reg, mem);	Conditional move if (unsigned) not greater (see cmova for details).
cmovnae(mem, reg); cmovnae(reg, reg); cmovnae(reg, mem);	Conditional move if (unsigned) not greater or equal (see cmova for details).
cmovnb(mem, reg); cmovnb(reg, reg); cmovnb(reg, mem);	Conditional move if (unsigned) not less than (see cmova for details).

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
cmovnbe(mem, reg); cmovnbe(reg, reg); cmovnbe(reg, mem);	Conditional move if (unsigned) not less than or equal (see cmova for details).
cmovnc(mem, reg); cmovnc(reg, reg); cmovnc(reg, mem);	Conditional move if no carry/carry clear (see cmova for details).
cmovne(mem, reg); cmovne(reg, reg); cmovne(reg, mem);	Conditional move if not equal (see cmova for details).
cmovng(mem, reg); cmovng(reg, reg); cmovng(reg, mem);	Conditional move if (signed) not greater (see cmova for details).
cmovnge(mem, reg); cmovnge(reg, reg); cmovnge(reg, mem);	Conditional move if (signed) not greater or equal (see cmova for details).
cmovnl(mem, reg); cmovnl(reg, reg); cmovnl(reg, mem);	Conditional move if (signed) not less than (see cmova for details).
cmovnle(mem, reg); cmovnle(reg, reg); cmovnle(reg, mem);	Conditional move if (signed) not less than or equal (see cmova for details).
cmovno(mem, reg); cmovno(reg, reg); cmovno(reg, mem);	Conditional move if no overflow / overflow flag = 0 (see cmova for details).
cmovnp(mem, reg); cmovnp(reg, reg); cmovnp(reg, mem);	Conditional move if no parity / parity flag = 0 / odd parity (see cmova for details).
cmovns(mem, reg); cmovns(reg, reg); cmovns(reg, mem);	Conditional move if no sign / sign flag = 0 (see cmova for details).
cmovnz(mem, reg); cmovnz(reg, reg); cmovnz(reg, mem);	Conditional move if not zero (see cmova for details).
cmovo(mem, reg); cmovo(reg, reg); cmovo(reg, mem);	Conditional move if overflow / overflow flag = 1 (see cmova for details).

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
cmovp(mem, reg); cmovp(reg, reg); cmovp(reg, mem);	Conditional move if parity flag = 1 (see cmova for details).
cmovpe(mem, reg); cmovpe(reg, reg); cmovpe(reg, mem);	Conditional move if even parity / parity flag = 1 (see cmova for details).
cmovpo(mem, reg); cmovpo(reg, reg); cmovpo(reg, mem);	Conditional move if odd parity / parity flag = 0 (see cmova for details).
cmovs(mem, reg); cmovs(reg, reg); cmovs(reg, mem);	Conditional move if sign flag = 1 (see cmova for details).
cmovz(mem, reg); cmovz(reg, reg); cmovz(reg, mem);	Conditional move if zero flag = 1 (see cmova for details).
cmp(imm, reg); cmp(imm, mem); cmp(reg, reg); cmp(reg, mem); cmp(mem, reg);	Compare. Compares the first operand against the second operand. The two operands must be the same size. This instruction sets the condition code flags as appropriate for the condition jump and set instructions. This instruction does not change the value of either operand.
cmpsb(); repe.cmpsb(); repne.cmpsb();	Compare string of bytes. Compares the byte pointed at by ESI with the byte pointed at by EDI and then adjusts ESI and EDI by ± 1 depending on the value of the direction flag. Sets the flags according to the result. With the REPNE (repeat while not equal) flag, this instruction compares up to ECX bytes until all the first byte it finds in the two string that are equal. With the REPE (repeat while equal) prefix, this instruction compares two strings up to the first byte that is different. See the chapter on the String Instructions for more details.
cmpsw() repe.cmpsw(); repne.cmpsw();	Compare a string of words. Like cmpsb except this instruction compares words rather than bytes and adjusts ESI/EDI by ± 2 .
cmpsd() repe.cmpsd(); repne.cmpsd();	Compare a string of double words. Like cmpsb except this instruction compares double words rather than bytes and adjusts ESI/EDI by ± 4 .
cpxchg(reg, mem); cpxchg(reg, reg);	Reg and mem must be the same size. They can be eight, 16, or 32 bit objects. This instruction compares the value in the accumulator (al, ax, or eax) against the second operand. If the two values are equal, this instruction copies the source (first) operand to the destination (second) operand. Otherwise, this instruction copies the second operand into the accumulator.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
cmpxchg8b(mem64);	Compares the 64-bit value in EDX:EAX with the memory operand. If the values are equal, then this instruction stores the 64-bit value in ECX:EBX into the memory operand and sets the zero flag. Otherwise, this instruction copies the 64-bit memory operand into the EDX:EAX registers and clears the zero flag.
cpuid();	CPU Identification. This instruction identifies various features found on the different Pentium processors. See the Intel documentation on this instruction for more details.
cwd();	Convert Word to Double. Sign extends AX to DX:AX.
cwde();	Convert Word to Double Word Extended. Sign extends AX to EAX.
daa();	Decimal Adjust after Addition. Adjusts value in AL after a decimal addition.
das();	Decimal Adjust after Subtraction. Adjusts value in AL after a decimal subtraction.
dec(reg); dec(mem);	Decrement. Subtracts one from the destination memory location or register.
div(reg); div(reg8, ax); div(reg16, dx:ax); div(reg32, edx:eax); div(mem); div(mem8, ax); div(mem16, dx:ax); div(mem32, edx:eax); div(imm8, ax); div(imm16, dx:ax); div(imm32, edx:eax);	Divides accumulator or extended accumulator (dx:ax or edx:eax) by the source operand. Note that the instructions involving an immediate operand are HLA extensions. HLA creates a memory object holding these constants and then divides the accumulator or extended accumulator by the contents of this memory location. Note that the accumulator operand is twice the size of the source (divisor) operand. This instruction computes the quotient and places it in AL, AX, or EAX and it computes the remainder and places it in AH, DX, or EDX (depending on the divisor's size). This instruction raises an exception if you attempt to divide by zero or if the quotient doesn't fit in the destination register (AL, AX, or EAX). This instruction performs an unsigned division.
enter(imm16, imm8);	Enter a procedure. Creates an activation record for a procedure. The first constant specifies the number of bytes of local variables. The second parameter (in the range 0..31) specifies the static nesting level (lex level) of the procedure.
idiv(reg); idiv(reg8, ax); idiv(reg16, dx:ax); idiv(reg32, edx:eax); idiv(mem); idiv(mem8, ax); idiv(mem16, dx:ax); idiv(mem32, edx:eax); idiv(imm8, ax); idiv(imm16, dx:ax); idiv(imm32, edx:eax);	Divides accumulator or extended accumulator (dx:ax or edx:eax) by the source operand. Note that the instructions involving an immediate operand are HLA extensions. HLA creates a memory object holding these constants and then divides the accumulator or extended accumulator by the contents of this memory location. Note that the accumulator operand is twice the size of the source (divisor) operand. This instruction computes the quotient and places it in AL, AX, or EAX and it computes the remainder and places it in AH, DX, or EDX (depending on the divisor's size). This instruction raises an exception if you attempt to divide by zero or if the quotient doesn't fit in the destination register (AL, AX, or EAX). This instruction performs a signed division. The condition code bits are undefined after executing this instruction.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
imul(reg); imul(reg8, al); imul(reg16, ax); imul(reg32, eax); imul(mem); imul(mem8, al); imul(mem16, ax); imul(mem32, eax); imul(imm8, al); imul(imm16, ax); imul(imm32, eax);	<p>Multiplies the accumulator (AL, AX, or EAX) by the source operand. The source operand will be the same size as the accumulator. The product produces an operand that is twice the size of the two operands with the product winding up in AX, DX:AX, or EDX:EAX. Note that the instructions involving an immediate operand are HLA extensions. HLA creates a memory object holding these constants and then multiplies the accumulator by the contents of this memory location.</p> <p>This instruction performs a signed multiplication. Also see INTMUL.</p> <p>This instruction sets the carry and overflow flag if the H.O. portion of the result (AH, DX, EDX) is not a sign extension of the L.O. portion of the product. The sign and zero flags are undefined after the execution of this instruction.</p>
in(imm8, al); in(imm8, ax); in(imm8, eax); in(dx, al); in(dx, ax); in(dx, eax);	<p>Input data from a port. These instructions read a byte, word, or double word from an input port and place the input data into the accumulator register. Immediate port constants must be in the range 0..255. For all other port addresses you must use the DX register to hold the 16-bit port number. Note that this is a privileged instruction that will raise an exception in many Win32 Operating Systems.</p>
inc(reg); inc(mem);	<p>Increment. Adds one to the specified memory or register operand. Does not affect the carry flag. Sets the overflow flag if there was signed overflow. Sets the zero and sign flags according to the result. Note that Z=1 indicates an unsigned overflow.</p>
int(imm8);	<p>Call an interrupt service routine specified by the immediate operand. Note that Windows does not use this instruction for system calls, so you will probably never use this instruction under Windows. Note that INT(3); is the user breakpoint instruction (that raises an appropriate exception). INT(0) is the divide error exception. INT(4) is the overflow exception. However, it's better to use the HLA RAISE statement than to use this instruction for these exceptions.</p>
intmul(imm, reg); intmul(imm, reg, reg); intmul(imm, mem, reg); intmul(reg, reg); intmul(mem, reg);	<p>Integer multiply. Multiplies the destination (last) operand by the source operand (if there are only two operands; or it multiplies the two source operands together and stores the result in the destination operand (if there are three operands). The operands must all be 16 or 32-bit operands and they must all be the same size.</p> <p>This instruction computes a signed product. This instruction sets the overflow and carry flags if there was a signed arithmetic overflow; the zero and sign flags are undefined after the execution of this instruction.</p>
into();	<p>Raises an exception if the overflow flag is set. Note: the HLA pseudo-variable "@into" controls the code generation for this instruction. If @into is false, HLA ignores this instruction; if @into is true (default), then HLA emits the object code for this instruction. Note that if the overflow flag is set, this instruction behaves like the "INT(4);" instruction.</p>
iret();	<p>Return from an interrupt. This instruction is not generally usable from an application program. It is for use in interrupt service routines only.</p>

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
<i>ja label;</i>	Conditional jump if (unsigned) above. You would generally use this instruction immediately after a CMP instruction to test to see if one operand is greater than another using an unsigned comparison. Control transfers to the specified label if this condition is true, control falls through to the next instruction if the condition is false.
<i>jae label;</i>	Conditional jump if (unsigned) above or equal. See JA above for details.
<i>jb label;</i>	Conditional jump if (unsigned) below. See JA above for details.
<i>jbe label;</i>	Conditional jump if (unsigned) below or equal. See JA above for details.
<i>jc label;</i>	Conditional jump if carry is one. See JA above for details.
<i>je label;</i>	Conditional jump if equal. See JA above for details.
<i>jg label;</i>	Conditional jump if (signed) greater. See JA above for details.
<i>jge label;</i>	Conditional jump if (signed) greater or equal. See JA above for details.
<i>jl label;</i>	Conditional jump if (signed) less than. See JA above for details.
<i>jle label;</i>	Conditional jump if (signed) less than or equal. See JA above for details.
<i>jna label;</i>	Conditional jump if (unsigned) not above. See JA above for details.
<i>jnae label;</i>	Conditional jump if (unsigned) not above or equal. See JA above for details.
<i>jnb label;</i>	Conditional jump if (unsigned) below. See JA above for details.
<i>jnb label;</i>	Conditional jump if (unsigned) below or equal. See JA above for details.
<i>jnc label;</i>	Conditional jump if carry flag is clear (no carry). See JA above for details.
<i>jne label;</i>	Conditional jump if not equal. See JA above for details.
<i>jng label;</i>	Conditional jump if (signed) not greater. See JA above for details.
<i>jnge label;</i>	Conditional jump if (signed) not greater or equal. See JA above for details.
<i>jnl label;</i>	Conditional jump if (signed) not less than. See JA above for details.
<i>jnle label;</i>	Conditional jump if (signed) not less than or equal. See JA above for details.
<i>jno label;</i>	Conditional jump if no overflow (overflow flag = 0). See JA above for details.
<i>jnp label;</i>	Conditional jump if no parity/parity odd (parity flag = 0). See JA above for details.
<i>jns label;</i>	Conditional jump if no sign (sign flag = 0). See JA above for details.
<i>jnz label;</i>	Conditional jump if not zero (zero flag = 0). See JA above for details.
<i>jo label;</i>	Conditional jump if overflow (overflow flag = 1). See JA above for details.
<i>jp label;</i>	Conditional jump if parity (parity flag = 1). See JA above for details.
<i>jpe label;</i>	Conditional jump if parity even (parity flag = 1). See JA above for details.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
<i>jpo label;</i>	Conditional jump if parity odd (parity flag = 0). See JA above for details.
<i>js label;</i>	Conditional jump if sign (sign flag = 0). See JA above for details.
<i>jz label;</i>	Conditional jump if zero (zero flag = 0). See JA above for details.
<i>jcxz label;</i>	Conditional jump if CX is zero. See JA above for details. Note: the range of this branch is limited to ± 128 bytes around the instruction. HLA does not check for this (MASM reports the error when it assembles HLA's output). Since this instruction is slower than comparing CX to zero and using JZ, you probably shouldn't even use this instruction. If you do, be sure that the target label is nearby in your code.
<i>jecxz label;</i>	Conditional jump if ECX is zero. See JA above for details. Note: the range of this branch is limited to ± 128 bytes around the instruction. HLA does not check for this (MASM reports the error when it assembles HLA's output). Since this instruction is slower than comparing ECX to zero and using JZ, you probably shouldn't even use this instruction. If you do, be sure that the target label is nearby in your code.
<i>jmp label;</i> <i>jmp(label);</i> <i>jmp ProcedureName;</i> <i>jmp(mem32);</i> <i>jmp(reg32);</i>	<p>Jump Instruction. This instruction unconditionally transfers control to the specified destination operand. If the operand is a 32-bit register or memory location, the JMP instruction transfers control to the instruction whose address appears in the register or the memory location.</p> <p>Note: you should exercise great care when jumping to a procedure label. The JMP instruction does not push a return address or any other data associated with a procedure's activation record. Hence, when the procedure attempts to return it will use data on the stack that was pushed prior to the execution of the JMP instruction; it is your responsibility to ensure such data is present on the stack when using JMP to transfer control to a procedure.</p>
<i>lahf();</i>	Load AH from FLAGS. This instruction loads the AH register with the L.O. eight bits of the FLAGS register. See SAHF for the flag layout.
<i>lea(reg32, mem);</i> <i>lea(mem, reg32);</i>	Load Effective Address. These instructions, which are both semantically identical, load the 32-bit register with the address of the specified memory location. The memory location does not need to be a double word object. Note that there is never any ambiguity in this instruction since the register is always the destination operand and the memory location is always the source.
<i>leave();</i>	Leave procedure. This instruction cleans up the activation record for a procedure prior to returning from the procedure. You would normally use this instruction to clean up the activation record created by the ENTER instruction.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
lock prefix	<p>The lock prefix asserts a special pin on the processor during the execution of the following instruction. In a multiprocessor environment, this ensures that the processor has exclusive use of a shared memory object while the instruction executes. The lock prefix may only precede one of the following instructions: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. Furthermore, this prefix is only valid for the forms that have a memory operand as their destination operand. Any other instruction or addressing mode will raise an undefined opcode exception.</p> <p>HLA does not directly support the LOCK prefix on these instructions (if it did, you would normally write instructions like “lock.add(;)” and “lock.bts(;)” However, you can easily add this instruction to HLA’s instruction set through the use of the following macro:</p> <pre data-bbox="537 730 1214 814">#macro lock; byte \$F0; // \$F0 is the opcode for the lock prefix. #endmacro;</pre> <p>To use this macro, simply precede the instruction you wish to lock with an invocation of the macro, e.g.,</p> <pre data-bbox="537 957 805 982">lock add(al, mem);</pre> <p>Note that a LOCK prefix will dramatically slow an instruction down since it must access main memory (i.e., no cache) and it must negotiate for the use of that memory location with other processors in a multiprocessor system. The LOCK prefix has very little value in single processor systems.</p>
lods b();	Load String Byte. Loads AL with the byte whose address appears in ESI. Then it increments or decrements ESI by one depending on the value of the direction flag. See the chapter on string instructions for more details. Note: HLA does not allow the use of any repeat prefix with this instruction.
lodsw();	Load String Word. Loads AX from [ESI] and adds ± 2 to ESI. See LODSB for more details.
loadsd();	Load String Double Word. Loads EAX from [ESI] and adds ± 4 to ESI. See LODSB for more details.
loop <i>label</i> ;	Decrements ECX and jumps to the target label if ECX is not zero. See JA for more details. Like JECX, this instruction is limited to a range of ± 128 bytes around the instruction and only MASM will catch the range error. Since this instruction is actually slower than a DEC/JNZ pair, you should probably avoid using this instruction.
loope <i>label</i> ;	Check the zero flag, decrement ECX, and branch if the zero flag was set and ECX did not become zero. Same limitations as LOOP. See LOOP above for details.
loopne <i>label</i> ;	Check the zero flag, decrement ECX, and branch if the zero flag was clear and ECX did not become zero. Same limitations as LOOP. See LOOP above for details.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
loopnz <i>label</i> ;	Same instruction as LOOPNE.
loopz <i>label</i> ;	Same instruction as LOOPZ.
mov(imm, reg); mov(imm, mem); mov(reg, reg); mov(reg, mem); mov(mem, reg); mov(mem16, mem16); mov(mem32, mem32);	Move. Copies the data from the source (first) operand to the destination (second) operand. The operands must be the same size. Note that the memory to memory moves are an HLA extension. HLA compiles these statements into a push(source)/pop(dest) instruction pair.
movsb(); rep.movsb();	Move string of bytes. Copies the byte pointed at by ESI to the byte pointed at by EDI and then adjusts ESI and EDI by ± 1 depending on the value of the direction flag. With the REP (repeat) prefix, this instruction moves ECX bytes. See the chapter on the String Instructions for more details.
movsw(); rep.movsw();	Move string of words. Like MOVSB above except it copies words and adjusts ESI/EDI by ± 2 .
movsd(); rep.movsd();	Move string of double words. Like MOVSB above except it copies double words and adjusts ESI/EDI by ± 4 .
movsx(reg, reg); movsx(mem, reg);	Move with sign extension. Copies the (smaller) source operand to the (larger) destination operand and sign extends the value to the size of the larger operand. The source operand must be smaller than the destination operand.
movzx(reg, reg); movzx(mem, reg);	Move with zero extension. Copies the (smaller) source operand to the (larger) destination operand and zero extends the value to the size of the larger operand. The source operand must be smaller than the destination operand.
mul(reg); mul(reg8, al); mul(reg16, ax); mul(reg32, eax); mul(mem); mul(mem8, al); mul(mem16, ax); mul(mem32, eax); mul(imm8, al); mul(imm16, ax); mul(imm32, eax);	<p>Multiplies the accumulator (AL, AX, or EAX) by the source operand. The source operand will be the same size as the accumulator. The product produces an operand that is twice the size of the two operands with the product winding up in AX, DX:AX, or EDX:EAX. Note that the instructions involving an immediate operand are HLA extensions. HLA creates a memory object holding these constants and then multiplies the accumulator by the contents of this memory location.</p> <p>This instruction performs a signed multiplication. Also see INTMUL. The carry and overflow flags are cleared if the H.O. portion of the result is zero, they are set otherwise. The sign and zero flags are undefined after this instruction.</p>

