The **MOON** Processor and Assembly Language

Peter Grogono

January 1995

1 The MOON Processor

The MOON is an imaginary processor based on recent RISC architectures.[1] The architecture is similar to, but simpler than, the DLX architecture described by John Hennessy and David Patterson in their textbook.[2] This document describes the architecture, instruction set, and assembly language of the MOON processor.

1.1 Architecture

The MOON is a RISC (Reduced Instruction Set Computer). The number of different instructions is small, and individual instructions are simple. All instructions occupy one word and require one memory cycle to execute (additional time may be required for data access).

1.1.1 Processor

The processor has a few instructions that access memory and many instructions that perform operations on the contents of registers. Since register access is much faster than memory access, it is important to make good use of registers to use the MOON efficiently.

There are sixteen registers, $R0, R1, \ldots, R15$. $R0$ always contains zero. There is a 32-bit *program counter* that contains the address of the next instruction to be executed.

1.1.2 Memory

A memory address is a value in the range $0, 1, \ldots, 2^{32} - 1$. The amount of memory actually available is typically less than this.

Each address identifies one 8-bit byte. The addresses $0, 4, \ldots, 4N$ are *word addresses*. The processor can load and store bytes and words.

---

[1] A MOON is similar to a SUN, but not as bright.

[2] *Computer Architecture: a Quantitative Approach*, John Hennessy and David Patterson, Morgan Kaufmann, 1990.

1.2 Terminology and Notation

A **word** has 32 bits. The bits are numbered from 0 (the most significant) to 31 (the least significant).

An **integer** is a 32-bit quantity that can be stored in a word. An integer value N satisfies the inequality $2^{-31} \leq N < 2^{31}$. Bit 0 is the **sign bit**. Integers are stored in two's-complement form.

An **address** has 32 bits. Address calculations may involve signed numbers, but the result is interpreted as an unsigned, 32-bit quantity.

A **byte** has 8 bits. The bits are numbered from 0 (the most significant) to 7 (the least significant). Up to four bytes may be stored in a word.

The name of the memory is $\mathcal{M}$. The expression $\mathcal{M}_8[K]$ denotes the byte stored at address K. The expression $\mathcal{M}_{32}[K]$ denotes the word stored at addresses $K, K+1, K+2$, and $K+3$.

An address is **legal** if the addressed byte exists. Legal addresses form a contiguous sequence $0, 1, \ldots, N$, where N depends on the processor or simulator.

An address is **aligned** if it is a multiple of 4. The aligned addresses are therefore $0, 4, 8, \ldots, 4N$.

The names $R0, R1, \ldots, R15$ denote **registers**. Each register can store a 32-bit word. We write $\mathcal{R}(i)$ to denote the contents of register Ri and $\mathcal{R}_{a..b}(i)$ to denote the contents of bits $a, a+1, \ldots, b$ of register Ri. At all times, $\mathcal{R}(0) = 0$.

The name PC denotes the **program counter**. The program counter stores the 32-bit address of the current instruction.

The symbol $\longleftarrow$ stands for data transfer, or assignment. A numeric superscript indicates the number of bits transferred. For example, $\mathcal{R}_{24..31}(3) \overset{8}{\longleftarrow} \mathcal{M}_8[1000]$ means that 8 bits (one byte) is transferred from memory location 1000 to the least significant byte of $R3$.

1.3 Instruction Set

1.3.1 Instruction Formats

Each instruction occupies one word (32 bits) of memory. There are two instruction formats, A and B, shown in Figure 1. Both formats contain a 6-bit operation code. Format A contains three register operands and Format B contains two register operands and a 16-bit data operand. Formats are not mentioned further in this document because they are not relevant to assembly language programming. In general, however, an instruction is Format B if and only if it contains a K operand.

Instructions are divided into three classes: data access, arithmetic, and control. The following subsections describe the effects of each instruction. Unless otherwise stated, PC is incremented by 4 during the execution of an instruction. That is, the operation $PC \overset{32}{\longleftarrow} PC+4$ is performed implicitly.

