目　录

第 1 章　处理器设计导论

本章内容综述

同多数工程任务一样，通用处理器的设计需要对很多方面进行权衡和折衷。在这一章中，将讨论处理器指令集与逻辑设计的基本原理，以及设计工程师可采用的、有助于达到设计目标的各种技术。

抽象是理解复杂计算机的基础。本章介绍计算机硬件设计师使用的抽象方法，其中最重要的是逻辑门。本章介绍了一个简单微处理器的设计，从指令集、寄存器传输级描述，一直到逻辑门设计。

精简指令集计算机（RISC，Reduced Introduction Set Computer）的思想起源于 1980 年斯坦福大学的一项处理器研究项目，而其中一些核心思想可以追溯到更早的计算机。在这一章中我们将介绍 RISC 产生的思想。这些思想也影响了 ARM 处理器设计。ARM 处理器的设计将在第 2 章介绍。

随着基于计算机的便携式产品市场的快速发展，数字电路的功耗越来越重要。在本章的末尾，我们将介绍低功耗、高性能设计的原理。

1.1　处理器体系结构和组织

所有现代通用计算机都使用**存储程序数字计算机**的原理。存储程序的概念源于 1940 年普林斯顿先进学科研究所，并在 Baby 计算机中得到实现。该计算机于 1948 年在英国的曼彻斯特大学首次运行。

50 年的发展导致处理器性能的急剧增长，而其价格也同样大幅度下降。这段时期虽然计算机的性能价格比迅速提高，但操作原理只变化了一点点。多数改进都是由电子技术的进步造成的。由真空管到分立的晶体管，再到由几个双极晶体管组成的集成电路（IC，Integrated Circuit），然后经过几代 IC 技术的发展，到今天的超大规模集成电路（VLSI，Very Large Scale Integrated），把数百万个场效应晶体管集成到同一个芯片上。随着晶体管的变小，晶体管也越来越便宜、快速和省电。在过去 30 年间，这种相互促进的局面支撑着计算机工业的进步，并至少在今后几年内还会继续这个趋势。

但是在过去 50 年内，并非所有的发展都源于电子技术的进步。有时候，在技术应用方法上的一个新见解也会作出重大的贡献。这些见解以计算机体系结构和组织方式

来描述。下面介绍一些体系结构与组织方面的术语。

计算机体系结构

计算机体系结构描述从用户角度看到的计算机。指令集、可见寄存器、存储器管理表结构和异常处理模式都是体系结构的一部分。

计算机组织

计算机组织描述用户不能看到的体系结构的实现方式。流水线结构、透明的Cache(高速缓存)、步行表(table-walking)硬件以及转换后备缓冲(TLB, Translation Look-aside Buffer)都是计算机组织的问题。

在计算机设计的这些进步中,20 世纪 60 年代早期引入的虚拟存储器、透明的Cache 和流水线等都曾是计算机发展的里程碑。作为里程碑之一,RISC 思想大幅度提高了计算机性能价格比。

什么是处理器

通用处理器是一个执行存储器中指令的有限状态机。系统的状态是由存储器中的数据和计算机本身的某些寄存器的数据定义的(参见图 1.1。以 16 进制表示的存储器地址将在 6.2 节解释)。每一条指令都规定了总状态变化的特定方式,还指定随后该执行哪一条指令。

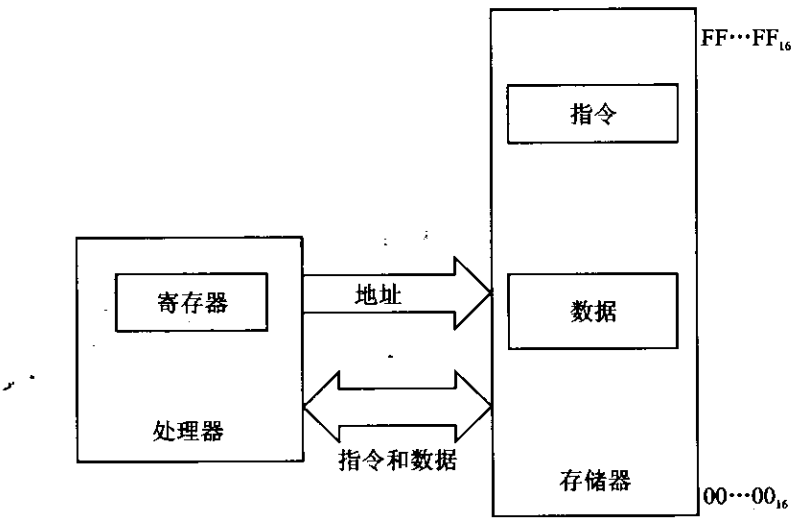


图 1.1 存储程序数字计算机中的状态

存储程序计算机

存储程序(stored-program)数字计算机把指令和数据存放在同一个存储器系统中,必要时可以将指令作为数据处理。这使处理器本身能够产生指令以供后面执行。在现在看来,在细粒度下(**自修改**代码)进行这种操作的程序是一种不良的形式,因为它

非常难于调试，但以粗粒度使用却是多数计算机操作的基础方式。只要计算机可以从磁盘装载新的程序（覆盖老的程序），然后执行，那么计算机就可以用这种方法改变自己的程序。

计算机应用

因为可编程，可以说存储程序数字计算机是**万能的**。也就是说，它可以承担任何能用适当算法描述的任务。这有时体现在，对于台式计算机，用户在不同时间运行不同的程序；而有时体现在，对于同一个处理器可用于不同的应用范围，而每种应用运行不同的固定程序。典型的应用是嵌入式产品，如移动电话、汽车发动机管理系统等。

1.2　硬件设计中的抽象

计算机是非常复杂的设备，它以非常高的速度工作。现在，一个微处理器可能由数百万个晶体管构成，每个晶体管每秒钟可以开关 1 亿次。看着一个文件在台式 PC 或工作站上卷屏，设想一下每一秒钟的动作是如何使用着数百亿次晶体管的开关动作。现在可以意识到，每一个这样的开关动作在某种意义上说都是有意设计的结果，其中没有随机或不可控的。实际上，在这些跳变中的单个错误就有可能使计算机崩溃而无法使用。那么，怎样才能设计这么复杂的系统，使它可以如此可靠地工作呢？

晶体管

答案的线索可能就在问题本身。我们用晶体管的观点描述了计算机的工作。但是什么是晶体管呢？晶体管是由仔细选择的、具有复杂电学性质的化学物质组成的古怪结构。这些电学性质只有参考量子力学理论才能搞清楚。然而晶体管的总体行为可以不用参考量子力学就可以描述为一系列方程式，它们给出了施加在晶体管端口上的电压与流经它的电流的关系。这些方程式是器件物理原理的本质行为的**抽象**。

逻辑门

描述晶体管行为的方程式还是相当复杂的。当一组晶体管连接到一起组成一个特殊结构时，例如图 1.2 所示的 CMOS（互补金属氧化物半导体）与非门，对这一组晶体管的行为有一个特别简单的描述。

如果每一条输入线（A 和 B）保持接近于 Vdd 或 Vss 的电压，那么输出也会依据下述规则接近 Vdd 或 Vss，即

- 如果 A 和 B 全都接近 Vdd，则输出将会接近 Vss。
- 如果 A 或 B（或两者都）接近 Vss，则输出将会接近 Vdd。

再详细一点，我们可以规定这些规则中的“接近”是什么意思。把接近 Vdd 的值作为“**真**（true）”，而把接近 Vss 的值作为“**假**（false）”。这样这个电路就实现了布尔逻辑的“与非”功能：

$$\text{output} = \overline{A \cdot B} \tag{1}$$

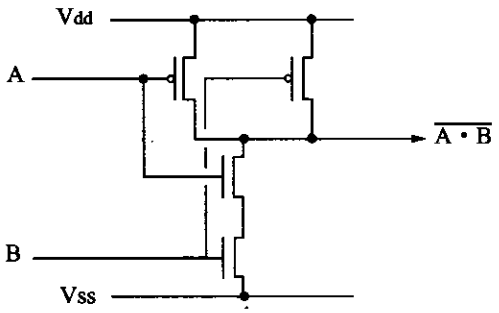
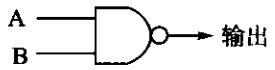


图 1.2 静态 2 输入 CMOS 与非门的晶体管电路图

尽管还有许多其他工程设计方法用 4 个晶体管可靠地实现这个表达式，但其可靠性已足够高，逻辑设计师可以只用逻辑门思考。逻辑设计师使用的概念如图 1.3 所示，这些概念还包括逻辑门的下列视图(view)：



逻辑符号

A	B	输 出
0	0	1
0	1	1
1	0	1
1	1	0

真值表

图 1.3 “与非”门逻辑符号和真值表

逻辑符号

逻辑符号：这是在电路原理图中代表“与非”门的符号。还有类似的符号代表其他逻辑门(例如，输出端去掉小圆圈就得到“与”门，其输出和“与非”门相反。其他例子在附录中给出)。

真值表

真值表：真值表描述门的逻辑功能，它包含了逻辑设计师对于大多数应用所需要了解的与门相关的所有内容。真值表的意义在于它比 4 组晶体管方程要简单得多。

(在真值表中，布尔变量的一般处理方法是以“1”表示“真”，以“0”表示“假”。)

门的抽象

门抽象(abstraction)的作用不仅在于它极大地简化了采用大量晶体管设计电路的过程，而且在于它真正地使设计师不必了解门是由晶体管组成的。不管门是用场效应晶体管(CMOS 工艺采用的晶体管)、双极晶体管、继电器、射流逻辑，或是用其他逻辑形式实现的，逻辑电路应该有同样的逻辑行为。实现技术会影响电路的性能，但不应影响功能。尽可能接近理想地支持门级抽象，以便让逻辑电路设计师不再需要理解晶体管方程，这是晶体管级电路设计师的责任。

抽象的级

理解这个问题可能会有些困难，特别是多年用逻辑门设计电路的那些读者。但是门级抽象所体现的概念可以在计算机科学中不同的层次中应用，并且绝对是我们从本节开始所要考虑的过程的基础，它就是复杂事物的管理方法。

把同一层级的几个元件集合在一起，把它们构造的本质行为抽象出来，在更高层次上隐去那些不必要的细节。这个抽象的过程使我们能够把复杂的系统划分为很少的几个层级。例如，像门的模型一样，每个层级包括 4 个低一级的元件，那么就可以从一只晶体管开始，仅用 10 个层次实现由百万只晶体管组成的微处理器。在很多情况下处理的元件多于 4 个，因此，抽象的级数会大大减少。

在硬件级别上典型抽象层次可以是这样的：

1）晶体管；

2）逻辑门、存储器单元和专用电路；

3）1 位加法器、多路器、译码器和触发器；

4）1 字宽加法器、多路器、译码器、寄存器和总线；

5）ALU(算术逻辑单元)、桶式移位器、寄存器堆和存储体；

6）处理器、Cache 和存储器管理组织；

7）处理器、外围单元、Cache 存储器和存储器管理单元；

8）集成系统芯片；

9）印刷电路板；

10）移动电话、PC 和发动机控制器。

如果用硬件来描述设计，那么以抽象级别的观点来理解设计的过程是相当具体的。这个过程不但可以用于硬件，抽象也是理解软件的基础。我们将在相应的课程中探讨软件的抽象。

门级设计

由逻辑门再往上是构造由几个门构成的常用单元库。就如前面所列出的，典型的单元有加法器、多路器、译码器和触发器。每个单元的宽度都是 1 位。本书主要介绍与处理器核设计和使用相关的内容，而不对逻辑设计作过多的介绍。本书认为想了解 ARM 处理器的读者已对传统的逻辑设计非常熟悉。

对于那些尚不熟悉逻辑设计或需要复习这些知识的读者，请阅读附录(计算机逻辑)。这是下一节中内容的基础要点。附录中包括下列内容的相关细节，即

- 布尔代数和符号；
- 二进制数；
- 二进制加法；
- 多路器；
- 时钟；
- 时序电路；

- 锁存器和触发器；
- 寄存器。

如果对这些术语的任何一个不熟悉，简要地阅读附录就可以了解到后面学习所需要的足够信息。

注意，在附录中，这些电路功能虽然是以简单逻辑门描述的，但通常会有其他更有效的、基于晶体管电路的 CMOS 实现形式。使用 CMOS 芯片的互补晶体管，可以有许多方法实现逻辑设计的基本要求，并且新的晶体管电路还在不断发布。

进一步的信息可参阅逻辑设计教程。在本书的参考文献中推荐了适当的参考书。

1.3 MU0——一个简单的处理器

一个简单的处理器可以由一些基本的部件构成，即

- 程序计数器(PC)寄存器：用来保存当前指令的地址。
- 累加器(ACC)寄存器：用来保存正在处理的数据。
- 算术逻辑单元(ALU)：可以对二进制操作数进行若干操作，如加、减、增值等。
- 指令寄存器(IR)：保存当前执行的指令。
- 指令译码器和控制逻辑：它根据指令控制上述部件产生所需的结果。

这些有限的部件可以实现的指令集是有限的。在许多年里，曼彻斯特大学用这个设计说明处理器设计的原理。曼彻斯特设计的机器经常被称为 MUn，其中 $1 \leqslant n \leqslant 6$，所以，这个简单的机器被称为 MU0。这个设计只是为了教学而开发的，并不是作为科研项目而制造的大规模机器。它与第一台曼彻斯特机很相似，并且被大学生以多种方式实现过。

MU0 指令集

MU0 是具有 12 位地址空间的 16 位机。它的存储器可以组织为 4 096 个可分别寻址的、16 位字长的存储区。也就是说，它的寻址范围可以达到 8 KB。指令长度为 16 位，如图 1.4 所示。其中 4 位为操作码(**opcode**)，12 位为地址域(S)。最简单的指令集只使用了 16 种可用操作码中的 8 种，如表 1.1 所列。

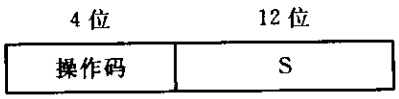


图 1.4 MU0 的指令格式

例如，指令“ACC := ACC + mem_{16}[S]”的意思就是“把地址为 S 的(16 位宽的)存储器单元的内容加到累加器”。指令从存储器的 0 地址开始顺序读取，直到执行到修改 PC 的指令为止。到那时则开始从 jump 指令给出的地址读取指令。

表 1.1　MU0 指令集

指　令	操作码	效　果
LDA　S	0000	ACC := mem_{16}[S]
STO　S	0001	mem_{16}[S] := ACC
ADD　S	0010	ACC := ACC+mem_{16}[S]
SUB　S	0011	ACC := ACC−mem_{16}[S]
JMP　S	0100	PC := S
JGE　S	0101	If ACC≥0　PC := S
JNE　S	0110	If ACC≠0　PC := S
STP	0111	stop

MU0 逻辑设计

为了理解这个指令集是如何实现的，要把这个设计过程在逻辑级走一遍。采取的方法是把设计划分为两个部分：

1. 数据通路

所有并行地传送、存储或处理多位二进制数的部件都属于数据通路，包括累加器、程序计数器、ALU 和指令寄存器。对于这类部件，将采用基于寄存器、多路器等器件的寄存器传输级(RTL，Register Transfer Level)设计方式。

2. 控制逻辑

所有不属于数据通路的部件都属于控制逻辑，将采用有限状态机(FSM，Finite State Machine)的方式进行设计。

数据通路设计

把实现 MU0 指令集所需的基本部件连接起来的方法有很多种。在进行选择时，需要一个指导性原则来帮助我们做出正确的决定。这里将遵循这样的原则，即在设计中存储器将成为限制因素，并且存储器的一次读写总是占用 1 个时钟周期。因此，在设计中，则有：

每条指令占用的时钟周期数严格地由它必须访问存储器的次数决定。

重新看表 1.1，我们可以看到，前 4 条指令各需要两次存储器访问(一次是读取指令本身，另一次是读取或保存操作数)，而后 4 条指令可以在 1 个周期内执行，因为它们不需要操作数(实际上，我们可能不关心 STP 指令的效率，因为它使处理器永远停止)。因此，我们需要这样一个数据通路，它有足够的资源使这些指令可以在 1 个或 2 个时钟周期内完成。图 1.5 给出了一个适宜的数据通路。

(有些读者可能希望在这个数据通路中有一个专用的 PC 增值器。这些读者应注意，所有不改变 PC 的指令都占用 2 个时钟周期，所以，可以使用主 ALU 在其中一个周期内使 PC 加 1。)

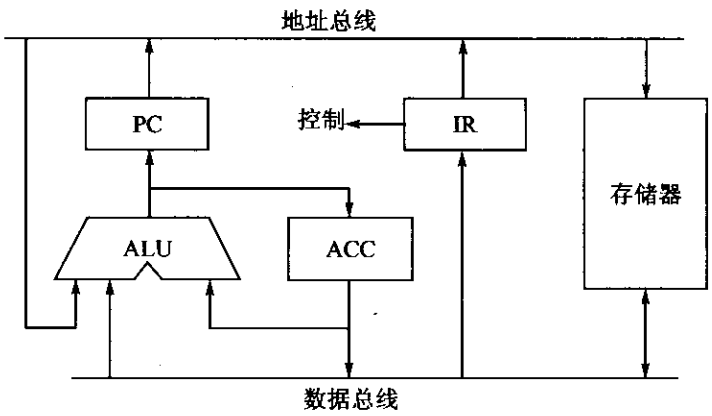


图 1.5 MU0 数据通路示例

数据通路操作

在我们的设计中，假定每条指令都是从它到达指令寄存器时开始。因为在它到达指令寄存器之前我们不知道将处理什么指令，所以，指令分两步执行，也可以忽略第一步。

1. 访问存储器中的操作数并执行所需的操作

送出指令寄存器中的地址，然后，或者从存储器读出一个操作数，与 ALU 中累加器完成相应操作后，将结果写回累加器；或者把累加器的数据输出保存到存储器中。

2. 读取下一步要执行的指令

送出 PC 或者指令寄存器中的地址，读取下一条指令。无论是哪种情形，这个地址都会在 ALU 中加 1，并将增值存入 PC。

初始化

处理器必须在确定状态下开始工作。通常这需要输入复位信号，使处理器从一个确定的地址开始执行指令。我们将 MU0 设计为从地址 000_{16} 开始执行。这可以有多种方法实现。方法之一就是用复位信号将 ALU 的输出清 0，然后由时钟把它打入 PC 寄存器。

寄存器传输级设计

下一步就是严格地设计控制信号。这些信号要控制数据通路的全部操作。假定所有寄存器都是在输入时钟的下降沿改变状态，而且在需要时可以用控制信号来防止它们在某些时钟沿改变。例如 PC，当 PCce 为 1 时，它会在时钟周期末改变；但是当 PCce 为 0 时，它将保持原值。

一种比较适合的寄存器组织如图 1.6 所示。图中给出了所有寄存器的使能信号、ALU 功能的选择信号（精确的信号数和说明在后面确定）、两个多路器的选择控制信号、把 ACC 值送到存储器的三态驱动器的控制信号，以及存储器的请求（MEMrq）和读/写（RnW）控制信号。图中绘出的其他信号从数据通路输出到控制逻辑，包括操作

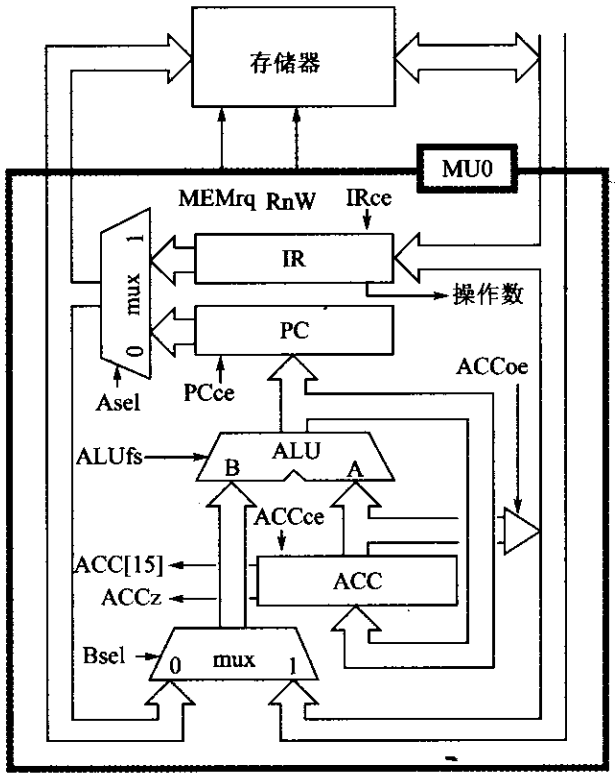


图 1.6　MU0 寄存器传输级组织

码位和指示 ACC 是否为零或负值的信号。后一个信号控制相应的条件转移指令。

控制逻辑

控制逻辑对当前指令进行译码，并产生数据通路的控制信号，必要时会使用来自数据通路的控制输入。控制逻辑是一个有限状态机，原则上开始设计时应该画出状态转换图。但是，这个设计中的 FSM 非常简单，不值得画状态转换图。设计只需要两个状态："取指令"和"执行"，因此，用 1 位足以描述这两个状态(Ex/ft)。

控制逻辑可以用表 1.2 来表述。在这个表中，x 表示"不关心"的条件。只要确定了 ALU 的功能选择码，这个表就可以直接用 PLA(Programmable Logic Array，可编程逻辑阵列)实现，也可以转换成组合逻辑用标准门来实现。

认真研究表 1.2 可以发现一些很容易简化的地方。程序计数器和指令寄存器的时钟使能(PCce 和 IRce)总是相同的。这是有道理的，因为只要读取一条新指令，ALU 就计算程序计数器的下一个值，并将它锁存。因此，这两个控制信号可以合并成一个。同样，凡是当累加器驱动数据总线时(ACCoe 为高)，存储器应该执行写操作(RnW 为低)。因此，这两个信号中的一个可以由另一个信号的反相产生。

在这些简化之后，控制逻辑的设计就几乎完成了，剩下的只是确定 ALU 功能的编码。

表 1.2 MU0 控制逻辑

输入						输出									
指令	操作码	复位	Ex/ft	ACCz	ACC15	Asel	Bsel	ACCce	PCce	IRce	ACCoe	ALUfs	MEMrq	RnW	Ex/ft
Reset	xxxx	1	x	x	x	0	0	1	1	1	0	=0	1	1	0
LDA S	0000	0	0	x	x	1	1	1	0	0	0	=B	1	1	1
	0000	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
STO S	0001	0	0	x	x	1	x	0	0	0	1	x	1	0	1
	0001	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
ADD S	0010	0	0	x	x	1	1	1	0	0	0	A+B	1	1	1
	0010	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
SUB S	0011	0	0	x	x	1	1	1	0	0	0	A−B	1	1	1
	0011	0	1	x	x	0	0	0	1	1	0	B+1	1	1	0
JMP S	0100	0	x	x	x	1	0	0	1	1	0	B+1	1	1	0
JGE S	0101	0	x	x	0	1	0	0	1	1	0	B+1	1	1	0
	0101	0	x	x	1	0	0	0	1	1	0	B+1	1	1	0
JNE S	0110	0	x	0	x	1	0	0	1	1	0	B+1	1	1	0
	0110	0	x	1	x	0	0	0	1	1	0	B+1	1	1	0
STOP	0111	0	x	x	x	1	x	0	0	0	0	x	0	1	0

ALU 设计

图 1.6 中的大多数寄存器传输级功能都有直接的逻辑实现方式(有怀疑的读者可参阅附录“计算机逻辑”)。但是,MU0 的 ALU 比附录中讲述的简单加法器稍微复杂一些。

所需的 ALU 功能列于表 1.2,共有 5 种(A+B、A−B、B、B+1 和 0)。最后一项只在复位信号有效时产生,因而可用复位信号直接控制。控制逻辑只需要产生 2 位的功能选择码来选择其他 4 项功能。如果基本的 ALU 输入是操作数 A 和 B,那么增加一个传统的二进制加法器就可以产生所有的功能。

- A+B 是加法器的标准输出(假设进位输入为 0)。
- A−B 可以用 $A+\overline{B}+1$ 来实现。这需要把输入 B 反相,并将进位输入强制为 1。
- B 可以通过强制输入 A 和进位输入都为 0 来实现。
- B+1 可以通过强制 A 为 0 并将进位输入强制为 1 来实现。