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
neg(reg); neg(mem);	Negate. Computes the two's complement of the operand and leaves the result in the operand. This instruction clears the carry flag if the result is zero, it sets the carry flag otherwise. It sets the overflow flag if the original value was the smallest possible negative value (which has no positive counterpart). It sets the sign and zero flags according to the result obtained.
nop();	No Operation. Consumes space and time but does nothing else. Same instruction as "xchg(eax, eax);"
not(reg); not(mem);	Bitwise NOT. Inverts all the bits in its operand. Note: this instruction does not affect any flags.
or(imm, reg); or(imm, mem); or(reg, reg); or(reg, mem); or(mem, reg);	Bitwise OR. Logically ORs the source operand with the destination operand and leaves the result in the destination. The two operands must be the same size. Clears the carry and overflow flags and sets the sign and zero flags according to the result.
out(al, imm8); out(ax, imm8); out(eax, imm8); out(al, dx); out(ax, dx); out(eax, dx);	Outputs the accumulator to the specified port. See the IN instruction for limitations under Win32.
pop(reg); pop(mem);	Pop a value off the stack. Operands must be 16 or 32 bits.
popa();	Pops all the 16-bit registers off the stack. The popping order is DI, SI, BP, SP, BX, DX, CX, AX.
popad();	Pops all the 32-bit registers off the stack. The popping order is EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX.
popf();	Pops the 16-bit FLAGS register off the stack. Note that in user (application) mode, this instruction ignores the interrupt disable flag value it pops off the stack.
popfd();	Pops the 32-bit EFLAGS register off the stack. Note that in user (application) mode, this instruction ignores many of the bits it pops off the stack.
push(reg); push(mem);	Pushes the specified 16-bit or 32-bit register or memory location onto the stack. Note that you cannot push eight-bit objects.
pusha();	Pushes all the 16-bit general purpose registers onto the stack in the order AX, CX, DX, BX, SP, BP, SI, DI.
pushad();	Pushes all the 32-bit general purpose registers onto the stack in the order EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
pushd(imm); pushd(reg); pushd(mem);	Pushes the 32-bit operand on to the stack. Generally used to push constants or anonymous variables. Note that this is a synonym for PUSH if you specify a register or typed memory operand.
pushf();	Pushes the value of the 16-bit FLAGS register onto the stack.
pushfd();	Pushes the value of the 32-bit FLAGS register onto the stack.
pushw(imm); pushw(reg); pushw(mem);	Pushes the 16-bit operand on to the stack. Generally used to push constants or anonymous variables. Note that this is a synonym for PUSH if you specify a register or typed memory operand.
rcl(imm, reg); rcl(imm, mem); rcl(cl, reg); rcl(cl, mem);	Rotate through carry, left. Rotates the destination (second) operand through the carry the number of bits specified by the first operand, shifting the bits from the L.O. to the H.O. position (i.e., rotate left). The carry flag contains the last bit shifted into it. The overflow flag, which is valid only when the shift count is one, is set if the sign changes as a result of the rotate. This instruction does not affect the other flags. In particular, note that this instruction does not affect the sign or zero flags.
rcr(imm, reg); rcr(imm, mem); rcr(cl, reg); rcr(cl, mem);	Rotate through carry, right. Rotates the destination (second) operand through the carry the number of bits specified by the first operand, shifting the bits from the H.O. to the L.O. position (i.e., rotate right). The carry flag contains the last bit shifted into it. The overflow flag, which is valid only when the shift count is one, is set if the sign changes as a result of the rotate. This instruction does not affect the other flags. In particular, note that this instruction does not affect the sign or zero flags.
rdtsc();	Read Time Stamp Counter. Returns in EDX:EAX the number of clock cycles that have transpired since the last reset of the processor. You can use this instruction to time events in your code (i.e., to determine whether one instruction sequence takes more time than another).
ret(); ret(imm16);	Return from subroutine. Pops a return address off the stack and transfers control to that location. The second form of the instruction adds the immediate constant to the ESP register to remove the procedure's parameters from the stack.
rol(imm, reg); rol(imm, mem); rol(cl, reg); rol(cl, mem);	Rotate left. Rotates the destination (second) operand the number of bits specified by the first operand, shifting the bits from the L.O. to the H.O. position (i.e., rotate left). The carry flag contains the last bit shifted into it. The overflow flag, which is valid only when the shift count is one, is set if the sign changes as a result of the rotate. This instruction does not affect the other flags. In particular, note that this instruction does not affect the sign or zero flags.
ror(imm, reg); ror(imm, mem); ror(cl, reg); ror(cl, mem);	Rotate right. Rotates the destination (second) operand the number of bits specified by the first operand, shifting the bits from the H.O. to the L.O. position (i.e., rotate right). The carry flag contains the last bit shifted into it. The overflow flag, which is valid only when the shift count is one, is set if the sign changes as a result of the rotate. This instruction does not affect the other flags. In particular, note that this instruction does not affect the sign or zero flags.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
sahf();	Store AH into FLAGS. Copies the value in AH into the L.O. eight bits of the FLAGS register. Note that this instruction will not affect the interrupt disable flag when operating in user (application) mode. Bit #7 of AH goes into the Sign flag, bit #6 goes into the zero flag, bit #4 goes into the auxiliary carry (BCD carry) flag, bit #2 goes into the parity flag, and bit #0 goes into the carry flag. This instruction also clears bits one, three, and five of the FLAGS register. It does not affect any other bits in FLAGS or EFLAGS.
sal(imm, reg); sal(imm, mem); sal(cl, reg); sal(cl, mem);	Shift Arithmetic Left. Same instruction as SHL. See SHL for details.
sar(imm, reg); sar(imm, mem); sar(cl, reg); sar(cl, mem);	Shift Arithmetic Right. Shifts the destination (second) operand to the right the specified number of bits using an arithmetic shift right algorithm. The carry flag contains the value of the last bit shifted out of the second operand. The overflow flag is only defined when the bit shift count is one, this instruction always clears the overflow flag. The sign and zero flags are set according to the result.
sbb(imm, reg); sbb(imm, mem); sbb(reg, reg); sbb(reg, mem); sbb(mem, reg);	Subtract with borrow. Subtracts the source (first) operand and the carry from the destination (second) operand. Sets the condition code bits according to the result it computes. This instruction sets the flags the same way as the SUB instruction. See SUB for details.
scasb(); repe.scasb(); repne.scasb();	Scan string byte. Compares the value in AL against the byte that EDI points at and sets the flags accordingly (same as the CMP instruction). Adds ± 1 to EDI after the comparison (based on the setting of the direction flag). With the REPE (repeat while equal) prefix, this instruction will scan through as many as ECX bytes in memory as long as each byte that EDI points at is equal to the value in AL (i.e., it scans for the first value not equal to the value in AL). With the REPNE prefix, this instruction scans through as many as ECX bytes as long as the value that EDI points at is not equal to AL (i.e., it scans for the first byte matching AL's value). See the chapter on string instructions for more details.
scasw(); repe.scasw(); repne.scasw();	Scan String Word. Compares the value in AX against the word that EDI points at and set the flags. Adds ± 2 to EDI after the operation. Also supports the REPE and REPNE prefixes (see SCASB above).
scasd(); repe.scasd(); repne.scasd();	Scan String Double word. Compares the value in EAX against the double word that EDI points at and set the flags. Adds ± 4 to EDI after the operation. Also supports the REPE and REPNE prefixes (see SCASB above).
seta(reg); seta(mem);	Conditional set if (unsigned) above (Carry=0 and Zero=0). Stores a one in the destination operand if the result of the previous comparison found the first operand to be greater than the second using an unsigned comparison. Stores a zero into the destination operand otherwise.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
setae(reg); setae(mem);	Conditional set if (unsigned) above or equal (Carry=0). See SETA for details.
setb(reg); setb(mem);	Conditional set if (unsigned) below (Carry=1). See SETA for details.
setbe(reg); setbe(mem);	Conditional set if (unsigned) below or equal (Carry=1 or Zero=1). See SETA for details.
setc(reg); setc(mem);	Conditional set if carry set (Carry=1). See SETA for details.
sete(reg); sete(mem);	Conditional set if equal (Zero=1). See SETA for details.
setg(reg); setg(mem);	Conditional set if (signed) greater (Sign=Overflow and Zero=0). See SETA for details.
setge(reg); setge(mem);	Conditional set if (signed) greater or equal (Sign=Overflow or Zero=1). See SETA for details.
setl(reg); setl(mem);	Conditional set if (signed) less than (Sign<>Overflow). See SETA for details.
setle(reg); setle(mem);	Conditional set if (signed) less than or equal (Sign<>Overflow or Zero = 1). See SETA for details.
setna(reg); setna(mem);	Conditional set if (unsigned) not above (Carry=1 or Zero=1). See SETA for details.
setnae(reg); setnae(mem);	Conditional set if (unsigned) not above or equal (Carry=1). See SETA for details.
setnb(reg); setnb(mem);	Conditional set if (unsigned) not below (Carry=0). See SETA for details.
setnbe(reg); setnbe(mem);	Conditional set if (unsigned) not below or equal (Carry=0 and Zero=0). See SETA for details.
setnc(reg); setnc(mem);	Conditional set if carry clear (Carry=0). See SETA for details.
setne(reg); setne(mem);	Conditional set if not equal (Zero=0). See SETA for details.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
setng(reg); setng(mem);	Conditional set if (signed) not greater (Sign<>Overflow or Zero = 1). See SETA for details.
setnge(reg); setnge(mem);	Conditional set if (signed) not greater than (Sign<>Overflow). See SETA for details.
setnl(reg); setnl(mem);	Conditional set if (signed) not less than (Sign=Overflow or Zero=1). See SETA for details.
setnle(reg); setnle(mem);	Conditional set if (signed) not less than or equal (Sign=Overflow and Zero=0). See SETA for details.
setno(reg); setno(mem);	Conditional set if no overflow (Overflow=0). See SETA for details.
setnp(reg); setnp(mem);	Conditional set if no parity (Parity=0). See SETA for details.
setns(reg); setns(mem);	Conditional set if no sign (Sign=0). See SETA for details.
setnz(reg); setnz(mem);	Conditional set if not zero (Zero=0). See SETA for details.
seto(reg); seto(mem);	Conditional set if Overflow (Overflow=1). See SETA for details.
setp(reg); setp(mem);	Conditional set if Parity (Parity=1). See SETA for details.
setpe(reg); setpe(mem);	Conditional set if Parity even (Parity=1). See SETA for details.
setpo(reg); setpo(mem);	Conditional set if Parity odd (Parity=0). See SETA for details.
sets(reg); sets(mem);	Conditional set if sign set(Sign=1). See SETA for details.
setz(reg); setz(mem);	Conditional set if zero (Zero=1). See SETA for details.
shl(imm, reg); shl(imm, mem); shl(cl, reg); shl(cl, mem);	Shift left. Shifts the destination (second) operand to the left the number of bit positions specified by the first operand. The carry flag contains the value of the last bit shifted out of the second operand. The overflow flag is only defined when the bit shift count is one, this instruction sets overflow flag if the sign changes as a result of this instruction's execution. The sign and zero flags are set according to the result.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
shld(imm8, reg, reg); shld(imm8, reg, mem); shld(cl, reg, reg); shld(cl, reg, mem);	Shift Left Double precision. The first operand is a bit count. The second operand is a source and the third operand is a destination. These operands must be the same size and they must be 16- or 32-bit values (no eight bit operands). This instruction treats the second and third operands as a double precision value with the second operand being the L.O. word or double word and the third operand being the H.O. word or double word. The instruction shifts this double precision value the specified number of bits and sets the flags in a manner identical to SHL. Note that this instruction does not affect the source (second) operand's value.
shr(imm, reg); shr(imm, mem); shr(cl, reg); shr(cl, mem);	Shift right. Shifts the destination (second) operand to the right the number of bit positions specified by the first operand. The last bit shifted out goes into the carry flag. The overflow flag is set if the H.O. bit originally contained one. The sign flag is cleared and the zero flag is set if the result is zero.
shrd(imm8, reg, reg); shrd(imm8, reg, mem); shrd(cl, reg, reg); shrd(cl, reg, mem);	Shift Right Double precision. The first operand is a bit count. The second operand is a source and the third operand is a destination. These operands must be the same size and they must be 16- or 32-bit values (no eight bit operands). This instruction treats the second and third operands as a double precision value with the second operand being the H.O. word or double word and the third operand being the L.O. word or double word. The instruction shifts this double precision value the specified number of bits and sets the flags in a manner identical to SHR. Note that this instruction does not affect the source (second) operand's value.
stc();	Set Carry. Sets the carry flag to one.
std();	Set Direction. Sets the direction flag to one. If the direction flag is one, the string instructions decrement ESI and/or EDI after each operation.
sti();	Set interrupt enable flag. Generally this instruction is not usable in user (application) mode. In kernel mode it allows the CPU to begin processing interrupts.
stosb(); rep.stosb();	Store String Byte. Stores the value in AL at the location whose address EDI contains. Then it adds ± 1 to EDI. If the REP prefix is present, this instruction repeats the number of times specified in the ECX register. This instruction is useful for quickly clearing out byte arrays.
stosw(); rep.stosw();	Store String Word. Stores the value in AX at location [EDI] and then adds ± 2 to EDI. See STOSB for details.
stosd(); rep.stosd();	Store String Double word. Stores the value in EAX at location [EDI] and then adds ± 4 to EDI. See STOSB for details.
sub(imm, reg); sub(imm, mem); sub(reg, reg); sub(reg, mem); sub(mem, reg);	Subtract. Subtracts the first operand from the second operand and leaves the difference in the destination (second) operand. Sets the zero flag if the two values were equal (which produces a zero result), sets the carry flag if there was unsigned overflow or underflow; sets the overflow if there was signed overflow or underflow; sets the sign flag if the result is negative (H.O. bit is one). Note that SUB sets the flags identically to the CMP instruction, so you can use conditional jump or set instructions after SUB the same way you would use them after a CMP instruction.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
test(imm, reg); test(imm, mem); test(reg, reg); test(reg, mem); test(mem, reg);	Test operands. Logically ANDs the two operands together and sets the flags but does not store the computed result (i.e., it does not disturb the value in either operand). Always clears the carry and overflow flags. Sets the sign flag if the H.O. bit of the computed result is one. Sets the zero flag if the computed result is zero.
xadd(mem, reg); xadd(reg, reg);	Adds the first operand to the second operand and then stores the original value of the second operand into the first operand: <pre>xadd(source, dest); temp := dest dest := dest + source source := temp</pre> <p>This instruction sets the flags in a manner identical to the ADD instruction.</p>
xchg(reg, reg); xchg(reg, mem); xchg(mem, reg);	Swaps the values in the two operands which must be the same size. Does not affect any flags.
xlat();	Translate. Computes AL := [EBX + AL]; That is, it uses the value in AL as an index into a lookup table whose base address is in EBX. It copies the specified byte from this table into AL.
xor(imm, reg); xor(imm, mem); xor(reg, reg); xor(reg, mem); xor(mem, reg);	Exclusive-OR. Logically XORs the source operand with the destination operand and leaves the result in the destination. The two operands must be the same size. Clears the carry and overflow flags and sets the sign and zero flags according to the result.

Table 2: Floating Point Instruction Set

Instruction	Description
f2xm1();	Compute $2^x - 1$ in ST0, leaving the result in ST0.
fabs();	Computes the absolute value of ST0.
fadd(mem); fadd(sti, st0); fadd(st0, sti);	Add operand to st0 or add st0 to destination register (sti, i=0..7). If the operand is a memory operand, it must be a <i>real32</i> or <i>real64</i> object.
faddp(); faddp(st0, sti);	With no operands, this instruction adds st0 to st1 and then pops st0 off the FPU stack.

Table 2: Floating Point Instruction Set

Instruction	Description
fbld(mem80);	This instruction loads a ten-byte (80-bit) packed BCD value from memory and converts it to a real80 object. This instruction does not check for an invalid BCD value. If the BCD number contains illegal digits, the result is undefined.
fbstp(mem80);	This instruction pops the real80 object off the top of the FPU stack, converts it to an 80-bit BCD value, and stores the result in the specified memory location (tbyte).
fchs();	This instruction negates the floating point value on the top of the stack (st0).
fclex();	This instruction clears the floating point exception flags.
fcmova(sti, st0); ^a	Floating point conditional move if above. Copies sti to st0 if c=0 and z=0 (unsigned greater than after a CMP).
fcmovae(sti, st0);	Floating point conditional move if above or equal. Copies sti to st0 if c=0 (unsigned greater or equal after a CMP).
fcmovb(sti, st0);	Floating point conditional move if below. Copies sti to st0 if c=1 (unsigned less than after a CMP).
fcmovbe(sti, st0);	Floating point conditional move if below or equal. Copies sti to st0 if c=1 or z=1 (unsigned less than or equal after a CMP).
fcmove(sti, st0);	Floating point conditional move if equal. Copies sti to st0 if z=1 (equal after a CMP).
fcmovna(sti, st0);	Floating point conditional move if not above. Copies sti to st0 if c=1 or z=1 (unsigned not above after a CMP).
fcmovnae(sti, st0);	Floating point conditional move if not above or equal. Copies sti to st0 if c=1 (unsigned not above or equal after a CMP).
fcmovnb(sti, st0);	Floating point conditional move if not below. Copies sti to st0 if c=0 (unsigned not below after a CMP).
fcmovnbe(sti, st0);	Floating point conditional move if not below or equal. Copies sti to st0 if c=0 and z=0 (unsigned not below or equal after a CMP).
fcmovne(sti, st0);	Floating point conditional move if not equal. Copies sti to st0 if z=0 (not equal after a CMP).
fcmovnu(sti, st0);	Floating point conditional move if not unordered. Copies sti to st0 if the last floating point comparison did not produce an unordered result (parity flag = 0).
fcmovu(sti, st0);	Floating point conditional move if not unordered. Copies sti to st0 if the last floating point comparison produced an unordered result (parity flag = 1).
fcom(); fcom(mem); fcom(st0, sti);	Compares the value of ST0 with the operand and sets the floating point condition bits based on the comparison. If the operand is a memory operand, it must be a <i>real32</i> or <i>real64</i> value. Note that to test the condition codes you will have to copy the floating point status word to the FLAGS register; see the chapter on floating point arithmetic for details.

Table 2: Floating Point Instruction Set

Instruction	Description
fcomi(st0, sti); ^b	Compares the value of ST0 with the second operand and sets the appropriate bits in the FLAGS register.
fcomip(st0, sti);	Compares the value of ST0 with the second operand, sets the appropriate bits in the FLAGS register, and then pops ST0 off the FPU stack.
fcomp(); fcomp(mem); fcomp(sti);	Compares the value of ST0 with the operand, sets the floating point status bits, and then pops ST0 off the floating point stack. With no operands, this instruction compares ST0 to ST1. Memory operands must be <i>real32</i> or <i>real64</i> objects.
fcompp();	Compares ST0 to ST1 and then pops both values off the stack. Leaves the result of the comparison in the floating point status register.
fcos();	Computes $ST0 = \cos(ST0)$.
fdecstp();	Rotates the items on the FPU stack.
fdiv(mem); fdiv(sti, st0); fdiv(st0, sti);	Floating point division. If a memory operand is present, it must be a <i>real32</i> or <i>real64</i> object; FDIV will divide ST0 by the memory operand and leave the quotient in ST0. If the FDIV operands are registers, FDIV divides the destination (second) operand by the source (first) operand and leaves the result in the destination operand.
fdivp(); fdivp(sti);	With no operands, this instruction divides ST1 by ST0, pops ST0, and replaces the new top of stack with the quotient (replacing the previous ST1 value).
fdivr(mem); fdivr(sti, st0); fdivr(st0, sti);	Floating point divide with reversed operands. Like FDIV, but computes operand/ST0 rather than ST0/operand.
fdivrp(); fdivrp(sti);	Floating point divide and pop, reversed. Like FDIVP except it computes operand/ST0 rather than ST0/operand.
ffree(sti);	Frees the specified floating point register.
fiadd(mem);	Memory operand must be a 16-bit or 32-bit signed integer. This instruction converts the integer to a real, pushes the value, and then executes FADDP();
ficom(mem);	Floating point compare to integer. Memory operand must be an <i>int16</i> or <i>int32</i> object. This instruction converts the memory operand to a <i>real80</i> value and compares ST0 to this value and sets the status bits in the floating point status register.
ficom(mem);	Floating point compare to integer and pop. Memory operand must be an <i>int16</i> or <i>int32</i> object. This instruction converts the memory operand to a <i>real80</i> value and compares ST0 to this value and sets the status bits in the floating point status register. After the comparison, this instructions pop ST0 from the FPU stack.
fidiv(mem);	Floating point divide by integer. Memory operand must be an <i>int16</i> or <i>int32</i> object. These instructions convert their integer operands to a <i>real80</i> value and then divide ST0 by this value, leaving the result in ST0.

Table 2: Floating Point Instruction Set

Instruction	Description
<code>fidivr(mem);</code>	Floating point divide by integer, reversed. Like FIDIV above, except this instruction computes $\text{mem}/\text{ST0}$ rather than $\text{ST0}/\text{mem}$.
<code>fld(mem);</code>	Floating point load integer. Mem operand must be an <code>int16</code> or <code>int32</code> object. This instructions converts the integer to a <code>real80</code> object and pushes it onto the FPU stack.
<code>fmul(mem);</code>	Floating point multiply by integer. Converts <code>int16</code> or <code>int32</code> operand to a <code>real80</code> value and multiplies <code>ST0</code> by this result. Leaves product in <code>ST0</code> .
<code>fincstp();</code>	Rotates the registers on the FPU stack.
<code>finit();</code>	Initializes the FPU for use.
<code>fist(mem);</code>	Converts <code>ST0</code> to an integer and stores the result in the specified memory operand. Memory operand must be an <code>int16</code> or <code>int32</code> object.
<code>fistp(mem);</code>	Floating point integer store and pop. Pops <code>ST0</code> value off the stack, converts it to an integer, and stores the integer in the specified location. Memory operand must be a word, double word, or quad word (64-bit integer) object.
<code>fisub(mem);</code>	Floating point subtract integer. Converts <code>int16</code> or <code>int32</code> operand to a <code>real80</code> value and subtracts it from <code>ST0</code> . Leaves the result in <code>ST0</code> .
<code>fisubr(mem);</code>	Floating point subtract integer, reversed. Like FISUB except this instruction compute $\text{mem}-\text{ST0}$ rather than $\text{ST0}-\text{mem}$. Still leaves the result in <code>ST0</code> .
<code>fld(mem);</code> <code>fld(sti);</code>	Floating point load. Loads (pushes) the specified operand onto the FPU stack. Memory operands must be <code>real32</code> , <code>real64</code> , or <code>real80</code> objects. Note that <code>FLD(ST0)</code> duplicates the value on the top of the floating point stack.
<code>fld1();</code>	Floating point load 1.0. This instruction pushes 1.0 onto the FPU stack.
<code>fldcw(mem16);</code>	Load floating point control word. This instruction copies the word operand into the floating point control register.
<code>fldenv(mem28);</code>	This instruction loads the FPU status from the block of 28 bytes specified by the operand. Generally, only an operating system would use this instruction.
<code>fldl2e();</code>	Floating point load constant. Loads $\log_2(e)$ onto the stack.
<code>fldl2t();</code>	Floating point load constant. Loads $\log_2(10)$ onto the stack.
<code>fldlg2();</code>	Floating point load constant. Loads $\log_{10}(2)$ onto the stack.
<code>fldln2();</code>	Floating point load constant. Loads $\log_e(2)$ onto the stack.
<code>fldpi();</code>	Floating point load constant. Loads the value of pi (π) onto the stack.
<code>fldz();</code>	Floating point load constant. Pushes the value 0.0 onto the stack.