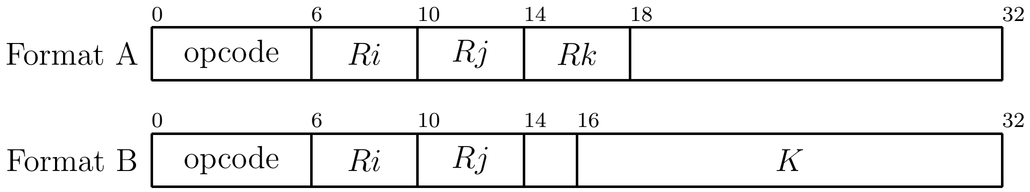

Figure 1: Instruction Formats

1.3.2 Data Access Instructions

See Figure 2. The effective address produced by the operand $K(Rj)$ is $\mathcal{R}(j)+K$. The effective address must be legal; otherwise the processor halts with an error condition. The data field K is interpreted as a signed, 16-bit quantity: $-16384 \le K < 16384$.

The effective address of a load word (lw) or store word (sw) instruction must be aligned; otherwise the processor halts with an error condition.

A lb instruction affects only the 8 low-order bits of the register; the 24 high-order bits are unaffected.

Function	Operation		Effect
Load word	lw	$Ri, K(Rj)$	$\mathcal{R}(i) \xleftarrow{32} \mathcal{M}_{32}[\mathcal{R}(j)+K]$
Load byte	lb	$Ri, K(Rj)$	$\mathcal{R}_{24..31}(i) \xleftarrow{8} \mathcal{M}_{8}[\mathcal{R}(j)+K]$
Store word	sw	$K(Rj), Ri$	$\mathcal{M}_{32}[\mathcal{R}(j)+K] \xleftarrow{32} \mathcal{R}(i)$
Store byte	sb	$K(Rj), Ri$	$\mathcal{M}_{8}[\mathcal{R}(j)+K] \xleftarrow{8} \mathcal{R}_{24..31}(i)$

Figure 2: Data Access Instructions

1.3.3 Arithmetic Instructions

Most of the arithmetic instructions have three operands. The first two operands are registers and the third is either a register (Figure 3) or an immediate operand (Figure 4) whose value is stored in the instruction. The first operand receives the result of the operation; the other operands are not affected by the operation.

The operands need not be distinct. For example, the instruction sub $R2, R2, R2$ could be used to set register 2 to zero.

The MOON processor does not detect carry or overflow in arithmetic instructions.

The "logical" operations, and, or, and not, operate on each bit of the word, with the usual interpretations.

The comparison instructions (c___) are similar to the other binary operators except that the value they store in the result register is either 1 (if the comparison yields true), or 0 (if the comparison yields false).

Function	Operation	Effect
Add	`add` Ri, Rj, Rk	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) + \mathcal{R}(k)$
Subtract	`sub` Ri, Rj, Rk	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) - \mathcal{R}(k)$
Multiply	`mul` Ri, Rj, Rk	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \times \mathcal{R}(k)$
Divide	`div` Ri, Rj, Rk	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \div \mathcal{R}(k)$
Modulus	`mod` Ri, Rj, Rk	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \bmod \mathcal{R}(k)$
And	`and` Ri, Rj, Rk	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \wedge \mathcal{R}(k)$
Or	`or` Ri, Rj, Rk	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \vee \mathcal{R}(k)$
Not	`not` Ri, Rj	$\mathcal{R}(i) \xleftarrow{32} \neg\mathcal{R}(j)$
Equal	`ceq` Ri, Rj, Rk	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) = \mathcal{R}(k)$
Not equal	`cne` Ri, Rj, Rk	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \neq \mathcal{R}(k)$
Less	`clt` Ri, Rj, Rk	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) < \mathcal{R}(k)$
Less or equal	`cle` Ri, Rj, Rk	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \leq \mathcal{R}(k)$
Greater	`cgt` Ri, Rj, Rk	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) > \mathcal{R}(k)$
Greater or equal	`cge` Ri, Rj, Rk	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \geq \mathcal{R}(k)$