ALU 的门级逻辑如图 1.7 所示。Aen 使能操作数 A,或将其强制为 0;Binv 控制操作数 B 是否反相。1 位的进位输出(Cout)连接到下一位的进位输入(Cin);第一位进位输入由 ALU 的功能选择(如同 Aen 和 Binv)来控制,最后一位的进位输出未用。ALU 与多路器、寄存器、控制逻辑和总线缓冲器(用来将累加器的值送到数据总线)组合在一起,处理器就完成了。加上标准的存储器,就有了一台可以工作的计算机。

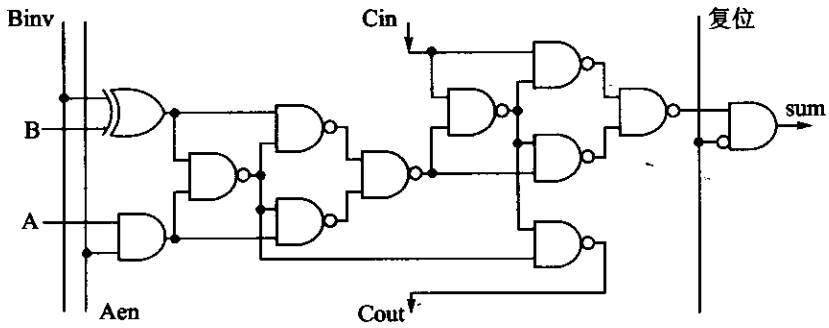


图 1.7　MU0 ALU 逻辑的一位

MU0 的扩展

尽管 MU0 微处理器非常简单，并且不适合作为高级语言编译器的目标，但它说明了处理器设计的基本原理。与第一个 ARM 处理器的设计过程相比，主要的区别在于复杂性而不是原理。基于微码控制逻辑的 MU0 设计也开发出来了。它在 MU0 的基础上进行了扩展，加入了变址寻址方式。同所有好的处理器一样，MU0 在指令空间上留有余地，以便将来可以扩充指令集。

要使 MU0 成为一个有用的处理器，还需要做许多工作。以下的扩展看来是最重要的，即

- 扩展地址空间。
- 增加更多的寻址方式。
- 能够保存 PC，以便支持子程序调用。
- 增加更多的寄存器，支持中断等等。

总的来说，如果要设计的是一个高性能的、很适合于高级语言编译器的处理器，MU0 不是一个合适的起始平台。

1.4　指令集的设计

如果 MU0 的指令集不适合高性能处理器，那么什么样的指令集才适用呢？

我们从最初的原理开始。先查看基本的机器操作，例如将两个数相加产生一个结果的指令。

4 地址指令

按照最通常的形式，一条指令需要一些位使其区别于其他指令。一些位指定操作数的地址，一些位指定把结果放在哪里(目的)，一些位指定下一条要执行的指令的地址。这样一条指令的汇编语言格式可以如下：

```
ADD     d, s1, s2,  next_i          ; d := s1 + s2
```

这种指令在存储器中可用如图 1.8 所示的二进制格式表示。在这种格式中，每条指令需要 4n+f 位。其中每个操作数需要 n 位，指定 ADD 操作码需要 f 位。

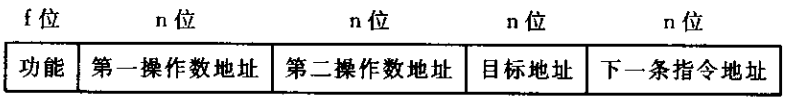


图 1.8 4 地址指令格式

3 地址指令

减少每条指令所需位数的首要方法是把下一条指令的地址变为隐含的(除非是转移指令，其作用就是明确地修改指令的顺序)。若假定下一条指令的缺省地址可以由指令的大小加上 PC 值得到，则指令变为 3 地址格式。它的汇编语言格式如下：

```
ADD     d, s1, s2                ; d := s1 + s2
```

这样的指令的二进制表示如图 1.9 所示。

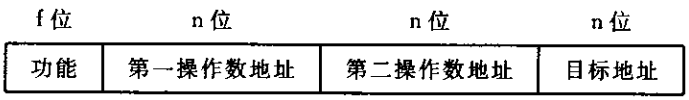


图 1.9 3 地址指令格式

2 地址指令

若目的与一个源操作数共用一个寄存器，则可以进一步节省存储一条指令所需的位数。汇编语言格式可如下：

```
ADD     d, s1                    ; d := d + s1
```

现在，二进制表示压缩为如图 1.10 所示。

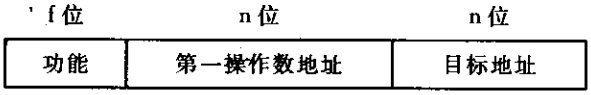


图 1.10 2 地址指令格式

1 地址指令

如果目的寄存器是隐含的，则通常称之为**累加器**(例如前一节讲过的 MU0)。指令则只需要指定一个操作数，即

```
ADD     s1                       ; accumulator := accumulator + s1
```

二进制表示则进一步简化为如图 1.11 所示。

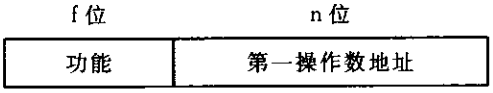


图 1.11　1 地址（累加器）指令格式

0 地址指令

最后，可以采用求值堆栈式(evaluation stack)的体系结构，从而使全部操作数变为隐含的。汇编语言的格式如下：

```
ADD                    ; top_of_stack := top_of_stack + next_on_stack
```

二进制表示如图 1.12 所示。

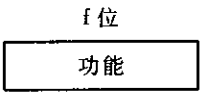


图 1.12　0 地址指令格式

n 地址应用举例

除了 4 地址格式外，其他形式都已在处理器的指令集设计中使用过。虽然 4 地址指令也用于一些内部微码设计，但是对于机器级指令集来说，没必要如此浪费。例如：

- Inmos Trasputer 处理器使用 0 地址的求值堆栈结构。
- 前面的 MU0 是一种简单的 1 地址结构的例子。
- 一些 ARM 处理器中为提高代码密度而采用的 Thumb 指令集使用了以 2 地址形式为主的结构。
- 标准的 ARM 指令集使用了 3 地址结构。

地　址

MU0 结构中的地址是操作数所在存储器的直接“绝对”地址。但 ARM 的 3 地址指令格式中的 3 个地址指定的是寄存器，而不是存储器地址。通常，所谓“3 地址结构”指的是这样的指令集，即两个源操作数和一个目的操作数可以互相独立地指定，但通常只能在有限的可能值之间指定。

指令类型

我们刚刚查看了几种定义 ADD 指令的方式。一个完整的指令集不仅要完成存储器中操作数的算术操作，还需要做更多的事。通常一个通用的指令集应包括以下几类指令：

- 数据处理指令。例如加、减和乘。
- 数据传送指令。这类指令把数据从存储器中一个地方复制到另一个地方，或者从存储器复制到处理器的寄存器等等。

- 流控制指令。这类指令把程序的执行从一部分切换到另一部分。切换有可能取决于数据的值。
- 控制处理器执行状态的特殊指令。例如，切换到特权模式以执行操作系统功能。

有时一条指令属于一个以上的类别。例如，“减 1，如果非 0 则转移”这条在控制程序循环时是很有用的指令，它既对循环变量进行某些数据处理，又完成流控制功能。与此类似，从存储器某地址读取操作数并把结果送到寄存器的数据处理指令，可以看作是进行数据传送功能。

正交指令

如果构造一条指令的每一种选择都独立于其他的选择，那么指令集就是**正交的**。由于加法和减法是类似的操作，可以预料能够在类似的条件下使用它们。如果加法使用寄存器地址的 3 地址格式，那么减法也应可以，并且两者都不应对可以使用的寄存器有任何特殊的限制。

正交指令集对汇编语言编程者来说比较容易学习，针对它的编译器也容易编写，硬件的实现通常也更有效。

寻址模式

当数据处理或数据传送指令访问操作数时，有几种标准的方法用于指定所需数据的位置。多数处理器支持这些**寻址模式**中的几种(但是很少会支持所有模式)。

1) 立即寻址：指令中给出所需的数值(二进制形式)。
2) 绝对寻址：指令中包含所需数据在存储器中的全部地址(二进制)。
3) 间接寻址：指令中包含一个存储器位置的二进制地址。在该位置存有所需数据的二进制地址。
4) 寄存器寻址：所需数据在一个寄存器中，指令包含这个寄存器的编号。
5) 寄存器间接寻址：指令中包含寄存器的编号，而该寄存器的内容是数据在存储器中的地址。
6) 基址偏移寻址：指令指定寄存器(**基址**)和二进制偏移量。偏移量与基址相加得到存储器地址。
7) 基址变址寻址：指令指定基址寄存器和另外一个寄存器(**变址**)。变址与基址相加得到存储器地址。
8) 基址比例变址寻址：类似前一种方式，但变址在与基址相加之前要乘以一个常数(通常为数据项的长度，通常是 2 的幂)。
9) 堆栈寻址：一个隐含或指定的寄存器(**堆栈指针**)指向存储器中某处(**堆栈**)，数据项以后进先出的原则写入(**压入**)或读出(**弹出**)。

注意，对这些寻址模式，不同的处理器厂商采用的名称可能有所不同。寻址模式几乎可以无限地扩充，例如，增加更多的间接层次，增加基址变址加偏移等等。但是，以上所列举的模式涵盖了大多数通常使用的寻址模式。

控制流指令

如果程序必须偏离缺省的(通常为顺序执行)指令执行顺序,则使用控制流指令显式地修改程序计数器(PC)。这类指令中最简单的指令通常称为“转移”或“跳转”。由于大多数的转移范围相对较小,所以常见的形式是 PC 相对转移。典型的汇编语言格式如下:

```
        B       LABEL
        ...
LABEL   ...
```

在这里,编译程序计算出加到 PC 值的位移量,在转移执行时强制 PC 指向 LABEL。转移的最大范围取决于分配给以二进制格式表示的位移量的位数。如果要求的转移超出范围,则编译程序应报告错误。

条件转移

数字信号处理(DSP)程序可能总是执行固定的指令序列,但是,通用处理器经常需要根据数据的变化而改变其程序。一些处理器(包括 MU0)允许指令根据通用寄存器的值决定是否执行转移。例如:

- 如果特定寄存器的值为零(或非零,或为负等等),则转移。
- 如果两个指定的寄存器相等(或不等),则转移。

条件码寄存器

然而,最常用的转移机制是基于条件码寄存器。条件码寄存器是处理器中的一种专用寄存器。只要执行数据处理指令(也可能只是专用指令,或者专门设置条件码寄存器的指令),条件码寄存器就记录其结果是否为零、为负、溢出、产生进位输出等等。条件转移指令则由条件码寄存器的状态控制。

子程序调用

有时转移指令用于调用子程序。在子程序执行结束后,指令的执行顺序须返回到调用位置。因为子程序可能会从许多不同的地方被调用,所以,必须保存有关的调用地址。保存调用地址有很多方法:

- 在执行转移之前,由调用程序计算出适当的返回地址,并把它存到标准存储器,以便子程序作为返回地址使用。
- 可把返回地址压入堆栈。
- 可把返回地址放入寄存器。

子程序调用使用频繁,因此,大多数体系结构都使用专用指令以提高效率。与简单的转移相比,典型的子程序调用会跳过更多的存储器空间,所以,当然要单独处理。调用经常是无条件的。当需要时,也可以编程为有条件地调用子程序,方法是插入一个无条件调用,并用相反条件的转移指令来绕开它。

子程序返回

返回指令将返回地址从它存储的地方(存储器,也许为堆栈或寄存器)回送到 PC。

系统调用

另一类控制流指令是系统调用,也就是转移到操作系统例程,并且经常伴随着当前正在执行程序的特权(privilege)级别的改变。处理器的一些功能,可能包括所有的输入和输出外围器件,都被保护着以防止用户代码访问。因此,如果用户程序需要访问这些功能,就必须启动系统调用。

系统调用以受控的方式穿过保护壁垒。一个设计良好的处理器会确保在多用户操作系统下,一个用户程序能被保护以防止其他用户,也可能是个别恶意用户的攻击。这就要求恶意用户不能改变系统代码,而且当需要访问被保护的功能时,系统代码必须进行检查,确认被请求的功能是经过授权的。

这是硬件和软件设计的复杂领域。多数嵌入式系统(以及许多桌面系统)不使用全部的硬件保护能力。但是,如果处理器不支持保护的系统模式,那么它在那些要求有这种功能的应用中将不会被考虑。因此,目前多数微处理器都支持这种模式。虽然在了解指令集的基本设计时不需要理解支持安全操作系统的全部含义,但是,即使见闻不广的读者也应了解这个问题,因为除非将这个潜在的安全保护的目标牢记在心,否则,商业处理器结构的一些特性就没有什么意义。

异 常

最后一类控制流指令包含这样一些情况,即控制流的改变不是按程序员的意愿发生,而是程序执行时发生了一些未预料的(可能还是不希望的)的结果。例如,可能因为检测到存储器子系统的故障,试图访问存储器失败了。为了解决这个问题,程序必须偏离原计划的进程。

这种计划外的控制流改变称为**异常**(exception)。

1.5 处理器设计中的权衡

处理器设计的艺术就是定义一个指令集,它要支持对程序员有用的功能,同时它的实现要尽可能有效率。最好是这个指令集还应使以后更复杂的实现也有同样的效率。

程序员一般都希望以尽可能抽象的方式表达他的程序,使用的高级语言应支持那些适合于解决问题的所要使用概念的处理方式。当前的趋势是功能的和面向对象的语言,与以前的命令式语言(例如 C 语言)相比,这种语言的抽象级别更高。即使是以前的语言,离通常的机器指令也已经相当远了。

高级语言结构和机器指令之间在语义学上的**缝隙**(semantic gap)由**编译器**来链接。编译器是(通常是复杂的)计算机程序,它把高级语言程序翻译成一系列机器指令。因此,处理器的设计者所定义的指令集,应是一个好的编译对象,而不是那种让程序员直

接用来手工解决问题的东西。那么，什么样的指令集才是好的编译对象呢？

复杂指令集计算机

1980 年以前，指令集设计的主要趋势是增加复杂度，以减小必须由编译器搭接的语义学缝隙。在指令集中加入单指令过程的进入和退出，一条指令在多个时钟周期内完成一个复杂的操作序列。处理器的卖点是其寻址模式和数据类型等等的技巧和数量。

这种趋势的起因是 20 世纪 70 年代发展起来的小型计算机。这些计算机的主存储器速度相对较慢，与其相连的处理器是由很多简单的集成电路搭成的。处理器由比主存储器速度快的微码 ROM（只读存储器）控制。因此，将经常使用的操作以微码序列实现，而不使用需要从主存储器读取几条指令的方式是非常有意义的。

贯穿整个 20 世纪 70 年代，微处理器的性能不断提高。这些单片处理器依赖先进的半导体技术使得在单个芯片上集成尽可能多的的晶体管，所以，它的发展是发生在半导体行业，而不是在计算机行业。结果，微处理器的设计缺乏在结构级上独创的思想，特别是其实现技术的需求。设计师们最多从小型计算机工业取得想法，而小型计算机的实现技术是非常不同的。特别是全部复杂例程所需要的微码 ROM 占据了过多的芯片面积，给其他能增强性能的部件没有留下多少空间。

这个方法产生了 20 世纪 70 年代晚期的单片复杂指令集计算机（CISC，Complex Instruction Set Computer）。这是带有小型计算机指令集的微处理器。而这个指令集又是以有限的可用硅资源为代价的。

RISC 革命

精简指令集计算机（RISC，Reduced Instruction Set Computer）诞生在指令集日益复杂的时候。RISC 的概念对 ARM 处理器的设计有重大影响。RISC 实际上就是 ARM 的别名。但是，在更详细地研究 RISC 或 ARM 之前，我们需要对处理器做些什么和怎样设计处理器以使它更快地工作这些问题稍微多作一些了解。

如果减小处理器指令集与高级语言之间的语义学缝隙不是使计算机更有效率的正确途径，那么设计师还有哪些选择呢？

处理器做些什么

如果想要处理器运行得更快，首先必须弄明白它花费时间在做什么。有一个普遍的误解，就是认为计算机花费时间在进行计算。也就是说，它在对用户的数据进行算术操作。实际上，它只用很少的时间进行这个意义上的“计算”。尽管它进行相当数量的算术运算，但是，这些运算多数需要寻址，以便找到相关数据与程序的位置。找到用户的数据后，多数的工作是把它们移来移去，而不是进行转换意义上的处理。

在指令集的级别上，可以测量各个不同指令的使用频率。重要的是获得**动态的**测量，就是测量被执行的指令的频率，而不是由各类型二进制指令的计数得到的**静态**频率。一个典型的统计如表 1.3 所列。该统计是通过在 ARM 指令仿真器上运行打印预览程序来提取的，但是对其他程序和指令集也有广泛的典型意义。

表 1.3 典型的动态指令使用率

指令类型	动态使用率/%
数据移动	43
控制流	23
算术操作	15
比较	13
逻辑操作	5
其他	1

这些采样统计表明，应予以优化的最重要的指令是与数据移动相关的指令，无论是在处理器寄存器与存储器之间的移动，还是从寄存器到寄存器的移动。这些指令几乎占据了被执行指令的一半。使用频率第二高的指令是控制流指令。例如，转移和进程调用，它们占据了 1/4。算术指令低至 15%，比较指令与之相似。

现在我们初步了解了计算机花费时间在做什么，就可以考虑使它们运行更快的方法。其中最重要的是流水线。另一个重要的技术是使用将在 10.3 节介绍的 Cache(高速缓存)。第 3 个技术——超标量指令执行——非常复杂，没有应用在 ARM 处理器中，本书也没有介绍。

流水线

处理器按照一系列步骤来执行每一条指令。典型的步骤如下：

1) 从存储器读取指令(fetch)。

2) 译码以鉴别它是哪一类指令(dec)。

3) 从寄存器堆取得所需的操作数(reg)。

4) 将操作数进行组合以得到结果或存储器地址(ALU)。

5) 如果需要，则访问存储器以存取数据(mem)。

6) 将结果回写到寄存器堆(res)。

并不是所有的指令都需要每一个步骤，但是，多数指令需要其中的多数步骤。这些步骤往往使用不同的硬件功能，例如，ALU 可能只在第 4 步中用到。因此，如果一条指令不是在前一条结束之前就开始，那么在每一个步骤内处理器只有少部分的硬件被使用。

有一个明显的方法可以改善硬件资源的使用率和处理器的吞吐量，这就是在当前指令结束之前就开始执行下一条指令。该技术被称为**流水线**，是在通用处理器中采用并行算法且非常有效的途径。

采用上述操作顺序，处理器可以这样来组织：当一条指令刚刚执行完步骤1并转向步骤2时，下一条指令就开始执行步骤1。图1.13说明了这个过程。从原理上来

说，这样的流水线应该比没有重叠的指令执行快 6 倍，但实际上事情并没有这么好，下面我们将会看到原因。

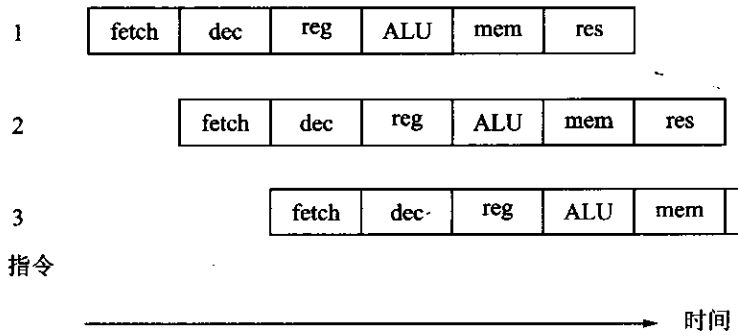


图 1.13　流水线的指令执行

流水线中的冒险

在典型的计算机程序中经常会遇到这样的情形，即一条指令的结果被用做下一条指令的操作数。当这种情形发生时，图 1.13 所示的流水线操作就中断了，因为第一条指令的结果在第二条指令取操作数时还没有产生。第二条指令必须停止，直到结果产生为止。这时流水线的行为如图 1.14 所示。这是流水线的“**写后读**”冒险(hazard)。

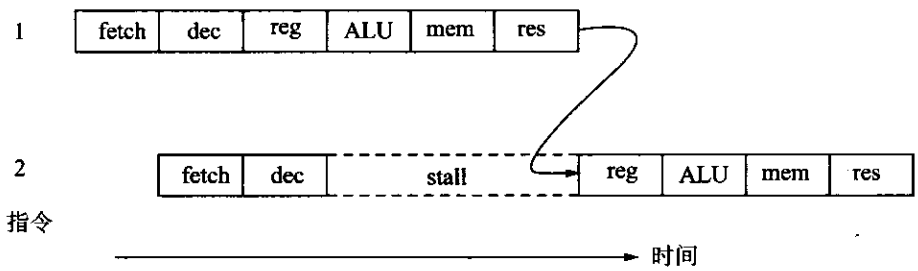


图 1.14　先写后读的流水线冒险

转移指令更会破坏流水线的行为，因为后续指令的取指步骤受到转移目标计算的影响，因而必须推迟。不幸的是，当转移指令正在被译码时，在它被确认为是转移指令之前，后续的取指操作就发生了。这样一来，读取到的指令就不得不丢弃。如果转移目标的计算是在图 1.13 中流水线的 ALU 阶段完成的，那么，在得到转移目标之前已经有 3 条指令按照原有的指令流读取(见图 1.15)。如果有可能，最好早一些计算转移目标，尽管这可能需要专门的硬件。如果转移指令具有固定的格式，那么可以(也就是说在确认该指令是转移指令之前)在 dec 阶段**预测**计算转移目标，从而将转移的执行时间减少到单个周期。

但是要注意，由于条件转移与前一条指令的条件码结果有关，在这个流水线中还会有条件转移的冒险。

一些 RISC 体系结构(尽管不是 ARM)规定，不管是否进行了转移，转移之后的指令都要执行。这个技术称为**延迟转移**。

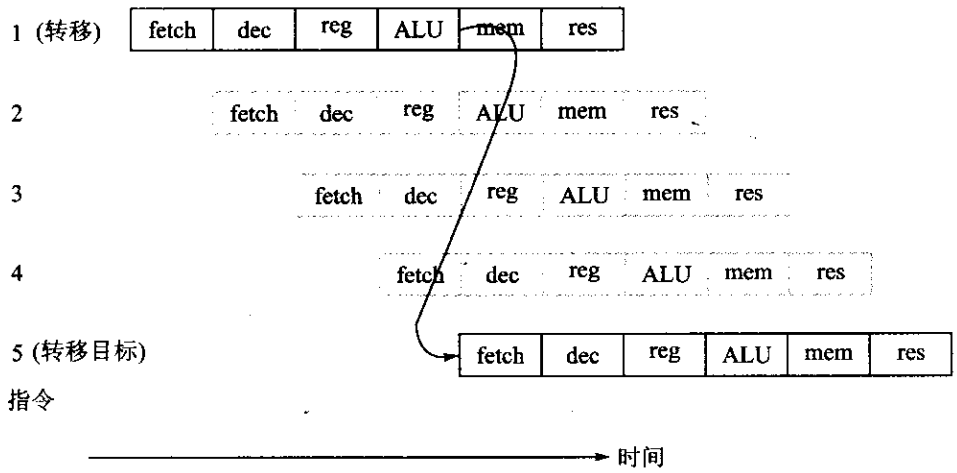


图 1.15 流水线的转移行为

流水线效率

尽管有些技术可以减少这些流水线问题的影响，但是，不能完全消除这些困难。流水线越深(就是流水线的级数越多)，问题就越严重。对于相对简单的处理器，使用3～5级流水线效果会更好。但是，超过了这个级数，收益递减的法则开始生效，增加的成本和复杂度将超过收益。

显然，只有当所有指令都依相似的步骤执行时，流水线才能带来好处。如果处理器的指令非常复杂，每一条指令的行为都与下一条指令不同，那么就很难用流水线实现。1980 年，因为有限的硅资源、有限的设计资源，以及设计一个复杂指令集的流水线的高度复杂性，当时的复杂指令集微处理器没有采用流水线。

1.6 精简指令集计算机

1980 年，Patterson 和 Ditzel 完成了一篇题为“精简指令集计算机概述”的论文(完整的参考见参考文献)。在这篇开创性的论文中，他们详细说明了这样的观点，即单片处理器的优化结构不必像多芯片处理器的优化结构一样。随后一个处理器设计项目取得的结果支持了他们的论点。这个项目是伯克利一个研究生班承担的，他们联合研究精简指令集计算机(RISC)体系结构。这项设计，即伯克利 RISC Ⅰ，比当时的商业 CISC 处理器简单得多，开发中投入的设计力量也少一个数量级，但仍然达到了相似的性能。