Table 2: Floating Point Instruction Set

Instruction	Description
fmul(mem); fmul(sti, st0); fmul(st0, sti);	Floating point multiply. If the operand is a memory operand, it must be a real32 or real64 value; in this case, FMUL multiplies the memory operand and ST0, leaving the product in ST0. For the other two forms, the FMUL instruction multiplies the first operand by the second and leaves the result in the second operand.
fmulp(); fmulp(st0, sti);	Floating point multiply and pop. With no operands this instruction computes $ST1:=ST0*ST1$ and then pops ST0. With two register operands, this instruction computes ST0 times the destination register and then pops ST0.
fnop();	Floating point no-operation.
fpatan();	Floating point partial arctangent. Computes $ATAN(ST1/ST0)$, pops ST0, and then stores the result in the new TOS value (previous ST1 value).
fprem();	Floating point remainder. This instruction is retained for compatibility with older programs. Use the FPREM1 instruction instead.
fprem1();	Floating point partial remainder. This instruction computes the remainder obtained by dividing ST0 by ST1, leaving the result in ST0 (it does not pop either operand). If the C2 flag in the FPU status register is set after this instruction, then the computation is not complete; you must repeatedly execute this instruction until C2 is cleared.
fptan();	Floating point partial tangent. This instruction computes $TAN(ST0)$ and replaces the value in ST0 with this result. Then it pushes 1.0 onto the stack. This instruction sets the C2 flag if the input value is outside the acceptable range of $\pm 2^{63}$.
frndint();	Floating point round to integer. This instruction rounds the value in ST0 to an integer using the rounding control bits in the floating point control register. Note that the result left on TOS is still a real value. It simply doesn't have a fractional component. You may use this instruction to round or truncate a floating point value by setting the rounding control bits appropriately. See the chapter on floating point arithmetic for details.
frstor(mem108);	Restores the FPU status from a 108-byte memory block.
fsave(mem108);	Writes the FPU status to a 108-byte memory block.
fscale();	Floating point scale by power of two. ST1 contains a scaling value. This instruction multiplies ST0 by 2^{st1} .
fsin();	Floating point sine. Replaces ST0 with $\sin(ST0)$.
fsincos();	Simultaneously computes the sin and cosine values of ST0. Replaces ST0 with the sine of ST0 and then it pushes the cosine of (the original value of) ST0 onto the stack. Original ST0 value must be in the range $\pm 2^{63}$.
fsqrt();	Floating point square root. Replaces ST0 with the square root of ST0.

Table 2: Floating Point Instruction Set

Instruction	Description
fst(mem); fst(sti);	Floating point store. Stores a copy of ST0 in the destination operand. Memory operands must be <i>real32</i> or <i>real64</i> objects. When storing the value to memory, FST converts the value to the smaller format using the rounding control bits in the floating point control register to determine how to convert the <i>real80</i> value in ST0 to a <i>real32</i> or <i>real64</i> value.
fstcw(mem16);	Floating point store control word. Stores a copy of the floating point control word in the specified <i>word</i> memory location.
fstenv(mem28);	Floating point store FPU environment. Stores a copy of the 28-byte floating point environment in the specified memory location. Normally, an OS would use this when switch contexts.
fstp(mem); fstp(sti);	Floating point store and pop. Stores ST0 into the destination operand and then pops ST0 off the stack. If the operand is a memory object, it must be a <i>real32</i> , <i>real64</i> , or <i>real80</i> object.
fstsw(ax); fstsw(mem16);	Stores a copy of the 16-bit floating point status register into the specified word operand. Note that this instruction automatically places the C1, C2, C3, and C4 condition bits in appropriate places in AH so that a following SAHF instruction will set the processor flags to allow the use of a conditional jump or conditional set instruction after a floating point comparison. See the chapter on floating point arithmetic for more details.
fsub(mem); fsub(st0, sti); fsub(sti, st0);	Floating point subtract. With a single memory operand (which must be a <i>real32</i> or <i>real64</i> object), this instruction subtracts the memory operand from ST0. With two register operands, this instruction computes $dest := dest - src$ (where <i>src</i> is the first operand and <i>dest</i> is the second operand).
fsubp(); fsubp(st0, sti);	Floating point subtract and pop. With no operands, this instruction computes $ST1 := ST0 - ST1$ and then pops ST0 off the stack. With two operands, this instruction computes $STi := STi - ST0$ and then pops ST0 off the stack.
fsubr(mem); fsubr(st0, sti); fsubr(sti, st0);	Floating point subtract, reversed. With a <i>real32</i> or <i>real64</i> memory operand, this instruction computes $ST0 := mem - ST0$. For the other two forms, this instruction computes $dest := src - dest$ where <i>src</i> is the first operand and <i>dest</i> is the second operand.
fsubrp(); fsubrp(st0, sti);	Floating point subtract and pop, reversed. With no operands, this instruction computes $ST1 := ST0 - ST1$ and then pops ST0. With two operands, this instruction computes $STi := ST0 - STi$ and then pops ST0 from the stack.
ftst();	Floating point test against zero. Compares ST0 with 0.0 and sets the floating point condition code bits accordingly.

Table 2: Floating Point Instruction Set

Instruction	Description
fucom(<i>sti</i>); fucom();	Floating point unordered comparison. With no operand, this instruction compares ST0 to ST1. With an operand, this instruction compares ST0 to STi and sets floating point status bits accordingly. Unlike FCOM, this instruction will not generate an exception if either of the operands is an illegal floating point value; instead, this sets a special status value in the FPU status register.
fucomi(<i>sti</i> , <i>st0</i>);	Floating point unordered comparison.
fucomp(); fucomp(<i>sti</i>);	Floating point unordered comparison and pop. With no operands, compares ST0 to ST1 using an unordered comparison (see FUCOM) and then pops ST0 off the stack. With an FPU operand, this instruction compares ST0 to the specified register and then pops ST0 off the stack. See FUCOM for more details.
fucompp(); fucompp(<i>sti</i>);	Floating point unordered compare and double pop. Compares ST0 to ST1, sets the condition code bits (without raising an exception for illegal values, see FUCOM), and then pops both ST0 and ST1.
fwait();	Floating point wait. Waits for current FPU operation to complete. Generally an obsolete instruction. Used back in the days when the FPU was on a different chip than the CPU.
fxam();	Floating point Examine ST0. Checks the value in ST0 and sets the condition code bits according to the type of the value in ST0. See the chapter on floating point arithmetic for details.
fxch(); fxch(<i>sti</i>);	Floating point exchange. With no operands this instruction exchanges ST0 and ST1 on the FPU stack. With a single register operand, this instruction swaps ST0 and STi.
fextract();	Floating point exponent/mantissa extraction. This instruction breaks the value in ST0 into two pieces. It replaces ST0 with the real representation of the binary exponent (e.g. 2 ⁵ becomes 5.0) and then it pushes the mantissa of the value with an exponent of zero.
fyl2x();	Floating point partial logarithm computation. Computes ST1 := ST1 * log ₂ (ST0); and then pops ST0.
fyl2xp1();	Floating point partial logarithm computation. Computes ST1 := ST1 * log ₂ (ST0 + 1.0) and then pops ST0 off the stack. Original ST0 value must be in the range $\left(-\left(1 - \frac{\sqrt{2}}{2}\right), \left(1 - \frac{\sqrt{2}}{2}\right)\right)$

a. Floating point conditional move instructions are only available on Pentium Pro and later processors.

b. FCOMIx instructions are only available on Pentium Pro and later processors.

The following table uses these abbreviations:

Reg32- A 32-bit general purpose (integer) register.

mmi- One of the eight MMX registers, MM0..MM7.

imm8- An eight-bit constant value; some instructions have smaller ranges than 0..255. See the particular instruction for details.

mem64- A memory location (using an arbitrary addressing mode) that references a qword value.

Note: Most instructions have two operands. Typically the first operand is a source operand and the second operand is a destination operand. For exceptions, see the description of the instruction.

Table 3: MMX Instruction Set

Instruction	Description
emms();	Empty MMX State. You must execute this instruction when you are finished using MMX instructions and before any following floating point instructions.
movd(reg32, mmi); movd(mem32, mmi); movd(mmi, reg32); movd(mmi, mem32);	Moves data between a 32-bit integer register or dword memory location and an MMX register (mm0..mm7). If the destination operand is an MMX register, then the source operand is zero-extended to 64 bits during the transfer. If the destination operand is a dword memory location or 32-bit register, this instruction copies only the L.O. 32 bits of the MMX register to the destination.
movq(mem64, mmi); movq(mmi, mem64); movq(mmi, mmi);	This instruction moves 64 bits between an MMX register and a <i>qword</i> variable in memory or between two MMX registers.
packssdw(mem64, mmi); packssdw(mmi, mmi);	Pack and saturate two signed double words from source and two signed double words from destination and store result into destination MMX register. This process involves taking these four double words and “saturating” them. This means that if the value is in the range -32768..32768 the value is left unchanged, but if it’s greater than 32767 the value is set to 32767 or if it’s less than -32768 the value is clipped to -32768. The four double words are packed into a single 64-bit MMX register. The source operand supplies the upper two words and the destination operand supplies the lower two words of the packed 64-bit result. See the chapter on the MMX instructions for more details.
packsswb(mem64, mmi); packsswb(mmi, mmi);	Pack and saturate four signed words from source and four signed words from destination and store the result as eight signed bytes into the destination MMX register. See the chapter on the MMX instructions for more details. The bytes obtained from the destination register wind up in the L.O. four bytes of the destination; the bytes computed from the signed saturation of the source register wind up in the H.O. four bytes of the destination register.

Table 3: MMX Instruction Set

Instruction	Description
<p>packusdw(mem64, mmi); packusdw(mmi, mmi);</p>	<p>Pack and saturate two unsigned double words from source and two unsigned double words from destination and store result into destination MMX register. This process involves taking these four double words and “saturating” them. This means that if the value is in the range 0..65535 the value is left unchanged, but if it’s greater than 65535 the value is clipped to 65535. The four double words are packed into a single 64-bit MMX register. The source operand supplies the upper two words and the destination operand supplies the lower two words of the packed 64-bit result. See the chapter on the MMX instructions for more details.</p>
<p>packuswb(mem64, mmi); packuswb(mmi, mmi);</p>	<p>Pack and saturate four unsigned words from source and four unsigned words from destination and store the result as eight unsigned bytes into the destination MMX register. Word values greater than 255 are clipped to 255 during the saturation operation. See the chapter on the MMX instructions for more details. The bytes obtained from the destination register wind up in the L.O. four bytes of the destination; the bytes computed from the signed saturation of the source register wind up in the H.O. four bytes of the destination register.</p>
<p>paddb(mem64, mmi); paddb(mmi, mmi);</p>	<p>Packed Add of Bytes. This instruction adds together the individual bytes of the two operands. The addition of each byte is independent of the other eight bytes; there is no carry from byte to byte. If an overflow occurs in any byte, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.</p>
<p>padd(mem64, mmi); padd(mmi, mmi);</p>	<p>Packed Add of Double Words. This instruction adds together the individual dwords of the two operands. The addition of each dword is independent of the other two dwords; there is no carry from dword to dword. If an overflow occurs in any dword, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.</p>
<p>paddsb(mem64, mmi); paddsb(mmi, mmi);</p>	<p>Packed Add of Bytes, signed saturated. This instruction adds together the individual bytes of the two operands. The addition of each byte is independent of the other eight bytes; there is no carry from byte to byte. If an overflow or underflow occurs in any byte, then the value saturates at -128 or +127. This instruction does not affect any flags.</p>
<p>paddsw(mem64, mmi); paddsw(mmi, mmi);</p>	<p>Packed Add of Words, signed saturated. This instruction adds together the individual words of the two operands. The addition of each word is independent of the other four words; there is no carry from word to word. If an overflow or underflow occurs in any word, the value saturates at either -32768 or +32767. This instruction does not affect any flags.</p>
<p>paddusb(mem64, mmi); paddusb(mmi, mmi);</p>	<p>Packed Add of Bytes, unsigned saturated. This instruction adds together the individual bytes of the two operands. The addition of each byte is independent of the other eight bytes; there is no carry from byte to byte. If an overflow or underflow occurs in any byte, then the value saturates at 0 or 255. This instruction does not affect any flags.</p>

Table 3: MMX Instruction Set

Instruction	Description
paddsw(mem64, mmi); paddsw(mmi, mmi);	Packed Add of Words, unsigned saturated. This instruction adds together the individual words of the two operands. The addition of each word is independent of the other four words; there is no carry from word to word. If an overflow or underflow occurs in any word, the value saturates at either 0 or 65535. This instruction does not affect any flags.
paddw(mem64, mmi); paddw(mmi, mmi);	Packed Add of Words. This instruction adds together the individual words of the two operands. The addition of each word is independent of the other four words; there is no carry from word to word. If an overflow occurs in any word, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.
pand(mem64, mmi); pand(mmi, mmi);	Packed AND. This instruction computes the bitwise AND of the source and the destination values, leaving the result in the destination. This instruction does not affect any flags.
pandn(mem64, mmi); pandn(mmi, mmi);	Packed AND NOT. This instruction makes a temporary copy of the first operand and inverts all of the bits in this copy; then it ANDs this value with the destination MMX register. This instruction does not affect any flags.
pavgb(mem64, mmi); pavgb(mmi, mmi);	Packed Average of Bytes. This instruction computes the average of the eight pairs of bytes in the two operands. It leaves the result in the destination (second) operand.
pavgw(mem64, mmi); pavgw(mmi, mmi);	Packed Average of Words. This instruction computes the average of the four pairs of words in the two operands. It leaves the result in the destination (second) operand.
pcmpeqb(mem64, mmi); pcmpeqb(mmi, mmi);	Packed Compare for Equal Bytes. This instruction compares the individual bytes in the two operands. If they are equal this instruction sets the corresponding byte in the destination (second) register to \$FF (all ones); if they are not equal, this instruction sets the corresponding byte to zero.
pcmpeqd(mem64, mmi); pcmpeqd(mmi, mmi);	Packed Compare for Equal Double Words. This instruction compares the individual double words in the two operands. If they are equal this instruction sets the corresponding double word in the destination (second) register to \$FFFF_FFFF (all ones); if they are not equal, this instruction sets the corresponding dword to zero.
pcmpeqw(mem64, mmi); pcmpeqw(mmi, mmi);	Packed Compare for Equal Words. This instruction compares the individual words in the two operands. If they are equal this instruction sets the corresponding word in the destination (second) register to \$FFFF (all ones); if they are not equal, this instruction sets the corresponding word to zero.

Table 3: MMX Instruction Set

Instruction	Description
<p><code>pcmpgtb(mem64, mmi);</code> <code>pcmpgtb(mmi, mmi);</code></p>	<p>Packed Compare for Greater Than, Bytes. This instruction compares the individual bytes in the two operands. If the destination (second) operand byte is greater than the source (first) operand byte, then this instruction sets the corresponding byte in the destination (second) register to \$FF (all ones); if they are not equal, this instruction sets the corresponding byte to zero. Note that there is no PCMPLEB instruction. You can simulate this instruction by swapping the operands in the PCMPGTB instruction (i.e., compare in the opposite direction). Also note that these operands are, in a sense, backwards compared with the standard CMP instruction. This instruction compares the second operand to the first rather than the other way around. This was done because the second operand is always the destination operand and, unlike the CMP instruction, this instruction writes data to the destination operand.</p>
<p><code>pcmpgtd(mem64, mmi);</code> <code>pcmpgtd(mmi, mmi);</code></p>	<p>Packed Compare for Greater Than, Double Words. This instruction compares the individual dwords in the two operands. If the destination (second) operand dword is greater than the source (first) operand dword, then this instruction sets the corresponding dword in the destination (second) register to \$FFFF_FFFF (all ones); if they are not equal, this instruction sets the corresponding dword to zero. Note that there is no PCMPLED instruction. You can simulate this instruction by swapping the operands in the PCMPGTD instruction (i.e., compare in the opposite direction). Also note that these operands are, in a sense, backwards compared with the standard CMP instruction. This instruction compares the second operand to the first rather than the other way around. This was done because the second operand is always the destination operand and, unlike the CMP instruction, this instruction writes data to the destination operand.</p>
<p><code>pcmpgtw(mem64, mmi);</code> <code>pcmpgtw(mmi, mmi);</code></p>	<p>Packed Compare for Greater Than, Words. This instruction compares the individual words in the two operands. If the destination (second) operand word is greater than the source (first) operand word, then this instruction sets the corresponding word in the destination (second) register to \$FFFF (all ones); if they are not equal, this instruction sets the corresponding dword to zero. Note that there is no PCMPLEW instruction. You can simulate this instruction by swapping the operands in the PCMPGTW instruction (i.e., compare in the opposite direction). Also note that these operands are, in a sense, backwards compared with the standard CMP instruction. This instruction compares the second operand to the first rather than the other way around. This was done because the second operand is always the destination operand and, unlike the CMP instruction, this instruction writes data to the destination operand.</p>

Table 3: MMX Instruction Set

Instruction	Description
pextrw(imm8, mmi, reg32);	Packed Extraction of a word. The imm8 value must be a constant in the range 0..3. This instruction copies the specified word from the MMX register into the L.O. word of the destination 32-bit integer register. This instruction zero extends the 16-bit value to 32 bits in the integer register. Note that there are no extraction instructions for bytes or dwords. However, you can easily extract a byte using PEXTRW and an AND or XCHG instruction (depending on whether the byte number is even or odd). You can use MOVD to extract the L.O. dword. to extract the H.O. dword of an MMX register requires a bit more work; either extract the two words and merge them or move the data to memory and grab the dword you're interested in.
pinsw(imm8, reg32, mmi);	Packed Insertion of a word. The imm8 value must be a constant in the range 0..3. This instruction copies the L.O. word from the 32-bit integer register into the specified word of the destination MMX register. This instruction ignores the H.O. word of the integer register.
pmaddwd(mem64, mmi); pmaddwd(mmi, mmi);	Packed Multiple and Accumulate (Add). This instruction multiplies together the corresponding words in the source and destination operands. Then it adds the two double word products from the multiplication of the two L.O. words and stores this double word sum in the L.O. dword of the destination MMX register. Finally, it adds the two double word products from the multiplication of the H.O. words and stores this double word sum in the H.O. dword of the destination MMX register.
pmaxw(mem64, mmi); pmaxw(mmi, mmi);	Packed Signed Integer Word Maximum. This instruction compares the four words between the two operands and stores the signed maximum of each corresponding word in the destination MMX register.
pmaxub(mem64, mmi); pmaxub(mmi, mmi);	Packed Unsigned Byte Maximum. This instruction compares the eight bytes between the two operands and stores the unsigned maximum of each corresponding byte in the destination MMX register.
pminw(mem64, mmi); pminw(mmi, mmi);	Packed Signed Integer Word Minimum. This instruction compares the four words between the two operands and stores the signed minimum of each corresponding word in the destination MMX register.
pminub(mem64, mmi); pminub(mmi, mmi);	Packed Unsigned Byte Minimum. This instruction compares the eight bytes between the two operands and stores the unsigned minimum of each corresponding byte in the destination MMX register.
pmovmskb(mmi, reg32);	Move Byte Mask to Integer. This instruction creates a byte by extracting the H.O. bit of the eight bytes from the MMX source register. It zero extends this value to 32 bits and stores the result in the 32-bit integer register.
pmulhuw(mem64, mmi); pmulhuw(mmi, mmi);	Packed Multiply High, Unsigned Words. This instruction multiplies the four unsigned words of the two operands together and stores the H.O. word of the resulting products into the corresponding word of the destination MMX register.