Figure 3: Arithmetic Instructions with Register Operands

Function	Operation	Effect
Add immediate	`addi` Ri, Rj, K	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) + K$
Subtract immediate	`subi` Ri, Rj, K	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) - K$
Multiply immediate	`muli` Ri, Rj, K	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \times K$
Divide immediate	`divi` Ri, Rj, K	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \div K$
Modulus immediate	`modi` Ri, Rj, K	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \bmod K$
And immediate	`andi` Ri, Rj, K	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \wedge K$
Or immediate	`ori` Ri, Rj, K	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \vee K$
Equal immediate	`ceqi` Ri, Rj, K	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) = K$
Not equal immediate	`cnei` Ri, Rj, K	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \neq K$
Less immediate	`clti` Ri, Rj, K	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) < K$
Less or equal immediate	`clei` Ri, Rj, K	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \leq K$
Greater immediate	`cgti` Ri, Rj, K	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) > K$
Greater or equal immediate	`cgei` Ri, Rj, K	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(j) \geq K$
Shift left	`sl` Ri, K	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(i) \ll K$
Shift right	`sr` Ri, K	$\mathcal{R}(i) \xleftarrow{32} \mathcal{R}(i) \gg K$

Figure 4: Arithmetic Instructions with an Immediate Operand

In instructions with immediate operands (___i), the operand K is a signed, 16-bit quantity. Negative numbers are sign-extended. For example, the operand -1 is interpreted as $-1_{0..31}$, not as 65535 (its 16-bit value).

The shift instructions (s_) are useful if $0 \le K \le 31$; their effect is undefined otherwise. The operators $\ll$ and $\gg$ have the same effect as << and >> in C.

1.3.4 Input and Output Instructions

See Figure 5. The instruction getc reads one byte from *stdin*, the standard input stream. Similarly, putc writes to *stdout*, the standard output stream.

Function	Operation	Effect
Get character	getc Ri	$\mathcal{R}_{24..31}(i) \xleftarrow{8} \text{Stdin}$
Put character	putc Ri	$\text{Stdout} \xleftarrow{8} \mathcal{R}_{24..31}(i)$

Figure 5: Input and Output Instructions

1.3.5 Control Instructions

See Figure 6. The target of a branch instruction (that is, the value assigned to PC if the branch is taken) must be a legal address; otherwise the processor halts with an error condition.

The jump-and-link instructions are used to call subroutines; they store the return address in the specified register and then jump to the given location.

Function	Operation	Effect
Branch if zero	bz Ri, K	if $\mathcal{R}(i) = 0$ then $PC \xleftarrow{16} PC + K$
Branch if non-zero	bnz Ri, K	if $\mathcal{R}(i) \neq 0$ then $PC \xleftarrow{16} PC + K$
Jump	j K	$PC \xleftarrow{16} PC + K$
Jump (register)	jr Ri	$PC \xleftarrow{32} \mathcal{R}(i)$
Jump and link	jl Ri, K	$\mathcal{R}(i) \xleftarrow{32} PC + 4; PC \xleftarrow{16} PC + K$
Jump and link (register)	jlr Ri, Rj	$\mathcal{R}(i) \xleftarrow{32} PC + 4; PC \xleftarrow{16} \mathcal{R}(j)$
No-op	nop	Do nothing
Halt	hlt	Halt the processor

Figure 6: Control Instructions

1.4 Timing

The time required to run a program is measured in clock cycles and dominated by memory access. There are two paths to the memory; one is used to read instructions and the other is used to read and write data.

Before each instruction is executed, the processor must load a 32-bit word containing the instruction. This requires 10 clock cycles.

For data, the processor uses a *memory address register* (MAR) and a 32-bit *memory data register* (MDR). The processor loads an address into MAR and issues a read or write directive to the memory controller. The memory controller either obtains a word of data from the memory and stores it in MDR (read) or copies the contents of MDR to the memory.

A read or write operation requires 10 clock cycles. If the data required for a read operation is already in the MDR, the read operation requires only 1 clock cycle. For example, loading the four bytes of a word using lb instructions requires 10 clock cycles for the first byte and 1 clock cycle for each of the other three bytes, provided that no other data access intervenes.

2 MOON Assembly Language

Programs for the MOON processor are written in its *assembly language*. We use the following typographical conventions to describe the grammar of the assembly language. Figure 7 shows the grammar.

- Non-terminal symbols are written in *slanted type* and have an initial upper case letter. Examples: *Program*, *Instr*.
- Terminal symbols are written in a sans serif font. Punctuation symbols are quoted. Examples: eol, “`,`”.
- The following symbols are metasymbols of the grammar:

⟶	separates the defined symbol from the defining expression;
|	indicates alternatives;
[. . .]	enclose an optional item (zero or one occurrences);
{ . . . }	enclose a repeated item (zero or more occurrences).