RISC 体系结构

- 固定的(32 位)指令长度，指令类型很少。而 CISC 处理器指令集的长度一般可变，指令类型也很多。

- Load-Store 结构，数据处理指令只访问寄存器，与访问存储器的指令是分开的。而 CISC 处理器一般允许将存储器中的数据作为数据处理指令的操作数。
- 由 32 个 32 位寄存器构成大的寄存器堆(register bank)，其中所有的寄存器都可以用于任何用途，以使 Load-Store 结构有效地工作。虽然 CISC 寄存器集也加大了，但是没有这么大，而且大都是不同的寄存器用于不同的用途(例如，Motorola 公司 MC68000 的数据寄存器和地址寄存器)。

这些差别极大地简化了处理器的设计，使设计者在实现体系结构时可以采用以下这些对提高原型机性能有很大作用的、组织方面的特点。

RISC 的组织

- 硬连线的指令译码逻辑。而 CISC 处理器使用大的微码 ROM 进行指令译码。
- 流水线执行。而 CISC 处理器即使有，也只允许在连续指令间有极少的重叠(尽管它们现在允许)。
- 单周期执行。而 CISC 处理器执行 1 条指令一般需要多个时钟周期。

结合这些体系结构和组织上的变化，伯克利 RISC 微处理器有效地摆脱了在渐进的改善中无法回避的问题，即陷于性能函数的局部最大值的危险。

RISC 的优点

Patterrson 和 Ditzel 认为 RISC 有 3 个基本优点：

1. 管芯面积小

简单的处理器需要的晶体管少，需要的硅片面积也小。因此，整个 CPU 在工艺技术发展的较早阶段即可容纳在一个芯片内。一旦技术发展超过这一阶段，RISC CPU 就能省下更多的面积用于实现可以提高性能的功能部件，例如高速缓存、存储器管理和浮点硬件等等。

2. 开发时间短

简单的处理器会占用较少的设计力量，因而设计费用低。它还会更好地与投放市场时的工艺技术相适应(因为开发周期越短，越容易在开发时预测工艺技术的发展)。

3. 性能高

这个优点比较微妙。前面两条优点容易接受，但看看我们周围，高性能总要通过不断增加复杂度来实现，说 RISC 有高性能的优点有些使人难以接受。

可以这样来看这个问题：较小的东西具有较高的自然频率(昆虫煽动翅膀的频率高于小鸟，小鸟煽动翅膀的频率高于大鸟等等)，所以，简单的处理器应该容许较高的时钟频率。让我们来设计一个复杂的处理器，但开始时先设计一个简单的，然后每次增加一条复杂的指令。每增加一条复杂的指令，都会使某些高级的功能更有效率，但是，它也会降低所有指令所用的时钟频率。我们可以度量对于典型程序总的得失。当我们这样做的时候，会发现所有复杂的指令都使程序执行变慢了。因此，我们坚持最初的简单处理器。

这些论点得到了试验结果和处理器原型机(伯克利 RISC Ⅰ之后不久开发的伯克利 RISC Ⅱ)的支持。商业的处理器公司最初是怀疑的，但是，多数为了各自的用途而设计

处理器的新公司都看到了降低开发成本和战胜对手的机会。这些商业的 RISC 设计(ARM 是其中第一个)证明了这个想法是成功的。从 1980 年以来,所有新的通用处理器体系结构都或多或少地采用了 RISC 的概念。

RISC 回顾

因为现在 RISC 已经在商业应用中被普遍接受,我们有可能回顾和更清楚地看一看,究竟它对微处理器的发展有哪些贡献。

1. 流水线

流水线是在处理器中实现并行操作的最简单形式,而且可以使速度提高 2～3 倍。精简指令集极大地简化了流水线的设计。

2. 高时钟频率和单周期执行

在 1980 年,标准的半导体存储器 DRAM(Dynamic Random Access Memory,动态随机访问存储器)在随机访问时可工作于 3 MHz,而在顺序访问(页面模式)时可工作于 6 MHz。当时的 CISC 微处理器最多可以 2 MHz 访问存储器,所以,存储器的带宽还没有用满。RISC 处理器的结构简单,它可以工作在较高的时钟频率,充分使用已有的存储器带宽。

这些特性都不是体系结构的特征,但是,这些特性都依赖于体系结构。正是体系结构足够简单,才使它的实现可以具有这些特性。RISC 体系结构获得成功是由于它足够简单,使设计者能够采用这些组织方面的技术。使用微码、多周期执行和非流水线实现一个指令长度固定的 Load-Store 体系结构是完全可行的,但是,这样的实现比起 CISC 没有任何优点。在那个时代,不可能实现一个硬连线的、单周期执行的流水线 CISC。但是现在可以了!

时钟频率

作为以上分析的注脚,关于时钟频率的讨论还有两个方面需要进一步说明,即

- 在 20 世纪 80 年代,CISC 的时钟频率通常比早期 RISC 的高。但是,它们访问一次存储器需要几个时钟周期,所以,它们访问存储器的速率低。切勿单以时钟频率来评价处理器。
- CISC 访问存储器的速率与它的带宽不相匹配。这似乎与 1.5 节"复杂指令集计算机"中的注释相冲突。在那一部分中曾证明,在 20 世纪 70 年代早期的小型计算机中,在主存储器速度相对低于处理器速度的情况下,微码是正确的。要化解这个冲突,我们则要注意到,在其后的 10 年中,存储器技术发展得非常迅速,而早期的 CISC 微处理器比典型的小型计算机处理器还要慢。之所以微处理器速度较慢,是由于必须舍弃较快速的双极工艺,而采用慢得多的 NMOS 工艺,以便使整个处理器集成到单个芯片所需的逻辑密度。

RISC 的缺点

RISC 处理器在性能竞争中明显胜出,设计成本又低。那么 RISC 就什么都好吗?随着时间的推移,两条缺点开始显现出来,即

- 与 CISC 相比，通常 RISC 的代码密度低。
- RISC 不能执行 x86 代码。

其中第二条很难改变，尽管也有为许多 RISC 平台开发的 PC 仿真软件。然而，只有当你想建造一个 IBM PC 的兼容机时，这才是个问题。而在其他应用中，完全可以忽略它。

代码密度低是指令集长度固定的结果，而且当应用领域较宽时，这个问题会更加严重。如果没有 Cache，则代码密度低会导致在取指时使用更大的主存储器带宽，造成更高的存储器功耗。当处理器集成一定规模的片上 Cache 时，代码密度低会导致在任何时候都只有少部分正在工作的指令集能够装入 Cache，会降低 Cache 的命中率，造成对主存储器带宽的需求以及功耗有更大的增长。

ARM 代码密度和 Thumb

ARM 处理器是根据 RISC 原理设计的，但是由于各种原因，在低代码密度问题上它比其他多数 RISC 要好一些，然而它的代码密度仍然不如某些 CISC 处理器的。在代码密度特别重要的场合，ARM 公司在某些版本的 ARM 处理器中，加入了一个称为 **Thumb** 结构的新型机构。Thumb 指令集是原来 32 位 ARM 指令集的 16 位压缩形式，并在指令流水线中使用了动态解压缩硬件。Thumb 代码密度优于多数 CISC 处理器达到的代码密度。

Thumb 结构将在第 7 章中介绍。

RISC 之后

RISC 似乎不可能会成为最后一个表述计算机体系结构的词汇。那么，有没有任何迹象表明将出现其他突破，而使 RISC 方法变为陈旧的技术呢？

直到本书写作时，还没看到任何发展带来了像 RISC 那样深刻的改变。但是，指令集在不断地发展，以便为有效的实现和为多媒体等新的应用提供更好的支持。

1.7　低功耗设计

自从 50 年前引入了数字计算机以来，它的性价比就一直在得到持续的改进，其速率与其他任何技术努力都无法相比。作为提高性能过程的一个副产品，计算机的功耗也同样引人注目地降低了。然而，直到最近，对降低功耗的需求才像对提高性能的需求那样重要，在某些应用领域甚至更加重要了。这种变化大约来源于电池驱动的便携式设备的市场增长，例如，由高性能计算部件组成的数字移动电话、膝上电脑等。

随着集成电路的引入，集成电路与计算机业相互促进。计算机业被一种双赢的局面所驱动。由此，较少的晶体管使成本降低，性能提高，而且功耗也降低了。现在设计者开始专门为了低功耗而设计，在某些情况下，甚至为了达到低功耗而牺牲性能。

在为有效利用功率而进行的努力中，ARM 处理器处于中心地位。因而，考虑一下与低功耗有关的问题是适宜的。

功率到哪里去了

低功耗设计的起点是要搞清楚功率在电路中耗费到哪里去了。CMOS 是现代高性能电子器件的主流工艺，它本身就具有一些适于低功耗设计的优良特性。因此，我们首先看一看在 CMOS 电路中功率耗费到哪里去了。

一种典型的 CMOS 电路是静态的“与非”门，如图 1.2 所示。所有信号都在电源电压 Vdd 和地电压 Vss 之间变化。我们把 Vdd 和 Vss 称为“轨(rail)”。直到最近，5 V 电源都是标准电源，但是，很多现代 CMOS 工艺要求 3 V 左右更低的电源，而最新的技术则工作于 1～2 V 之间，而且将来还会进一步降低。

当门电路工作时，将输出端或者通过由 p 型晶体管组成的上拉网络连接到 Vdd，或者通过 n 型晶体管组成的下拉网络连接到 Vss。当两个输入端都接近某一个轨时，上述两个网络之一就会导通，而另一个则会有效地不导通。因此，在门电路中没有从 Vdd 到 Vss 的通路。此外，输出端一般只连接到相似门的输入端，因而只有电容性负载。一旦输出端被驱动到某个轨，它不需要电流来保持这个状态。因此，在短时间后，门切换电路将达到稳定状态，而且不再从电源中吸取电流。

CMOS 电路只有切换时才消耗功率。这个特征并不是其他许多逻辑技术所共有的。它是使 CMOS 成为高密度集成电路首选技术的主要因素。

CMOS 的功耗组成

CMOS 电路的总功耗由 3 个部分组成，即

1. 切换功耗

这是对门的输出电容 CL 进行充电和放电所消耗的功率，代表由门完成的有用功。每次输出跳变的能量如下：

$$E_t = \frac{1}{2} \cdot C_L \cdot V_{dd}^2 \approx 1 \text{ pJ} \tag{2}$$

2. 短路功耗

当门的输入端处于中间电平时，p 型和 n 型网络都可能导通。这将导致从 Vdd 到 Vss 出现短时间导通通路。如果电路设计正确(一般指能够避免信号缓慢转变的设计)，则短路功耗应该比切换功耗小得多。

3. 漏电流

当晶体管网络处于关断状态时，也会通过很小的电流。尽管按常规工艺这个电流很小(每个门的漏电流比 nA 还小得多)，但是，它是在接通电源但不活动的电路中惟一的功耗，而且可以长时间地使供电电池漏电。它在活动电路中一般可以忽略。

在设计良好的活动电路中，切换功耗是主要的；短路功耗或许在总功耗中加上 10%～20%；漏电流只有在电路不活动时才是重要的。然而，正如下面要讨论的，低电压操作的趋势导致性能和漏电流之间的折衷，在未来的低功耗、高性能设计中，漏电流越来越受到关注。

CMOS 电路的功耗

忽略短路功耗和漏电流部分，则 CMOS 电路的总功耗 P_C 为电路 C 中每个门 g 的功耗的总和，即

$$P_C = \frac{1}{2} \cdot f \cdot V_{dd}^2 \cdot \sum_{g \in C} Ag \cdot C_L^g \tag{3}$$

式中：f——时钟频率；

Ag——门的活跃因数(反映这样一个事实，即不是所有的门都在每一个时钟周期切换)。

C_L^g——门的负载电容。

注意在这个公式中，如果 1 个时钟周期内有两次跳变，则活跃因数为 2。

低功耗电路设计

典型的门负载电容是工艺技术的函数，因而它不受设计者的直接控制。由式(3)的其他参数可想到低功耗设计的各种方法。下面以重要性为序列出这些方法。

1. 降低电源电压

电源电压对功耗的贡献是 2 次方的，这使降低电源电压成为明显的目标。下面将会进一步讨论。

2. 降低电路活跃因数 A

例如门控时钟等技术属于这类方法。只要电路的功能不是必需的，都应消除活动度。

3. 减少门数

在参数相同的情况下，简单电路比复杂电路省电，因为简单电路的总功耗是较少门电路功耗贡献之和。

4. 降低时钟频率

避免不必要的高速时钟是可取的。但是，虽然低速时钟会降低功耗，它也会降低性能。它对功耗效率(例如，可以用 MIPS/W 数来度量)的影响是中性的。然而，如果降低时钟频率后可以在降低了的 Vdd 下工作，那么这将对功耗效率非常有益。

降低 Vdd

缩小 CMOS 工艺的特征尺寸则要求降低电源电压。这是由于形成晶体管的材料不能经受无限强的电场。随着晶体管变小，如果电源电压保持不变，则电场强度就会增加。

但是，随着低功耗设计重要性的提高，可以预期，降低电源电压的需求远超过只是为防止电击穿所需要的。现在有什么妨碍我们使用非常低的电源电压呢？

降低 Vdd 的问题在于这也会降低电路的性能。饱和的晶体管电流由下式给出，即

$$I_{sat} \propto (V_{dd} - V_t)^2 \tag{4}$$

式中：V_t——晶体管的阈值电压。

电路节点的电荷正比于 Vdd，所以，最高的工作频率为

$$f_{max} \propto \frac{(Vdd - V_t)^2}{Vdd} \tag{5}$$

因此，当 Vdd 降低时，最高工作频率也会降低。对于亚微米工艺，性能的损失可能不像式(5)所示的那么严重，因为在高电压下电流会受到速度饱和效应的限制，但是性能将会有一定程度的损失。由式(5)可以看出，降低 V_t 是改善性能损失的一个明显的途径。然而，漏电流强烈的依赖于 V_t，即

$$I_{leak} \propto \exp\left(-\frac{V_t}{35\ mV}\right) \tag{6}$$

即使 V_t 降低很少，也可能会显著地增大漏电流，增大电池经过非活动电路的漏电，因而需要在最高性能和最低待机功耗之间进行折衷。如果系统中这两个特性都是很重要的，那么系统设计师就必须认真地考虑这个问题。

即使待机功耗不重要，设计师也必须明白，使用非常低阈值的晶体管来提高性能可能会将漏电功耗增加到可与动态功耗相比的程度。因此，在选择封装和冷却系统时，必须考虑漏电功耗。

低功耗设计策略

作为低功耗设计技术初步介绍的总结，下面列出低功耗设计策略的一些建议：

1. 降低 Vdd

选择能满足所需性能的最低时钟频率，然后，在时钟频率和各种系统部件要求的限制范围内，设定尽量低的电源电压。降低电源电压时要小心谨慎，使漏电不超出待机功耗的要求。

2. 降低片外活动度

片外电容比片内负载大得多，所以，任何时候都要降低片外活动度。要避免瞬态脉冲驱动片外负载，使用 Cache 来减少对片外存储器的访问。

3. 降低片内活动度

这一项的优先级低于降低片外活动度。避免给不必要的电路模块施加时钟信号(例如，使用门控时钟)以及在可能时使用睡眠模式仍然是很重要的。

4. 采用并行技术

如果电源电压是自由的，则可以采用各种并行技术来改善功耗效率。并行技术可以使两个电路在原电路一半的时钟频率下达到同样的性能，同时，可以用较低的电源电压达到所需功能。

低功耗设计是一个活跃的研究领域，也是一个新思想快速涌现的领域。可以预期，在未来 10 年中，依靠工艺与设计技术进步的结合，将使高速数字电路的功耗效率得到进一步的显著改善。

1.8 例题与练习

(更实际的练习将要求读者使用某种形式的硬件仿真环境。)

例题　1.1

使用逻辑门和 4 位寄存器设计一个 4 位二进制计数器。

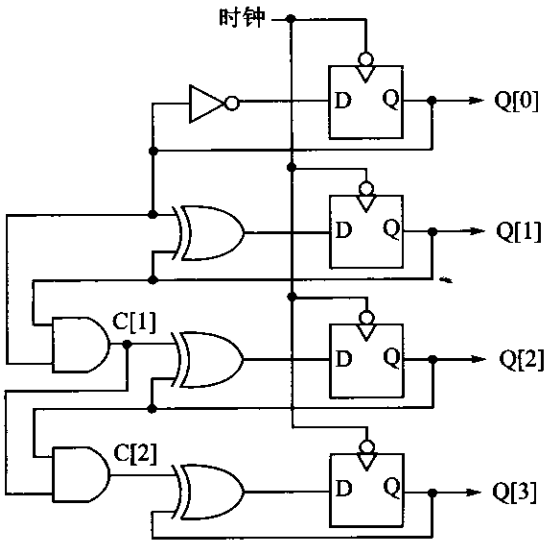
若寄存器的输入以 D[3：0]表示，而它的输出以 Q[3：0]表示，则可构造一个组合逻辑产生 D[3：0]= Q[3：0]+1 来实现计数器。二进制加法器的逻辑方程在附录中给出(式(20)给出了和，式(21)给出了进位)。当第二操作数为常数时，这些式子简化为

$$D[0]=\overline{Q[0]} \tag{7}$$

$$D[i]=Q[i]\cdot\overline{C[i-1]}+\overline{Q[i]}\cdot C[i-1] \tag{8}$$

$$C[i]=Q[i]\cdot C[i-1] \tag{9}$$

式中：$1\leqslant i\leqslant 3$，$C[0]=1$(C[3]不需要)。这些方程可画成逻辑图，图中还包括了寄存器。



练习　1.1.1

修改二进制计数器使之从 0 计数到 9，然后在下一个时钟沿又从 0 开始计数(这是**模 10** 计数器)。

练习　1.1.2

修改二进制计数器使之包含**同步清 0** 功能。这就是增加一个新的输入端(“clear”)。当它有效时，不管计数器当前的计数值如何，在下一个时钟沿之后都要将输出清 0。

练习　1.1.3

修改二进制计数器使之包含 **up/down** 输入端。当该输入端为高时，计数器的行为像练习 1.1.2 中描述的那样；当它变低时，计数器应减计数(以与 up 模式相反的次序)。

例题　1.2

在 MU0 指令集中加入变址寻址。

这里有用的最小扩展是引入一个新的 12 位变址寄存器(X)，以及一些能够将 X 寄

存器初始化和用于 Load 及 Store 指令的新指令，参见表 1.1。在原设计中有 8 个没有使用的操作码，所以，能增加多达 8 条新指令而不会超出空间。变址操作的基本集为

```
LDX     S          ; X := mem16[S]
LDA     S, X       ; ACC := mem16[S+X]
STA     S, X       ; mem16[S+X] := ACC
```

如果有办法修改变址寄存器，它就会更有用。例如在表中步进：

```
INX                ; X := X+1
DEX                ; X := X-1
```

这给出了变址寄存器的基本功能。若有方法能使 X 保存到存储器，则会使 X 更为有用。这样 X 就能够作为临时寄存器使用。为简单起见，不再详述。

练习 1.2.1

修改图 1.6 中的 RTL 组织，使之包含 X 寄存器，指出所需的新的控制信号。

练习 1.2.2

修改表 1.2 的控制逻辑以支持变址寻址。如果你有硬件仿真器，则检测你的设计。（这不是没有价值的！）

例题 1.3

估算单周期延迟转移给性能带来的好处。

延迟转移使转移指令之后的指令不管转移是否实现都要执行。转移指令之后的指令在延迟槽(delay slot)中。假定动态指令频率如表 1.3 所列，流水线结构如图 1.13 所示。忽略寄存器冒险，假定所有延迟槽都能被填充(多数单延迟槽能够被填充)。

如果在译码阶段有一个专用的转移目标加法器，转移指令有 1 个周期的延迟效应，则单个单延迟槽将消除所有浪费的周期。4 条指令中就有 1 条转移指令。因此，如果没有延迟槽，则 4 条指令占用 5 个时钟周期；而如果有延迟槽，则占用 4 个时钟周期，性能提高了 25%。

如果没有专用的转移目标加法器，而是使用主 ALU 阶段来计算转移目标，那么 1 个转移将浪费 3 个周期。因此，平均 4 条指令包括 1 条转移指令，占用 7 个时钟周期。若有单延迟槽，则占用 6 个时钟周期。延迟槽使性能提高了 17%(但是即使没有单延迟槽，专用的转移目标加法器也是有益的)。

练习 1.3.1

评估 2 周期延迟转移对性能的好处。假定第 1 延迟槽能全部被填充，而第 2 延迟槽只有 50%能被填充。

为什么只是在没有专用转移目标加法器时才会使用 2 周期延迟转移？

练习 1.3.2

以上讨论的 1 周期和 2 周期延迟转移对代码的大小有什么影响(所有未填充的转移延迟槽必须被非操作码填充)？

第 2 章　ARM 体系结构

本章内容综述

ARM 处理器是**精简指令集计算机**(RISC)。正如在第 1 章中所说的,RISC 的概念源于斯坦福(Stanford)大学和伯克利(Berkeley)大学在 1980 年前后进行的处理器研究计划。

在本章中我们将看到 RISC 的概念如何有助于 ARM 处理器的成型。最初 ARM 是 1983—1985 年间在英国剑桥的 Acorn Computer 公司开发的。它是第一个为商业用途而开发的 RISC 微处理器,与后来的 RISC 体系结构有明显的不同。这里以综述方式给出 ARM 体系结构的主要特征。详细内容则推迟到后续几章。

1990 年,ARM 特别为扩大开发 ARM 技术而成立了独立的公司。从那以后,ARM 已被授权给世界各地的许多半导体制造厂。它已经成为低功耗和追求成本的嵌入式应用的市场领导者。

没有软、硬件开发工具的支持,处理器就不会特别有用。支持 ARM 的工具套件包括用于硬件建模和软件测试与基准测试的指令集模拟器、汇编器、C 和 C++编译器、链接器和符号调试器。

2.1　Acorn RISC 机器

第一个 ARM 处理器是 1983 年 10 月—1985 年 4 月间在英国剑桥的 Acorn Computer 公司开发的。那时候 ARM 代表 **Acorn RISC Machine** 公司,并一直持续到Advanced RISC Machine limited(后来将名称简化为 ARM Limited)在 1990 年成立之前。

Acorn 因为 BBC(英国广播公司)微型计算机的成功而在英国个人计算机市场占据了强有力的位置。BBC 微型计算机是以 8 位 6502 微处理器为核心的机器。随着 BBC 在 1982 年 1 月一系列电视节目的介绍,它迅速成为英国学校的主流机器。它还在计算机爱好者的市场上享有热烈的支持,并找到了进入一些研究性实验室和高等教育组织的途径。

随着 BBC 微机的成功,Acorn 的工程师考虑用不同的微处理器去构造另一种机器。他们发现所有的商业供货均不充足。1983 年可得到的 16 位 CISC 微处理器比标准的存储器部件还慢。它们也有一些多时钟周期完成的指令(在一些情况下,需要数百

个时钟周期),使其有很长的中断等待时间。BBC 微机很大程度上得益于 6502 的快速中断响应。因此,Acorn 的设计者不愿意接受处理器性能方面的退步。

由于在商业微处理器的供货方面遭受的这些挫折,专有微处理器的设计被提到议事日程。主要的障碍是 Acorn 小组知道商业微处理器需要花费数百个人年的设计努力。Acorn 不可能指望这样规模的投资,因为它是一个总共仅有 400 多雇员的公司。他们必须用少量的设计成本生产出比较好的产品,而且除了为 BBC 微机设计过少量的小规模门阵列之外,它们在全定制芯片设计方面没有任何经验。

在这明显不可能的情况下,伯克利 RISC Ⅰ的论文带来一线生机。它是由少数研究生在一年内设计完成的处理器,品质与领先的商业货源不相上下。它的结构简单,因而没有复杂的指令来损害中断执行时间。还有一些支持的论据,认为它能指引未来的道路,虽然技术的优点不论怎样得到学术界的支持也不能保证商业的成功。

ARM 由于各种因素的偶然组合而诞生,成为 Acorn 产品线的核心部分。后来,在明智地将缩写字 ARM 的意义修改为 **Advanced RISC Machine** 以后,它把它的名字借给新组成的公司去在 Acorn 的产品范围之外扩展市场。尽管名称变化了,体系结构仍保持同原 Acorn 的设计相近。