Table 3: MMX Instruction Set

Instruction	Description
<p><code>pmulhw(mem64, mmi);</code> <code>pmulhw(mmi, mmi);</code></p>	<p>Packed Multiply High, Signed Words. This instruction multiplies the four signed words of the two operands together and stores the H.O. word of the resulting products into the corresponding word of the destination MMX register.</p>
<p><code>pmullw(mem64, mmi);</code> <code>pmullw(mmi, mmi);</code></p>	<p>Packed Multiply Low, Signed Words. This instruction multiplies the four signed words of the two operands together and stores the L.O. word of the resulting products into the corresponding word of the destination MMX register.</p>
<p><code>por(mem64, mmi);</code> <code>por(mmi, mmi);</code></p>	<p>Packed OR. Computes the bitwise OR of the two operands and stores the result in the destination (second) MMX register.</p>
<p><code>psadbw(mem64, mmi);</code> <code>psadbw(mmi, mmi);</code></p>	<p>Packed Sum of Absolute Differences. This instruction computes the absolute value of the difference of each of the unsigned bytes between the two operands. Then it adds these eight results together to form a word sum. Finally, the instruction zero extends this word to 64 bits and stores the result in the destination (second) operand.</p>
<p><code>pshufw(imm8,mem64,mmi);</code></p>	<p>Packed Shuffle Word. This instruction treats the <code>imm8</code> value as an array of four two-bit values. These bits specify where the destination (third) operand's words obtain their values. Bits zero and one tell this instruction where to obtain the L.O. word, bits two and three specify where word #1 comes from, bits four and five specify the source for word #2, and bits six and seven specify the source of the H.O. word in the destination operand. Each pair of bytes specifies a word number in the source (second) operand. For example, an immediate value of <code>%00011011</code> tells this instruction to grab word #3 from the source and place it in the L.O. word of the destination; grab word #2 from the source and place it in word #1 of the destination; grab word #1 from the source and place it in word #2 of the destination; and grab the L.O. word of the source and place it in the H.O. word of the destination (i.e., swap all the words in a manner similar to the <code>BSWAP</code> instruction).</p>
<p><code>pshld(mem, mmi);</code> <code>pshld(mmi, mmi);</code> <code>pshld(imm8, mmi);</code></p>	<p>Packed Shift Left Logical, Double Words. This instruction shifts the destination (second) operand to the left the number of bits specified by the first operand. Each double word in the destination is treated as an independent entity. Bits are not carried over from the L.O. dword to the H.O. dword. Bits shifted out are lost and this instruction always shifts in zeros.</p>
<p><code>pshlq(mem, mmi);</code> <code>pshlq(mmi, mmi);</code> <code>pshlq(imm8, mmi);</code></p>	<p>Packed Shift Left Logical, Quad Word. This instruction shifts the destination operand to the left the number of bits specified by the first operand.</p>
<p><code>pshlw(mem, mmi);</code> <code>pshlw(mmi, mmi);</code> <code>pshlw(imm8, mmi);</code></p>	<p>Packed Shift Left Logical, Words. This instruction shifts the destination (second) operand to the left the number of bits specified by the first operand. Bits shifted out are lost and this instruction always shifts in zeros. Each word in the destination is treated as an independent entity. Bits are not carried over from the L.O. words into the next higher word. Bits shifted out are lost and this instruction always shifts in zeros.</p>

Table 3: MMX Instruction Set

Instruction	Description
psard(mem, mmi); psard(mmi, mmi); psard(imm8, mmi);	Packed Shift Right Arithmetic, Double Word. This instruction treats the two halves of the 64-bit register as two double words and performs separate arithmetic shift rights on them. The bit shifted out of the bottom of the two double words is lost.
psarw(mem, mmi); psarw(mmi, mmi); psarw(imm8, mmi);	Packed Shift Right Arithmetic, Word. This instruction operates independently on the four words of the 64-bit destination register and performs separate arithmetic shift rights on them. The bit shifted out of the bottom of the four words is lost.
psrld(mem, mmi); psrld(mmi, mmi); psrld(imm8, mmi);	Packed Shift Right Logical, Double Words. This instruction shifts the destination (second) operand to the right the number of bits specified by the first operand. Each double word in the destination is treated as an independent entity. Bits are not carried over from the H.O. dword to the L.O. dword. Bits shifted out are lost and this instruction always shifts in zeros.
pslrq(mem, mmi); pslrq(mmi, mmi); pslrq(imm8, mmi);	Packed Shift Right Logical, Quad Word. This instruction shifts the destination operand to the right the number of bits specified by the first operand.
pslrw(mem, mmi); pslrw(mmi, mmi); pslrw(imm8, mmi);	Packed Shift Right Logical, Words. This instruction shifts the destination (second) operand to the right the number of bits specified by the first operand. Bits shifted out are lost and this instruction always shifts in zeros. Each word in the destination is treated as an independent entity. Bits are not carried over from the H.O. words into the next lower word. Bits shifted out are lost and this instruction always shifts in zeros.
psubb(mem64, mmi); psubb(mmi, mmi);	Packed Subtract of Bytes. This instruction subtracts the individual bytes of the source (first) operand from the corresponding bytes of the destination (second) operand. The subtraction of each byte is independent of the other eight bytes; there is no borrow from byte to byte. If an overflow or underflow occurs in any byte, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.
psubd(mem64, mmi); psubd(mmi, mmi);	Packed Subtract of Double Words. This instruction subtracts the individual dwords of the source (first) operand from the corresponding dwords of the destination (second) operand. The subtraction of each dword is independent of the other; there is no borrow from dword to dword. If an overflow or underflow occurs in any dword, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.
psubsb(mem64, mmi); psubsb(mmi, mmi);	Packed Subtract of Bytes, signed saturated. This instruction subtracts the individual bytes of the source operand from the corresponding bytes of the destination operand, saturating to -128 or +127 if overflow or underflow occurs. The subtraction of each byte is independent of the other seven bytes; there is no carry from byte to byte. This instruction does not affect any flags.

Table 3: MMX Instruction Set

Instruction	Description
<p><code>psubsw(mem64, mmi);</code> <code>psubsw(mmi, mmi);</code></p>	<p>Packed Subtract of Words, signed saturated. This instruction subtracts the individual words of the source operand from the corresponding words of the destination operand, saturating to -32768 or +32767 if overflow or underflow occurs. The subtraction of each word is independent of the other three words; there is no carry from word to word. This instruction does not affect any flags.</p>
<p><code>psubusb(mem64, mmi);</code> <code>psubusb(mmi, mmi);</code></p>	<p>Packed Subtract of Bytes, unsigned saturated. This instruction subtracts the individual bytes of the source operand from the corresponding bytes of the destination operand, saturating to 0 if underflow occurs. The subtraction of each byte is independent of the other seven bytes; there is no carry from byte to byte. This instruction does not affect any flags.</p>
<p><code>psubusw(mem64, mmi);</code> <code>psubusw(mmi, mmi);</code></p>	<p>Packed Subtract of Words, unsigned saturated. This instruction subtracts the individual words of the source operand from the corresponding words of the destination operand, saturating to 0 if underflow occurs. The subtraction of each word is independent of the other three words; there is no carry from word to word. This instruction does not affect any flags.</p>
<p><code>psubw(mem64, mmi);</code> <code>psubw(mmi, mmi);</code></p>	<p>Packed Subtract of Words. This instruction subtracts the individual words of the source (first) operand from the corresponding words of the destination (second) operand. The subtraction of each word is independent of the others; there is no borrow from word to word. If an overflow or underflow occurs in any word, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.</p>
<p><code>punpckhbw(mem64, mmi);</code> <code>punpckhbw(mmi, mmi);</code></p>	<p>Unpack high packed data, bytes to words. This instruction unpacks and interleaves the high-order four bytes of the source (first) and destination (second) operands. It places the H.O. four bytes of the destination operand at the even byte positions in the destination and it places the H.O. four bytes of the source operand in the odd byte positions of the destination operand.</p>
<p><code>punpckhdq(mem64, mmi);</code> <code>punpckhdq(mmi, mmi);</code></p>	<p>Unpack high packed data, dwords to qword. This instruction copies the H.O. dword of the source operand to the H.O. dword of the destination operand and it copies the (original) H.O. dword of the destination operand to the L.O. dword of the destination.</p>
<p><code>punpckhwd(mem64, mmi);</code> <code>punpckhwd(mmi, mmi);</code></p>	<p>Unpack high packed data, words to dwords. This instruction unpacks and interleaves the high-order two words of the source (first) and destination (second) operands. It places the H.O. two words of the destination operand at the even word positions in the destination and it places the H.O. words of the source operand in the odd word positions of the destination operand.</p>
<p><code>punpcklbw(mem64, mmi);</code> <code>punpcklbw(mmi, mmi);</code></p>	<p>Unpack low packed data, bytes to words. This instruction unpacks and interleaves the low-order four bytes of the source (first) and destination (second) operands. It places the L.O. four bytes of the destination operand at the even byte positions in the destination and it places the L.O. four bytes of the source operand in the odd byte positions of the destination operand.</p>

Table 3: MMX Instruction Set

Instruction	Description
<p><code>punpckldq(mem64, mmi);</code> <code>punpckldq(mmi, mmi);</code></p>	<p>Unpack low packed data, dwords to qword. This instruction copies the L.O. dword of the source operand to the H.O. dword of the destination operand and it copies the (original) L.O. dword of the destination operand to the L.O. dword of the destination (i.e., it doesn't change the L.O. dword of the destination).</p>
<p><code>punpcklwd(mem64, mmi);</code> <code>punpcklwd(mmi, mmi);</code></p>	<p>Unpack low packed data, words to dwords. This instruction unpacks and interleaves the low-order two words of the source (first) and destination (second) operands. It places the L.O. two words of the destination operand at the even word positions in the destination and it places the L.O. words of the source operand in the odd word positions of the destination operand.</p>
<p><code>pxor(mem64, mmi);</code> <code>pxor(mmi, mmi);</code></p>	<p>Packed Exclusive-OR. This instruction exclusive-ORs the source operand with the destination operand leaving the result in the destination operand.</p>

The HLA Language Reference

Appendix E

To be written when the HLA language settles down a little bit.

In the meantime, please consult the HLA Language Reference Manual (a separate document that is part of the HLA distribution).

The HLA Standard Library Reference

Appendix F

To be written when the HLA language settles down a bit. See the HLA Standard Library documentation in the meantime.

HLA Exceptions

Appendix G

The HLA Standard Library provides the following exception types¹:

ex.StringOverflow

HLA raises this exception if you attempt to store too many characters into preallocated string variable. The following standard library routines can raise this exception:

pat.extract, strealloc, stdin.gets, str.cpy, str.setstr, str.cat, str.substr, str.insert, console.gets, cStrToStr, dToStr, e80ToStr, hToStr, u64ToStr, i64ToStr, qToStr, r80ToStr, tbToStr, wToStr, date.print, date.toString, and date.a_toString.

ex.StringIndexError

HLA raises this exception if a routine attempts to use an index that is beyond the last valid character in a string. The following standard library routines can raise this exception:

str.span2, str.rspan2, str.brk2, str.rbrk2, str.substr, str.a_substr, strToFlt, StrToi8, StrToi16, StrToi32, StrToi64, StrTou8, StrTou16, StrTou32, StrTou64, StrToh, StrTow, StrTod, and StrToq.

ex.ValueOutOfRange

HLA raises this exception if an arithmetic overflow occurs, if an input parameter is out of range, or if user input is too great for the destination variable. The following standard library routines can raise this exception:

arg.v, arg.delete, rand.urange, rand.range, stdin.geti8, stdin.geti16, stdin.geti32, stdin.geti64, stdin.getu8, stdin.getu16, stdin.getu32, stdin.getu64, stdin.geth, stdin.getw, stdin.getd, stdin.getq, stdin.getf, table.create, console.a_getRect, console.fillRect, console.fillRectAttr, console.getc, console.getRect, console.gets, console.gotoxy, console.putRect, console.scrollDnRect, console.scrollUpRect, atof, atoh, atoi8, atoi16, atoi32, atoi64, atou8, atou16, atou32, atou64, e80ToStr, r80ToStr, StrToi8, StrToi16, StrToi32, StrToi64, StrTou8, StrTou16, StrTou32, StrTou64, StrToh, StrTow, StrTod, StrToq, fileio.getd, fileio.geth, fileio.getw, fileio.getq, fileio.geti8, fileio.geti16, fileio.geti32, fileio.geti64, fileio.getu8, fileio.getu16, fileio.getu32, fileio.getu64, fileio.pute80pad, fileio.pute64pad, fileio.pute32pad, fileio.putr32Pad, fileio.putr64Pad, and fileio.putr80Pad.

ex.IllegalChar

Several HLA routines raise this exception if they encounter a non-ASCII character (character code \$80..\$FF) where a delimiter character is expected. Generally, you can treat this error as though it were a conversion error. Routines that raise this exception include:

atoh, atoi8, atoi16, atoi32, atoi64, atou8, atou16, atou32, and atou64.

ex.ConversionError

Routines in the HLA Standard Library raise this exception if there is an error conversion data from one format to another. Typically this occurs when converting strings to numeric data. Routines that raise this exception include:

1. Please note that the HLA Standard Library is under constant revision and the list appearing in this chapter may be slightly out of date. Please consult the HLA Standard Library documentation for an up-to-date listing of exceptions and the routines that raise them.

`stdin.geti8`, `stdin.geti16`, `stdin.geti32`, `stdin.geti64`, `stdin.getu8`, `stdin.getu16`, `stdin.getu32`, `stdin.getu64`, `stdin.geth`, `stdin.getw`, `stdin.getd`, `stdin.getq`, `stdin.getf`, `atof`, `atoh`, `atoi8`, `atoi16`, `atoi32`, `atoi64`, `atou8`, `atou16`, `atou32`, `atou64`, `fgetf`, `fileio.geti8`, `fileio.geti16`, `fileio.geti32`, `fileio.geti64`, `fileio.getu8`, `fileio.getu16`, `fileio.getu32`, `fileio.getu64`, `fileio.geth`, `fileio.getw`, `fileio.getd`, and `fileio.getq`.

ex.BadFileHandle

The file class method `file.handle` raises this exception if the handle associated with a file class object is illegal (has not been initialized).

ex.FileOpenFailure

The `fileio.Open`, `fileio.OpenNew`, `file.Open`, and `file.OpenNew` procedures raise this exception if there is an error opening a file.

ex.FileCloseError

The `fileio.Close` and `file.Close` procedures raise this exception if there is some sort of error when attempting to close a file.

ex.FileWriteError

Those routines that write data to a file (e.g., `fputi8`) raise this exception if there is an error writing data to the output file.

ex.FileReadError

Those routines that read data from a file (e.g., `fileio.geti8`) raise this exception if there is a physical error reading the data from the file.

ex.DiskFullError

Those routines that write data to a file will raise this exception if an attempt is made to write data to a full disk.

ex.EndOfFile

Those routines that read data from a file will raise this exception if your program attempts to read data beyond the end of the file.

ex.MemoryAllocationFailure

Routines that allocation storage (e.g., `malloc` and `realloc`) will raise this exception if Windows cannot satisfy the memory allocation request. Several routines in the standard library may raise this exception since they indirectly call the HLA `malloc` routine. Examples include `stdin.a_gets` and almost any other Standard Library routine that has “a_” as a prefix to the name.

ex.AttemptToDerefNULL

Several routines in the Standard Library that expect a string pointer will raise this exception if the string pointer (or other pointer) contains NULL (zero). Examples include:

`getf`, `str.cpy`, `str.a_cpy`, `str.setstr`, `str.cat`, `str.a_cat`, `str.index`, `str.rindex`, `str.chpos`, `str.rchpos`, `str.span`, `str.span2`, `str.rspan`, `str.rspan2`, `str.brk`, `str.brk2`, `str.rbrk`, `str.rbrk2`, `str.eq`, `str.ne`, `str.lt`, `str.le`, `str.gt`, `str.ge`, `str.substr`, `str.a_substr`, `str.insert`, `str.a_insert`, `str.delete`, `str.a_delete`, `str.ieq`, `str.ine`, `str.ilt`, `str.ile`, `str.igt`, `str.ige`,

str.upper, str.a_upper, str.lower, str.a_lower, str.delspace, str.a_delspace, str.trim, str.a_trim, str.tokenize, str.tokenize2, atof, atoi8, atoi16, atoi32, atoi64, atou8, atou16, atou32, atou64, dtostr, htostr, wtostr, qtostr, strToFlt, StrToi8, StrToi16, StrToi32, StrToi64, StrTou8, StrTou16, StrTou64, StrToh, StrTow, StrTod, StrToq, and tbtostr.

ex.WidthTooBig

HLA routines that print a numeric value within a field width will raise this exception if the specified width exceeds 1,024 characters. Routines that raise this exception include *stdout.put*, *stdout.puti8Size*, *fputu32Size*, and all other *xxxxSize* output routines.

ex.TooManyCmdLnParms

The *arg.CmdLn* procedure raises this exception if it determines that there are more than 256 command line parameters on the command line (a virtual impossibility since command lines are generally limited to 128 characters). Since all of the other routines in the *args* module can call *arg.CmdLn*, it is possible for any of the routines in this module to raise this exception.

ex.ArrayShapeViolation

Routines in the arrays module raise this exception if the dimension on some array are inappropriate for the specified operation. For example, the *array.cpy* code will raise this exception if you attempt to copy a source array to a destination whose dimensions don't exactly match the source array.

ex.InvalidDate

The routines in the datetime module will raise this exception if you pass them an illegal date value as a parameter.

ex.InvalidDateFormat

The date output routines (*date.print* and *date.toString*) raise this exception if you attempt convert a date to a string but the current (internal) date format variable contains an invalid value. This usually implies that you've passed an incorrect parameter to the *date.SetFormat* procedure.

ex.TimeOverflow

The *time.secsToHMS* procedure raises this overflow if there is an error converting time in seconds to hours, minutes, and seconds (very rare, only occurs above two billion seconds).

ex.AccessViolation

This is a hardware exception that Windows raise if your program attempts to access an illegal memory location (generally a NULL reference or an uninitialized pointer).

ex.Breakpoint

This exception is raised by Windows for debugger programs; you should never see this exception unless you are writing a debugger program.

ex.SingleStep

This is another exception that is raised by Windows for debugger programs; you should never see this exception unless you are writing a debugger program.

ex.PrivInstr

Windows raises this instruction if you attempt to execute a special instruction that is illegal in “user mode” (that is, can only be executed by Windows). Since HLA only compiles a few privileged instructions, and this book doesn’t discuss them at all, the only way you’ll probably see this exception occur is if your program jumps off into (non-code) memory somewhere and begins executing data as instructions.

ex.IllegalInstr

Windows raises this exception if the CPU attempts to execute some code that is not a valid instruction. This generally implies that your program has jumped off into data memory and is attempting to execute data as machine instructions.

ex.BoundInstr

Windows raises this exception if you execute the BOUND instruction (see “Some Additional Instructions: INTMUL, BOUND, INTO” on page 393) and the register value is outside the specified range.

ex.IntoInstr

Windows raises this exception if you execute the INTO instruction and the overflow flag is set (see “Some Additional Instructions: INTMUL, BOUND, INTO” on page 393).

ex.DivideError

Windows raises this exception if you attempt an integer division by zero, or if the quotient of a division is too large to fit within the destination operand(AL, AX, or EAX).

ex.fDenormal**ex.fDivByZero****ex.fInexactResult****ex.fInvalidOperation****ex.fOverflow****ex.fStackCheck****ex.fUnderflow**

Windows raises one of these exceptions if you’ve enabled exceptions on the FPU and one of the specified conditions occurs (e.g., a floating point division by zero will raise the *ex.fDivByZero* exception).

InvalidHandle

Windows will raise this exception if you pass an uninitialized or otherwise invalid handle value to a Windows API (application programmer’s interface) routine. Many of the HLA Standard Library routines pass handles to Windows, so this exception could occur as a result of a call to an input/output routine.

StackOverflow

Windows raises this exception if the stack exceeds the storage allocated to it (16Mbytes in a typical HLA program).

ControlC

HLA raises this exception if the user presses control-C or control-Break during program execution.

HLA Compile-Time Functions

Appendix H

H.1 Conversion Functions

The conversion functions translate data from one format to another. For example, functions in this group can convert integers to strings or strings to integers. These routines provide the compile-time equivalent of the HLA Standard Library CONV module.

The compile-time conversion routines are unusual in the set of compile-time function insofar as they do not require a leading "@" symbol. Instead, the conversion routines use the names of several of the built-in data types. The following table describes each of these functions:

Table 1: Compile-Time Data Conversion Functions

Function	Parameters ^a	Description
boolean	boolean(<i>constExpr</i>) ConstExpr can be a boolean, integer, character, or string operand.	If <i>constExpr</i> is numeric, this function returns false for zero and true for any other value. For characters, "t" or "f" returns true or false (respectively), anything else is an error. For strings, the operands must be "true" or "false" (else an error occurs). The boolean function returns boolean values unchanged and returns an error for any other type.
int8	<i>xxxx</i> (<i>constExpr</i>) Note: <i>xxxx</i> represents one of the function names to the left. <i>constExpr</i> can be any constant expression that evaluates to a numeric, character, boolean, or string operand.	These functions will convert their operand to the specified data type. These functions generate an error if the resulting value will not fit in the specified data type (e.g., int8(-1000) will generate an error). Note that HLA treats <i>byte</i> , <i>word</i> , and <i>dword</i> functions identically to <i>uns8</i> , <i>uns16</i> , and <i>uns32</i> (respectively). For boolean operands, true returns one and false returns zero. If the operand is a real value, then these functions truncate the value to obtain the corresponding integer return value. For character operands, these function return the corresponding ASCII code of the character. For string operands, the string must be a legal sequence of characters that form a decimal number.
int16		
int32		
uns8		
uns16		
uns32		
byte		
word		
dword		

Table 1: Compile-Time Data Conversion Functions

Function	Parameters ^a	Description
real32	<i>xxxxxx</i> (<i>constExpr</i>)	These functions convert their specified operand to the corresponding real value. These functions convert integer operands to the corresponding real value. If the operand is a string expression, it must be a valid sequence of characters that corresponds to an HLA floating point value. These functions convert that string to the corresponding real value.
real64	Note: <i>xxxxxx</i> represents one of the function names to the left. <i>constExpr</i> can be any constant expression that evaluates to a numeric or string operand.	
real80		
char	<i>char</i> (<i>constExpr</i>) <i>constExpr</i> must be a positive integer value, a character, or a string.	If the parameter is an integer value, this function returns the character with that ASCII code. If the parameter is a string, this function returns the first character of that string. If the operand is a character, this function simply returns that character value.
string	<i>string</i> (<i>constExpr</i>) <i>constExpr</i> can be any legal constant data type.	This function returns the string representation of the specified parameter. For real values, this function returns the scientific notation format for the value. For boolean expressions, this function returns the value "true" or "false". For character operands, this function returns a string containing the single character specified as the operand. For integer parameters, this function returns a string containing the decimal equivalent of that value. For character set operands, this function returns a string listing all the characters in the character set. If the operand is a string expression, this function simply returns that string.
cset	<i>cset</i> (<i>constExpr</i>) <i>constExpr</i> can be a character, string, or a cset.	This function returns a character set containing the characters specified by the operand. If the operand is a character, then this function returns the singleton set containing that single character. If the operand is a string, this function returns the union of all the characters in that string. If the operand is a character set, this function returns that character set.

Table 1: Compile-Time Data Conversion Functions

Function	Parameters ^a	Description
text	text(<i>constExpr</i>) <i>constExpr</i> : same as for string.	This function takes the same parameters as the string function. Instead of returning a string constant, however, this function expands the text in-line at the point of the function. This function is equivalent to the @text function.
@odd	@odd(<i>constExpr</i>) <i>constExpr</i> must be an integer value.	This function returns true if the integer operand is an odd number, it returns false otherwise.

a. Integer operands can be any of the *intXX*, *unsXX*, *byte*, *word*, or *dword* types.

Internally, HLA generally maintains all numeric constant values as *int32*, *uns32*, *dword*, or *real80*. Therefore, it is unlikely that you would want to use the *int8*, *int16*, *uns8*, *uns16*, *real32*, or *real64* compile-time functions in your program unless you want to force an error if a value is out of range.

Of these conversion functions, the *string* function is, perhaps, the most useful. Many compile-time functions and statements accept only string operands. You can use the *string* function to translate other data types to a string in a situation where you wish to use one of these other data types. For example, the #ERROR statement only allows a single string parameter. However, you can construct complex error messages by using the string concatenation operator ("+") with the *string* function, e.g.,

```
#error( "Constant value i32=" + string(i32) + " and that is out of range" )
```

H.2 Numeric Functions

These functions provide common mathematical functions. Remember, these functions compute their values at compile-time. Their parameters are constants and they return constant values. These functions are not useable with variables at run-time. See the HLA Standard Library for comparable functions you can call from your assembly language programs while they are running.