Program ⟶ { *Line* } eof
Line ⟶ [*Symbol*] [*Instr* | *Directive*] [*Comment*] eol
Directive ⟶ *DirCode* [*Operand* { “`,`” *Operand* }]
Instr ⟶ *Opcode* [*Operand* { “`,`” *Operand* }]
Operand ⟶ *Register* | *Constant* [“`(`” *Register* “`)`”] | *String*
Register ⟶ (“`r`” | “`R`”) *Digit* [*Digit*]
Constant ⟶ *Number* | *Symbol*
Number ⟶ [“`+`” | “`-`”] *Digit* { *Digit* }
String ⟶ “`"`” { *Char* } “`"`”
Symbol ⟶ *Letter* { *Letter* | *Digit* }

Figure 7: Assembly Language Grammar

The symbols eof and eol denote “end of file” and “end of line”, respectively.

A *Symbol* is a string consisting of the following characters: letters, digits, and _. The first character of a symbol must not be a digit. Directives and instruction codes must not be used as symbols. Strings of the form "`R` { *Digit* }" and "`r` { *Digit* }" are not legal symbols (cf. register syntax below).

A *Comment* starts with the character "%" and continues to the end of the current line.

A *Constant* is a signed, decimal number.

The registers are "`R0`" through "`R15`". The letter "R" may be either upper or lower case.

The predefined symbol *topaddr* has $M + 1$ as its value, where M is the highest legal address. This symbol can be used to check for addressing errors or to initialize a stack or frame pointer. For example, the following instruction could be used to initialize the frame pointer:

```
addi    r14,r0,topaddr
```

The syntax of *Directive* depends on the particular directive, as shown in Figure 8.

Directive		Effect
entry		The following instruction will be the first to execute
align		The next address will be aligned
org	K	The next address will be K
dw	$K_1, K_2, \ldots$	Store words in memory
db	$K_1, K_2, \ldots$	Store bytes in memory
res	K	Reserve K bytes of memory

Figure 8: Directives

The operands of a dw directive are either symbols or integers.

The operands of a db directive are bytes (unsigned numbers in the range $0, 1, \ldots, 255$) or strings enclosed in quotes (`" ... "`). The characters in the string must be ASCII graphic characters (codes 32 through 126) only. The MOON simulator does not recognize escape characters in strings.

The operand of a res directive is a positive integer, K. The assembler requires $K < 2^{31}$, but in practice the maximum value of K will be limited by the amount of memory available.

Figure 9 shows a listing that might be generated by the assembler for a simple program. The addresses in the left column would not be included in the input file generated by a programmer or compiler.

The program begins with a directive, org, specifying that the data labelled "`message`" will be stored at address 103. Since the message is a byte string, it is in fact stored at that address, without alignment.

The processor and assembly language do not require any particular format for strings. The convention used in this program is that strings are null-terminated, as in C. An alternative would be to prefix a string with a number giving its length, as Pascal does. The bytes 13 and 10 are RETURN and LINEFEED, respectively.

The directive org 217 sets the current address to 217. The align directive changes the current address to the next word boundary, 220. The directive entry immediately before this

```
  0         org     103
103 message db      "Hello, world!", 13, 10, 0
119         org     217
217         align
220         entry                   % Start here
220         add     r2,r0,r0
224 pri     lb      r3,message(r2)  % Get next char
228         ceqi    r4,r3,0
232         bnz     r4,pr2          % Finished if zero
236         putc    r3
240         addi    r2,r2,1
244         j       pri             % Go for next char
248 pr2     addi    r2,r0,name      % Go and get reply
252         jl      r15,getname
256         hlt                     % All done!
260
260 % Subroutine to read a string
260 name    res     59              % Name buffer
319         align
320 getname getc    r3              % Read from keyboard
324         ceqi    r4,r3,10
328         bnz     r4,endget       % Finished if CR
332         sb      0(r2),r3        % Store char in buffer
336         addi    r2,r2,1
340         j       getname
344 endget  sb      0(r2),r0        % Store terminator
348         jr      r15             % Return
352
352 data    dw      1000, -35
360         dw      99, getname
```

Figure 9: An Assembly Language Program

instruction indicates that it is the first instruction to be executed.

The directive `res 59` at address 260 reserves 59 bytes of memory. The following directive, `align`, ensures that the next instruction will be aligned on a word boundary.