2.2 体系结构的继承

在设计第一片 ARM 芯片时,尽管早期的机器,如 Digital PDP-8、Cray-1 和 IBM 801,早就提出了 RISC 的概念,并且具有许多后来融入 RISC 的特征,但 RISC 惟一的例子仍为伯克利的 RISC Ⅰ和Ⅱ及斯坦福的 **MIPS**(Microprocessor without Interlocking Pipeline Stages,无互锁流水线微处理器)。

采用的技术特征

ARM 体系结构采用了若干 Berkeley RISC 设计中的特征,但也放弃了若干其他特征。这些采用的特征如下:

- Load/Store 体系结构;
- 固定的 32 位指令;
- 3 地址指令格式。

未采用的特征

在 Berkeley RISC 设计采用的特征中被 ARM 设计者放弃的如下:

1. 寄存器窗口

Berkeley RISC 处理器的寄存器堆中有大量的寄存器,任何时候总有 32 个寄存器是可见的。进程进入和退出的指令移动可见寄存器的"窗口",以使每一个进程都访问新一组寄存器,因此,减少了在处理器与存储器之间因寄存器的保存和恢复而导致的数据拥堵。

寄存器窗口带来的主要问题是大量的寄存器占用很大的芯片面积。这些特征因成

本因素而未被采用，尽管在 ARM 中用来处理异常的影子(shadow)寄存器在概念上没有太大的不同。

在 RISC 早期，由于 Berkeley 原型机中包含了寄存器窗口，使得寄存器窗口的机制密切伴随着 RISC 的概念。但是，后来只有 Sun SPARC 的体系结构在它的原型机中采纳了它。

2. 延迟转移

由于转移中断了指令的平滑流动，因此它造成了流水线的问题。多数 RISC 处理器采用延迟转移来改善这一问题，即在后续指令执行后才进行转移。

延迟转移的问题在于它们消除了单个指令的可分性。在单发射流水线处理器中工作得很好，但它们不能扩展到超标量的处理器，而且可能与转移预测机制产生不良的相互作用。

在原来的 ARM 中延迟转移没有采用，因为它使异常处理更加复杂。从长远观点来看，这是一个好的决定，因为当采用不同流水线重新实现体系结构时，它能使任务简化。

3. 所有指令单周期执行

尽管 ARM 大多数数据处理指令都是在单一时钟周期内执行的，但许多其他指令需要多个时钟周期。

基本原理是基于这样的观察：当数据和指令使用同一个存储器时，即便是最简单的 Load 和 Store 指令也最少需要访问两次存储器(一次指令，一次数据)。这样，只有使用分开的数据和指令存储器才有可能使所有指令都单周期执行。但这对于 ARM 的应用领域来说这太昂贵了。

ARM 被设计为使用最少的时钟周期来访问存储器，而不是所有的指令都单周期执行。当访问存储器需要超过 1 个周期时，就多用 1 个周期。有可能时，做一些有用的事，如支持自动变址寻址模式。这减少了完成任何操作序列所需要的 ARM 指令总数，并提高了性能和代码密度。

简单性

最初的 ARM 设计小组所最关心的是必须保持设计的简单性。在第一片 ARM 芯片之前，Acorn 的设计者仅有设计复杂度约 2 000 门门阵列的经验，因此，探讨了全定制的 CMOS 设计方法的某些问题。当冒险进入未知领域时，可取的做法是把你能够控制的风险减到最小。因为这样做了以后，由于不够了解或根本上不可控制的因素，仍然会有重大的风险。

ARM 的简单性在 ARM 的硬件组织和实现(在第 4 章叙述)上比指令集的结构上体现得更明显。从编程者的观点来看，在指令集设计上可能看到更多的是保守主义，虽然它接收了 RISC 方法基本的原则，但不如许多后来的 RISC 设计更彻底。

把简单的硬件和指令集结合起来，这是 RISC 思想的基础。但仍然保留一些 CISC 的特征，并且因此达到了比纯粹 RISC 更高的代码密度，使得 ARM 获得了功耗效率和较小的核面积。

2.3 ARM 编程模型

处理器指令集定义了操作。程序员可以用这些操作来改变集成了处理器的系统的状态。这些状态是由处理器可见寄存器和系统存储器中数据项的值构成的。每一条指令可以看作是完成从指令执行前的状态到指令完成后的状态所定义的转换。注意，尽管典型的处理器有许多不可见寄存器参与指令的执行，在指令执行前后这些寄存器的值是不重要的。只有可见寄存器中的值有某些意义。ARM 处理器的可见寄存器如图 2.1 所示。

当编写用户级程序时，仅有 15 个通用 32 位寄存器(r0～r14)、程序计数器(r15)和当前程序状态寄存器(CPSR，Current Program Status Register)需要考虑。其余寄存器仅用于系统级编程和异常处理(如中断)。

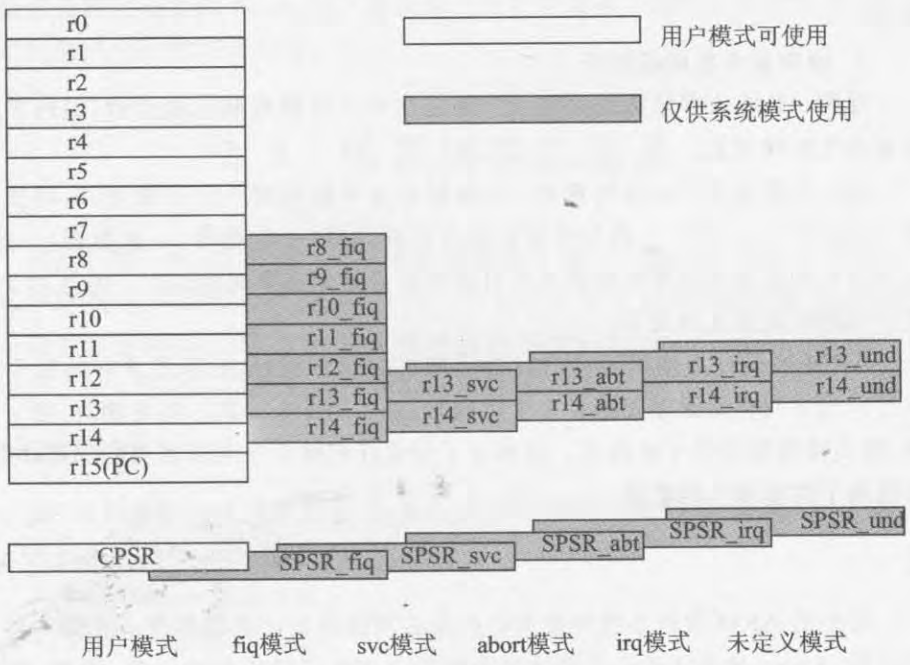


图 2.1 ARM 的可见寄存器

当前程序状态寄存器(CPSR)

CPSR 在用户级编程时用于存储条件码。例如，这些位可用来记录比较操作的结果和控制条件转移是否发生。用户级程序员通常不需要关心该寄存器是如何配置的。但为了完整，将该寄存器示于图 2.2。寄存器的低位用于控制处理器的模式(见 5.1 节)、指令集("T"，见 7.1 节)和中断使能("I"和"F"，见 5.2 节)，而且被保护以防止用户级程序改变它们。条件码标志位位于寄存器的高 4 位，意义如下：

· N：负数。改变标志位的最后的 ALU 操作产生负数结果(32 位结果的最高位

是 1)。

- Z：零。改变标志位的最后的 ALU 操作产生 0 结果(32 位结果的每一位都是 0)。
- C：进位。改变标志位的最后的 ALU 操作产生进位输出，不管是 ALU 中算术操作的结果，还是来自移位器的结果。
- V：溢出位。改变标志位的最后的算术 ALU 操作产生到符号位的溢出。

注意，尽管以上 C 和 V 的定义看起来颇为复杂，但使用时不需要详细理解它们的操作。在大多数情况下有一个简单的条件测试，不需要编程员计算出条件码的精确值即可得到需要的结果。

图 2.2　ATM CPSR 寄存器的格式

存储器系统

除了处理器的寄存器状态外，ARM 系统还有存储器状态。存储器可以看作是序号为 $0\sim2^{32}-1$ 的线性字节阵列。数据项可以是 8 位字节、16 位半字和 32 位字。字总是以 4 字节的边界对准(也就是说最低有效两位地址为 0)，半字则以偶数字节的边界对准。

存储器组织如图 2.3 所示。图中画出了存储器中的一个小区域，其中每一个字节都有惟一的号码。字节可以占有任一位置，图中给出了几个例子。长度为 1 个字的数据项占有一组 4 字节的位置，该位置开始于 4 的倍数的字节地址。图 2.3 包含了两个这样的例子。半字占有两个字节位置，该位置开始于偶数字节地址。

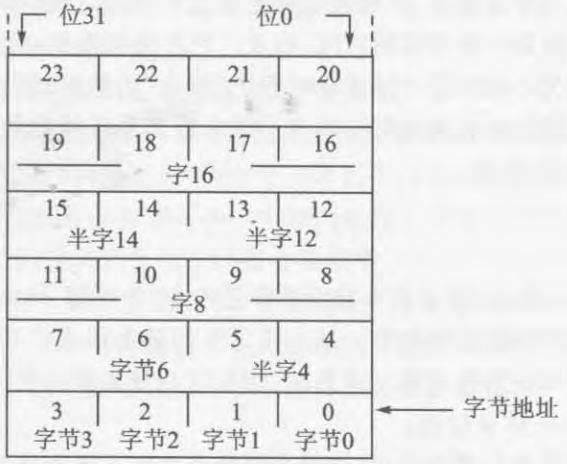


图 2.3　ARM 的存储器组织

(这是标准的“小端”格式，即 ARM 使用的存储器组织。ARM 也同样可以配置为“大端”格式存储器组织，将在第 5 章再讨论这个问题。)

Load-Store 体系结构

同大多数 RISC 处理器一样，ARM 使用 Load-Store 体系结构。这就意味着指令集仅能处理(指加、减等)寄存器中(或指令中直接指定)的值，而且总是将这些处理的结果放回到寄存器中。针对存储器状态的惟一操作是将存储器的值拷贝到寄存器(Load 指令)或将寄存器中的值拷贝到存储器(Store 指令)。

典型的 CISC 处理器允许将存储器中的值加到寄存器中的值，有时还允许将寄存器中的值加到存储器中的值。ARM 不支持这类“存储器-存储器”操作。因此，所有的 ARM 指令都属于下列 3 种类型之一，即：

1) 数据处理指令。这类指令只能使用和改变寄存器中的值。例如，一条指令能使两个寄存器相加，并将结果放到一个寄存器中。
2) 数据传送指令。这类指令把存储器中的值拷贝到寄存器中(Load 指令)或把寄存器的值拷贝到存储器中(Store 指令)。另外一种形式仅在系统代码中有用，即交换存储器和寄存器的值。
3) 控制流指令。一般指令在执行时使用存储于连续的存储器地址中的指令。控制流指令使执行切换到不同的地址。切换或者是永久的(转移指令)，或者保存返回地址以恢复原来的执行顺序(转移和链接指令)，或者陷入系统代码(监控调用)。

监控模式

ARM 处理器支持被保护的监控模式(supervisor mode)。保护机制确保用户代码在未经适当检查以确保它不会执行非法操作的情况下，不能得到监控特权。

对于用户级程序员来说，只能通过特定的监控调用才能访问系统级函数。通常这些函数包括对外围硬件寄存器的访问，以及广泛使用的操作，例如字符输入与输出。用户级程序员主要关心如何设计算法来对程序“拥有”的数据进行操作，并通过操作系统来处理与程序外部世界有关的所有事务。请求操作系统函数的指令将在 3.3 节的“监控程序调用”一段中介绍。

ARM 指令集

所有的 ARM 指令(除了将在第 7 章讲述的 16 位压缩 Thumb 指令)都是 32 位宽，在存储器中以 4 字节的边界对准。基本指令集的基本用法将在第 3 章讲述，包括 2 进制指令格式的全部细节将在第 5 章给出。ARM 指令集最鲜明的特征如下：

- Load-Store 体系结构；
- 3 地址的数据处理指令(亦即两个源操作数寄存器和结果寄存器都独立设定)；
- 每条指令都是条件执行；
- 包含非常强大的多寄存器 Load 和 Store 指令；
- 能用在单时钟周期内执行的单条指令来完成一项普通的移位操作和一项普通的 ALU 操作；
- 通过协处理器指令集来扩展 ARM 指令集，包括在编程模式中增加了新的寄存

器和数据类型；

· 在 Thumb 体系结构中以高密度 16 位压缩形式表示的指令集。

对熟悉现代 RISC 指令集的读者来说，ARM 指令集可能看起来比商用的 RISC 处理器有更多的格式。确实如此，而且这也使指令译码更复杂，但同时也带来了较高的代码密度。大多数 ARM 处理器都应用于小型嵌入式系统。对于这些系统，代码密度的优势超过了因译码复杂而导致的损失。Thumb 代码扩展这一优势，使 ARM 比大多数 CISC 处理器有更好的代码密度。

I/O 系统

ARM 将 I/O(输入/输出)外设(如磁盘控制器、网络接口等)作为支持中断的存储器映射设备来处理。这些设备中的内部寄存器如同是 ARM 存储器映射中的可寻址单元，且可以像其他存储器单元一样使用同样的(Load-Store)指令来读和写。

外设可以通过使用一般中断(IRQ)或快速中断(FIQ)发出中断请求来引起处理器的注意。这两种中断输入是电平敏感的，且可屏蔽。一般多数中断源共用 IRQ 输入端，只有一个或两个时间要求紧迫的中断源连接在中断级较高的 FIQ 输入端。

一些系统可能在处理器外部包含直接存储器访问(DMA，Direct Memory Access)硬件，以处理高带宽 I/O 的拥堵。这一点将在 11.9 节进行进一步的讨论。

中断是异常的一种形式，采用下述方法来处理。

ARM 异常

ARM 体系结构支持一系列中断、陷阱和监控调用。所有这些都归结为异常。在任何情况下处理异常的方法都是一样的，即

1) 通过将 PC 拷贝到 r14_exc 以及将 CPSR 拷贝到 SPSR_exc(在此 exc 表示异常类型)来保存当前状态。

2) 将处理器操作模式改变为适当的异常模式。

3) 将 PC 强制变为 00_{16}～$1C_{16}$ 范围内某个与异常类型有关的特殊值。

这个特殊值位置(向量地址)的指令通常是指向异常处理程序的转移指令。通常将 r13_exc 初始化，使其指向存储器中一个专用堆栈。异常处理程序将使用 r13_exc 来保存一些用户寄存器，使其能作为工作寄存器使用。

返回到用户程序是通过恢复用户寄存器，再使用指令自动地恢复 PC 和 CPSR 来完成的。这可能涉及到对存储在 r14_exc 的 PC 值进行某些调整，以便对异常发生时的流水线状态进行补偿。这将在 5.2 节中进行更详细的描述。

2.4　ARM 开发工具

ARM 公司开发了一系列工具以支持 ARM 软件的开发，也有许多第三方和公共领域的工具可以使用，诸如 ARM 后端用于 gcc 的 C 编译器。

因为 ARM 广泛地用做嵌入式控制器，其目标硬件将不会为软件准备良好的开发

环境，所以，开发工具可进行来自一个平台的**交叉开发**(这就是在不同的体系结构上运行，从其中之一产生代码)。例如，平台是一个运行 Windows 的 PC，或者是一台合适的 UNIX 工作站。ARM 交叉开发工具套件的整体结构如图 2.4 所示。将 C 或汇编语言源文件编译或汇编成 ARM 的目标格式(.aof)文件，进而链接为 ARM 映射格式(.aif)文件。映射格式文件可以被构造为包括 ARM 符号调试器(ARM 符号调试器 **ARMsd** 可以在诸如 **ARM 开发板**的硬件上或使用 ARM 的软件仿真器 **ARMulator** 来装入、运行和调试程序)所需的调试表。ARMulator 被设计为允许容易地扩展软件模型，使之包括诸如 Cache、特殊存储器时序特性等系统特征。

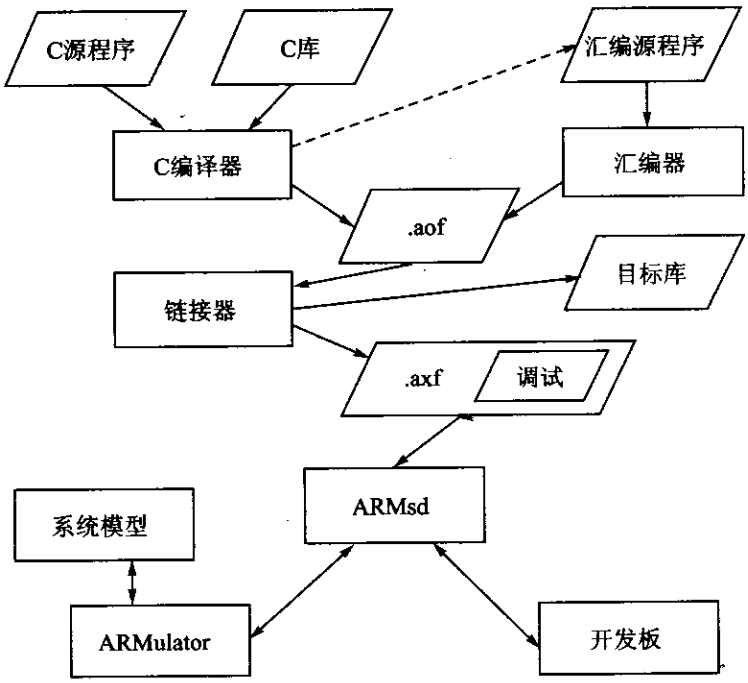


图 2.4 ARM 交叉开发工具套件的结构

ARM C 编译器

ARM C 编译器同 **ANSI**(American National Standards Institute)的 C 标准兼容，并得到适当的标准函数库的支持。对于所有可用的外部函数，它使用 ARM 过程调用标准(参看 6.8 节)。它可以产生汇编源程序输出，而不是 ARM 目标格式，因此，对代码可以检查，甚至手工优化，接着进行汇编。

编译器也可以产生 Thumb 代码。

ARM 汇编器

ARM 汇编器是完全宏汇编器，它产生 ARM 目标格式输出。输出可以与 C 编译器的输出链接。

汇编源语言接近机器级语言，大多数汇编指令被翻译成单条 ARM(或 Thumb)指令。ARM 汇编语言编程将在第 3 章介绍；ARM 指令集的全部细节，包括汇编语言格

式，将在第 5 章给出；Thumb 指令集和汇编语言格式将在第 7 章给出。

链接器

链接器将一个或多个目标文件组合为可执行的程序。它解决目标文件之间的符号引用，以及在程序需要时从库中提取目标模块。它可以用许多不同方式来汇编各种程序部件，这取决于程序代码是否运行在 RAM(Random Access Memory，它可被读和写)中，或是 ROM(Read Only Memory)中，是否需要覆盖等。

一般情况下，链接器在输出文件中包含调试表。如果目标文件在编译时使用了全部调试信息，这将包含全部的符号调试表(因此，可以使用源程序中的变量名来调试程序)。链接器也可以产生目标库模块，它不是可执行的，但可在将来同目标文件有效的链接。

ARMsd

ARMsd(ARM 符号调试器)是协助调试程序的前端接口。调试程序既可以在仿真环境下(在 ARMulator 上)运行，又可以在远程的目标系统(如 ARM 开发板)上运行。远程系统必须通过串行线或通过 JTag 测试接口(见 8.6 节)支持相应的远程调试协议。在处理器核被嵌入到专用系统芯片的情况下，系统调试非常复杂，这将在第 8 章讨论。

总的来说，ARMsd 允许可执行程序装入 ARMulator 或开发板上运行。它允许设置断点。断点是代码中的地址，如果执行到这里，它就会使执行停止，以便检查处理器的状态。在 ARMulator 上或在适当支持下在硬件中运行时，也同样允许设置观察点。观察点是存储器地址，如果作为数据地址来访问，它则会使执行以同样方式停止。

在更高层级上，ARMsd 支持完全源代码调试，使 C 程序员在调试程序时可以使用源文件来定义断点以及使用源程序中的变量名。

ARMulator

ARMulator(ARM 仿真器)是一套在主系统上用软件模拟各种 ARM 处理器核行为的程序。它可以在不同的精度级别上运行。

- 指令精度模型给出系统状态的确切行为，而不考虑处理器的精确时序特性。
- 周期精度模型逐个周期地给出处理器的确切行为，能准确给出程序所需的时钟周期数。
- 时序精度模型给出信号在周期内的准确时序，能够考虑逻辑延时。

所有这些方式的运行都比实际硬件慢得多。但是，指令精度模型的速度最快，最适合于软件开发。

简单地说，ARMulator 使得用 C 编译器或汇编器开发的 ARM 程序能够在没有 ARM 处理器连接的主机上进行测试和调试。使得程序执行所用的时钟周期数得以精确测量，因此，目标系统的性能得以评估。

复杂地说，ARMulator 可以作为目标系统的完全的、具有时序精度的 C 模型来使用，它具有全部的 Cache 细节和存储器管理功能，运行某一操作系统。

在这些两个极端的 ARMulator 之间，是一系列模型原型的模块，包括快速原型存

储器模型和协处理器接口支持(更详细的内容见 8.5 节)。

ARMulator 还可以用做在基于如 VHDL(VHDL 是标准的、被广泛支持的硬件描述语言)等语言的硬件仿真环境中具有时序精度的 ARM 行为模型的核。必须生成一个 VHDL 的"壳"以作为 ARMulator C 代码到 VHDL 环境的接口。

ARM 开发板

ARM 开发板是包含一系列元件和接口的电路板,支持基于 ARM 的系统的开发。它包括一个 ARM 核(例如 ARM7TDMI)、能配置为与目标系统存储器性能和总线宽度相匹配的存储器元件,以及能配置成为用于仿真专用外设的电可编程器。在拿到最终的专用硬件之前,能同时支持硬件和软件的开发。

软件工具套件

ARM 公司提供以上介绍的全套工具,以及一些实用程序和文档,作为 ARM 软件开发工具套件。工具套件的 CD-ROM 包括 PC 版本的工具集,它们可以运行于多数版本的 Windows 操作系统,也包括所有基于 Windows 的项目管理器。当出现新版本的 ARM 时,工具集就会更新。

ARM 项目管理器是上述工具的图形化前端。它支持由组成特殊工程的文件列表来构建单一库或可执行映射。这些文件可能是:

- 源文件(C、汇编语言等);
- 目标文件;
- 库文件。

可以在项目管理器中编辑源文件,产生相关列表,构建库输出或可执行映射。构建时有许多选项可以选择,例如:

- 输出代码大小或执行时间是否被优化。
- 用调试格式还是用发布格式输出。
  (由于从源代码到全面优化的输出之间的映射过于含糊而不适合于调试,因此,为了源级调试而编译的代码不能被充分优化。)
- 目标是哪一种 ARM 处理器(特别是它是否支持 Thumb 指令集)。

CD-ROM 还包含运行在 Sun 或 HP UNIX 主机上的工具版本,它们使用命令行形式的界面。所有的版本都有在线帮助。

JumpStart

VLSI 技术公司的 JumpStart 工具包括相似的基本开发系列工具,不过提供的是在适当工作站上的完全 X-Windows 界面,而不是标准的 ARM 工具套件的命令行界面。

还有许多其他公司提供支持 ARM 开发的工具。

2.5　例题与练习

例题　2.1

描述 ARM 体系结构的主要特征。

ARM 体系结构的主要特征如下：

- 大量的寄存器，它们都可以用于多种用途；
- Load-Store 体系结构；
- 3 地址指令（亦即两个源操作数寄存器和结果寄存器都独立设定）；
- 每条指令都条件执行；
- 包含非常强大的多寄存器 Load 和 Store 指令；
- 能在单时钟周期执行的单条指令内完成一项普通的移位操作和一项普通的 ALU 操作；
- 通过协处理器指令集来扩展 ARM 指令集，包括在编程模式中增加了新的寄存器和数据类型；