Table 2: Numeric Compile-Time Functions

Function	Parameters ^a	Description
@abs	@abs(<i>constExpr</i>) <i>constExpr</i> must be a numeric value.	Returns the absolute value of the parameter. The return type is the same as the parameter type.

Table 2: Numeric Compile-Time Functions

Function	Parameters ^a	Description
@byte	<p>@byte(<i>Expr</i>, <i>select</i>)</p> <p><i>Expr</i> must be a 32-bit ordinal expression, a real expression (32, 64, or 80 bits), or a <i>cset</i> expression.</p> <p><i>select</i> must be less than the size of <i>Expr</i>'s data type.</p>	<p>This function extracts the specified byte from the value of <i>Expr</i> as selected via the <i>select</i> parameter. If <i>select</i> is zero, this function returns the L.O. byte, higher values for <i>select</i> return the corresponding higher-order bytes of the object.</p> <p>Note that HLA usually extends literal constants to the largest representation possible, e.g., HLA treats 1.234 as a real80 value. Use coercion to change this, if necessary (e.g., @byte(real32(1.234), 3))</p>
@ceil	<p>@ceil(<i>constExpr</i>)</p> <p><i>constExpr</i> must be a numeric value.</p>	<p>If the parameter is an integer value, this function simply converts it to a real value and returns that value. If the parameter is a real value, then this function returns the smallest integer value larger than or equal to the parameter's value (i.e., this function rounds a real value to the next highest integer if the real value contains a fractional part).</p>
@cos	<p>@cos(<i>constExpr</i>)</p> <p><i>constExpr</i> must be a numeric value expressing an angle in radians.</p>	<p>This function returns the cosine of the specified parameter value.</p>
@exp	<p>@exp(<i>constExpr</i>)</p> <p><i>constExpr</i> must be a numeric value.</p>	<p>This function return <i>e</i> raised to the specified power.</p>
@floor	<p>@floor(<i>constExpr</i>)</p> <p><i>constExpr</i> must be a numeric value.</p>	<p>This function returns the largest integer value that is less than or equal to the numeric value passed as a parameter. For positive real numbers this is equivalent to truncation (for negative numbers, it rounds towards negative infinity).</p>
@log	<p>@log(<i>constExpr</i>)</p> <p><i>constExpr</i> is a non-negative numeric value.</p>	<p>This function computes the natural (base <i>e</i>) logarithm of its operand.</p>

Table 2: Numeric Compile-Time Functions

Function	Parameters ^a	Description
@log10	@log10(<i>constExpr</i>) <i>constExpr</i> is a non-negative numeric value.	This function computes the base-10 logarithm of its operand.
@max	@max(<i>list</i>) <i>list</i> is a list of two or more comma separated numeric expressions.	This function returns the maximum of a set of numeric values. The values in the list must all be the same type.
@min	@min(<i>list</i>) <i>list</i> is a list of two or more comma separated numeric expressions.	This function returns the minimum of a set of numeric values. The values in the list must all be the same type.
@random	@random(<i>intExpr</i>) <i>intExpr</i> must be a positive integer value.	This function returns a pseudo-random number between zero and <i>intExpr-1</i> . Currently HLA uses the random function provided by the C standard library; so don't expect a high quality random number generator here. In particular, if <i>intExpr</i> is a small value, the quality of the random number generator is very low.
@randomize	@randomize(<i>intExpr</i>) <i>intExpr</i> must be a positive integer value.	This function attempts to "randomize" the random number generator seed. Then it returns a random number between zero and <i>intExpr-1</i> . You should not call this function multiple times in your source file.
@sin	@sin(<i>constExpr</i>) <i>constExpr</i> must be a numeric value expressing an angle in radians.	This function returns the sine of the specified parameter value.
@sqrt	@sqrt(<i>constExpr</i>) <i>constExpr</i> must be a non-negative numeric value.	This function returns the square root of the specified operand value.

Table 2: Numeric Compile-Time Functions

Function	Parameters ^a	Description
@tan	@tan(<i>constExpr</i>) <i>constExpr</i> must be a numeric value expressing an angle in radians.	This function returns the tangent of the specified parameter value.

a. Numeric parameters are *intX*, *unsX*, *byte*, *word*, *dword*, or *realX* values.

H.3 Date/Time Functions

The Date/Time compile-time functions return strings holding the current date and time.

Table 3: HLA Compile-Time Date/Time Functions

Function	Parameters	Description
@date	@date	This function returns a string specifying the current date. This string typically takes the form "year/month/day", e.g., "2000/12/31".
@time	@time	This function returns a string specifying the current time. This string typically takes the form "HH:MM:SS xM" (x= A or P).

H.4 Classification Functions

These functions test a character or string to see if the characters belong to a certain class (e.g., alphabetic characters). These functions return true or false depending upon the result of the comparison.

If the operand is a character expression, these functions return true if that character belongs to the specified class. If the operand is a string, these functions return true if all characters in the string belong to the specified class.

Also see the HLA Compile-Time Pattern Matching Functions for some different ways to test characters in a string. Remember, these are compile-time functions. The HLA Standard Library contains comparable routines for use at run-time in your programs. See the HLA Standard Library documentation for more details.

Table 4: HLA Compile-Time Character Classification Functions.

Function	Parameters	Description
@isalpha	@isalpha(<i>constExpr</i>) <i>constExpr</i> must be a character or a string expression.	This function returns true if the character operand is an alphabetic character. If the parameter is a string, this function returns true if all characters in the string are alphabetic.
@isalphanum	@isalphanum(<i>constExpr</i>) <i>constExpr</i> must be a character or a string expression.	This function returns true if the character operand is an alphanumeric character. If the parameter is a string, this function returns true if all characters in the string are alphanumeric.
@isdigit	@isdigit(<i>constExpr</i>) <i>constExpr</i> must be a character or a string expression.	This function returns true if the character operand is a decimal digit character. If the parameter is a string, this function returns true if all characters in the string are digits.
@islower	@islower(<i>constExpr</i>) <i>constExpr</i> must be a character or a string expression.	This function returns true if the character operand is a lower case alphabetic character. If the parameter is a string, this function returns true if all characters in the string are lower case alphabetic characters.
@isspace	@isspace(<i>constExpr</i>) <i>constExpr</i> must be a character or a string expression.	This function returns true if the character operand is a space character ^a . If the parameter is a string, this function returns true if all characters in the string are spaces.
@isupper	@isupper(<i>constExpr</i>) <i>constExpr</i> must be a character or a string expression.	This function returns true if the character operand is an upper case alphabetic character. If the parameter is a string, this function returns true if all characters in the string are upper case alphabetic characters

Table 4: HLA Compile-Time Character Classification Functions.

Function	Parameters	Description
@isxdigit	@isxdigit(<i>constExpr</i>) <i>constExpr</i> must be a character or a string expression.	This function returns true if the character operand is a hexadecimal digit character {0-9, a-f, A-F}. If the parameter is a string, this function returns true if all characters in the string are hexadecimal digits.

a. "Space" means any white space character. This includes tabs, newlines, etc.

H.5 String and Character Set Functions

The following functions provide a very powerful set of string manipulation functions to the HLA compile-time language. These functions let you easily manipulate data like macro parameters and #text..#end-text blocks.

Remember, these are compile-time functions. The HLA Standard Library contains comparable routines for use at run-time in your programs. See the HLA Standard Library documentation for more details.

The @extract function behaves a little differently than the *cs.extract* function in the HLA Standard Library. @extract does not actually remove the specified character from the character set. Keep this in mind if you use both @extract and *cs.extract* frequently.

Table 5: HLA Compile-Time String Functions

Function	Parameters	Description
@delete	@delete(<i>strExpr</i> , <i>start</i> , <i>len</i>) <i>strExpr</i> must be a string expression. <i>start</i> and <i>len</i> must be positive integer expressions.	This function returns a string consisting of the <i>strExpr</i> parameter with <i>len</i> characters removed starting at position <i>start</i> in the string. The first character in the string is at position zero. Therefore, @delete("Hello", 2, 3) returns the string "He".

Table 5: HLA Compile-Time String Functions

Function	Parameters	Description
@extract	@extract(<i>csetExpr</i>) <i>csetExpr</i> must be a character set value.	This function returns an arbitrary character from the specified character set. Note that this function does not remove that character from the set. You must manually remove the character if you do not want the next call to @extract (with the same parameter) to return the same character, e.g., val c := @extract(someSet); someSet := someSet - {c};
@index	@index(<i>strExpr</i> , <i>start</i> , <i>findStr</i>) <i>strExpr</i> and <i>findStr</i> must be string expressions. <i>start</i> must be a non-negative integer value.	This function searches for the string specified by <i>findStr</i> within the <i>strExpr</i> string starting at character position <i>start</i> . If this function finds the string, it returns the index into <i>strExpr</i> where it located <i>findStr</i> . If it does not find the string, it returns -1.
@insert	@insert(<i>destStr</i> , <i>start</i> , <i>strToIns</i>) <i>destStr</i> and <i>strToIns</i> must be string expressions. <i>start</i> must be a non-negative integer expression.	This function returns a string consisting of the combination of the <i>destStr</i> and <i>strToIns</i> strings. This function inserts <i>strToIns</i> into <i>destStr</i> at the position specified by <i>start</i> .
@length	@length(<i>strExpr</i>) <i>strExpr</i> must be a string expression.	This function returns the length of the specified string as an integer result.
@lowercase	@lowercase(<i>strExpr</i> , <i>start</i>) <i>strExpr</i> must be a string expression. <i>start</i> must be a non-negative integer expression.	This function translates all characters from position <i>start</i> to the end of the string to lower case (if they were previously upper case alphabetic characters).

Table 5: HLA Compile-Time String Functions

Function	Parameters	Description
@rindex	@rindex(<i>strExpr</i> , <i>start</i> , <i>findStr</i>) <i>strExpr</i> and <i>findStr</i> must be string expressions. <i>start</i> must be a non-negative integer value.	This function searches backwards to find the last occurrence of the string specified by <i>findStr</i> within the <i>strExpr</i> string. This function will only search back to position <i>start</i> . If this function finds the string, it returns the index into <i>strExpr</i> where it located <i>findStr</i> . If it does not find the string, it returns -1.
@strbrk	@strbrk(<i>strExpr</i> , <i>start</i> , <i>csetExpr</i>) <i>strExpr</i> must be a string expression. <i>start</i> must be a non-negative integer expression. <i>csetExpr</i> must be a character set expression.	Starting at position <i>start</i> within <i>strExpr</i> , this function searches for the first character of <i>strExpr</i> that is a member of the <i>csetExpr</i> character set. It returns -1 if no such characters exist.
@strset	@strset(<i>charExpr</i> , <i>len</i>) <i>charExpr</i> must be a character value. <i>len</i> must be a non-negative integer value.	This function returns the string consisting of <i>len</i> copies of <i>charExpr</i> concatenated together.
@strspan	@strspan(<i>strExpr</i> , <i>start</i> , <i>csetExpr</i>) <i>strExpr</i> must be a string expression. <i>start</i> must be a non-negative integer expression. <i>csetExpr</i> must be a character set expression.	Starting at position <i>start</i> within <i>strExpr</i> , this function searches for the first character of <i>strExpr</i> that is not a member of the <i>csetExpr</i> character set. It returns the length of the string if all remaining characters are in the specified character set.
@substr	@substr(<i>strExpr</i> , <i>start</i> , <i>len</i>) <i>strExpr</i> must be a string expression. <i>start</i> and <i>len</i> must be non-negative integer values.	This function returns the sequence of <i>len</i> characters (the "substring") starting at position <i>start</i> in the <i>strExpr</i> string.

Table 5: HLA Compile-Time String Functions

Function	Parameters	Description
@tokenize	<p>@tokenize(<i>strExpr</i>, <i>start</i>, <i>Delims</i>, <i>Quotes</i>)</p> <p><i>strExpr</i> must be a string expression.</p> <p><i>start</i> must be a non-negative integer value.</p> <p><i>Delims</i> and <i>Quotes</i> must be character set expressions.</p>	<p>This function returns an array of strings consisting of the various substrings in <i>strExpr</i> obtained by separating the substrings using the <i>Delims</i> character set.</p> <p>This "lexical scan" operation begins at position <i>start</i> in <i>strExpr</i>. The function first skips over all characters in <i>Delims</i> and the collects all characters until it finds another delimiter character from the <i>Delims</i> set. It repeats this process, adding each scanned string to the array it returns, until it reaches the end of the string.</p> <p>The fourth parameter, <i>Quotes</i>, specifies special quoting characters. Typically, this character set is the empty set. However, if it contains a character, then that character is a quote character and all characters between two occurrences of the quote symbol constitute a single string, even if delimiters appear within that string. Typical characters in the <i>Quotes</i> character set would be apostrophes or quotation marks. If "(", "[", or "{" appears in the <i>Quotes</i> set, then the corresponding closing symbol must also appear and <i>tokenize</i> uses the pair of quote objects to surround a quoted item.</p> <p>This function is unusual insofar as it returns an array constant as its result. Typically you would assign this to a VAL or CONST object so you can gain access to the individual items in the array. To determine the number of elements in this array, use the @elements function.</p>

Table 5: HLA Compile-Time String Functions

Function	Parameters	Description
@trim	@trim(<i>strExpr</i>) <i>strExpr</i> must be a string expression.	This function returns the specified string operand with leading and trailing spaces removed.
@uppercase	@uppercase(<i>strExpr</i>) <i>strExpr</i> must be a string expression.	The function returns a copy of its string operand with any lower case alphabetic characters converted to upper case.

H.6 Pattern Matching Functions

The HLA compile-time language provides a large set of pattern matching functions similar to the run-time functions available in the PATTERNS.HHF module of the HLA Standard Library. These pattern matching functions, combined with the string processing functions of the previous section, provide you with the ability to write your own scanners and parsers (i.e., compilers) within the HLA compile-time language. Indeed, one of the primary purposes of the HLA compile-time language is to let you expand the HLA language to suit your needs. The pattern matching functions, along with macros, provide the backbone for this capability in HLA.

Note that there are some pretty serious differences between the HLA compile-time pattern matching functions and the routines in the PATTERNS.HHF module. Perhaps the biggest difference is the fact that the compile-time functions do not support backtracking while the run-time routines do. Fortunately, it is not that difficult to simulate backtracking on your own within the compile-time language.

Another difference between the compile-time functions and their run-time counterparts is the parameter list. The compile-time functions typically have a couple of optional parameters that let you extract information about the pattern match if it was successful. Consider, for a moment, the @matchStr function:

```
@matchStr( Str, tstStr )
```

When you call this function using the syntax above, this function will return true if the string expression *Str* begins with the sequence of characters found in *tstStr*. It returns false otherwise. Now consider the following invocation:

```
@matchStr( Str, tstStr, remainder )
```

This call to the function also returns true if *Str* begins with the characters found in the *tstStr* string. If this function returns true, then this function also copies the remaining characters (i.e., those characters in *Str* that following the *tstStr* characters at the beginning of *Str*) to the *remainder* VAL object. If @matchStr returns false, the value of *remainder* is undefined (with the single exception noted below).

It is perfectly legal to specify the same string as the *Str* and *remainder* parameters. For example, consider the following invocation:

```
@matchStr( str, "Hello ", str )
```

If *str* begins with the string "Hello " then this function will return true and replace *str*'s value with the string containing all characters beyond the sixth character of the string. If *str* does not begin with "Hello " then this function does not modify *str*'s value.

Most of the HLA compile-time pattern matching functions also allow a second optional parameter. Consider the following invocation of the @oneOrMoreCset function:

```
@oneOrMoreCset( str, {'0'..'9'}, remainder, matched )
```

If this function returns true it will copied the sequence of characters at the beginning of *str* that are members of the specified character set (i.e., digits) to the *matched* VAL object. This function returns all characters beyond the digits in the *remainder* VAL object. If this function returns false, then *remainder* and *matched* will contain undefined values and this function will not affect *str*.

H.6.1 String/Cset Pattern Matching Functions

Among the most useful of the pattern matching functions are those that checking the leading characters of a string to see if they are members of a particular character set. The following rich set of functions provides this capability in the compile-time language.

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@peekCset	<p>@peekCset(<i>str</i>, <i>cset</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function returns true if the first character of <i>str</i> is a member of <i>cset</i>; it returns false otherwise.</p> <p>If <i>rem</i> is present, this function copies <i>str</i> to <i>rem</i> upon return. If <i>matched</i> is present and the function returns true, this function stores a copy of the first character into the <i>matched</i> VAL object. The value of <i>matched</i> is undefined if this function returns false.</p>

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@oneCset	<p>@oneCset(<i>str</i>, <i>cset</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function returns true if the first character of <i>str</i> is a member of <i>cset</i>; it returns false otherwise.</p> <p>If <i>rem</i> is present, this function copies all characters of <i>str</i> beyond the first character to <i>rem</i> upon return. If <i>matched</i> is present and the function returns true, this function stores a copy of the first character into the <i>matched</i> VAL object. The value of <i>matched</i> is undefined if this function returns false.</p>
@uptoCset	<p>@uptoCset(<i>str</i>, <i>cset</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function copies all characters up to, but not including, a single character from the <i>cset</i> parameter. If the <i>str</i> parameter does not contain a character in the cset set, this function returns false. If it succeeds, and the <i>matched</i> parameter is present, it copies all characters it matches to the <i>matched</i> parameter and it copies all remaining characters to the <i>rem</i> parameter (if present).</p>
@zeroOrOneCset	<p>@zeroOrOneCset(<i>str</i>, <i>cset</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function always returns true. This function copies the single character it matches (if any) to the <i>matched</i> string and any remaining characters in the string to the <i>rem</i> object (assuming <i>rem</i> and <i>matched</i> appear in the parameter list).</p>

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@exactlyNCset	<p>@exactlyNCset(<i>str</i>, <i>cset</i>, <i>n</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>n</i> must be a non-negative integer value.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function returns true if the first <i>n</i> characters of <i>str</i> are members of <i>cset</i>. The character at position <i>n+1</i> must not be a member of <i>cset</i>.</p> <p>If this function returns true, it copies the first <i>n</i> characters of <i>str</i> to <i>matched</i> and copies any remaining characters to <i>rem</i> (assuming <i>rem</i> and <i>matched</i> are present).</p>
@firstNCset	<p>@firstNCset(<i>str</i>, <i>cset</i>, <i>n</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>n</i> must be a non-negative integer value.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function returns true if the first <i>n</i> characters of <i>str</i> are members of <i>cset</i>. The character at position <i>n+1</i> may or may not be a member of <i>cset</i>.</p> <p>If this function returns true, it copies the first <i>n</i> characters of <i>str</i> to <i>matched</i> and copies any remaining characters to <i>rem</i> (assuming <i>rem</i> and <i>matched</i> are present).</p>

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@nOrLessCset	<p>@nOrLessCset(<i>str</i>, <i>cset</i>, <i>n</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>n</i> must be a non-negative integer value.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function always returns true. This function matches up to the first <i>n</i> characters of <i>str</i> are members of <i>cset</i>. The character at position <i>n+1</i> may or may not be a member of <i>cset</i>.</p> <p>If this function returns true, it copies the matching (up to <i>n</i>) characters of <i>str</i> to <i>matched</i> and copies any remaining characters to <i>rem</i> (assuming <i>rem</i> and <i>matched</i> are present).</p>
@nOrMoreCset	<p>@nOrMoreCset(<i>str</i>, <i>cset</i>, <i>n</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>n</i> must be a non-negative integer value.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function returns true if it matches at least <i>n</i> characters from <i>str</i> in <i>cset</i>.</p> <p>If <i>rem</i> and <i>matched</i> appear in the parameter list, this function will copy all characters it matches to <i>matched</i> and copy any remaining characters into <i>rem</i>.</p>

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@nToMCset	<p>@nToMCset(<i>str</i>, <i>cset</i>, <i>n</i>, <i>m</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>n</i> and <i>m</i> must be non-negative integer values.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function returns true if there are at least <i>n</i> characters in <i>cset</i> at the beginning of <i>str</i>. If <i>rem</i> and <i>matched</i> are present, this function will copy all the characters it matches (up to the <i>m</i>th position) into the <i>matched</i> string and copy any remaining characters into the <i>rem</i> string. The character at position <i>m+1</i> may or may not be a member of <i>cset</i>.</p>
@exactlyNToMCset	<p>@ExactlyNToMCset(<i>str</i>, <i>cset</i>, <i>n</i>, <i>m</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>n</i> and <i>m</i> must be non-negative integer values.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function returns true if there are at least <i>n</i> characters in <i>cset</i> at the beginning of <i>str</i>. If <i>rem</i> and <i>matched</i> are present, this function will copy all the characters it matches (up to the <i>m</i>th position) into the <i>matched</i> string and copy any remaining characters into the <i>rem</i> string. If this function matches <i>m</i> characters, the character at position <i>m+1</i> must not be a member of <i>cset</i> or else this function will return false.</p>
@zeroOrMoreCset	<p>@zeroOrMoreCset(<i>str</i>, <i>cset</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function always returns true. If <i>rem</i> and <i>matched</i> are present, this function will copy all characters from the beginning of <i>str</i> that are members of <i>cset</i> to the <i>matched</i> string. It will copy all remaining characters to the <i>rem</i> string.</p>

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@oneOrMoreCset	<p>@OneOrMoreCset(<i>str</i>, <i>cset</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function returns true if the first character of <i>set</i> is a member of <i>cset</i>. If <i>rem</i> and <i>matched</i> are present, this function will copy all characters from the beginning of <i>str</i> that are members of <i>cset</i> to the <i>matched</i> string. It will copy all remaining characters to the <i>rem</i> string.</p>

H.6.2 String/Character Pattern Matching Functions

Though not always as useful as the character set pattern matching functions, the HLA compile-time character matching functions are more efficient than the character set routines when matching single characters.