3 The MOON Simulator

The assembler/simulator is a program that assembles a MOON program and simulates its execution. The name of the program is `moon`. In more detail, `moon` performs the following actions.

- Read the assembly language files indicated on the command line and store them in the simulated memory. There will typically be two files, a program with subroutines and a subroutine library. The loader checks for syntax errors; if any are found, `moon` reports the errors and returns without further processing.
- By default, start executing the program at the entry point and continue simulating until a `hlt` instruction has been executed. If the user selects the trace option, the simulator enters trace mode.

The simulator is invoked by a command of the form

moon $a_1, \ldots, a_n$

in which $a_1, a_2, \ldots$ are command-line arguments. The arguments may appear in any order; Figure 10 describes the permitted values and effect of each argument. The default values of arguments are indicated by bullets between the argument and its description.

The `p` directive may be used to generate listings of selected files. For example, the command

```
moon +p main −p lib
```

would generate a listing of `main` but not of `lib`. The listing will be written to `moon.prn` unless a file name is provided with a `+o` argument. There must not be any blanks between the `o` directive and the file name.

⟨*filename*⟩		Read assembly language from the file ⟨*filename*⟩, assemble it, and store it in memory. If the filename has no extension, MOON adds `.m`.
+p		Generate a listing.
−p	•	Do not generate a listing.
+s		Display values of symbols.
−s	•	Do not display values of symbols.
−t	•	Execute the program in normal mode.
+t		Execute the program in trace mode.
−x		Do not execute the program.
+x	•	Execute the program.
+o⟨*filename*⟩		Write listings to ⟨*filename*⟩`.prn`.

Figure 10: Command-line Arguments

If moon is started in trace mode, it responds interactively to the commands described in Figure 11. Command letters may be entered in upper or lower case. The operand of a trace command, shown as $\langle m \rangle$, may be given as a number or a symbol. For example, if we were tracing the program of Figure 9, either of the commands

b320
bgetname

would set a breakpoint at address 320. The case of letters in symbol names is significant.

RETURN	Execute k instructions, where k is 10 by default but can be changed by the k command.
b	Show all breakpoints.
b$\langle m \rangle$	Set a breakpoint at memory location m.
c	Clear all breakpoints.
c$\langle m \rangle$	Clear the breakpoint at memory location m.
d	Dump memory locations $PC \pm 20$.
d$\langle m \rangle$	Dump memory locations $m \pm 20$.
i	Set PC to entry point.
i$\langle m \rangle$	Set PC to m.
h	Display a help screen.
k	Set k to its default value of 10.
k$\langle m \rangle$	Set k to m.
q	Quit the simulator.
r	Display register values.
s	Display symbol values.
x	Run until next breakpoint.
x$\langle m \rangle$	Run until $PC = m$.

Figure 11: Interactive Commands for Trace Mode

In trace mode, moon initializes the value of PC to the entry point and maintains it in accordance with instructions executed thereafter. As each instruction is executed, the interpreter displays the instruction and the values of changed registers or memory locations. The command i sets the value of PC to the given value, or to the entry point if no value is given.

The command d displays values of memory words and the command r displays values of registers. Each value is displayed as a hexadecimal number, as a string of four characters, and as a 32-bit signed integer. In the character display, non-graphic characters are shown as dots.

3.1 Programming Conventions

The MOON architecture does not restrict programmers to any particular pattern of use. The addressing mode $K(Ri)$ is suitable for addressing a stack, with Ri as the stack pointer and K as an offset computed by the compiler. Any register can be used as the link for subroutine calls. Arguments can be passed either in registers or on the stack.

3.2 Defects of the Current Simulator (`moon.c`)

The precise behaviour of the MOON simulator depends on the architecture of the processor on which it is running and also on the C compiler used to compile it.

- The order in which bytes are stored in a word is inherited from the host processor. This does not affect the execution of MOON instructions but does affect the order in which characters are displayed during tracing.
- The shift instructions (sl and sr) of the MOON processor are simulated using the C operators << and >>. The effect of >> is undefined when the most significant bit of the left operand is set. Right-shifting a negative number may yield either a positive or a negative number.
- The effect of the putc and getc instructions depends on whether the simulator is running in normal or trace mode. In normal mode, getc reads a string from the keyboard and yields one character of the string each time it is executed. In trace mode, you should enter characters one at a time, as getc asks for them.