如果把 Thumb 指令集也当做 ARM 体系结构的一部分，那么还可以加上：

- 在 Thumb 体系结构中以高密度 16 位压缩形式表示指令集。

练习　2.1.1

哪条特征是 ARM 和许多其他 RISC 体系结构所共有的？

练习　2.1.2

ARM 体系结构中哪条特征是大多数其他 RISC 体系结构所不具有的？

练习　2.1.3

大多数其他 RISC 体系结构中哪条特征是 ARM 所不具有的？

第 3 章　ARM 汇编语言编程

本章内容综述

虽然在许多应用中更适于采用 C 或 C++等高级语言编程，但 ARM 处理器在汇编级编程是非常容易的。

汇编语言编程需要程序员在逐条机器指令的层次上考虑。ARM 指令长度为 32 位，因此，大致有 40 亿种不同的二进制机器指令。幸运的是，指令空间有良好的结构，程序员不必每个人都熟悉所有这 40 亿种二进制编码。即使如此，每条指令还是有很多需要掌握的细节。汇编器会为程序员处理大多数这类细节。

在这一章中，我们将着重于用户级的 ARM 汇编语言编程，并指出如何编写一个简单的程序，使它可以运行于 ARM 开发板或 ARM 仿真器(例如，作为 ARM 开发工具套件之一的 ARMulator)。熟悉了基本指令集之后，将在第 5 章中着重于介绍系统级的编程，以及 ARM 指令集的细节，包括指令的二进制编码。

一些 ARM 处理器支持压缩成 16 位 Thumb 指令的指令集，这些将在第 7 章讨论。

3.1　数据处理指令

ARM 的数据处理指令使得程序员能够完成寄存器中数据的算术和逻辑操作。其他指令只是传送数据和控制程序执行的顺序。因此，数据处理指令是惟一可以修改数据值的指令。这些指令的典型特征是需要两个操作数，产生单个结果。不过上述两个特征也有例外。典型的操作是将两个数加在一起产生单个结果，即它们的和。

下面是一些应用于 ARM 数据处理指令的原则：

- 所有的操作数是 32 位宽，或来自寄存器，或在指令中定义的立即数。
- 如果有结果，则结果为 32 位宽，放在一个寄存器中。(有一个例外是：长乘指令产生 64 位的结果，这将在 5.8 节讨论。)
- 每一个操作数寄存器和结果寄存器都在指令中独立地指定，这就是说，ARM 指令中使用 3 地址模式。

简单的寄存器操作

典型的 ARM 数据处理指令用汇编语言写出的格式如下：

```
ADD     r0, r1, r2                    ; r0 := r1 + r2
```

这一行的分号表示它的右边是注释，应该被汇编器忽略。在汇编源代码中加入注释使它易读和易理解。

这个例子只是简单地取两个寄存器(r1 和 r2)的值，将它们加起来，将结果放在第 3 个寄存器。源寄存器的值是 32 位宽，可以认为是无符号整数或有符号整数的 2 的补码值。加法可能产生进位输出，或在有符号数的 2 的补码的加法中产生到符号位的内部溢出。但这两种情况都会被忽略。

在写汇编语言源代码时，必须注意操作数的正确顺序，第一个是结果寄存器，然后是第一操作数，最后是第二操作数(对于交换操作，第一和第二操作数都是寄存器，它们的次序并不重要)。当这些指令执行时，对系统状态而言惟一的变化是目的寄存器 r0 的值(CPSR 中的标志位 N、Z、C 和 V 的值也会有选择地变化，这些稍后我们将看到)。

按照这种格式，各种指令分类排列如下：

1. 算术操作

这类指令对两个 32 位操作数进行二进制算术操作(加、减和反向减，后者指把操作数次序颠倒后相减)。操作数可以是无符号的或以 2 为补码的有符号整数。当使用进位位时，进位位为当前 CPSR 中 C 位的值。

```
ADD     r0, r1, r2                    ; r0 := r1 + r2
ADC     r0, r1, r2                    ; r0 := r1 + r2 + C
SUB     r0, r1, r2                    ; r0 := r1 - r2
SBC     r0, r1, r2                    ; r0 := r1 - r2 + C - 1
RSB     r0, r1, r2                    ; r0 := r2 - r1
RSC     r0, r1, r2                    ; r0 := r2 - r1 + C - 1
```

ADD 是简单的加法，ADC 是带进位的加法，SUB 是减法，SBC 是带进位的减法，RSB 是反向减法，RSC 带进位的反向减法。

2. 按位逻辑操作

这类指令对输入操作数的对应位进行指定的布尔逻辑操作。例如，下面的第一个例子即是对第 0～31 位执行“r0[i] :=r1[i] AND r2[i]”操作。式中 r0[i]是 r0 的第 i 位。

```
AND     r0, r1, r2                    ; r0 := r1 and r2
ORR     r0, r1, r2                    ; r0 := r1 or r2
EOR     r0, r1, r2                    ; r0 := r1 xor r2
BIC     r0, r1, r2                    ; r0 := r1 and not r2
```

3. 寄存器传送操作

这些指令不用第一操作数，它在汇编语言格式中被省略。传送指令只是简单地将第二操作数(可能按位取反)传送到目的寄存器。

```
MOV     r0, r2                        ; r0 := r2
MVN     r0, r2                        ; r0 := not r2
```

MVN 助记符意为“取反传送”，它是把源寄存器的每一位取反，将得到的值置入结果寄存器。

4. 比较操作

这类指令不产生结果(因此,结果在汇编语言格式中被省略),仅根据所选择的操作来设置 CPSR 中的条件代码位(N、Z、C 和 V)。

```
CMP     r1, r2          ; 根据 r1－r2 的结果设置 cc
CMN     r1, r2          ; 根据 r1－r2 的结果设置 cc
TST     r1, r2          ; 根据 r1 and r2 的结果设置 cc
TEQ     r1, r2          ; 根据 r1 xor r2 的结果设置 cc
```

助记符表示比较(CMP)、取反比较(CMN)、(位)测试(TST)、测试相等(TEQ)。

立即数操作

如果只是希望把一个常数加到寄存器,而不是两个寄存器相加,可以用立即数值取代第二操作数。立即数值是常量(literal),前面加一个"#"。

```
ADD     r3, r3, #1          ; r3 := r3 + 1
AND     r8, r7, #&ff        ; r8 := r7[7:0]
```

第一个例子也说明虽然 3 地址格式允许分别指定源和目的操作数,但是,它们可以使用同一个寄存器。第二个例子中,在"#"后的"&"表示以 16 进制(基于 16)的形式定义立即数。

因为立即数的值是在 32 位指令字内编码,所示不可能将所有可能的 32 位值作为立即数。只能是一个 8 位数按 2 位为边界进行的调整。多数有效的立即数可表示为

$$\text{立即数}=(0\rightarrow 255)\times 2^{2n} \tag{10}$$

式中:$0\leqslant n\leqslant 12$。汇编器也会用 MVN 代替 MOV,用 SUB 代替 ADD 等等。这样可以把立即数置于可以设置的范围之内。

这样做看来会对立即数的值带来复杂的限制,但实际上它确实覆盖了大多数一般情形,例如在 32 位字 4 个字节中任何一个字节的值,2 的任何次幂等等。在任何情况下,如果遇到无法编码的数,则汇编器都会报告。

(对立即数的值限制的原因在于它是在 2 进制指令的级别上指定的。在第 5 章中将说明这一点。想要充分理解这个问题的读者应到第 5 章寻找完整的解释。)

寄存器移位操作

第三种定义数据操作的方式同第一种类似,但允许第二个寄存器操作数在同第一操作数运算之前完成移位操作,例如:

```
ADD     r3, r2, r1, LSL #3          ; r3 := r2 + 8 × r1
```

注意它是一条 ARM 指令,在单个时钟周期内执行。许多处理器采用独立的指令提供移位操作,但 ARM 将它们同基本的 ALU 操作合并在单条指令中。

这里 LSL 意指"逻辑左移指定的位数",在该例中为 3。可以指定 0～31 范围内所有的数,但是,"0"相当于忽略移位操作。如前所述,"#"表示立即数。可以得到的移位操作如下:

- LSL：逻辑左移 0～31 位，空出的最低有效位用 0 填充。
- LSR：逻辑右移 0～32 位，空出的最高有效位用 0 填充。
- ASL：算术左移，与 LSL 同义。
- ASR：算术右移 0～32 位。如果源操作数是正数，则空出的最高有效位用 0 填充；如果是负数，则用 1 填充。
- ROR：循环右移 0～32 位，移出的字的最低有效位依次填入空出的最高有效位。
- RRX：扩展 1 位的循环右移，空位（第 31 位）是用原来标志位 C 填充，操作数右移 1 位。适当地使用条件码（见下），可以执行操作数和标志位 C 的 33 位循环操作。

这些移位操作如图 3.1 所示。

同样可以使用寄存器值定义第二操作数的移位位数，例如：

```
ADD     r5, r5 ,r3, LSL r2              ; r5 := r5 + r3 × 2^r2
```

这是 4 地址指令。只有 r2 的低 8 位是有意义的。但由于移位超过 32 位不是非常有用，所以，这种限制对于许多用途是不重要的。

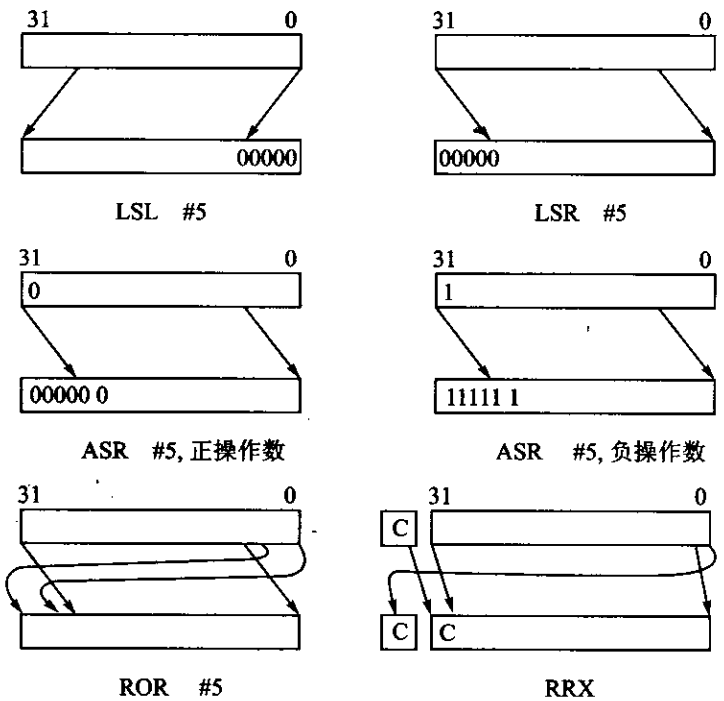


图 3.1　ARM 移位操作

设置条件码

如果程序员需要，那么任何数据处理指令都能设置条件码（N、Z、C 和 V）。比较操作只能设置条件码，因此，对比较操作来说别无选择。但对所有其他数据处理指令则必

须专门提出要求。在汇编语言级，这种要求以增加 S 操作码来指明，意为“设置条件码”。例如，下面的代码完成两个数的 64 位加法，一个数存于 r0＋r1，另一个数存于 r2＋r3，用 C 条件码标志位存立即数进位，即

```
ADDS    r2, r2, r0          ; 32 位进位输出→C…
ADC     r3, r3, r1          ; …再加到高位字中
```

由于操作码的 S 扩展使程序员能够控制指令是否修改条件码，所以，在适当时，可以在长指令序列中把条件代码保护起来。

算术操作(在此包含 CMP 和 CMN)根据算术运算的结果设置所有标志位。逻辑或传送操作不产生有意义的 C 或 V 值，这些操作根据结果来设置 N 和 Z，保留 V。当没有移位操作时，保留 C；或者当移位时，将最后移位移出的最后位设置为 C。这些细节通常意义不大。

条件码的使用

我们已看到标志位 C 作为算术数据处理指令的输入来使用的情形，然而还未看到条件码最重要的用途，这就是通过条件转移指令来控制程序流。这些将在 3.3 节中介绍。

乘　法

有专门的数据处理指令支持乘法，即

```
MUL     r4, r3, r3          ; r4 := ( r3 × r2 )[31:0]
```

它同其他算术指令有一些重要的不同，即

- 不支持第二操作数为立即数。
- 结果寄存器不允许同为第一源寄存器。
- 如果设置位 S，则标志位 V 保留(如同逻辑指令)，而且标志位 C 不再有意义。

两个 32 位整数相乘得到 64 位结果，将低 32 位有效位放在结果寄存器，其余的忽略。这可以被看作是模 2^{32} 算法的乘法，无论操作数被看作是有符号的还是无符号的整数，都可以给出正确的结果。(ARM 也支持长乘，高 32 位有效位放入第二个结果寄存器，这些将在 5.8 节描述。)

受到同样制约的还有一种指令，把乘积加到一个滚动的和数。这就是乘加指令：

```
MLA     r4, r3, r2, r1      ; r4 := ( r3 × r2 + r1)[31:0]
```

乘以一个常数可以由调一个常数到寄存器，然后使用这些指令中的一种来实现。但是，使用移位和加法或减法构成一小段数据处理指令通常更加有效。例如，将 r0 乘以 35：

```
ADD     r0, r0, r0, LSL #2  ; r0′ := 5 × r0
RSB     r0, r0, r0, LSL #3  ; r0″ := 7 × r0′(=35×r0)
```

3.2　数据传送指令

数据传送指令在 ARM 寄存器和存储器中间传送数据。ARM 指令集中有 3 种基本的数据传送指令：

1. 单寄存器的 Load 和 Store 指令

这些指令在 ARM 寄存器和存储器之间提供最灵活的单数据项传送方式。数据项可以是字节、32 位字或 16 位半字(原 ARM 芯片可能不支持半字)。

2. 多寄存器 Load 和 Store 指令

这些指令的灵活性比单寄存器传送指令差，但可以使大量的数据更有效地传送。它们用于进程的进入和退出，保存和恢复工作寄存器，以及拷贝存储器中一块数据。

3. 单寄存器交换指令

这些指令允许寄存器和存储器中的数值进行交换，在 1 条指令中有效地完成 Load 和 Store 操作。它们在用户级编程中很少使用，在本节中不作进一步的讨论。它的主要用途是在多处理器系统中实现信号标(semaphores)，以保证不会同时访问公用的数据结构。如果读者现在还不明白这些话的意思，也不必担心。

寄存器间接寻址

在 1.4 节的后部，是有关存储器寻址机制的讨论，这对处理器指令集的设计者是有用的。ARM 的数据传送指令都是基于寄存器间接寻址，还包括基址偏移和基址变址的寻址模式。

寄存器间接寻址利用一个寄存器的值(基址寄存器)作为存储器地址，或者从该地址取值存放到寄存器，或者将另一个寄存器的值存入该存储器地址。

这些指令的汇编语言格式如下：

```
LDR     r0, [r1]        ; r0 := mem32[r1]
STR     r0, [r1]        ; mem32[r1] := r0
```

其他形式的寻址都是建立在这种形式上，并在基址上加上立即数或寄存器偏移量。在任何情况下都需要有一个 ARM 寄存器来寄存地址。该地址靠近需要传送数据的地址，所以，我们将首先探讨将地址放入寄存器的方式。

初始化地址指针

要访问一个特定的存储器单元，必须把一个 ARM 寄存器初始化，使之包含这个单元的地址。在单寄存器传送指令的情况下，也可以是距这个单元地址 4 KB 之内的一个地址(后面将对 4 KB 的范围进行解释)。

如果要存取的数据地址靠近正被执行的代码，那么程序计数器 r15 的内容则接近所需的数据地址。通常都可以利用这种情形。可以用一条数据处理指令给 r15 增加微小的偏移量，但是，偏移量的计算可能不是那么简单。然而，这类复杂的计算正是汇编

器的所长。而且 ARM 汇编器有一个内置的伪指令 ADR，它使计算更加简单。在汇编源代码中，伪指令就像一般的指令，只是它没有直接对应于特定的 ARM 指令。在使用伪指令时，汇编器有一系列规则，它能按照这些规则选择最适当的指令或几条相连的指令。（事实上，ADR 总是被编译成单条 ADD 或 SUB 指令。）

作为一个例子来考虑一个程序，它必须从 TABLE1 向 TABLE2 拷贝数据。TABLE1和 TABLE2 都接近代码。

```
COPY    ADR     r1, TABLE1          ; r1 指向 TABLE1
        ADR     r2, TABLE2          ; r2 指向 TABLE2
        ...
TABLE1                              ; <数据源>
        ...
TABLE2                              ; <目标>
        ...
```

这里引入了**标号**（COPY、TABLE1 和 TABLE2），它只是在汇编语言代码中给一个特定位置赋予的名字。第一个 ADR 伪指令使得 r1 包含 TABLE1 之后的数据地址；第二个 ADR 同样使得 r2 包含从 TABLE2 开始的存储器地址。

当然 ARM 指令能用来计算存储器数据项的地址。但是，想要缩短程序，则需要发挥 ADR 伪指令的作用。

单寄存器 Load 和 Store 指令

这些指令使用基址寄存器来计算传送数据的地址。基址寄存器应该包含一个接近目标地址的地址，还要计算偏移量。它可能是另外一个寄存器或者立即数。

下面仅看一下这些指令中最简单的，它没有使用地址偏移量。

```
LDR     r0, [r1]        ; r0 := mem32[r1]
STR     r0, [r1]        ; mem32[r1] := r0
```

这里使用的符号指出数据的属性是由 r1 为地址的 32 位存储器字，r1 中的地址应对准 4 字节的分界处，因此，r1 的两位最低有效位应是 0。现在可以把一个表中的第一个字拷贝到另一个表中，即

```
COPY    ADR     r1, TABLE1          ; r1 指向 TABLE1
        ADR     r2, TABLE2          ; r2 指向 TABLE2
        LDR     r0, [r1]            ; 加载第一个数据…
        STR     r0, [r2]            ; 将它存入 TABLE2
        ...
TABLE1                              ; <数据源>
        ...
TABLE2                              ; <目标>
        ...
```

现在能使用数据处理指令为下一次传送修改基址寄存器，即

```
COPY    ADR     r1, TABLE1          ; r1 指向 TABLE1
        ADR     r2, TABLE2          ; r2 指向 TABLE2
LOOP    LDR     r0, [r1]            ; 取 TABLE1 第一个数据
        STR     r0, [r2]            ; 拷贝到 TABLE2
        ADD     r1, r1, #4          ; r1 进 1 个字
        ADD     r2, r2, #4          ; r2 进 1 个字
        ???                         ; 若拷贝多字,则返回 LOOP
        …
TABLE1                              ; <数据源>
        …
TABLE2                              ; <目标>
        …
```

注意基址寄存器增加 4(字节),因为这是字的长度。如果基址寄存器在增加前是字对准的,那么增加后也是字对准的。

所有的 Load 和 Store 指令只能使用这种寄存器间接寻址的简单形式。而 ARM 指令集包含更多的寻址模式,使得代码更高效。

基址偏移寻址

如果基址寄存器不包含确切的地址,则可以在基址上加上不超过 4 KB 的偏移量来计算传送地址,即

```
LDR     r0, [r1, #4]                ; r0 := mem32[r1 + 4]
```

这是一个**前变址**(pre-indexed)寻址模式。采用这种模式可以使用一个基址寄存器来访问位于同一区域的多个存储器单元。

有时修改基址寄存器的内容使之指向数据传送地址是很有用处的。可以使用带有**自动变址**的前变址寻址来实现对基址寄存器的修改,这样可以让程序追踪一个数据表,即

```
LDR     r0, [r1, #4]!               ; r0 := mem32[r1 + 4]
                                    ; r1 := r1 + 4
```

上面程序中的惊叹号表示在开始传送数据后,基址寄存器将更新,在 ARM 中自动变址并不花费额外的时间,因为这个过程是在数据从存储器中取出的同时,在处理器的数据通路中完成的。它严格地等效于先执行一条简单的寄存器间接 Load 指令,再执行一条数据处理指令,以向基址寄存器加一个偏移量(本例为 4 字节),但避免了额外的指令时间和代码空间开销。

另一个有用的指令形式叫做**后变址**(post-indexed)寻址,它允许基址不加偏移即作为传送地址使用,而后再自动变址,即

```
LDR     r0, [r1], #4                ; r0 := mem32[r1]
                                    ; r1 := r1 + 4
```

在此不再需要感叹号,因为立即数偏移量的惟一用途是作为基址寄存器修改量。

这种形式的指令严格地等效于简单的寄存器间接寻址加载，后面再加一条数据处理指令，但它速度快并占用较小的代码空间。

现在能够使用最后这种形式来改进先前介绍的表拷贝程序，即

```
COPY    ADR    r1, TABLE1          ; r1 指向 TABLE1
        ADR    r2, TABLE2          ; r2 指向 TABLE2
LOOP    LDR    r0, [r1], #4        ; 取 TABLE1 第一个数据
        STR    r0, [r2], #4        ; 拷贝到 TABLE2
        ???                        ; 若拷贝多字，则返回 LOOP
        ...
TABLE1                             ; <数据源>
        ...
TABLE2                             ; <目标>
        ...
```

Load 和 Store 指令被重复执行，直到将所需数量的数据拷贝到 TABLE2 为止，然后循环退出。需要控制流指令来确定循环何时退出。下面会简单地介绍控制流指令。

在以上例子中，基址寄存器的地址偏移一直是一个立即数。它同样可以是另一个寄存器，并且在加到基址寄存器前还可以经过移位操作。但与立即数偏移形式相比，这种形式的指令很少使用。这在 5.10 节会全面讨论。

作为最后的一种变形，传送数据的大小可以是一个无符号 8 位字节，而不是 32 位字。在操作码中增加一个字母 B 即可选用这个选项，即

```
LDRB    r0, [r1]                   ; r0 := mem8[r1]
```

在这种情况下，传送的地址可以对准任意字节，而不限于 4 字节的分界处，因为字节可以存在任意字节的地址中。取出的字节被放在 r0 最低的字节，r0 的其余字节填充 0。

（除了最早期的外，所有的 ARM 处理器也支持**有符号**字节，字节的最高位表示该值是作为正数处理还是作为负数处理，也支持有符号和无符号的 16 位半字。当在5.11节再一次更详细地考虑指令集时，将会解释这些变量。）

多寄存器数据传送

当大量的数据需要传送时，最好能同时存取几个寄存器。这些指令允许 16 个寄存器的任意子集合（或全部）用单条指令传送。但是与单寄存器传送指令相比，多寄存器数据传送可用的寻址模式更加有限。

这类指令的一个简单例子如下：

```
LDMIA   r1, {r0, r2, r5}           ; r0 := mem32[r1]
                                   ; r2 := mem32[r1+4]
                                   ; r5 := mem32[r1+8]
```

因为传送的数据项总是 32 位字，基址地址（r1）应是字对准的。

在花括弧中的传送列表可以包含 r0～r15 的任意寄存器或全部寄存器。列表中寄

存器的次序是不重要的，它不影响传送的次序和指令执行后寄存器中的值。但是，一般的习惯是在列表中按递增的次序设定寄存器。

注意，如果在列表中含有 r15，将引起控制流发生变化，因为 r15 是 PC。当讨论控制流指令时，会再返回来考虑这种情况。在那时之前我们不作进一步的考虑。

上面的例子说明了所有这类指令的一般特征：最低的寄存器被保存到最低地址或从最低地址中取数；其次是其他寄存器按照寄存器号的次序保存到第一个地址后面的相邻地址或从中取数。然而依第一个地址形成的方式会产生几种变形，而且还可以使用自动变址（也是在基址寄存器后加“!”）。

堆栈寻址

寻址的变形基于这样的事实，即这类指令的一个用途就是在存储器中实现堆栈。堆栈是一种后进先出的存储形式，它支持简单的动态存储器分配。这就是说，对于被用来存储数据的存储区域，其存储器分配在编译或汇编时是不知道的。递归函数就是一个例子，递归的深度取决于参变量。堆栈经常作为一个线性的数据结构来实现。当加入数据时，它就向上增大存储空间（**递增**堆栈）或向下增大存储空间（**递减**堆栈）；而数据移走时，它又缩回来。**堆栈指针**总是保持在当前堆栈顶的地址。它指向最后压入堆栈的有效数据（**满**栈），或者指向将被压入下一个数据的空位（**空**栈）。

以上的描述说明，对于堆栈有 4 个变种，分别表征由递增、递减、满栈、空栈组成的所有组合。ARM 多寄存器传送指令支持全部 4 种形式的堆栈。

- *满递增*：堆栈随着增大存储器地址而向上增长，基址寄存器指向存储有效数据的最高地址。
- *空递增*：堆栈随着增大存储器地址而向上增长，基址寄存器指向堆栈上方的第一个空位。
- *满递减*：堆栈随着减小存储器地址而向下增长，基址寄存器指向存储有效数据的最低地址。
- *空递减*：堆栈随着减小存储器地址而向下增长，基址寄存器指向堆栈下方的第一个空位。