Table 7: HLA Compile-time Character Matching Functions

Function	Parameters	Description
@peekChar	<p>@peekChar(<i>str</i>, <i>char</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>char</i> must be a character expression.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function returns true if the first character of <i>str</i> is equal to <i>char</i>. If <i>rem</i> and <i>matched</i> are present and this function returns true, it also returns <i>str</i> in <i>rem</i> and <i>char</i> in <i>matched</i>.</p>

Table 7: HLA Compile-time Character Matching Functions

Function	Parameters	Description
@oneChar	<p>@oneChar(<i>str</i>, <i>char</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>char</i> must be a character expression.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function returns true if the first character of <i>str</i> is equal to <i>char</i>. If <i>rem</i> and <i>matched</i> are present and this function returns true, it also returns all the characters <i>str</i> beyond the first character in <i>rem</i> and <i>char</i> in <i>matched</i>.</p>
@uptoChar	<p>@uptoChar(<i>str</i>, <i>char</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>char</i> must be a character expression.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function returns true if the <i>char</i> exists somewhere in <i>str</i>. If <i>rem</i> and <i>matched</i> are present and this function returns true, it also returns, in <i>rem</i>, all the characters in <i>str</i> starting with the first instance of <i>char</i>. It also returns all the characters up to (but not including) the first instance of <i>char</i> in <i>matched</i>.</p>
@zeroOrOneChar	<p>@zeroOrOneChar(<i>str</i>, <i>char</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>char</i> must be a character expression.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function always returns true. If <i>rem</i> and <i>matched</i> are present, it sets <i>matched</i> to the matched string (i.e., an empty string or the single character <i>char</i>) and it sets <i>rem</i> to the characters after the matched character value (the whole string if the first character of <i>str</i> is not equal to <i>char</i>).</p>

Table 7: HLA Compile-time Character Matching Functions

Function	Parameters	Description
@zeroOrMoreChar	<p>@zeroOrMoreChar(<i>str</i>, <i>char</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>char</i> must be a character expression.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function always returns true. If the first character of <i>str</i> does not match <i>char</i>, then this function returns the empty string in <i>matched</i> and it returns <i>str</i> in <i>rem</i>. If <i>str</i> begins with a sequence of characters all equal to <i>char</i>, then this function returns that sequence in <i>matched</i> and it returns the remaining characters in <i>rem</i>.</p>
@oneOrMoreChar	<p>@oneOrMoreChar(<i>str</i>, <i>char</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>char</i> must be a character expression.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function returns true if the first character in <i>str</i> is equal to <i>char</i>; otherwise it returns false.</p> <p>If this function returns true, it copies all leading occurrences of <i>char</i> into <i>matched</i> and any remaining characters from <i>str</i> into <i>rem</i>.</p>
@exactlyNChar	<p>@exactlyNChar(<i>str</i>, <i>char</i>, <i>n</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>n</i> must be a non-negative integer value.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function returns true if the first <i>n</i> characters of <i>str</i> all match <i>char</i>. The $n+1^{th}$ character must not be equal to <i>char</i>.</p> <p>If this function returns true, it returns a string of <i>n</i> copies of <i>char</i> in <i>matched</i> and all remaining characters (position <i>n</i> and beyond) in <i>rem</i>.</p>

Table 7: HLA Compile-time Character Matching Functions

Function	Parameters	Description
@firstNChar	<p>@firstNChar(<i>str</i>, <i>char</i>, <i>n</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>n</i> must be a non-negative integer value.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function returns true if the first <i>n</i> characters of <i>str</i> all match <i>char</i>. The $n+1^{th}$ character may or may not be equal to <i>char</i>.</p> <p>If this function returns true, it returns a string of <i>n</i> copies of <i>char</i> in <i>matched</i> and all remaining characters (position <i>n</i> and beyond) in <i>rem</i>.</p>
@nOrLessChar	<p>@nOrLessChar(<i>str</i>, <i>char</i>, <i>n</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>n</i> must be a non-negative integer value.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function returns true as long as <i>str</i> begins with <i>n</i> or fewer (including zero) copies of <i>char</i>. It fails if <i>str</i> begins with more than <i>n</i> copies of <i>char</i>.</p> <p>If this function returns true, it returns a string, in <i>matched</i>, containing all copies of <i>char</i> that appear at the beginning of <i>str</i>. It returns all remaining characters in <i>rem</i>.</p>
@nOrMoreChar	<p>@nOrMoreChar(<i>str</i>, <i>char</i>, <i>n</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>n</i> must be a non-negative integer value.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function returns true if <i>str</i> begins with at least <i>n</i> copies of <i>char</i>. It returns false otherwise.</p> <p>If this function returns true, then it returns the leading characters that match <i>char</i> in <i>matched</i> and all following characters in <i>rem</i>.</p>

Table 7: HLA Compile-time Character Matching Functions

Function	Parameters	Description
@nToMChar	<p>@nToMChar(<i>str</i>, <i>char</i>, <i>n</i>, <i>m</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>n</i> and <i>m</i> must be non-negative integer values.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function matches any string that has at least <i>n</i> copies of <i>char</i> at the beginning of <i>str</i>. It will match up to <i>m</i> copies of <i>char</i>. The character at position <i>m+1</i> may be equal to <i>char</i>, but this function will not match that character. If this function returns true, it returns the string of matched characters in <i>matched</i> and any remaining characters in <i>rem</i>.</p>
@exactlyNToM-Char	<p>@exactlyNToMChar(<i>str</i>, <i>char</i>, <i>n</i>, <i>m</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression. <i>cset</i> must be a character set expression.</p> <p><i>n</i> and <i>m</i> must be non-negative integer values.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function matches any string that has at least <i>n</i> copies of <i>char</i> at the beginning of <i>str</i>. It will match up to <i>m</i> copies of <i>char</i>. The character at position <i>m+1</i> must not be equal to <i>char</i>. If this function returns true, it returns the string of matched characters in <i>matched</i> and any remaining characters in <i>rem</i>.</p>

H.6.3 String/Case Insensitive Character Pattern Matching Functions

HLA provides two sets of character matching routines: the previous section described the standard character matching routines, this section presents the case insensitive versions of those same routines.

Table 8: HLA Compile-time Case Insensitive Character Matching Functions

Function	Parameters	Description
@peekiChar	@peekiChar(<i>str</i> , <i>char</i> , <i>rem</i> , <i>matched</i>) See @peekChar for details.	Case insensitive version of @peekChar. See @peekChar for details.
@oneiChar	@oneiChar(<i>str</i> , <i>char</i> , <i>rem</i> , <i>matched</i>) See @oneChar for details.	Case insensitive version of @oneChar. See @oneChar for details.
@uptoiChar	@uptoiChar(<i>str</i> , <i>char</i> , <i>rem</i> , <i>matched</i>) See @uptoChar for details.	Case insensitive version of @uptoChar. See @uptoChar for details.
@zeroOrOneiChar	@zeroOrOneiChar(<i>str</i> , <i>char</i> , <i>rem</i> , <i>matched</i>) See @zeroOrOneChar for details.	Case insensitive version of @zeroOrOneChar. See @zeroOrOneChar for details.
@zeroOrMorei-Char	@zeroOrMoreiChar(<i>str</i> , <i>char</i> , <i>rem</i> , <i>matched</i>) See @zeroOrMoreChar for details.	Case insensitive version of @zeroOrMoreChar. See @zeroOrMoreChar for details.
@oneOrMoreiChar	@OneOrMoreiChar(<i>str</i> , <i>char</i> , <i>rem</i> , <i>matched</i>) See @OneOrMoreChar for details.	Case insensitive version of @OneOrMoreChar. See @OneOrMoreChar for details.
@exactlyNiChar	@exactlyNiChar(<i>str</i> , <i>char</i> , <i>n</i> , <i>rem</i> , <i>matched</i>) See @exactlyNChar for details.	Case insensitive version of @exactlyNChar. See @exactlyNChar for details.
@firstNiChar	@firstNiChar(<i>str</i> , <i>char</i> , <i>n</i> , <i>rem</i> , <i>matched</i>) See @firstNChar for details.	Case insensitive version of @firstNChar. See @firstNChar for details.
@nOrLessiChar	@nOrLessiChar(<i>str</i> , <i>char</i> , <i>n</i> , <i>rem</i> , <i>matched</i>) See @nOrLessChar for details.	Case insensitive version of @nOrLessChar. See @nOrLessChar for details.
@nOrMoreiChar	@nOrMoreiChar(<i>str</i> , <i>char</i> , <i>n</i> , <i>rem</i> , <i>matched</i>) See @nOrMoreChar for details.	Case insensitive version of @nOrMoreChar. See @nOrMoreChar for details.

Table 8: HLA Compile-time Case Insensitive Character Matching Functions

Function	Parameters	Description
@nToMiChar	@nToMiChar(<i>str, char, n, m, rem, matched</i>) See @nToMChar for details.	Case insensitive version of @nToMChar. See @nToMChar for details.
@exactlyNToMiChar	@exactlyNToMiChar(<i>str, char, n, m, rem, matched</i>) See @exactlyNToMChar for details.	Case insensitive version of @exactlyNToMChar. See @exactlyNToMChar for details.

H.6.4 String/String Pattern Matching Functions

Another set of popular pattern matching routines in the compile-time function set are the string matching routines. These routines check to see if a string begins with some specified sequence of characters. Next to the character set matching functions, these are probably the most commonly used pattern matching functions.

Table 9: Compile-Time String Matching Functions

Function	Parameters	Description
@matchStr	@matchStr(<i>str, tstStr, rem, matched</i>) <i>str</i> must be a string expression. <i>tstStr</i> must be a string expression. <i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if <i>str</i> begins with the characters found in <i>tstStr</i> . If this function returns true, it also returns <i>tstStr</i> in <i>matched</i> and all characters in <i>str</i> following the <i>tstStr</i> characters in <i>rem</i> .
@matchiStr	@matchiStr(<i>str, tstStr, rem, matched</i>) <i>Same parameters as matchStr, see that function for details.</i>	This is a case insensitive version of @matchStr.

Table 9: Compile-Time String Matching Functions

Function	Parameters	Description
@uptoStr	@uptoStr(<i>str</i> , <i>tstStr</i> , <i>rem</i> , <i>matched</i>) <i>Same parameters as matchStr; see that function for details.</i>	This function returns true if <i>tstStr</i> appears somewhere within <i>str</i> , it returns false otherwise. If this function returns true, then it returns all characters up to, but not including, the characters from <i>tstStr</i> in <i>matched</i> . It returns all following characters (including the <i>tstStr</i> substring) in <i>rem</i> .
@uptoiStr	@uptoiStr(<i>str</i> , <i>tstStr</i> , <i>rem</i> , <i>matched</i>) <i>Same parameters as matchStr; see that function for details.</i>	This is a case insensitive version of @uptoStr.
@matchToStr	@matchToStr(<i>str</i> , <i>tstStr</i> , <i>rem</i> , <i>matched</i>) <i>Same parameters as matchStr; see that function for details.</i>	This function is similar to @uptoStr except if it returns true it will copy all characters up to and including <i>tstStr</i> to <i>matched</i> and all following characters to <i>rem</i> .
@matchToiStr	@matchToiStr(<i>str</i> , <i>tstStr</i> , <i>rem</i> , <i>matched</i>) <i>Same parameters as matchStr; see that function for details.</i>	This is a case insensitive version of @matchToStr.

H.6.5 String/Misc Pattern Matching Functions

This last group of compile-time pattern matching functions check for certain special types of strings, such as HLA identifiers, numeric constants, whitespace, and the end of the string.

Table 10: Miscellaneous Compile-time Pattern Matching Functions

Function	Parameters	Description
@matchID	<p>@matchID(<i>str</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The types of <i>rem</i> and <i>matched</i> are irrelevant but they must be VAL objects. This function stores a result into these two objects.</p>	<p>This function returns true if the first sequence of characters in <i>str</i> match the definition of an HLA identifier.</p> <p>An HLA identifier is any sequence of characters that begins with an underscore or an alphabetic character that is followed by zero or more underscore or alphanumeric characters.</p> <p>If this function returns true, it copies the identifier to <i>matched</i> and all following characters to <i>rem</i>.</p>
@matchIntConst	<p>@matchIntConst(<i>str</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The type of <i>rem</i> is irrelevant but it must be a VAL object. If <i>matched</i> is present, it must be an <i>int32</i> or <i>uns32</i> object.</p> <p>Note: unlike most pattern matching functions, <i>matched</i> is not a string object.</p>	<p>This function returns true if the leading characters of <i>str</i> are decimal digits (as per HLA, underscores are legal in the interior of the integer string).</p> <p>If this function returns true, it converts the matched integer string to integer form and stores this value in <i>matched</i>. This function returns the remaining characters after the numeric digits in <i>rem</i>.</p>

Table 10: Miscellaneous Compile-time Pattern Matching Functions

Function	Parameters	Description
@matchRealConst	<p>@matchRealConst(<i>str</i>, <i>rem</i>, <i>matched</i>)</p> <p><i>str</i> must be a string expression.</p> <p><i>rem</i> and <i>matched</i> are optional arguments (both are optional, but if <i>matched</i> is present, <i>rem</i> must also be present). The type of <i>rem</i> is irrelevant but it must be a VAL object. If <i>matched</i> is present, it must be an <i>real80</i> object.</p> <p>Note: unlike most pattern matching functions, <i>matched</i> is not a string object.</p>	<p>This function returns true if the leading characters of <i>str</i> correspond to the HLA definition of a real literal constant.</p> <p>If this function returns true, it also returns the remaining characters in <i>rem</i> and it converts the real string to <i>real80</i> format and stores this value in <i>matched</i>.</p>
@matchNumericConst	<p>@matchNumericConst(<i>str</i>, <i>rem</i>, <i>matched</i>)</p> <p>Same parameters as @matchIntConst or @matchRealConst; see those functions for details.</p>	<p>This is a combination of @matchIntConst and @matchRealConst. If this function matches a string at the beginning of <i>str</i> that is a legal numeric constant, it will convert that value to numeric form (<i>int32</i> or <i>real80</i>) and store the value into <i>matched</i>. This function returns any following characters in <i>rem</i>.</p>
@matchStrConst	<p>@matchStrConst(<i>str</i>, <i>rem</i>, <i>matched</i>)</p> <p>Same parameters as @matchID; see @matchID for details.</p>	<p>This function returns true if <i>str</i> begins with an HLA compatible literal string constant. If this function matches such a constant, it will store the matched string, minus the delimiting quotes, into the <i>matched</i> variable. It will also store any following characters into <i>rem</i>.</p>

Table 10: Miscellaneous Compile-time Pattern Matching Functions

Function	Parameters	Description
@zeroOrMoreWS	<p>@zeroOrMoreWS(str, rem)</p> <p><i>str</i> must be a string expression.</p> <p><i>rem</i> is an optional argument. The type of <i>rem</i> is irrelevant but it must be a VAL object.</p>	<p>This function always returns true. It matches zero or more "whitespace" characters at the beginning of <i>str</i>.</p> <p>Whitespace includes spaces, newlines, tabs, and certain other special characters.</p> <p>Note that this function does not return the matched string. To return matched whitespace characters, use @zeroOrMoreCset.</p>
@oneOrMoreWS	<p>@oneOrMoreWS(str, rem)</p> <p><i>str</i> must be a string expression.</p> <p><i>rem</i> is an optional argument. The type of <i>rem</i> is irrelevant but it must be a VAL object. This function stores a result into this object.</p>	<p>This function returns true if it matches at least one whitespace character. If it returns true, it also returns any characters following the leading whitespace characters in the <i>rem</i> variable.</p>
@wsOrEOS	<p>@wsOrEOS(str, rem)</p> <p><i>str</i> must be a string expression.</p> <p><i>rem</i> is an optional argument. The type of <i>rem</i> is irrelevant but it must be a VAL object. This function stores a result into this object.</p>	<p>This function succeeds if there is whitespace at the beginning of <i>str</i> or if <i>str</i> is the empty string. If it succeeds and there is leading whitespace, this function returns the remaining characters in <i>rem</i>. If this function succeeds and the string was empty, this function returns the empty string in <i>rem</i>. This function fails if the string is not empty and it begins with non-whitespace characters.</p>

Table 10: Miscellaneous Compile-time Pattern Matching Functions

Function	Parameters	Description
@wsThenEOS	@wsThenEOS(<i>str</i>) <i>str</i> must be a string expression.	This function returns true if <i>str</i> contains zero or more whitespace characters followed by the end of the string. If fails if there are any other characters in the string.
@peekWS	@peekWS(<i>str</i> , <i>rem</i>) <i>str</i> must be a string expression. <i>rem</i> is an optional argument. The type of <i>rem</i> is irrelevant but it must be a VAL object. This function stores a result into this object.	This function returns true if the next character in <i>str</i> is a whitespace character. This function returns a copy of <i>str</i> in <i>rem</i> if it is successful.
@eos	@eos(<i>str</i>) <i>str</i> must be a string expression.	This function returns true if and only if <i>str</i> is the empty string.

H.7 HLA Information and Symbol Table Functions

The symbol table functions provide access to information in HLA's internal symbol table database. These functions are particularly useful within macros to determine how to generate code for a particular macro parameter.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@name	@name(<i>identifier</i>) <i>identifier</i> must be a defined symbol in the HLA program.	This function returns a string specifying the name of the specified identifier. The name this function returns is computed after macro and text constant expansion. This function is useful mainly in macros to determine the name of a macro parameter.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@type	@type(identifier) <i>identifier</i> must be a defined symbol in the HLA program.	This function returns a <i>dword</i> constant that should be unique for any given type in the program. You can use this to compare the types of two different objects. For guaranteed uniqueness, see @typeName.
@typeName	@typeName(identifier) <i>identifier</i> must be a defined symbol in the HLA program.	This function returns a string that specifies the type of of the parameter.
@pType	@pType(identifierOrExpression) <i>identifierOrExpression</i> may be a defined symbol in the HLA program or a constant expression.	This function returns a small numeric constant that specifies the primitive type of the object. See the ptXXXXX constants in the HLA.HHF header file for the actual values this function returns.
@class	@class(identifierOrExpression) <i>identifierOrExpression</i> may be a defined symbol in the HLA program or a constant expression.	This function returns a small numeric constant that classifies the identifier or expression as to whether it is a constant, VAL, variable, parameter, static object, procedure, etc. See the cXXXX constants in the HLA.HHF header file for a list of the possible return values.
@size	@size(identifierOrExpression) <i>identifierOrExpression</i> may be a defined symbol in the HLA program or a constant expression.	This function returns the size, in bytes of the specified object.
@offset	@offset(identifier) <i>identifier</i> must be a defined VAR or parameter symbol in the HLA program.	This function returns a numeric constant providing the offset into a procedure's activation record for a VAR or parameter object.
@staticName	@staticName(identifier) <i>identifier</i> must be a defined static, procedure, method, iterator, or external symbol in the HLA program.	This function returns a string that specifies the internal name that HLA uses for the object.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@lex	@lex(identifier) <i>identifier</i> must be a defined symbol in the HLA program.	This function returns a small integer constant that specifies the static nesting level for the specified symbol. All symbols appearing in the main program have a lex level of zero. Symbols you define in procedures within the main program have a lex level of one. Higher lex level values are possible if you define procedures inside procedures.
@isExternal	@isExternal(identifier) <i>identifier</i> must be a defined symbol in the HLA program.	This function returns true if the specified identifier is an external symbol. Note that this function will return true if the symbol is defined external and a declaration for the symbol appears later in the code.
@arity	@arity(identifier) <i>identifier</i> must be a defined symbol in the HLA program.	This function returns zero if the specified symbol is not an array. It returns a small integer constant denoting the number of dimensions if the identifier is an array object.
@dim	@dim(identifier) <i>identifier</i> must be a defined symbol in the HLA program.	If the specified identifier is an array object, this function returns an array constant with one element for each dimension in the array. Each element of this array constant specifies the number of elements for each dimension of the array. If the identifier is not an array object, this function returns an array with a single element and that element's value will be zero.
@elements	@elements(identifier) <i>identifier</i> must be a defined symbol in the HLA program.	This function returns the number of elements in an array object. If the specified identifier is not an array object, this function returns zero. For multi-dimensional arrays, this function returns the product of each of the dimensions.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@elementSize	@elementSize(identifier) <i>identifier</i> must be a defined symbol in the HLA program.	This function returns the size, in bytes, of an element of the specified array.
@defined	@defined(identifier) <i>identifier</i> must be a defined symbol in the HLA program.	This function returns true if the specified symbol is defined prior to that point in the program.
@pClass	@pClass(identifier) <i>identifier</i> must be a parameter in the current procedure of an HLA program.	This function returns a small integer constant specifying the parameter passing mechanism for the specified parameter. The HLA.HHF header file defines the return values for this function (see the XXXX_pc constant declarations).
@isConst	@isConst(identifierOrExpression) <i>identifierOrExpression</i> may be a defined symbol in the HLA program or a constant expression.	This function returns true if the expression is a constant expression that HLA can evaluate at that point in the program.
@isReg	@isReg(Expression) <i>Expression</i> may be arbitrary text, but it is typically a register object.	This function returns true if the operand corresponds to an 80x86 general purpose register.
@isReg8	@isReg8(Expression) <i>Expression</i> may be arbitrary text, but it is typically a register object.	This function returns true if the operand corresponds to an eight-bit 80x86 general purpose register.
@isReg16	@isReg16(Expression) <i>Expression</i> may be arbitrary text, but it is typically a register object.	This function returns true if the operand corresponds to a 16-bit 80x86 general purpose register.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@isReg32	@isReg32(Expression) <i>Expression</i> may be arbitrary text, but it is typically a register object.	This function returns true if the operand corresponds to a 32-bit 80x86 general purpose register.
@isFReg	@isFReg(Expression) <i>Expression</i> may be arbitrary text, but it is typically a register object.	This function returns true if the operand corresponds to an FPU register.
@isMem	@isMem(Expression) <i>Expression</i> may be arbitrary text, but it is typically a memory object (including various addressing modes).	This function returns true if the specified parameter corresponds to a legal 80x86 memory address.
@isClass	@isClass(Text) <i>Text</i> may be arbitrary text, but it is typically a class object.	This function returns true if the text parameter is a class name or a class object.
@isType	@isClass(Text) <i>Text</i> may be arbitrary text, but it is typically a type name.	This function returns true if the specified text is a type identifier.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@section	@section	<p>This function returns a 32-bit value that specifies which portion of the program HLA is currently processing. This information is mainly useful in macros to determine where a macro expansion is taking place. This function returns the following bit values:</p> <p>Bit 0: Currently in the CONST section. Bit 1: Currently in the VAL section. Bit 2: Currently in the TYPE section. bit 3: Currently in the VAR section. bit 4: Currently in the STATIC section. bit 5: In the READONLY section. Bit 6: In the STORAGE section. Bit 7: Currently in the DATA section.</p> <p>Bits 8-11: reserved.</p> <p>Bit 12: Processing statements in main. Bit 13: Statements in a procedure. Bit 14: Statements in a method. Bit 15: Statements in an iterator. Bit 16: Statements in a macro. Bit 17: Statements in a keyword macro. Bit 18: In a Terminator macro. Bit 19: Statements in a Thunk.</p> <p>Bits 20-22: reserved.</p> <p>Bit 23: Processing statements in a Unit. Bit 24: Statements in a Program.</p> <p>Bit 25: Processing a Record declaration. Bit 26: Processing a Union declaration. Bit 27: Processing a Class declaration. Bit 28: Processing a Namespace declaration.</p>

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@curLex	@curLex	Returns the current lex level of this statement within the program.
@curOffset	@curOffset	Returns the current offset into the activation record. This is the offset of the last VAR object declared in the current program/procedure.
@curDir	@curDir	Returns +1 if processing parameters, -1 otherwise. This corresponds to whether offsets are increasing or decreasing in an activation record during compilation. This function also returns +1 when processing fields in a record or class; it returns zero when processing fields of a union.
@addofs1st	@addofs1st	This function returns true when processing local variables, it returns false when processing parameters and record/class/union declarations.
@lastObject	@lastObject	This function returns a string containing the name of the last macro object processed.
@lineNumber	@lineNumber	This function returns an <i>uns32</i> value specifying the current line number in the file.