块拷贝寻址

虽然关于多寄存器传送指令的堆栈概念是很有用的，但是，有时其他的概念更容易理解。例如，当用这类指令把一个数据块从存储器的一个位置拷贝到另一个位置时，寻址过程的机器概念更有用。因此，ARM 汇编器支持关于寻址机制两种不同的概念。这两种概念都可映射为同样的基本指令，而且可以互换使用。块拷贝概念基于数据被存储到基址寄存器中的地址以上还是以下，以及地址的增减开始于存储了第一个数据之前还是之后。对于 Load 和 Store 操作，这两种概念的映射是不同的，详细映射情况列于表 3.1。

图 3.2 说明了块拷贝的概念。从图中可看出每条指令如何将 3 个寄存器的数据存入存储器，以及在使用自动变址的情况下基址寄存器是如何改变的。在执行指令之前基址寄存器为 r9，自动变址之后为 r9′。

表 3.1 多寄存器 Load 和 Store 指令的堆栈和块拷贝对照

		递增		递减	
		满	空	满	空
增值	先增	STMIB STMFA			LDMIB LDMED
	后增		STMIA STMEA	LDMIA LDMFD	
减值	先减		LDMDB LDMEA	STMDB STMFD	
	后减	LDMDA LDMFA			STMDA STMED

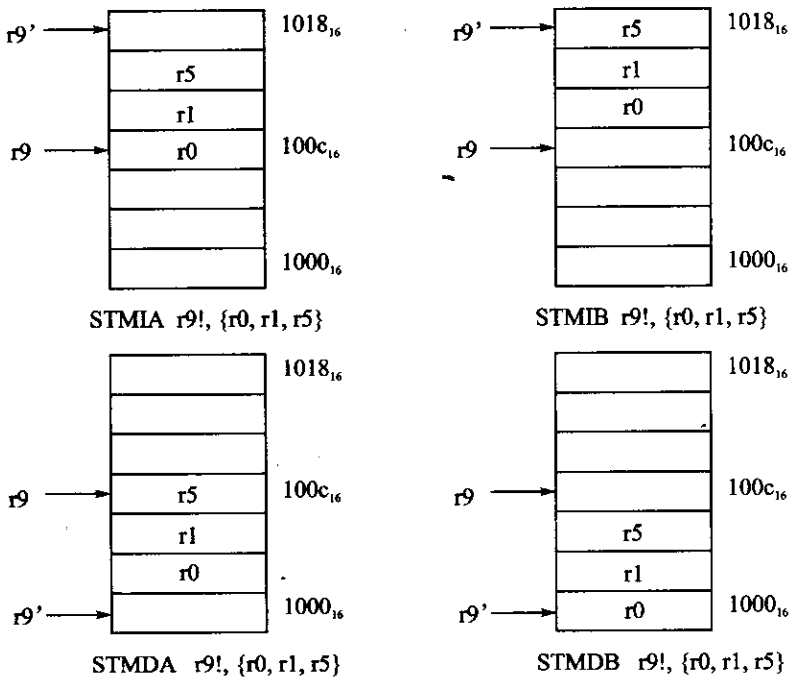


图 3.2 多寄存器传送寻址模式

下面用两条指令来说明这类指令的用途。它们把 8 个字从 r0 指向的位置拷贝到 r1 指向的位置，即

```
LDMIA    r0 !, {r2－r9}
STMIA    r1, {r2－r9}
```

这些指令执行后，r0 增加了 32。这是由于“!”使其自动变址 8 个字，而 r1 没有改变。如果 r2～r9 含有有用的数据，则可以把它们压入堆栈，从而在操作过程

中把它们保留起来，即

```
STMFD   r13!, {r2 －r9}          ; 将寄存器存入堆栈
LDMIA   r0!, {r2－r9}
STMIA   r1, {r2－r9}
LDMFD   r13!, {r2－r9}           ; 从堆栈中恢复
```

这里第一行和最后一行指令的后缀“FD”表示前面所述的满递减堆栈地址模式。注意，在堆栈操作中几乎总是要指定自动变址，以便保证堆栈指针具有一致的行为。

多寄存器 Load 和 Store 指令为保存和恢复处理器状态以及在存储器中移动数据块提供了一种很有效的方式。它节省代码空间，使操作速度比顺序执行等效的单寄存器 Load 和 Store 指令快达 4 倍(因改善后续行为而提高两倍，因减少指令数而提高将近两倍)。这个重要的优点说明，值得认真考虑数据在存储器内的组织方式，以便增大使用多寄存器传送指令去访问存储器的潜力。

或许这些指令不是纯 RISC 的，即使使用分开的指令和数据 Cache，也不能在单个时钟周期内执行。但是，其他 RISC 结构正在开始采用多寄存器传送指令，以便增加处理器的寄存器和存储器之间的数据带宽。

在另一方面，我们将会看到，多寄存器 Load 和 Store 指令实现起来是很复杂的。

ARM 多寄存器传送指令有独特的灵活性，能够传送 16 个当前可见寄存器的任何子集合。ARM 过程调用机制充分利用了这个优点，在 6.8 节将给予说明。

3.3　控制流指令

第 3 类指令既不处理数据，也不存取数据。它只是确定哪一条指令应在下一步执行。

转移指令

将程序的执行从一个位置切换到另一个位置最常用的方法是使用转移(branch)指令，即

```
        B       LABEL
        ...
LABEL   ...
```

通常，处理器顺序地执行指令。但当它执行到转移指令时，它直接执行 LABLE 处的指令，而不是执行紧跟在转移指令后面的指令。在本例中，程序的转移指令后面出现了 LABLE，所以，两者之间的指令就被跳过。但是，LABLE 也同样可以在转移指令前面出现。在这种情况下，处理器将返回到那里，并有可能重复执行一些早已执行过的指令。

条件转移

有时想让微处理器决定是否进行转移。例如，为了实现循环，需要转移回到循环的开始。但是，这种转移应该仅发生在执行到所需的循环次数之前。在这以后转移则应被跳过。

控制循环退出的机制是条件转移。这时，转移是同条件联系在一起的，只有条件码具有正确的值时转移才被执行。一种典型的循环控制指令序列可能如下：

```
        MOV     r0, #0          ; 计数器初始化
LOOP    ...
        ADD     r0, r0, #1      ; 循环计数器加 1
        CMP     r0, #10         ; 与循环的限制比较
        BNE     LOOP            ; 如果不相等，则返回
        ...                     ; 否则循环中止
```

这个例子给出了一类条件转移，即 BNE，或“不等则转移”。条件有许多种形式。表 3.2 中列出了所有这些形式，以及对它们的标准解释。列在表中同一行的一对条件

表 3.2 转移条件

转 移	解 释	一般应用
B	无条件的	总是执行转移
BAL	总是	总是执行转移
BEQ	相等	比较的结果为相等或零
BNE	不等	比较的结果为不等或非零
BPL	正	结果为正数或零
BMI	负	结果为负数
BCC	无进位	算术操作未得到进位
BLO	低于	无符号数比较，结果为低于
BCS	有进位	算术操作得到了进位
BHS	高于或相等	无符号数比较，结果为高于或相等
BVC	无溢出	有符号整数操作，未出现溢出
BVS	有溢出	有符号整数操作，出现溢出
BGT	大于	有符号整数比较，结果为大于
BGE	大于或相等	有符号整数比较，结果为大于或相等
BLT	小于	有符号整数比较，结果为小于
BLE	小于或相等	有符号整数比较，结果为小于或相等
BHI	高于	无符号数比较，结果为高于
BLS	低于或相等	无符号数比较，结果为低于或相等

(例如 BCC 和 BLO)的涵义相同,它们得出同样的二进制代码。但两者都是有用的,因为在特定的环境中,每一种条件都可能使汇编语言源代码的编译更加容易。当表中提到有符号数和无符号数的比较时,它并不是要选择比较指令本身,而只是支持操作数选择的解释。

条件执行

ARM 指令集有一条不寻常的特征,就是条件执行不仅应用于转移指令,也应用于所有的 ARM 指令。一条转移指令本来是用于跳过其后的几条指令的,但如果给予这些指令以相反的条件,则转移将被忽略。例如,考虑下面的指令序列:

```
        CMP     r0, #5
        BEQ     BYPASS                  ; if (r0 !=5) {
        ADD     r1, r1, r0              ; r1 := r1 + r0-r2
        SUB     r1, r1, r2              ; }
BYPASS  ...
```

这可以被替代为

```
        CMP     r0, #5                  ; if (r0 !=5) {
        ADDNE   r1, r1, r0              ; r1 := r1 + r0-r2
        SUBNE   r1, r1, r2              ; }
        ...
```

新的指令序列比原先的既短小又快速。不管条件序列是 3 条指令还是更少,使用条件执行都要比使用转移为好,即使被跳过的指令序列与其中的条件代码并不作什么复杂的操作。

(对这 3 条指令采取的方针是基于下述事实,即 ARM 转移指令一般要用 3 个周期来执行,而且这仅仅是个方针。如果代码被充分优化,那么是使用条件执行还是转移,必须根据代码动态行为的测量来作决定。)

要激活条件执行,须在 3 字符的操作码之后增加 2 字符的条件码(条件码应在其他任何修正码之前,例如在数据处理指令中控制设置条件码的"S"或指定字节存取的"B")。

要强调这种技术的应用范围,注意任何 ARM 指令,包括监控程序调用和协处理器指令都可以附加条件。如果条件不满足,指令便会被跳过。

有时巧妙地使用条件,有可能写出非常简练的代码,例如:

```
; if ( ( a = = b ) & & ( c = = d ) ) e++;

CMP     r0, r1
CMPEQ   r2, r3
ADDEQ   r4, r4, #1
```

注意,如果第一个比较发现操作数不同,则第二个比较被跳过,并导致加 1 指令也被跳过。由于第二个比较指令使用了条件执行,从而实现了 if 语句中的逻辑"与"。

转移和链接指令

在一个程序中通常需要能转移到子程序中，并且当子程序执行完毕时能确保恢复到原来的代码位置。这就需要把执行转移之前程序计数器的值保存下来。

ARM 使用转移和链接指令来提供这一功能。该指令完全像转移指令一样地执行转移，还把转移后面紧接的一条指令的地址保存到链接寄存器 r14 中。

```
        BL      SUBR                    ; 转移到 SUBR
        ...                             ; 返回到这里
SUBR    ...                             ; 子程序入口
        MOV     pc, r14                 ; 返回
```

注意，由于返回地址保存在寄存器里，在保存 r14 之前子程序不应再调用下一级的嵌套子程序；否则，新的返回地址将覆盖原来的返回地址，就无法返回到原来的调用位置。这时常规的机制是把 r14 压入存储器中的堆栈。由于子程序经常还需要一些工作寄存器，所以，可以使用多寄存器 Store 指令同时把这些寄存器中原有的数据一起存储。

```
        BL      SUB1
        ...
SUB1    STMFD   r13!, {r0-r2, r14}     ; 保存工作和链接寄存器
        BL      SUB2
        ...
SUB2    ...
```

不调用其他子程序的子程序(叶子程序)不需要存储 r14，因为它不会被覆盖。

子程序返回指令

为了返回调用程序，必须将转移链接指令保存到 r14 中的值拷贝回程序寄存器。对于在最简单的叶子程序，一条 MOV 指令，再利用程序计数器 r15 的可见性就足够了。

```
SUB2    ...
        MOV     pc, r14                 ; 把 r14 拷贝到 r15 来返回
```

事实上，程序计数器 r15 的可见性意味着任何数据处理指令都可以用来计算返回地址，尽管 MOV 指令是至今最常用的形式。

对于返回地址压入堆栈的情况，返回地址和任何保存的工作寄存器都可用多寄存器 Load 指令恢复。

```
SUB1    STMFD   r13!, {r0-r2, r14}     ; 保存工作寄存器和链接
        BL      SUB2
        ...
        LDMFD   r13!, {r0-r2, pc}       ; 恢复工作寄存器并返回
```

这里要注意返回地址是直接恢复到程序计数器，而不是链接寄存器。这种单条恢复和返回指令是非常有用的。还要注意多寄存器传送寻址模式的堆栈概念的使用。Store 和 Load 使用了同样的堆栈模式(在此为“满递减”，ARM 代码最常用的堆栈类型)，确保能收集到正确的数据。对于任何特定的堆栈，使用同样的寻址模式对于堆栈的任何应用都是非常重要的，除非你知道你正在干什么。

监控程序调用

只要程序需要输入或输出，例如把一些文本送到显示器，通常要调用监控程序。监控程序是一个运行于特权级别的程序，这就意味着它可以做用户级程序不能直接做的事情。对用户级程序效力的限制在不同的系统中是不同的。但是，在很多系统中，用户不能直接访问硬件设备。

监控程序提供了委托访问系统资源的方式，对用户级程序它更像一个专门的子程序入口。指令集包含一个专门的指令 SWI，用来调用这类功能。(SWI 代表软件中断，但人们通常将它视为“监控程序调用”。)

监控程序调用是在系统软件中实现的，因此，从一个 ARM 系统到另一个系统的监控程序调用可能会完全不同。尽管如此，大多数 ARM 系统在实现特定应用所需的专门调用之外，还实现了一个共同的调用子集。其中最有用的是把底部字节 r0 中的字符送到用户显示器件的一段程序：

```
SWI     SWI_WriteC                  ; 输出 r0[7:0]
```

另一个有用的调用把控制从用户程序返回到监视程序，即

```
SWI     SWI_Exit                    ; 返回到监视程序
```

SWI 的操作将在 5.6 节详细讲解。

跳转表

经验较少的编程人员通常不使用跳转表(jump table)。因此，如果读者在汇编级编程方面相对还是个新手，则可以忽略这一节。

跳转表的思想是程序员有时想调用一系列子程序中的一个，而决定究竟调用哪一个须由程序的计算值确定。显然用已有的指令完成这件事也是可能的。假设这个值在 r0 中，可以写成：

```
        BL      JUMPTAB
        ...
JUMPTAB CMP     r0, #0
        BEQ     SUB0
        CMP     r0, #1
        BEQ     SUB1
        CMP     r0, #2
        BEQ     SUB2
        ...
```

然而当子程序列表较长时，这种解决方案则变得非常慢，除非确认后面的选择很少使用。在此情况下，一个更有效的解决方案是利用程序计数器在通用寄存器文件中的可见性来实现。

```
        BL      JUMPTAB
        ...
JUMPTAB ADR     r1, SUBTAB              ; r1→SUBTAB
        CMP     r0, #SUBMAX             ; 检查超限…
        LDRLS   pc, [r1, r0, LSL #2]    ; …如果 OK，则跳转到表中
        B       ERROR                   ; …否则，发出错误信息
SUBTAB  DCD     SUB0                    ; 子程序表入口
        DCD     SUB1
        DCD     SUB2
        ...
```

DCD 指示汇编器保留一个存储字，将它初始化为右边表达式的值。在这种情况下只是标号的地址。

不管表中有多少子程序，以及它们使用的频繁度如何互不相关，这种方法的性能不变。但是要注意，在读跳转表时，若超出了表的末端，则结果很可能是可怕的，因此，检查越限是必须的！在此注意，越限检查是通过有条件地向 PC 置数来实现的。因此，越限时 Load 指令被跳过，并转移到错误处理。越限检查惟一的性能代价是同最大值比较。更直接的代码可以是：

```
        CMP     r0, #SUBMAX             ; 检查越限…
        BHI     ERROR                   ; …如果越限，则调用错误处理
        LDR     pc, [r1, r0, LSL #2]    ; …否则跳转到表中
        ...
```

但是在此要注意，每次使用跳转表都要承受有条件地跳过转移的代价。原版本更为有效，除非检测到越限的时候。但越限应该是不会频繁发生的，而且因为越限代表错误，在这种情况下性能则不是最关心的。

另一个可选用的、不太直接地实现跳转表的方法将在 6.6 节的“开关”部分讨论。

3.4 编写简单的汇编语言程序

现在我们有了写简单汇编语言程序的所有基本工具。面对任何一个编程任务，重要的是在把指令键入计算机之前对于你的算法有一个清楚的思路。几乎肯定的是，较大的程序用 C 或 C++编写比较好，因此，我们仅考虑小的汇编语言程序的例子。

即便是最有经验的程序员，开始时也要检查是否能够使一个很简单的程序运行起来，然后再去完成他们的实际任务。有很多复杂的事情要做，例如，学习使用文本编辑器，学会怎样使编译器运行，怎样将程序装入机器，怎样使它开始运行等等。这类简单的测试程序常常归类于 Hello World 程序，因为它所做的就是在程序结束之前，在显

示器上打印 Hello World。下面是 ARM 汇编语言的一个版本：

```
                AREA    HelloW, CODE, READONLY      ; 声明代码区
SWI_WriteC              EQU  &0                     ; 输出 r0 中的字符
SWI_Exit                EQU  &11                    ; 程序结束
                ENTRY                               ; 代码的入口
START           ADR     r1, TEXT                    ; r1→"Hello World"
LOOP            LDRB    r0, [r1], #1                ; 读取下一字节
                CMP     r0, #0                      ; 检查文本终点
                SWINE   SWI_WriteC                  ; 若非终点,则打印
                BNE     LOOP                        ; …并返回 LOOP
                SWI     SWI_Exit                    ; 执行结束
TEXT            =       "Hello World", &0a, &0d, 0
                END                                 ; 程序源代码结束
```

这段程序说明了 ARM 汇编语言和指令集的许多特征：

- 带有适当属性的代码 AREA 的声明。
- 系统调用的定义。这些系统调用将在程序中使用(在大的程序中，这些调用将在一个文件中定义，由其他代码文件引用)。
- 使用 ADR 伪指令，以将地址写入基址寄存器。
- 使用自动变址寻址，以扫描一系列字节。
- SWI 指令的条件执行，以避免额外的转移。

还要注意使用字节"0"标记字符串(跟在换行和回车特殊字符之后)的结束。只要采用循环结构，就必须确保它有结束的条件。

为了运行这一程序，将需要下列工具。它们都可以在 ARM 软件开发工具套件中得到，即

- 键入程序的文本编辑器。
- 将程序转变为 ARM 二进制代码的汇编器。
- 执行二进制代码的 ARM 系统或仿真器。ARM 系统必须有某种文本输出能力(例如，ARM 开发板将文本输出送回到主机，以便在主机的显示器上输出)。

一旦使这个程序运行了，就可以试一些更有用的事情。从现在起，惟一变化的是程序文本。编辑器、汇编器、测试系统或仿真器的使用与你已经做过的非常相似。而当到你的程序拒绝做你希望做的，而你又不明白它为什么拒绝时，你将需要使用调试器来搞清在程序中发生了什么。这就意味着学习怎样使用另一个复杂的工具。因此，我们将尽可能地推迟这个时刻。

在下一个例子中将完成块拷贝程序。这个程序已经在前面部分地开发过了。为了确保我们知道它已经正确地工作了，将用一个文本的源字符串，这样就能从目标地址输出它。我们还将把目标区初始化，使之与源区不相同。

```
                AREA    BlkCpy, CODE, READONLY
SWI_WriteC              EQU  &0                     ; 输出 r0 中的字符
SWI_Exit                EQU  &11                    ; 程序结束
```

```
                ENTRY                                   ; 代码的入口
                ADR     r1, TABLE1                      ; r1→TABLE1
                ADR     r2, TABLE2                      ; r2→TABLE2
                ADR     r3, T1END                       ; r3→T1END
LOOP1           LDR     r0, [r1], #4                    ; 读取 TABLE1 第一字
                STR     r0, [r2], #4                    ; 拷贝到 TABLE2
                CMP     r1, r3                          ; 结束?
                BLT     LOOP1                           ; 若非,则再拷贝
                ADR     r1, TABLE2                      ; r1→TABLE2
LOOP2           LDRB    r0, [r1], #1                    ; 读取下一个字
                CMP     r0, #0                          ; 检查文本终点
                SWINE   SWI_WriteC                      ; 若非终点,则打印
                BNE     LOOP2                           ; …并返回 LOOP
                SWI     SWI_Exit                        ; 结束
TABLE1          =       "This is the right string!", &0a, &0d, 0
T1END
                ALIGN                                   ; 保证字对准
TABLE2          =       "This is the wrong string!", 0
                END                                     ; 程序源代码结束
```

例题

这个程序使用字的 Load 和 Store 来拷贝表,这就是为什么表必须是字对准的。然后使用与在 Hello World 程序中使用过的相同的程序,采用字节 Load 把结果打印出来。

注意使用 BLT 来控制循环结束。如果 TABLE1 包含字节的数目且不是 4 的倍数,那么就有个危险,即 r1 将越过 T1END 而不是正好等于它。这样,基于 BNE 的结束条件就会失效。

如果你已经成功地使程序运行了,那么你在理解 ARM 指令集的基本操作上已经开始起步。你应该研究后面的例题与练习,以便加强理解。当尝试更加复杂的编程任务时,还会产生细节问题。这些将在第 5 章给出的全指令集描述中解答。

程序设计

对指令集有了基本的理解后,就可以比较顺利地编写和调试小的程序了。你只需把程序键入编辑器再看它是否工作即可。但是,如果以为这种简单的方法能用于成功地开发复杂的程序,可以指望这些程序会工作很多年,将来会由其他程序员改写,而且会传送到以意想不到的方式使用它的用户手中,那么,这种设想是很危险的。

本书不是程序设计的教程。但在提供了编程入门之后,如果不指出编写一个有用的程序与仅仅坐下来并编写代码相比,还有更多的事情要做,那将是一个很严重的疏忽。

认真地编程不应由写代码开始,而应由仔细地设计开始。开发过程的第一步是把要求搞清楚。令人惊讶的是,经常因为程序员没有很好地搞清要求,而使程序不能像预期的那样工作!然后应把要求(常常是不正规的)转化为含义明确的说明书。现在可以

开始设计了，定义程序结构、程序工作时使用的数据结构以及用来完成所需数据操作的算法。算法可以用**伪码**表达。所谓伪码是一种类程序的符号，它不遵循特定编程语言的语法，但可使程序的意思清晰。

仅当设计完成后，才可开始写代码。应该分别编写各个模块的代码，彻底地测试（这可能需要设计专门的程序作为“测试装置”）并写文档，再把程序一块一块地拼起来。

今天，差不多所有编程都是基于高级语言的，所以，大程序很少采用这里介绍的汇编语言编程。然而，有时可能需要用汇编语言开发小的软件组件，以达到关键应用所需的最佳性能。因此，了解如何编写汇编代码是很有用处的。

3.5　例题与练习

一旦读者对指令集有了基本的了解，学习编程最容易的方法就是阅读一些例子，然后尝试编写自己的程序，做一些稍有变化的工作。为了验证程序是否工作，需要 ARM 汇编器，以及 ARM 仿真器或者包含 ARM 处理器的硬件。本节包括 ARM 程序的例子和修改的建议。应该首先使原始程序正常工作，然后再看是否能编辑它，以完成练习中建议修改的功能。

例题　3.1

以 16 进制打印输出 r1。

这是一个很有用的小程序。它把一个寄存器的内容以 16 进制符号在显示器上打印出来。可以用它来帮助调试程序。做法是把寄存器的值打印出来，并与算法产生的预期结果进行核对。虽然在多数情况下使用调试程序来了解程序里面在做什么是更好的方法。

```
                AREA    Hex_Out, CODE, READONLY
SWI_WriteC      EQU     &0          ; 输出 r0 中的字符
SWI_Exit        EQU     &11         ; 程序结束
                ENTRY               ; 代码的入口
                LDR     r1, VALUE   ; 读取要打印的数据
                BL      HexOut      ; 调用 16 进制输出
                SWI     SWI_Exit    ; 结束
VALUE           DCD     &12345678   ; 测试数据
HexOut          MOV     r2, #8      ; 半字节数=8
LOOP            MOV     r0, r1, LSR #28 ; 读取顶半字节
                CMP     r0, #9      ; 0~9 还是 A~F?
                ADDGT   r0, r0, #"A"-10 ; ASCII 字母
                ADDLE   r0, r0, #"0"    ; ASCII 数字
                SWI     SWI_WriteC  ; 打印字符
                MOV     r1, r1, LSL #4  ; 左移 4 位
                SUBS    r2, r2, #1  ; 半字节数减 1
```

```
        BNE     LOOP                    ; 若还有,则进行下一个半字节
        MOV     pc, r14                 ; 返回
        END
```

练习 3.1.1

修改上面的程序,以 2 进制格式输出 r1。对于上例中读入 r1 的数值,应得到:

00010010001101000101011001111000

练习 3.1.2

以 HEXOUT 程序为基础显示存储器一个区域的内容。

例题 3.2

编写一段子程序,以输出紧接在调用指令后的文本字符串。

如果能输出一个文本字符串而不必为文本设置单独的数据区,则会非常有用(尽管若处理器像 StrongARM 那样有分开的数据和指令 Cache,这种方法则是低效的。在此情况下,最好设置一个单独的数据区)。一个调用应该如下:

```
        BL      TextOut
        =       "Test string", &0a, &0d, 0
        ALIGN
        ...                             ; 返回到此
```

这里的问题是,从子程序返回时不能直接回到由调用放入到链接寄存器中的地址,因为这会使程序在读下一个字符串时搁浅。下面是一个适当的子程序和测试程序:

```
                AREA    TestOut, CODE, READONLY
SWI_WriteC      EQU     &0              ; 输出 r0 中的字符
SWI_Exit        EQU     &11             ; 程序结束
                ENTRY                   ; 代码的入口
                BL      TextOut         ; 打印一下字符串
                =       "Test string", &0a, &0d, 0
                ALIGN
                SWI     SWI_Exit        ; 结束
TextOut         LDRB    r0, [r14], #1   ; 读取下一个字符
                CMP     r0, #0          ; 检测结束符
                SWINE   SWI_WriteC      ; 若未结束,则打印…
                BNE     TextOut         ; …并循环
                ADD     r14, r14, #3    ; 跨过下一个字的分界
                BIC     r14, r14, #3    ; 退回到字的分界
                MOV     pc, r14         ; 返回
                END
```

从这个例子可看出,r14 沿着字符串递增,在返回前被调整到下一个字的分界。如果这个调整(加 3,再将低两位清 0)看起来小了点,再请查对一下。只有 4 种情形。

练习　3.2.1

使用本例和上例的代码编写一个程序，按下列格式的 16 进制输出 ARM 寄存器。

r0＝12345678

r1＝9ABCDEF0

练习　3.2.2

试将你需要的寄存器在使用之前保存起来。例如，使用 PC 相对寻址方式将它们保存在代码附近。

第 4 章　ARM 的组织和实现

本章内容综述

从 Acorn Computer 公司在 1983—1985 年间开发的第一个 3 μm 器件，到 ARM 公司在 1990—1995 年间开发的 ARM6 和 ARM7，ARM 整数处理器核的组织结构变化很小。这些处理器都使用的是 3 级流水线，而这一时期 CMOS 工艺技术几乎将特征尺寸减小了一个数量级。因此，核的性能提高很快，但基本的操作原理大部分没有变化。

从 1995 年以来，ARM 公司推出了几个新的 ARM 核。它们采用 5 级流水线和分开的程序和数据存储器（通常采用的形式是将分开的 Cache 连接到指令和数据共享的主存储器系统），获得了显著的高性能。

这一章包括对这两种基本类型处理器核内部结构的描述，覆盖了 3 级和 5 级流水线的一般操作原理和一些实现细节。特殊核的细节将在第 9 章介绍。

4.1　3 级流水线 ARM 的组织

3 级流水线 ARM 的组织如图 4.1 所示，其主要的组成如下：

- 保存处理器状态的寄存器堆（register bank）。它有两个读端口和一个写端口，每个端口都可以访问任意寄存器，再加上专门访问程序计数器 r15 的一个附加读端口和一个附加写端口（r15 的附加写端口可以在取指地址增加后更新 r15，读端口可以在数据地址发出之后重新开始取指）。
- 桶式移位器。它可以把一个操作数移位或循环移位任意位数。
- ALU。完成指令集要求的算术或逻辑功能。
- 地址寄存器和增值器。它选择和保存所有的存储器地址，并在需要时产生顺序地址。
- 数据寄存器。保存传送到存储器或从存储器取出的数据。
- 指令译码器和相关的控制逻辑。

在单周期数据处理指令中，需访问两个寄存器操作数，B 总线上的数据移位后与 A 总线上的数据在 ALU 中组合，再将结果写回寄存器堆。程序计数器的数据放在地址寄存器中，地址寄存器的数据送入增值器。然后将增加后的数据拷贝到寄存器堆的 r15，同时还拷贝到地址寄存器，作为下一次取指的地址。

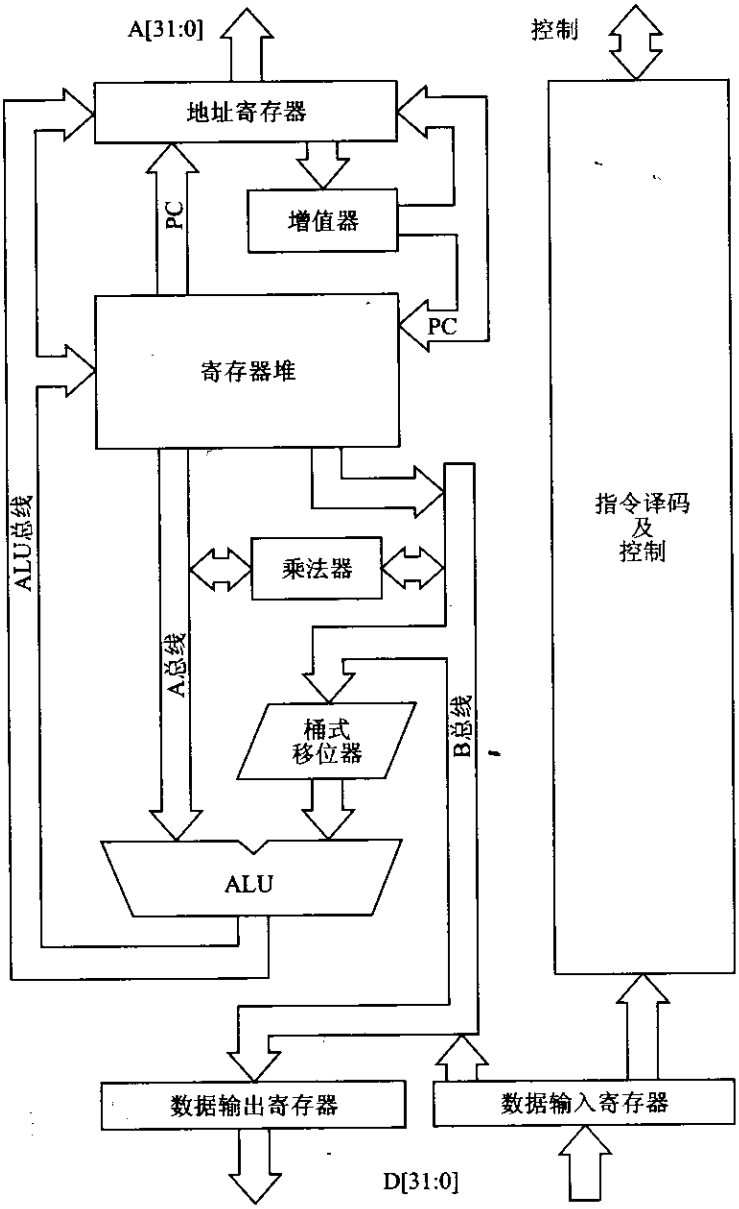


图 4.1　3 级流水线 ARM 的组织

3 级流水线

到 ARM7 为止的 ARM 处理器使用简单的 3 级流水线，包括下列流水线级：

1. 取　指

从存储器中取出指令，放入指令流水线。

2. 译　码

指令被译码，并为下一周期准备数据通路的控制信号。在这一级，指令占有译码逻辑，不占有数据通路。

3. 执　行

指令占有数据通路，寄存器堆被读取，操作数被移位，ALU 产生结果并回写到目的寄存器。

在任意时刻，可能有 3 种不同的指令占用这 3 级中的每一级，因此，每一级中的硬件必须能够独立操作。

当处理器执行简单的数据处理指令时，流水线使得每个时钟周期能完成 1 条指令。1 条指令用 3 个时钟周期来完成，因此，有 3 周期的**执行时间**(latency)，但**吞吐率**(throughput)是每个周期 1 条指令。单周期指令的 3 级流水线操作如图 4.2 所示。

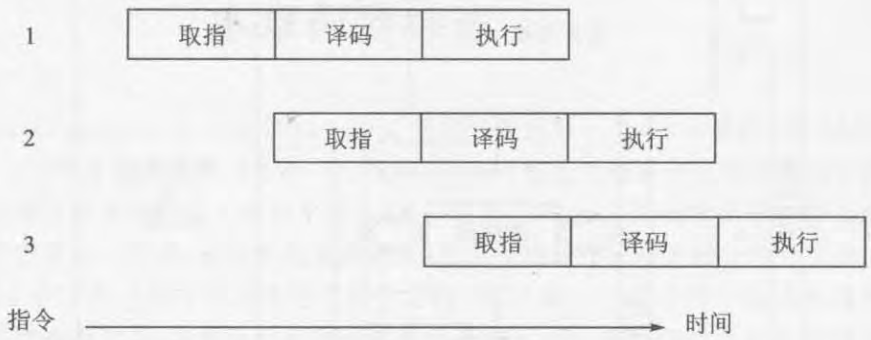


图 4.2　ARM 单周期指令的 3 级流水线操作

当执行多周期指令时，流程不太规则，如图 4.3 所示。图中表示了一组单周期指令 ADD，而在第一个 ADD 指令的后面出现一个数据存储指令 STR。访问主存储器的周期用浅阴影表示，因此，可以看到在每一个周期中都使用了存储器。同样，在每一个周期也使用了数据通路，这涉及到所有的执行周期、地址计算和数据传送。译码逻辑总是产生数据通路在下一周期使用的控制信号。因此，除译码周期外，在 STR 地址计算周期中也产生数据传送所需的控制信号。

图 4.3　ARM 多周期指令的 3 级流水线操作

这样，在这个指令序列中，处理器的所有部件在每个周期中都是活动的。而存储器是一个限制因素，它规定程序必须花费的周期数。

查看 ARM 流水线间断的最简单方式是观察：

- 所有指令都占用数据通路一个或多个相邻周期。
- 在指令占用数据通路的每一个周期，都在前面的相邻周期占有译码逻辑。
- 在第一个数据通路周期，每条指令为下下条指令发出取指信号。
- 转移指令清空(flush)和重填指令流水线。

PC 的行为

ARM 使用的流水线执行模式导致一个结果，就是程序计数器 PC(对使用者是可见的 r15)必须在当前指令之前计数。如前所述，如果指令在其第一个周期为下下条指令取指，这就意味着 PC 必须指向当前指令之后的 8 个字节(两条指令)。

事实确实是这样的，试图通过 r15 直接访问 PC 的程序员必须考虑到此时流水线的真实情况。然而对于大多数的正常应用，汇编器或编译器会处理所有的细节。

如果在指令的第一个周期之后再使用 r15，就会出现更复杂的行为，因为指令本身将在它的第一周期使 PC 增加。这样使用 PC 并不总是有益的。因此，ARM 在体系结构的定义中规定结果为“不可预测”，应该避免，特别是因为后期的 ARM 在这些情况下的行为并不相同。

4.2　5 级流水线 ARM 的组织

所有的处理器都要满足对高性能的要求。直到 ARM7 为止，在 ARM 核中使用的 3 级流水线的性价比是很高的。但是，为了得到更高的性能，需要重新考虑处理器的组织结构。执行一个给定程序需要的时间 T_{prog} 由下式确定：

$$T_{prog}=\frac{N_{inst}\times CPI}{f_{clk}} \tag{11}$$

式中：N_{inst}——在程序中执行的 ARM 指令数；

CPI——每条指令的平均时钟周期数；

f_{clk}——处理器的时钟频率。

因为 N_{inst} 对给定程序(使用给定的优化集并用给定的编译器来编译等)是常数，所以，仅有两种方法来提高性能，即

1. 提高时钟频率 f_{clk}

这就要求简化流水线每一级的逻辑，因而流水线的级数就要增加。

2. 减少每条指令的平均时钟周期数 CPI

这就要求重新考虑 3 级流水线 ARM 中多于 1 个流水线槽的指令的实现方法，以便使它占有较少的槽；或者减少因指令相关造成的流水线停顿，也可以将这两者结合起来。

存储器瓶颈

与 3 级核有关的减少 CPI 的根本问题与冯·诺曼(von Neumann)瓶颈有关——指令和数据放在同一个存储器的任何存储程序计算机，其性能将受到现有存储器带宽的

限制。3 级流水线 ARM 核(几乎)在每一个时钟周期都访问存储器,或者取指令,或者传输数据。只是抓紧存储器不用的几个时钟周期,只能使性能增加很少。为了明显改善 CPI,存储器系统必须在每个时钟周期中给出多于 1 个的数据。方法可以是在每个时钟周期从单个存储器中给出多于 32 位数据,或为指令和数据分别设置存储器。

作为上述讨论的结果,较高性能的 ARM 核使用 5 级流水线,且具有分开的指令和数据存储器。把指令的执行分割为 5 部分而不是 3 部分,这就减少了在每个时钟周期内必须完成的最大工作,进而允许使用较高的时钟频率(倘若其他的系统部件,特别是指令存储器,也重新设计以较高的时钟频率操作)。分开的指令和数据存储器(可能是分开的 Cache 连接到统一的指令和数据主存储器上)使核的 CPI 明显减少。

在 ARM9TDMI 中使用了典型的 5 级 ARM 流水线。ARM9TDMI 的组织结构如图 4.4 所示。

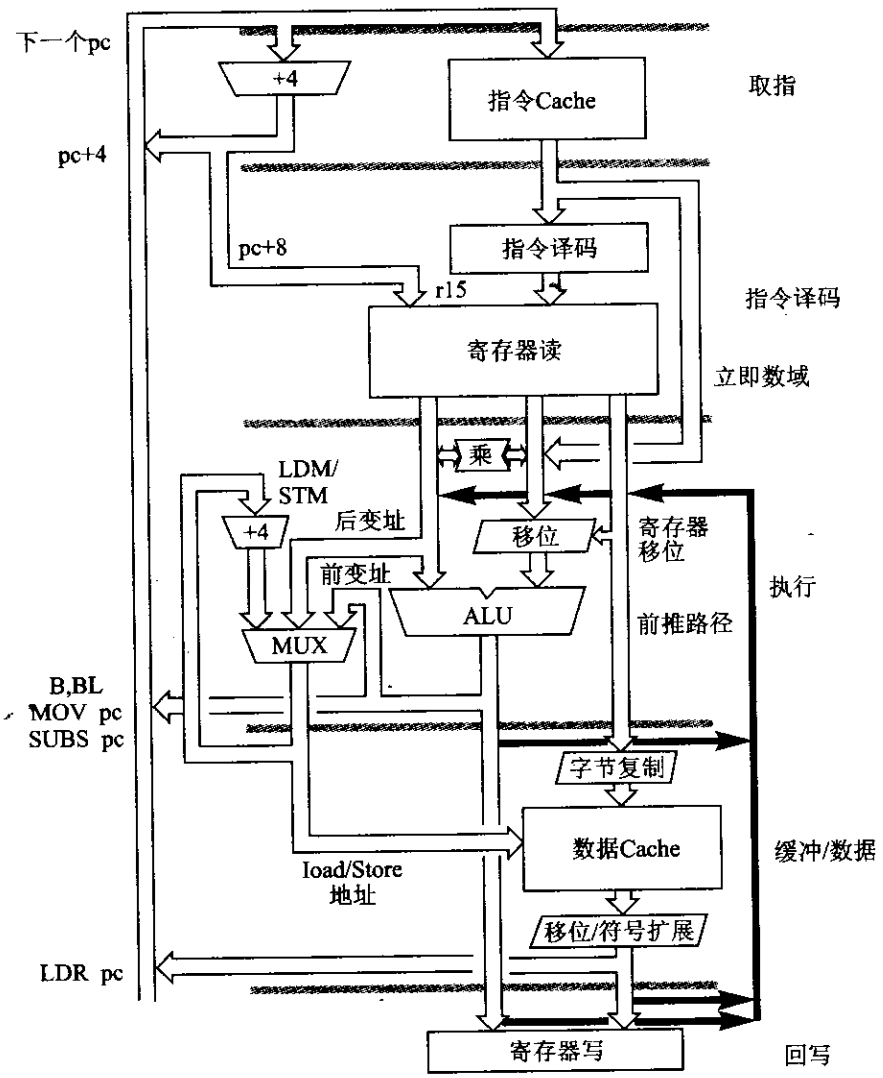


图 4.4 ARM9TDMI 5 级流水线的组织结构

5 级流水线

使用 5 级流水线的 ARM 处理器具有下面流水线级：

1. 取　指

从存储器中取出指令，并将其放入指令流水线。

2. 译　码

指令被译码，从寄存器堆中读取寄存器操作数。在寄存器堆中有 3 个操作数读端口，因此，大多数 ARM 指令能在 1 个周期内读取其操作数。

3. 执　行

把一个操作数移位，产生 ALU 的结果。如果指令是 Load 或 Store，则在 ALU 中计算存储器的地址。

4. 缓冲/数据

如果需要，则访问数据存储器；否则 ALU 的结果只是简单地缓冲 1 个时钟周期，以便使所有的指令具有同样的流水线流程。

5. 回　写

将指令产生的结果回写(write-back)到寄存器堆，包括任何从存储器中读取的数据。

这种 5 级流水线在许多 RISC 处理器中使用过，而且被认为是设计处理器的经典方法。尽管 ARM 指令集设计时并不是针对这样的流水线，但是，将它映射过来相对还是简单的。在组织结构(示于图 4.4)上对 ARM 指令集体系结构的主要妥协是寄存器堆有 3 个源操作数读端口和 2 个写端口(经典的 RISC 有 2 个读端口和 1 个写端口)以及在执行级包含了地址增值硬件，以支持多寄存器 Load 和 Store 指令。

数据前推

因为 5 级流水线的指令执行分布在 3 个流水级中，所以，解决数据相关而不使流水线停顿的惟一方法是引入**前推**(forwarding)通路，这是 5 级流水线(与 3 级流水线相比)复杂性的主要根源。

当指令需要使用前一条指令的结果，而这个结果还未回写到寄存器堆时，便产生数据相关(这些问题在前面 1.5 节的“流水线冒险”中讨论过)。前推通路使结果产生后可以立即在级间传送。5 级 ARM 流水线要求 3 个源操作数都能从任何 3 个中间结果寄存器中前推，如图 4.4 所示。

有一种情况，即使使用前推，也不可能避免流水线的停顿。考虑下列代码的序列：

```
LDR     rN, [···]          ; 从某处调入 rN
ADD     r2, r1, rN         ; 立即使用它
```

因为调入到 rN 的值在缓冲/数据级结束时刚刚进入处理器，而后面的指令在执行级的开始就需要它，所以，处理器无法避免 1 个周期的停顿。避免这种停顿的惟一方法是鼓励编译器(或汇编语言程序员)不要紧跟在 Load 指令后面放置一个相关的指令。

因为 3 级流水线 ARM 核不受这种代码序列的影响，已有的 ARM 程序通常会使用这种序列。这样的程序在 5 级 ARM 核上也能正确运行。但如果重新改写这个程

序，通过简单地重新排列指令来去除这些相关，程序会运行得更快。

PC 的产生

r15 的行为，正如程序员所看到的并在 4.1 节“PC 行为”一段中所描述的，是基于 3 级流水线的操作特性。5 级流水线在流水线中提前了 1 级来读取指令操作数，自然得到不同的值(PC+4 而不是 PC+8)。这产生的代码不兼容是无法接受的。但是 5 级流水线 ARM 全都仿真较早的 3 级流水线的行为。参考图 4.4，在取指级增加的 PC 值被直接送到译码级的寄存器文件(file)，穿过两级之间的流水线寄存器。下一条指令的 PC+4 等于当前指令的 PC+8，因此，未使用额外的硬件便得到了正确的 r15。

4.3 ARM 指令执行

参考图 4.1 给出的数据通路组织结构，可以很好地理解 ARM 指令的执行。我们将使用这个图的注释版本，省略控制逻辑部分，突出活跃的总线来表示操作数在处理器各个单元间的移动。下面从简单的数据处理指令开始。

数据处理指令

数据处理指令需要两个操作数，其中一个总是寄存器，另外一个是第二寄存器或立即数。第二操作数经过桶式移位器，在那里经过通常的移位操作后，在 ALU 中用一般的 ALU 操作同第一操作数结合。最后 ALU 的结果回写到目的寄存器(条件码寄存器可能被更新)。

所有的这些操作都发生在同一时钟周期，如图 4.5 所示。注意地址寄存器中的 PC 值是如何增值并拷贝到地址寄存器和寄存器堆的 r15，下下条指令是如何装入到指令流水线(i. pipe)底部的。在需要时，从指令流水线顶部的当前指令提取立即数。对于数据处理指令，只有指令的低 8 位(位[7：0])用做立即数。

数据传送指令

数据传送(Load 或 Store)指令计算存储器地址的方式与数据处理指令计算其结果的方式非常相似。一个寄存器用做基址，它的值加上(或者减去)偏移量。偏移量可能是另外的寄存器或立即数。但是，这次使用的是不移位的 12 位立即数，而不是移位的 8 位值。地址送到地址寄存器，并在第二周期进行数据传送。在数据传送周期没有在数据通路留下大量的空闲时间。ALU 保持从第一个周期得到的地址分量，如果需要，则可以用来计算自动变址对基址寄存器的修改。(如果不需要自动变址，则计算值在第二周期不被回写到基址寄存器。)

带有一个立即数偏移的数据 Store 指令(STR)在这两个周期中的数据通路操作如图 4.6 所示。注意，增加后的 PC 值如何在第一周期末存入寄存器堆，以使地址寄存器在第二周期能空下来接受数据传送地址，然后在第二周期末，再将 PC 送回地址寄存器，使预取指得以继续。