H.8 Compile-Time Variables

The HLA compile-time variables are special symbols that not only return values, but allow you to modify their internal values as well. You may use these symbols exactly like any VAL object in your program with the exception that you cannot change the type (generally *int32* or *boolean*) of these objects. By changing the values of these *pseudo-variables*, you can affect the way HLA generates code in your programs.

Table 12: HLA Compile-time Variables

Pseudo-Variable	Description
@parmOffset	This variable specifies the starting offset for parameters in a function. This should be eight for most procedures. If you change this value, HLA's automatic code generation for procedure calls may fail.
@localOffset	This variable specifies the starting offset for local variables in a procedure. This is typically zero. If you change this value, it will affect the offsets of all local symbols in the activation record, hence you should modify this value only if you really know what you're doing (and have a good reason for doing it).
@enumSize	This variable specifies the size (in bytes) of enum objects in your program. By default, this value is one. If you want word or dword sized enum objects you can change this variable to two or four. Other values may create problems for the compiler.
@minParmSize	This variable specifies the minimum number of bytes for each parameter. Under Windows, this should always be four.
@bound	This boolean variable, whose default value is true, controls the compilation of the BOUND instruction. If this variable contains false, HLA does not compile BOUND instructions. You can set this value to true or false throughout your program to control the emission of bound instructions.
@into	This boolean variable, whose default value is false, controls the compilation of the INTO instruction. If this variable contains false, HLA does not compile INTO instructions. You can set this value to true or false throughout your program to control the emission of INTO instructions.
@trace	Enables HLA's statement tracing facilities. See the separate appendix for details.
@exceptions	If true (the default) then HLA uses the full exception handling package in the HLA Standard Library. If false, HLA uses a truncated version. This allows programmers to write their own exception handling code.

H.9 Miscellaneous Compile-Time Functions

This last category contains various functions and objects that are quite useful in compile-time programs.

Table 13: Miscellaneous HLA Compile-time Functions and Objects

Function	Parameters	Description
@text	@text(<i>str</i>) <i>str</i> must be a string constant expression.	This function expands the string constant in-line as text, replacing this function invocation with the specified text.
@eval	@eval(<i>text</i>) <i>text</i> is an arbitrary sequence of textual characters.	This function is useable only within macro invocation parameter lists. It immediately evaluates the text object and passes the resulting text as the parameter to the macro. This provides eager evaluation capabilities for macro parameters.
@string: <i>identifier</i>	@string: <i>identifier</i> <i>identifier</i> must be a <i>text</i> constant.	This function returns a string constant corresponding to the string data in the specified identifier. It does not otherwise affect the value or type of <i>identifier</i> .
@toString: <i>identifier</i>	@string: <i>identifier</i> <i>identifier</i> must be a <i>text</i> constant.	This function converts the type of <i>identifier</i> from <i>text</i> to <i>string</i> and then returns the value of that string object.
@global: <i>identifier</i>	@string: <i>identifier</i> <i>identifier</i> must be an HLA identifier declared outside the current namespace.	@global is legal only within a namespace declaration. It provides access to identifiers outside the namespace.

Installing HLA on Your System

Appendix I

This information has been moved to Volume One, Chapter Two. Please see the HLA on-line documentation at <http://webster.cs.ucr.edu> if you require additional installation assistance.

Debugging HLA Programs

Appendix J

J.1 The @TRACE Pseudo-Variable

HLA v1.x has a few serious defects in its design. One major issue is debugging support. HLA v1.x emits MASM code that it runs through MASM in order to produce executable object files. Unfortunately, while this scheme makes the development, testing, and debugging of HLA easier, it effectively eliminates the possibility of using existing source level debugging tools to locate defects in an HLA program¹. Starting with v1.24, HLA began supporting a new feature to help you debug your programs: the “@trace” pseudo-variable. This appendix will explain the purpose of this compile-time variable and how you can use it to locate defects in your programs.

By default, the compile-time variable `@trace` contains false. You can change its value with any variation of the following statement:

```
?@trace := <<boolean constant expression>>;
```

Generally, “<<boolean constant expression>>” is either *true* or *false*, but you could use any compile-time constant expression to set `@trace`'s value.

Once you set `@trace` to true, HLA begins generating extra code in your program. In fact, before almost every executable statement HLA injects the following code:

```
_traceLine_( filename, linenumber );
```

The *filename* parameter is a string specifying the name of the current source file, the *linenumber* parameter is an uns32 value that is the current line number of the file. The `_traceLine_` procedure uses this information to display an appropriate trace value while the program is running.

HLA will automatically emit the above procedure call in between almost all instructions appearing in your program². Assuming that the `_traceLine_` procedure simply prints the filename and line number, when you run your application it will create a log of each statement it executes.

You can control the emission of the `_traceLine_` procedure calls in your program by alternately setting `@trace` to *true* or *false* throughout your code. This lets you selectively choose which portions of your code will provide trace information during program execution. This feature is very important because if you're displaying the trace information to the console, your program runs thousands, if not millions, of times slower during the trace operation. It wouldn't do to have to trace through a really long loop in order to trace through following code that you're concerned about. By setting `@trace` to *false* prior to the long loop and setting it to true immediately after the loop, you can execute the loop at full speed and then begin tracing the code you want to check once the loop completes execution.

HLA does not supply the `_traceLine_` procedure; it is your responsibility to write the code for this procedure. The following is a typical implementation:

```
procedure trace( filename:string; linenum:uns32 ); external( "_traceLine_" );
procedure trace( filename:string; linenum:uns32 ); nodisplay;
begin trace;

    pushfd();
    stdout.put( filename, ": #", linenum, nl );
    popfd();

end trace;
```

1. Of course, you can also debug the MASM output using a source level debugger, but this is not a very pleasant experience.

2. HLA does not emit this procedure call between statements that are composed and a few other miscellaneous statements. However, your programs probably get better than 95% coverage so this will be sufficient for most purposes.

This particular procedure simply prints the filename and line number each time HLA calls it. Therefore, you'll get a trace sent to the standard output device that looks something like the following:

```
t.hla: #19
t.hla: #20
t.hla: #21
t.hla: #22
etc.
```

A very important thing to note about this sample implementation of `_traceLine_` is that it preserves the FLAGS register. The `_traceLine_` procedure must preserve all registers including the flags. The example code above only preserves the FLAGS because the `stdout.put` macro always preserves all the registers. However, were this code to modify other registers or call some procedure that modifies some registers, you would want to preserve those register values as well. Remember, HLA calls this procedure between most instructions, if you do not preserve the registers and flags within this procedure, it will adversely affect the running of your program.

This is very important: the `@trace` variable must contain false while HLA is compiling your `_traceLine_` procedure. If HLA emits trace code inside the trace procedure, this will create an infinite loop via infinite recursion which will crash your program. Always make sure `@trace` is false across the compilation of your `_traceLine_` procedure.

Here's a sample program using the `@trace` variable and sample output for the program:

```
program t;
#includeOnce( "stdlib.hhf" )

procedure trace( filename:string; linenum:uns32 ); external( "_traceLine_" );

?@trace := false; // Must be false for TRACE routine.

procedure trace( filename:string; linenum:uns32 ); nodisplay;
begin trace;

    stdout.put( filename, ": #", linenum, nl );

end trace;

?@trace := true;

begin t;

    mov( 0, ecx );           // Line 22
    if( ecx == 0 ) then     // Line 23

        mov( 10, ecx );     // Line 25

    else

        mov( 5, ecx );

    endif;
    while( ecx > 0 ) do     // Line 32

        dec( ecx );        // Line 34

    endwhile;
    for( mov( 0, ecx ); ecx < 5; inc( ecx ) ) do // Line 37
```



```

#if( @defined( DEBUG )) // If DEBUG is not defined, then don't emit code.

    #if( DEBUG ) // DEBUG has to be a boolean const equal to true.

        #if( @isConst( runTimeFlag ))

            #if( runTimeFlag )

                stdout.put( msg, nl );

            #endif

        #else // runTimeFlag is a run-time variable

            if( runTimeFlag ) then

                stdout.put( msg, nl );

            endif;

        #endif // runTimeFlag is/is not Constant

    #endif // DEBUG

#endif // DEBUG is defined

#endifmacro

```

With this macro defined, you can write a statement like the following to print/not print a message at run-time:

```
myDebugMsg( BooleanValue, "This is a debug message" );
```

The *BooleanValue* expression is either a boolean constant expression, a boolean variable, or a register. If this value is true, then the program will print the message; if this value is false, the program does not print the message. Since this is a variable, you can control debugging output at run-time by changing the value of the *BooleanValue* parameter.

As *myDebugMsg* is written, you must define the boolean constant *DEBUG* and set it to true or the program will not compile the debugging statements. If *DEBUG* is a VAL object, you can actually

You can certainly expand upon this macro by providing support for a variable number of parameters. This would allow you to specify an list of values to display in the *stdout.put* macro invocation. See the chapter on Macros for more information about variable parameter lists in macros if you want to do this.

J.2 The Assert Macro

Another tool you may use to help locate defects in your programs is the assert macro. This macro is available in the “excepts.hhf” header file (included by “stdlib.hhf”). An invocation of this macro takes the following form:

```
assert( boolean_expression );
```

Note that you do not preface the macro with “except.” since the macro declaration appears outside the “except” namespace in the excepts.hhf header file. The *boolean_expression* component is any expression that is legal within an HLA HLL control statement like IF, WHILE, or REPEAT..UNTIL.

The assert macro evaluates the expression and simply returns if the expression is true. If the expression evaluates false, then this macro invocation will raise an exception (ex.AssertionFailed). By liberally sprinkling assert invocations through your code, you can test the behavior of your program and stop execution if an assertion fails (thus helping you to pinpoint problems in your code).

One problem with placing a lot of asserts throughout your code is that each assert takes up a small amount of space and execution time. Fortunately, the HLA Standard Library provides a mechanism by which you may control code generation of the assert macros. The `excepts.hhf` header file defines a VAL object, `ex.NDEBUG`, that is initialized with the value `false`. When this compile-time variable is `false`, HLA will emit the code for the assert macro; however, if you set this constant to `true`, then HLA will not emit any code for the assert macro. Therefore, you may liberally place assert macro invocations throughout your code and not worry about their effect on the final version of your program you ship; you can easily remove the impact of all assert macros in your program by sticking in a statement of the form `?ex.NDEBUG:=true;` in your source file.

Note that you may selectively turn asserts on or off by alternately placing `ex.NDEBUG:=false;` and `?ex.NDEBUG:=true;` throughout your code. This allows you to leave some important assertions active in your code, even when you ship the final version.

Since `assert` raises an exception, you may use the HLA `try..exception..endtry` statement to catch any exceptions that fail. If you do not handle a specific assertion failure, HLA will abort the program with an appropriate message that tells you the (source) file and line number where the assertion failed.

(More to come someday...)

Comparing HLA and MASM

Appendix K

To be written... Until then, check the output HLA produces to see the code that HLA emits for various statements.

HLA Code Generation for HLL Statements Appendix L

One of the principal advantages of using assembly language over high level languages is the control that assembly provides. High level languages (HLLs) represent an abstraction of the underlying hardware. Those who write HLL code give up this control in exchange for the engineering efficiencies enjoyed by HLL programmers. Some advanced HLL programmers (who have a good mastery of the underlying machine architecture) are capable of writing fairly efficient programs by recognizing what the compiler does with various high level control constructs and choosing the appropriate construct to emit the machine code they want. While this “low-level programming in a high level language” does leave the programmer at the mercy of the compiler-writer, it does provide a mechanism whereby HLL programmers can write more efficient code by choosing those HLL constructs that compile into efficient machine code.

Although the High Level Assembler (HLA) allows a programmer to work at a very low level, HLA also provides structured high-level control constructs that let assembly programmers use higher-level code to help make their assembly code more readable. Those assembly language programmers who need (or want) to exercise maximum control over their programs will probably want to avoid using these statements since they tend to obscure what is happening at a really low level. At the other extreme, those who would always use these high-level control structures might question if they really want to use assembly language in their applications; after all, if they’re writing high level code, perhaps they should use a high level language and take advantage of optimizing technology and other fancy features found in modern compilers. Between these two extremes lies the typical assembly language programmer. The one who realizes that most code doesn’t need to be super-efficient and is more interested in productively producing lots of software rather than worrying about how many CPU cycles the one-time initialization code is going to consume. HLA is perfect for this type of programmer because it lets you work at a high level of abstraction when writing code whose performance isn’t an issue and it lets you work at a low level of abstraction when working on code that requires special attention.

Between code whose performance doesn’t matter and code whose performance is critical lies a big gray region: code that should be reasonably fast but speed isn’t the number one priority. Such code needs to be reasonably readable, maintainable, and as free of defects as possible. In other words, code that is a good candidate for using high level control and data structures if their use is reasonably efficient.

Unlike various HLL compilers, HLA does not (yet!) attempt to optimize the code that you write. This puts HLA at a disadvantage: it relies on the optimizer between your ears rather than the one supplied with the compiler. If you write sloppy high level code in HLA then a HLL version of the same program will probably be more efficient if it is compiled with a decent HLL compiler. For code where performance matters, this can be a disturbing revelation (you took the time and bother to write the code in assembly but an equivalent C/C++ program is faster). The purpose of this appendix is to describe HLA’s code generation in detail so you can intelligently choose when to use HLA’s high level features and when you should stick with low-level assembly language.

L.1 The HLA Standard Library

The HLA Standard Library was designed to make learning assembly language programming easy for beginning programmers. Although the code in the library isn’t terrible, very little effort was made to write top-performing code in the library. At some point in the future this may change as work on the library progresses, but if you’re looking to write very high-performance code you should probably avoid calling routines in the HLA Standard Library from (speed) critical sections of your program.

Don’t get the impression from the previous paragraph that HLA’s Standard Library contains a bunch of slow-poke routines, however. Many of the HLA Standard Library routines use decent algorithms and data structures so they perform quite well in typical situations. For example, the HLA string format is far more efficient than strings in C/C++. The world’s best C/C++ *strlen* routine is almost always going to be slower than HLA *strlen* function. This is because HLA uses a better definition for string data than C/C++, it has little to do with the actual implementation of the *strlen* code. This is not to say that HLA’s *strlen* routine cannot be improved; but the routine is very fast already.

One problem with using the HLA Standard Library is the frame of mind it fosters during the development of a program. The HLA Standard Library is strongly influenced by the C/C++ Standard Library and libraries common in other high level languages. While the HLA Standard Library is a wonderful tool that can help you write assembly code faster than ever before, it also encourages you to think at a higher level. As any expert assembly language programmer can tell you, the real benefits of using assembly language occur only when you “think in assembly” rather than in a high level language. No matter how efficient the routines in the Standard Library happen to be, if you’re “writing C++ programs with MOV instructions” the result is going to be little better than writing the code in C++ to begin with.

One unfortunate aspect of the HLA Standard Library is that it encourages you to think at a higher level and you’ll often miss a far more efficient low-level solution as a result. A good example is the set of string routines in the HLA Standard Library. If you use those routines, even if they were written as efficiently as possible, you may not be writing the fastest possible program you can because you’ve limited your thinking to string objects which are a higher level abstraction. If you did not have the HLA Standard Library laying around and you had to do all the character string manipulation yourself, you might choose to treat the objects as character arrays in memory. This change of perspective can produce dramatic performance improvement under certain circumstances.

The bottom line is this: the HLA Standard Library is a wonderful collection of routines and they’re not particularly inefficient. They’re very easy and convenient to use. However, don’t let the HLA Standard Library lull you into choosing data structures or algorithms that are not the most appropriate for a given section of your program.

L.2 Compiling to MASM Code -- The Final Word

The remainder of this document will discuss, in general, how HLA translates various HLL-style statements into assembly code. Sometimes a general discussion may not provide specific answers you need about HLA’s code generation capabilities. Should you have a specific question about how HLA generates code with respect to a given code sequence, you can always run the compiler and observe the output it produces. To do this, it is best to create a simple program that contains only the construct you wish to study and compile that program to assembly code. For example, consider the following very simple HLA program:

```
program t;
begin t;
    if( eax = 0 ) then
        mov( 1, eax );
    endif;
end t;
```

If you compile this program using the command window prompt “hla -s t.hla” then HLA produces a (MASM) file similar to the following¹:

```
if      @Version lt 612
.586
else
.686
.mmx
.xmm
```

1. Because the code generator in HLA is changing all the time, this file may not reflect an accurate compilation of the above HLA code. However, the concepts will be the same.

```

endif
.model flat, syscall
offset32 equ <offset flat:>
assume fs:nothing
?ExceptionPtr equ <(dword ptr fs:[0])>
externdef ??HWexcept:near32
externdef ??Raise:near32

std_output_hndl equ -11

externdef __imp__ExitProcess@4:dword
externdef __imp__GetStdHandle@4:dword
externdef __imp__WriteFile@20:dword

cseg segment page public 'code'
cseg ends
readonly segment page public 'data'
readonly ends
strings segment page public 'data'
strings ends
dseg segment page public 'data'
dseg ends
bssseg segment page public 'data'
bssseg ends

strings segment page public 'data'

?dfltmsg byte "Unhandled exception error.",13,10
?dfltmsgsize equ 34
?absmsg byte "Attempted call of abstract procedure or method.",13,10
?absmsgsize equ 55
strings ends
dseg segment page public 'data'
?dfmwritten word 0
?dfmStdOut dword 0

public ?MainPgmCoroutine
?MainPgmCoroutine byte 0 dup (?)
dword ?MainPgmVMT
dword 0 ;CurrentSP
dword 0 ;Pointer to stack
dword 0 ;ExceptionContext
dword 0 ;Pointer to last caller
?MainPgmVMT dword ?QuitMain
dseg ends
cseg segment page public 'code'

?QuitMain proc near32
pushd 1
call dword ptr __imp__ExitProcess@4

?QuitMain endp

cseg ends

```

Appendix L

```

cseg          segment page public 'code'

??DfltExHndlr proc    near32

                pushd  std_output_hndl
                call   __imp__GetStdHandle@4
                mov    ?dfmStdOut, eax
                pushd  0          ;lpOverlapped
                pushd  offset32 ?dfmwritten      ;BytesWritten
                pushd  ?dfltmsgsize      ;nNumberOfBytesToWrite
                pushd  offset32 ?dfltmsg      ;lpBuffer
                pushd  ?dfmStdOut        ;hFile
                call   __imp__WriteFile@20

                pushd  0
                call   dword ptr __imp__ExitProcess@4

??DfltExHndlr endp

                public ??Raise
??Raise proc   near32
                jmp    ??DfltExHndlr
??Raise endp

                public ??HWexcept
??HWexcept    proc   near32
                mov    eax, 1
                ret
??HWexcept    endp

?abstract     proc   near32

                pushd  std_output_hndl
                call   __imp__GetStdHandle@4
                mov    ?dfmStdOut, eax
                pushd  0          ;lpOverlapped
                pushd  offset32 ?dfmwritten      ;BytesWritten
                pushd  ?absmsgsize      ;nNumberOfBytesToWrite
                pushd  offset32 ?absmsg      ;lpBuffer
                pushd  ?dfmStdOut        ;hFile
                call   __imp__WriteFile@20

                pushd  0
                call   dword ptr __imp__ExitProcess@4

?abstract     endp

                public ?HLAMain
?HLAMain      proc   near32

; Set up the Structured Exception Handler record
; for this program.

                push  offset32 ??DfltExHndlr
                push  ebp

```

```

        push    offset32 ?MainPgmCoroutine
        push    offset32 ??HWexcept
        push    ?ExceptionPtr
        mov     ?ExceptionPtr, esp
        mov     dword ptr ?MainPgmCoroutine+12, esp

        pushd   0                ;No Dynamic Link.
        mov     ebp, esp          ;Pointer to Main's locals
        push    ebp              ;Main's display.
        mov     [ebp+16], esp
        cmp     eax, 0
        jne    ?1_false
        mov     eax, 1
?1_false:
        push    0
        call   dword ptr __imp__ExitProcess@4
?HLAMain
cseg    endp
        ends
        end

```

The code of interest in this example is at the very end, after the comment “;Main’s display” appears in the text. The actual code sequence that corresponds to the IF statement in the main program is the following:

```

        cmp     eax, 0
        jne    ?1_false
        mov     eax, 1
?1_false:

```

Note: you can verify that this is the code emitted by the IF statement by simply removing the IF, recompiling, and comparing the two assembly outputs. You’ll find that the only difference between the two assembly output files is the four lines above. Another way to “prove” that this is the code sequence emitted by the HLA IF statement is to insert some comments into the assembly output file using HLA’s #ASM..#ENDASM directives. Consider the following modification to the “t.hla” source file:

```

program t;

begin t;

    #asm
    ; Start of IF statement:
    #endasm

    if( eax = 0 ) then

        mov( 1, eax );

    endif;

    #asm
    ; End if IF statement.
    #endasm

```

```
end t;
```

HLA's #asm directive tells the compiler to simply emit everything between the #asm and #endasm keywords directly to the assembly output file. In this example the HLA program uses these directives to emit a pair of comments that will bracket the code of interest in the output file. Compiling this to assembly code (and stripping out the irrelevant stuff before the HLA main program) yields the following:

```

                public  ?HLAMain
?HLAMain        proc    near32

; Set up the Structured Exception Handler record
; for this program.

                push    offset32 ??DfltExHndlr
                push    ebp
                push    offset32 ?MainPgmCoroutine
                push    offset32 ??HWexcept
                push    ?ExceptionPtr
                mov     ?ExceptionPtr, esp
                mov     dword ptr ?MainPgmCoroutine+12, esp

                pushd   0                ;No Dynamic Link.
                mov     ebp, esp         ;Pointer to Main's locals
                push    ebp             ;Main's display.
                mov     [ebp+16], esp

;#asm

                ; Start of IF statement:
                ;#endasm

                cmp     eax, 0
                jne     ?1_false
                mov     eax, 1

?1_false:

;#asm

                ; End if IF statement.
                ;#endasm

                push    0
                call   dword ptr __imp__ExitProcess@4
?HLAMain        endp
cseg            ends
                end
```

This technique (embedding bracketing comments into the assembly output file) is very useful if it is not possible to isolate a specific statement in its own source file when you want to see what HLA does during compilation.

L.3 The HLA `if..then..endif` Statement, Part I

Although the HLA IF statement is actually one of the more complex statements the compiler has to deal with (in terms of how it generates code), the IF statement is probably the first statement that comes to mind when something thinks about high level control structures. Furthermore, you can implement most of the other control structures if you have an IF and a GOTO (JMP) statement, so it makes sense to discuss the IF statement first. Nevertheless, there is a bit of complexity that is unnecessary at this point, so we'll begin our discussion with a simplified version of the IF statement; for this simplified version we'll not consider the ELSEIF and ELSE clauses of the IF statement.

The basic HLA IF statement uses the following syntax:

```
if( simple_boolean_expression ) then
    << statements to execute if the expression evaluates true >>
endif;
```

At the machine language level, what the compiler needs to generate is code that does the following:

```
<< Evaluate the boolean expression >>
<< Jump around the following statements if the expression was false >>
<< statements to execute if the expression evaluates true >>
<< Jump to this point if the expression was false >>
```

The example in the previous section is a good demonstration of what HLA does with a simple IF statement. As a reminder, the HLA program contained

```
if( eax = 0 ) then
    mov( 1, eax );
endif;
```

and the HLA compiler generated the following assembly language code:

```
        cmp     eax, 0
        jne     ?1_false
        mov     eax, 1
?1_false:
```

Evaluation of the boolean expression was accomplished with the single “`cmp eax, 0`” instruction. The “`jne ?1_false`” instruction jumps around the “`mov eax, 1`” instruction (which is the statement to execute if the expression evaluates true) if the expression evaluates false. Conversely, if EAX is equal to zero, then the code falls through to the MOV instruction. Hence the semantics are exactly what we want for this high level control structure.

HLA automatically generates a unique label to branch to for each IF statement. It does this properly even if you nest IF statements. Consider the following code:

```

program t;
begin t;
    if( eax > 0 ) then
        if( eax < 10 ) then
            inc( eax );
        endif;
    endif;
end t;

```

The code above generates the following assembly output:

```

                cmp     eax, 0
                jna     ?1_false
                cmp     eax, 10
                jnb     ?2_false
                inc     eax
?2_false:
?1_false:

```

As you can tell by studying this code, the INC instruction only executes if the value in EAX is greater than zero and less than ten.

Thus far, you can see that HLA's code generation isn't too bad. The code it generates for the two examples above is roughly what a good assembly language programmer would write for approximately the same semantics.

L.4 Boolean Expressions in HLA Control Structures

The HLA IF statement and, indeed, most of the HLA control structures rely upon the evaluation of a boolean expression in order to direct the flow of the program. Unlike high level languages, HLA restricts boolean expressions in control structures to some very simple forms. This was done for two reasons: (1) HLA's design frowns upon side effects like register modification in the compiled code, and (2) HLA is intended for use by beginning assembly language students; the restricted boolean expression model is closer to the low level machine architecture and it forces them to start thinking in these terms right away.

With just a few exceptions, HLA's boolean expressions are limited to what HLA can easily compile to a CMP and a condition jump instruction pair or some other simple instruction sequence. Specifically, HLA allows the following boolean expressions:

operand1 relop operand2

relop is one of:

```
= or ==      (either one, both are equivalent)
<> or !=    (either one, both are equivalent)
<
<=
>
>=
```

In the expressions above *operand₁* and *operand₂* are restricted to those operands that are legal in a CMP instruction. This is because HLA translates expressions of this form to the two instruction sequence:

```
cmp( operand1, operand2 );
jXX someLabel;
```

where “jXX” represents some condition jump whose sense is the opposite of that of the expression (e.g., “*eax > ebx*” generates a “JNA” instruction since “NA” is the opposite of “>”).

Assuming you want to compare the two operands and jump around some sequence of instructions if the relationship does not hold, HLA will generate fairly efficient code for this type of expression. One thing you should watch out for, though, is that HLA’s high level statements (e.g., IF) make it very easy to write code like the following:

```
if( i = 0 ) then
    ...
elseif( i = 1 ) then
    ...
elseif( i = 2 ) then
    ...
.
.
.
endif;
```

This code looks fairly innocuous, but the programmer who is aware of the fact that HLA emits the following would probably not use the code above:

```
    cmp( i, 0 );
    jne lbl;
    .
    .
    .
lbl: cmp( i, 1 );
    jne lbl2;
    .
    .
    .
lbl2: cmp( i, 2 );
```

·
·
·

A good assembly language programmer would realize that it's much better to load the variable "i" into a register and compare the register in the chain of CMP instructions rather than compare the variable each time. The high level syntax slightly obscures this problem; just one thing to be aware of.

HLA's boolean expressions do not support conjunction (logical AND) and disjunction (logical OR). The HLA programmer must manually synthesize expressions involving these operators. Doing so forces the programmer to link in lower level terms, which is usually more efficient. However, there are many common expressions involving conjunction that HLA could efficiently compile into assembly language. Perhaps the most common example is a test to see if an operand is within (or outside) a range specified by two constants. In a HLL like C/C++ you would typically use an expression like "(value >= low_constant && value <= high_constant)" to test this condition. HLA allows four special boolean expressions that check to see if a register or a memory location is within a specified range. The allowable expressions take the following forms:

```
register in constant .. constant
register not in constant .. constant
```

```
memory in constant .. constant
memory not in constant .. constant
```

Here is a simple example of the first form with the code that HLA generates for the expression:

```
if( eax in 1..10 ) then
    mov( 1, ebx );
endif;
```

Resulting (MASM) assembly code:

```
        cmp     eax, 1
        jb     ?1_false
        cmp     eax, 10
        ja     ?1_false
        mov     ebx, 1
?1_false:
```

Once again, you can see that HLA generates reasonable assembly code without modifying any register values. Note that if modifying the EAX register is okay, you can write slightly better code by using the following sequence:

```
        dec     eax
        cmp     eax, 9
        ja     ?1_false
        mov     ebx, 1
?1_false:
```

While, in general, a simplification like this is not possible you should always remember how HLA generates code for the range comparisons and decide if it is appropriate for the situation.

By the way, the "not in" form of the range comparison does generate slightly different code than the form above. Consider the following:

```

if( eax not in 1..10 ) then

    mov( 1, eax );

endif;

```

HLA generates the following (MASM) assembly language code for the sequence above:

```

                                cmp     eax, 1
                                jb      ?2_true
                                cmp     eax, 10
                                jna     ?1_false
?2_true:
                                mov     eax, 1
?1_false:

```

As you can see, though the code is slightly different it is still exactly what you would probably write if you were writing the low level code yourself.

HLA also allows a limited form of the boolean expression that checks to see if a character value in an eight-bit register is a member of a character set constant or variable. These expressions use the following general syntax:

```

reg8 in CSet_Constant
reg8 in CSet_Variable

reg8 not in CSet_Constant
reg8 not in CSet_Variable

```

These forms were included in HLA because they are so similar to the range comparison syntax. However, the code they generate may not be particularly efficient so you should avoid using these expression forms if code speed and size need to be optimal. Consider the following:

```

if( al in { 'A'..'Z', 'a'..'z', '0'..'9' } ) then

    mov( 1, eax );

endif;

```

This generates the following (MASM) assembly code:

```

strings          segment page public 'data'
?1_cset          byte 00h,00h,00h,00h,00h,00h,0ffh,03h
                 byte 0feh,0ffh,0ffh,07h,0feh,0ffh,0ffh,07h
strings          ends

                 push   eax
                 movzx  eax, al
                 bt     dword ptr ?1_cset, eax
                 pop    eax
                 jnc    ?1_false
                 mov    eax, 1
?1_false:

```

This code is rather lengthy because HLA never assumes that it cannot disturb the values in the CPU registers. So right off the bat this code has to push and pop EAX since it disturbs the value in EAX. Next, HLA doesn't assume that the upper three bytes of EAX already contain zero, so it zero fills them. Finally, as you can see above, HLA has to create a 16-byte character set in memory in order to test the value in the AL register. While this is convenient, HLA does generate a lot of code and data for such a simple looking expression. Hence, you should be careful about using boolean expressions involving character sets if speed and space is important. At the very least, you could probably reduce the code above to something like:

```

movzx( charToTest, eax );
bt( eax, { 'A'..'Z', 'a'..'z', '0'..'9' } );
jnc SkipMov;
mov(1, eax );
SkipMov:

```

This generates code like the following:

```

strings      segment page public 'data'
?cset_3      byte    00h,00h,00h,00h,00h,00h,0ffh,03h
              byte    0feh,0ffh,0ffh,07h,0feh,0ffh,0ffh,07h
strings      ends

              movzx   eax, byte ptr ?1_charToTest[0] ;charToTest
              bt      dword ptr ?cset_3, eax
              jnc     ?4_SkipMov
              mov     eax, 1

?4_SkipMov:

```

As you can see, this is slightly more efficient. Fortunately, testing an eight-bit register to see if it is within some character set (other than a simple range, which the previous syntax handles quite well) is a fairly rare operation, so you generally don't have to worry about the code HLA generates for this type of boolean expression.

HLA lets you specify a register name or a memory location as the only operand of a boolean expression. For registers, HLA will use the TEST instruction to see if the register is zero or non-zero. For memory locations, HLA will use the CMP instruction to compare the memory location's value against zero. In either case, HLA will emit a JNE or JE instruction to branch around the code to skip (e.g., in an IF statement) if the result is zero or non-zero (depending on the form of the expression).

```

register
!register

memory
!memory

```

You should not use this trick as an efficient way to test for zero or not zero in your code. The resulting code is very confusing and difficult to follow. If a register or memory location appears as the sole operand of a boolean expression, that register or memory location should hold a boolean value (true or false). Do not think that "if(eax) then..." is any more efficient than "if(eax<>0) then..." because HLA will actually emit the same exact code for both statements (i.e., a TEST instruction). The second is a lot easier to understand if you're really checking to see if EAX is not zero (rather than it contains the boolean value true), hence it is always preferable even if it involves a little extra typing.

Example:

```

if( eax != 0 ) then
    mov( 1, ebx );
endif;

if( eax ) then
    mov( 2, ebx );
endif;

```

The code above generates the following assembly instruction sequence:

```

                test    eax,eax ;Test for zero/false.
                je      ?2_false
                mov     ebx, 1
?2_false:
                test    eax,eax ;Test for zero/false.
                je      ?3_false
                mov     ebx, 2
?3_false:

```

Note that the pertinent code for both sequences is identical. Hence there is never a reason to sacrifice readability for efficiency in this particular case.

The last form of boolean expression that HLA allows is a flag designation. HLA uses symbols like @c, @nc, @z, and @nz to denote the use of one of the flag settings in the CPU FLAGS register. HLA supports the use of the following flag names in a boolean expression:

```

@c, @nc, @o, @no, @z, @nz, @s, @ns,
@a, @na, @ae, @nae, @b, @nb, @be, @nbe,
@l, @nl, @g, @ne, @le, @nle, @ge, @nge,
@e, @ne

```

Whenever HLA encounters a flag name in a boolean expression, it efficiently compiles the expression into a single conditional jump instruction. So the following IF statement's expression compiles to a single instruction:

```

if( @c ) then
    << do this if the carry flag is set >>
endif;

```

The above code is completely equivalent to the sequence:

```

jnc SkipStmts;

<< do this if the carry flag is set >>

SkipStmts:

```

The former version, however, is more readable so you should use the IF form wherever practical.

L.5 The JT/JF Pseudo-Instructions

The JT (jump if true) and JF (jump if false) pseudo-instructions take a boolean expression and a label. These instructions compile into a conditional jump instruction (or sequence of instructions) that jump to the target label if the specified boolean expression evaluates false. The compilation of these two statements is almost exactly as described for boolean expressions in the previous section.

The following are a couple of examples that show the usage and code generation for these two statements.

```
lbl2:
    jt( eax > 10 ) label;
label:
    jf( ebx = 10 ) lbl2;
```

; Translated Code:

```
?2_lbl2:
    cmp eax, 10
    ja ?4_label

?4_label:

    cmp ebx, 10
    jne ?2_lbl2
```

L.6 The HLA if..then..elseif..else..endif Statement, Part II

With the discussion of boolean expressions out of the way, we can return to the discussion of the HLA IF statement and expand on the material presented earlier. There are two main topics to consider: the inclusion of the ELSEIF and ELSE clauses and the HLA hybrid IF statement. This section will discuss these additions.

The ELSE clause is the easiest option to describe, so we'll start there. Consider the following short HLA code fragment:

```
if( eax < 10 ) then

    mov( 1, ebx );

else

    mov( 0, ebx );

endif;
```

HLA's code generation algorithm emits a JMP instruction upon encountering the ELSE clause; this JMP transfers control to the first statement following the ENDIF clause. The other difference between the IF/ELSE/ENDIF and the IF/ENDIF statement is the fact that a false expression evaluation transfers control to the ELSE clause rather than to the first statement following the ENDIF. When HLA compiles the code above, it generates machine code like the following:


```

        cmp     eax, 10
        jnb    ?2_false    ;Branch to ELSE section if false

        mov     ebx, 1
        jmp    ?2_endif    ;Skip over ELSE section

; This is the else section:

?2_false:
        mov     ebx, 0
?2_endif:

```

About the only way you can improve upon HLA's code generation sequence for an IF/ELSE statement is with knowledge of how the program will operate. In some rare cases you can generate slightly better performing code by moving the ELSE section somewhere else in the program and letting the THEN section fall straight through to the statement following the ENDIF (of course, the ELSE section must jump back to the first statement after the ENDIF if you do this). This scheme will be slightly faster if the boolean expression evaluates true most of the time. Generally, though, this technique is a bit extreme.

The ELSEIF clause, just as its name suggests, has many of the attributes of an ELSE and an IF clause in the IF statement. Like the ELSE clause, the IF statement will jump to an ELSEIF clause (or the previous ELSEIF clause will jump to the current ELSEIF clause) if the previous boolean expression evaluates false. Like the IF clause, the ELSEIF clause will evaluate a boolean expression and transfer control to the following ELSEIF, ELSE, or ENDIF clause if the expression evaluates false; the code falls through to the THEN section of the ELSEIF clause if the expression evaluates true. The following examples demonstrate how HLA generates code for various forms of the IF..ELSEIF.. statement:

Single ELSEIF clause:

```

if( eax < 10 ) then
    mov( 1, ebx );
elseif( eax > 10 ) then
    mov( 0, ebx );
endif;

```

; Translated code:

```

        cmp     eax, 10
        jnb    ?2_false
        mov     ebx, 1
        jmp    ?2_endif
?2_false:
        cmp     eax, 10
        jna    ?3_false
        mov     ebx, 0
?3_false:
?2_endif:

```

Single ELSEIF clause with an ELSE clause:

Appendix L

```
if( eax < 10 ) then
    mov( 1, ebx );
elseif( eax > 10 ) then
    mov( 0, ebx );
else
    mov( 2, ebx );
endif;
```

; Converted code:

```
                cmp     eax, 10
                jnb     ?2_false
                mov     ebx, 1
                jmp     ?2_endif
?2_false:
                cmp     eax, 10
                jna     ?3_false
                mov     ebx, 0
                jmp     ?2_endif
?3_false:
                mov     ebx, 2
?2_endif:
```

IF statement with two ELSEIF clauses:

```
if( eax < 10 ) then
    mov( 1, ebx );
elseif( eax > 10 ) then
    mov( 0, ebx );
elseif( eax = 5 ) then
    mov( 2, ebx );
endif;
```

; Translated code:

```
                cmp     eax, 10
                jnb     ?2_false
                mov     ebx, 1
                jmp     ?2_endif
?2_false:
                cmp     eax, 10
                jna     ?3_false
                mov     ebx, 0
                jmp     ?2_endif
```

```
?3_false:
        mov     ebx, 2
?2_endif:
```

IF statement with two ELSEIF clauses and an ELSE clause:

```
if( eax < 10 ) then
    mov( 1, ebx );
elseif( eax > 10 ) then
    mov( 0, ebx );
elseif( eax = 5 ) then
    mov( 2, ebx );
else
    mov( 3, ebx );
endif;
```

; Translated code:

```
        cmp     eax, 10
        jnb    ?2_false
        mov     ebx, 1
        jmp    ?2_endif
?2_false:
        cmp     eax, 10
        jna    ?3_false
        mov     ebx, 0
        jmp    ?2_endif
?3_false:
        cmp     eax, 5
        jne    ?4_false
        mov     ebx, 2
        jmp    ?2_endif
?4_false:
        mov     ebx, 3
?2_endif:
```

This code generation algorithm generalizes to any number of ELSEIF clauses. If you need to see an example of an IF statement with more than two ELSEIF clauses, feel free to run a short example through the HLA compiler to see the result.

In addition to processing boolean expressions, the HLA IF statement supports a hybrid syntax that lets you combine the structured nature of the IF statement with the unstructured nature of typical assembly language control flow. The hybrid form gives you almost complete control over the code generation process without completely sacrificing the readability of an IF statement. The following is a typical example of this form of the IF statement:

```

if
  ({
    cmp( eax, 10 );
    jna false;
  }) then

    mov( 0, eax );

endif;

```

; The above generates the following assembly code:

```

                cmp     eax, 10
                jna     ?2_false
?2_true:
                mov     eax, 0
?2_false:

```

Of course, the hybrid IF statement fully supports ELSE and ELSEIF clauses (in fact, the IF and ELSEIF clauses can have a potpourri of hybrid or traditional boolean expression forms). The hybrid forms, since they let you specify the sequence of instructions to compile, put the issue of efficiency squarely in your lap. About the only contribution that HLA makes to the inefficiency of the program is the insertion of a JMP instruction to skip over ELSEIF and ELSE clauses.

Although the hybrid form of the IF statement lets you write very efficient code that is more readable than the traditional “compare and jump” sequence, you should keep in mind that the hybrid form is definitely more difficult to read and comprehend than the IF statement with boolean expressions. Therefore, if the HLA compiler generates reasonable code with a boolean expression then by all means use the boolean expression form; it will probably be easier to read.

L.7 The While Statement

The only difference between an IF statement and a WHILE loop is a single JMP instruction. Of course, with an IF and a JMP you can simulate most control structures, the WHILE loop is probably the most typical example of this. The typical translation from WHILE to IF/JMP takes the following form:

```

while( expr ) do

    << statements >>

endwhile;

// The above translates to:

label:
    if( expr ) then

        << statements >>
        jmp label;

    endif;

```

Experienced assembly language programmers know that there is a slightly more efficient implementation if it is likely that the boolean expression is true the first time the program encounters the loop. That translation takes the following form:

```

    jmp testlabel;
label:

    << statements >>

testlabel:
    JT( expr ) label; // Note: JT means jump if expression is true.

```

This form contains exactly the same number of instructions as the previous translation. The difference is that a JMP instruction was moved out of the loop so that it executes only once (rather than on each iteration of the loop). So this is slightly more efficient than the previous translation. HLA uses this conversion algorithm for WHILE loops with standard boolean expressions.

L.8 repeat..until

L.9 for..endfor

L.10 forever..endfor

L.11 break, breakif

L.12 continue, continueif

L.13 begin..end, exit, exitif

L.14 foreach..endfor

L.15 try..unprotect..exception..anyexception..endtry, raise

Editorial Note: This document is a work in progress. At some future date I will finish the sections above. Until then, use the HLA “-s” compiler option to emit MASM code and study the MASM output as described in this appendix.