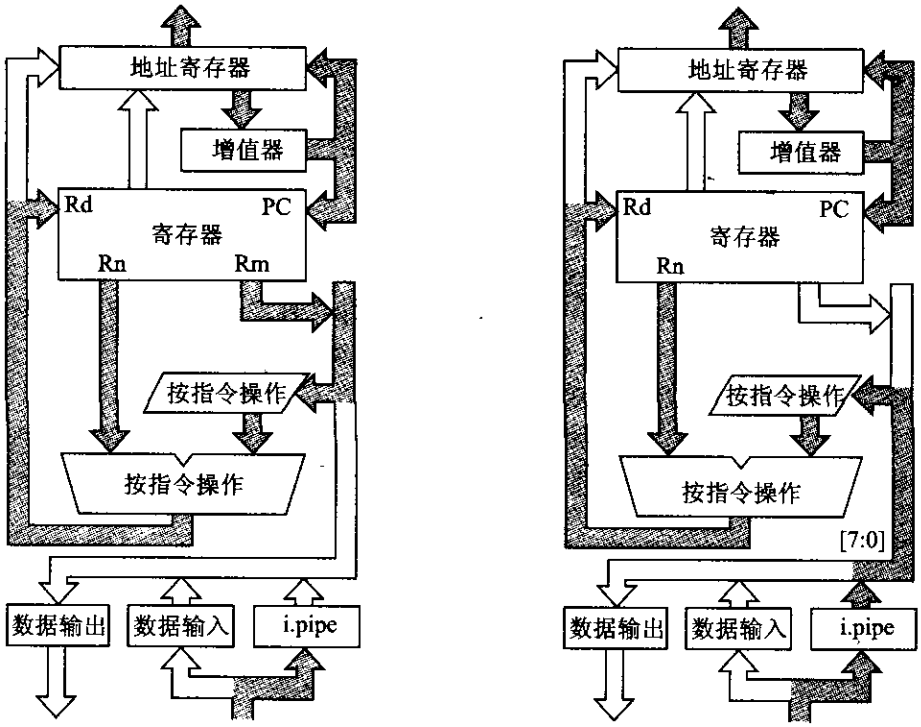


图 4.5　数据处理指令的数据通路行为

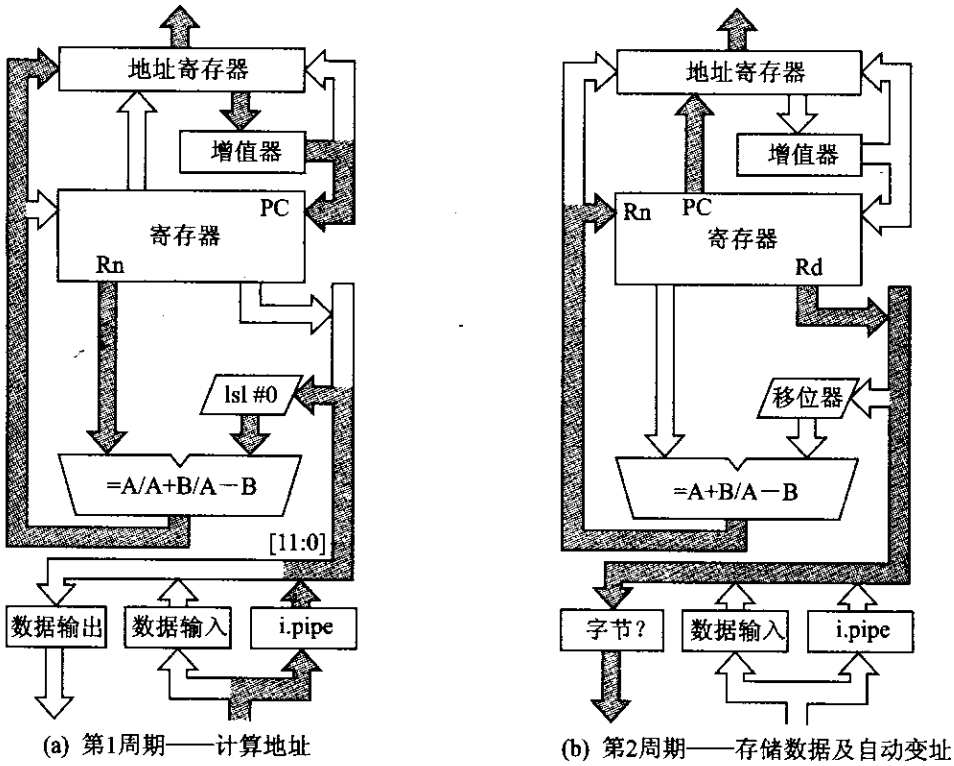


图 4.6　STR(存储寄存器)指令的数据通路行为

或许在这一级应该注意，在一个时钟周期中送到地址寄存器的值是用做下一个时钟周期存储器访问的值，地址寄存器在效果上是处理器的数据通路与外部存储器之间的流水线寄存器。

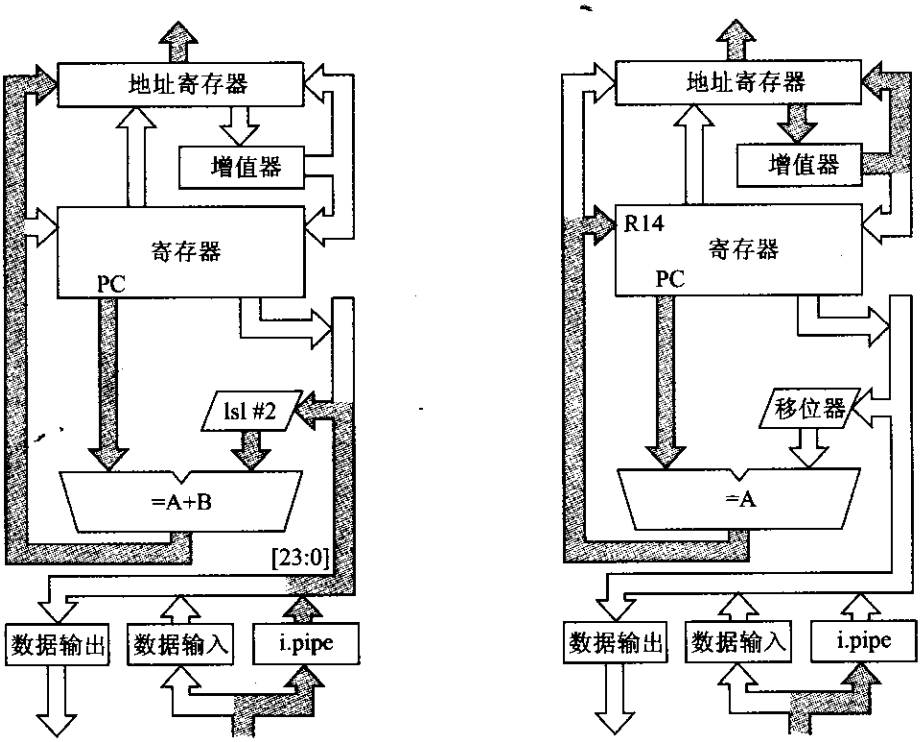
（地址寄存器能在临近当前周期结束前，产生下一个时钟周期的存储器地址。如果需要，则可以把流水线延迟的责任转移给存储器，这可以使一些存储器件以较高的性能工作。这里我们把这一细节留到以后讨论。现在我们将把地址寄存器看作是通向存储器的流水线寄存器。）

当指令指定存储的数据为字节类型时，数据输出模块便从寄存器提取最低字节，并在 32 位数据总线上把它重复 4 次。这时外部存储器控制逻辑使用地址总线的低两位来激活在存储器系统中适当的字节。

Load 指令也使用同样的模式，不同的只是来自存储器的数据在第 2 周期仅到达数据输入寄存器，还需要第 3 周期将数据从这里传送到目的寄存器。

转移指令

转移指令在第一周期计算目标地址，如图 4.7 所示。从指令中提取一个 24 位立即数，并左移两位产生字对准的偏移，再与 PC 相加。结果作为取指的地址发送出去。在重新填充指令流水线时，如果需要（这指的是，如果指令为转移链接），则将返回地址拷贝到链接寄存器（r14）。



(a) 第1周期——计算转移目标　　(b) 第2周期——保存返回地址

图 4.7　转移指令的 3 个周期中的前两个周期

要完成流水线再填充则还需要第 3 周期。这个周期也用来对存储在链接寄存器中的值作小的修正，以便能直接指向跟在转移指令后面的指令。这是必须的，因为 r15 包含 PC+8，而下一条指令是 PC+4(参看 4.1 节“PC 行为”一段)。

其他 ARM 指令的操作方式同上面所述类似。下面将转向更详细地讨论数据通路如何完成这些操作。

4.4　ARM 的实现

ARM 的实现方式与第 1 章中对 MU0 的概述相似。将设计划分为数据通路部分和控制部分，前者用寄存器传输级(RTL，Register Transfer Level)描述，后者可看成是一个有限状态机(FSM，Finite State Machine)。

时钟方案

与 1.3 节中给出 MU0 的例子不同，多数 ARM 不用边沿敏感的寄存器来工作，而是根据如图 4.8 所示的两相非重叠时钟来设计。两相非重叠时钟是在内部由输入的同一个时钟信号产生的。这个方案允许使用电平敏感的透明锁存器。数据移动是通过使数据交替地通过锁存器来进行控制的。这些锁存器有些在第 1 相导通，有些在第 2 相导通。第 1 相和第 2 相时钟的非重叠特征确保在电路中没有竞争条件。

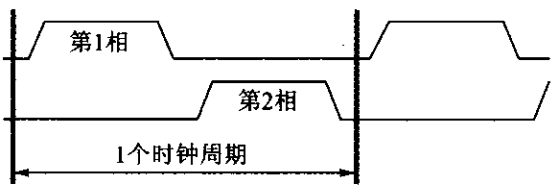


图 4.8　两相非重叠时钟方案

数据通路时序

3 级流水线数据通路的正常时序如图 4.9 所示。寄存器读总线是动态的，在第 2 相预充电(这里“动态”意指它们有时没有驱动，由电荷保持其逻辑值。使用充电-保持电路使它具有准静态的行为，即使时钟在周期内任意点停止，数据也不会丢失)。当第 1 相变高时，所选的寄存器就对读总线放电，在第 1 相的早期变为有效值。一个操作数通过桶式移位器(它也使用了动态技术)，移位器的输出在第 1 相稍晚变为有效值。

ALU 的输入锁存器在第 1 相导通，使操作数一旦有效，便可开始在 ALU 组合。但是它们在第 1 相末关断，使第 2 相预充电不会到达 ALU。这样 ALU 在第 2 相继续处理操作数，在相结束前产生有效的输出，并在第 2 相结束时锁存到目的寄存器。

但要注意数据如何通过 ALU 输入锁存。它们并不影响数据通路的时序，因为当有效的数据到达时它们导通。透明锁存器的这一特性在 ARM 设计中被用在许多地方，以确保时钟不减慢关键信号。

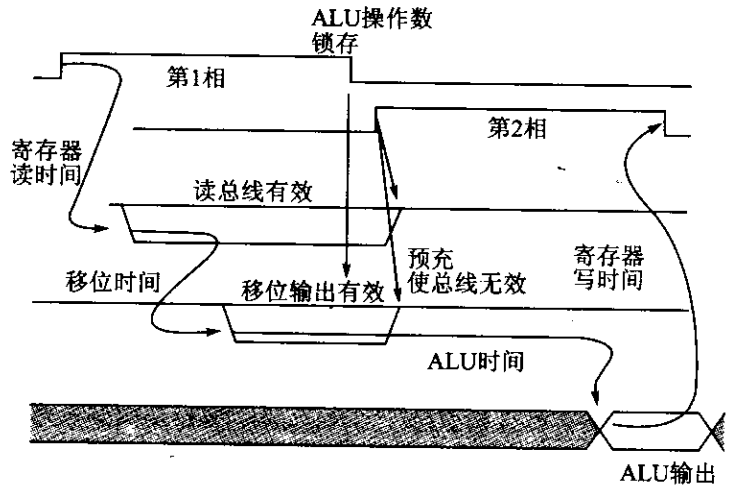


图 4.9 ARM 数据通路的时序(3 级流水线)

因此,最小数据通路周期时间是下列时间的总和,即

- 寄存器读时间;
- 移位延迟;
- ALU 延迟;
- 寄存器写建立时间;
- 第 1 相和第 2 相非重叠时间。

当然,ALU 延迟是主要因素。ALU 延迟的变化很大,依赖于它执行的操作。逻辑操作相对快,因为它不涉及进位传输。算术操作(加、减和比较)涉及较长的逻辑路径,因为进位的传输可能会穿越整个字宽。

加法器设计

因为 32 位加法时间明显影响数据通路周期时间,进而影响最大时钟速率和处理器性能,所以,在开发 ARM 系列处理器过程中,这一直是关注的焦点。

第一个 ARM 处理器原型使用的是如图 4.10 所示的简单行波进位(ripple-carry)加法器。使用 CMOS 的"与-或-非"门产生进位逻辑,再将"与"/"或"逻辑交换,以便偶数位使用图中的电路,奇数位使用对偶的电路。对偶电路的输入和输出都是原电路的反相,而且将"与"门同"或"门交换。最坏情况下的进位路径为 32 个门。

为了提高时钟频率,ARM2 使用了 4 位超前进位(carry look-ahead)方案以减少最坏情况下进位通路的长度,电路如图 4.11 所示。逻辑产生进位产生(G,Generate)和传递(P,Propagate)信号,它们控制 4 位进位输出。进位传递通路的长度减少到 8 个门延迟,也使用了归并的"与-或-非"门和交替的"与"/"或"逻辑。

ALU 的功能

ALU 不仅是将两个输入相加。它必须完成指令集定义的全部数据操作,包括存储器传送所需的地址计算、转移计算和位逻辑功能等。

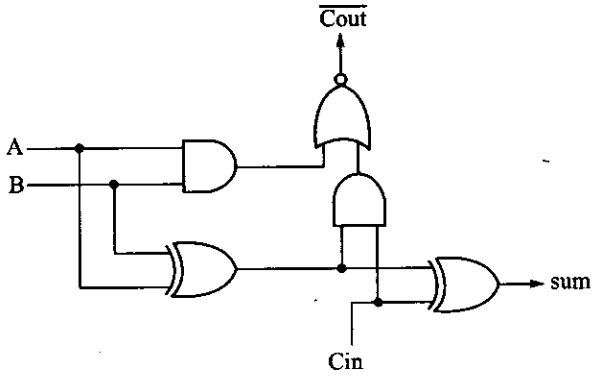


图 4.10　原用于 ARM1 的行波进位加法器电路

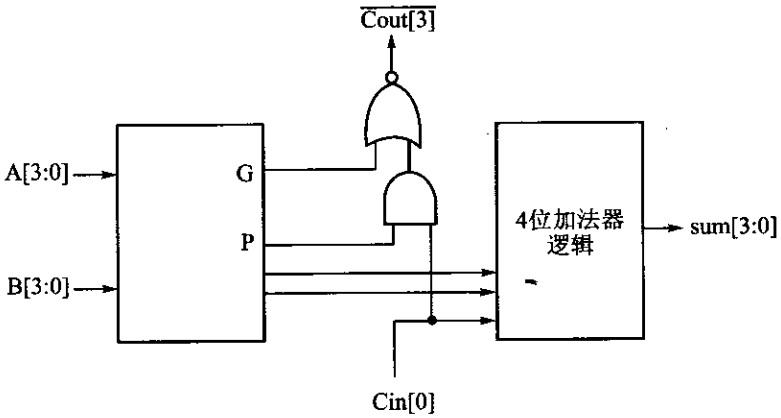


图 4.11　ARM2 的 4 位超前进位方案

ARM2 ALU 的全部逻辑如图 4.12 所示。这个 ALU 产生的全部功能和 ALU 功能选择的相关数值如表 4.1 所列。

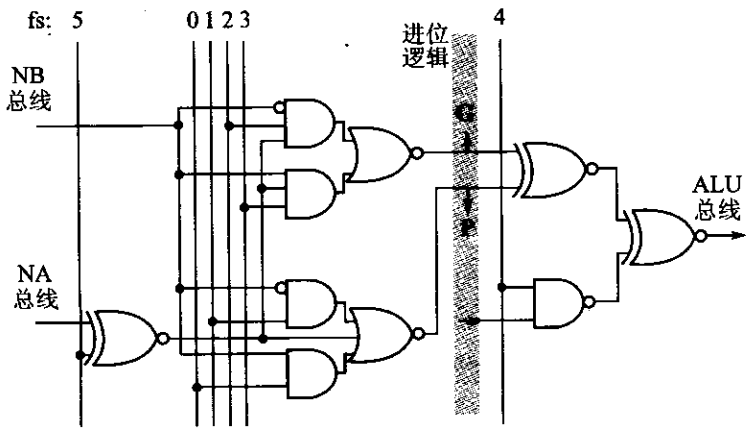


图 4.12　ARM2 的 ALU 中用于 1 位输出的逻辑

表 4.1 ARM2 ALU 的功能代码

fs5	fs4	fs3	fs2	fs1	fs0	ALU 输出
0	0	0	1	0	0	A and B
0	0	1	0	0	0	A and not B
0	0	1	0	0	1	A xor B
0	1	1	0	0	0	A + not B + carry
0	1	0	1	1	0	A + B + carry
1	1	0	1	1	0	not A + B + carry
0	0	0	0	0	0	A
0	0	0	0	0	1	A or B
0	0	0	1	0	1	B
0	0	1	0	1	0	not B
0	0	1	1	0	0	0

ARM6 的进位选择加法器

在 ARM6 中使用进位选择加法器使最坏情况下的加法时间得到进一步的提高。这种形式的加法器在计算各个字段的和时，同时计算进位输入为 0 及进位输入为 1 的两种情况。然后用正确的进位输入值来控制多路选择器，选择最终的结果。完整的电路方案如图 4.13 所示。

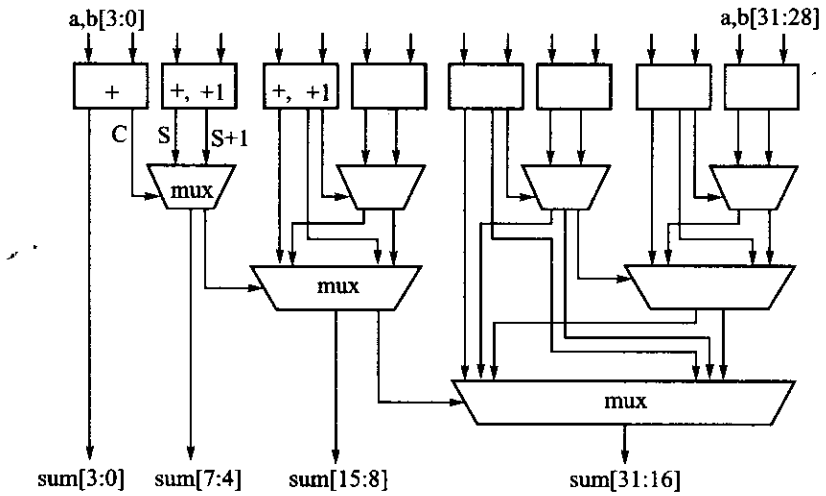


图 4.13 ARM6 的进位选择加法器方案

其关键路径是 O($\log_2$[字宽])个逻辑门的长度。尽管因为这些门中有些扇出(fan-out)较高，因而难于同先前的方案直接比较，但是，最坏情况的加法时间比 4 位超前进位加法器明显加快，代价是芯片面积也明显增加了。

ARM6 的 ALU 结构

ARM6 进位选择加法器使 ALU 的算术和逻辑功能不容易合并为像 ARM2 所用的那种单一结构。相反地，分离的逻辑单元和加法器并行运行，根据需要，用多路选择器从加法器或逻辑单元中选择输出。

完整的 ALU 结构如图 4.14 所示。每一个操作数可以选择是否取反，然后相加或在逻辑单元中组合。最终选出所需的结果，并发送到 ALU 的结果总线。

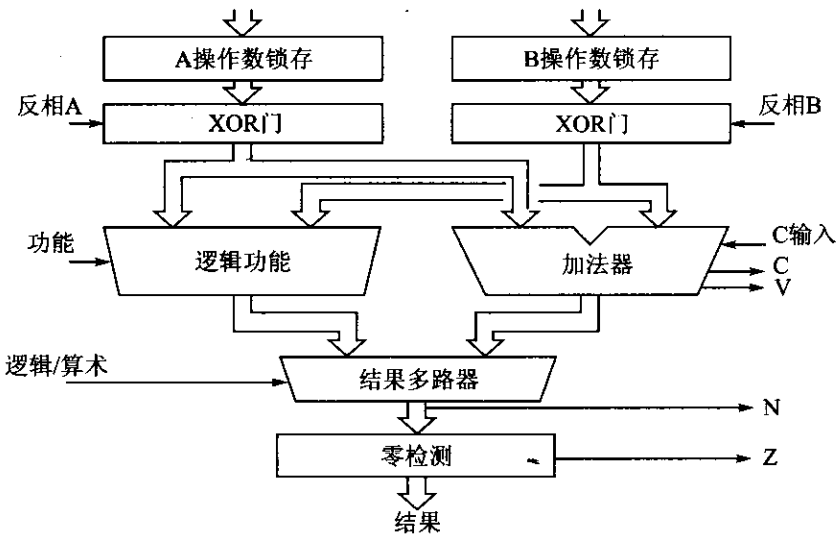


图 4.14　ARM6 中 ALU 的组织结构

标志位 C 和 V 在加法器(它们对逻辑操作是无意义的)中产生，标志位 N 来自结果的第 31 位的拷贝，标志位 Z 则从整个结果总线求值。注意产生标志位 Z 需要 32 输入端的“或非”门，这会很容易成为关键路径的信号。

进位判决加法器

加法器的逻辑在 ARM9TDMI 中得到进一步的改进。在那里使用了进位判决(carry arbitration)加法器。这种加法器使用一种非常快速的并行逻辑结构“并行-前置(parallel-prefix)”树来计算所有的中间进位。

进位判决方案使用两个新的变量 u 和 v 来记录传统的进位传递和进位产生信息。加法器的输入为 A 和 B，考虑一个特定位的进位输出 C 的计算算法。在进位输入确定之前，可以得到的信息如表 4.2 所列。该表还给出这些信息如何由 u 和 v 编码。这些信息可以用以下公式与来自相邻位的信息进行组合，即

$$(u,v)\bullet(u',v')=(v+u\bullet u',v+u\bullet v') \tag{12}$$

可以看出这个组合操作是相关的，因而可以使用规则的并行前置树来计算求和中所有位的 u 和 v。实现式(12)所需的的逻辑可以在单个 CMOS 门上将 4 对输入组合在一起，从单个晶体管结构中产生新的 u 和 v 输出。

表 4.2 ARM9 进位判决编码

A	B	C	u	v
0	0	0	0	0
0	1	未知	1	0
1	0	未知	1	0
1	1	1	1	1

此外，还可以看出，如果进位输入为 1，则 u 给出进位输出；如果进位输入为 0，则 v 给出进位输出。因此，u 和 v 可以用来产生混合的进位判决/进位选择加法器所需的(Sum，Sum+1)，产生一系列可以在性能、面积和功耗之间折衷考虑的、可能的设计。

桶式移位器

ARM 体系结构支持与 ALU 操作串行完成移位操作的指令，其组织结构如图 4.1 所示。因此，移位器的性能是关键的，因为移位时间直接计入数据通路的周期时间，如图 4.9 的数据通路时序图所示。

(其他处理器的体系结构趋向于使 ALU 与移位器并行。这样，只要移位器不比 ALU 慢，它就不影响数据通路的周期时间。)

为了减少通过移位器的延迟，使用一个交叉开关矩阵把每一个输入引导到适当的输出。交叉开关的原理如图 4.15 所示。图中显示了一个 4×4 的矩阵(ARM 处理器使用 32×32 矩阵)。每一个输入都通过开关与一个输出相连。如果使用预充(pre-charged)动态逻辑，正像它在 ARM 数据通路中那样，每一个开关可以用单 NMOS 管来实现。

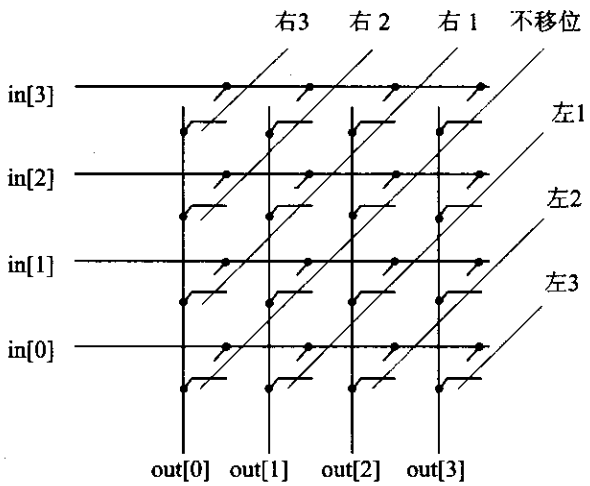


图 4.15 交叉开关桶式移位器原理

把沿着一条对角线的开关连接到共同的控制输入，即可实现移位功能，即

对于左移或右移功能，将一条对角线的开关导通。这将所有的输入位与所使用

的输出分别相连。(不是所有的都使用,因为一些位从末端移出。)在 ARM 中,桶式移位器的操作使用反逻辑,"1"代表接近地的电位,"0"代表接近电源的电位。预充电将所有的输出置为"0",因此,那些在特定的开关操作中没有同任何输入相连的输出保留为"0",得出了移位语义所需要的"0"填充。

- 对于循环右移功能,右移对角线同互补的左移对角线一起激活。例如,在 4 位矩阵中使用"右 1"和"左 3"(3=4−1)对角线来实现右循环 1 位。
- 对于未连接的输出位,算术右移使用符号扩展而不是 0 填充。使用另外的逻辑进行移位总量译码以及适当地为那些未连接的输出放电。

乘法器设计

与第一个原型机不同,所有的 ARM 处理器都实现了支持整数乘法的硬件。使用过两种类型的乘法器:

- 早期的 ARM 核使用低成本乘法器硬件,它只支持 32 位结果的乘法和乘-累加指令。
- 近期的 ARM 核具有高性能的乘法器硬件,支持 64 位结果的乘法和乘-累加指令。

低成本方式重复使用主要的数据通路,使用桶式移位器和 ALU 在每一个时钟周期产生 2 位乘积。当乘法寄存器不再有数据时,早终止(early-ermination)逻辑停止迭代。

乘法器使用改进的 Booth 算法,利用"×3"可以由"×(−1)+×4"这样的事实来产生 2 位积。这就使 2 位乘法器的全部 4 个值都能通过简单的移位以及加或减来实现,有可能将"×4"进位到下一个周期。

乘法运算第 N 个周期的控制设置情况如表 4.3 所列。(注意"×2"的情况也采用减法和进位来实现。同样也可以采用加法和不进位,但采用这种方法控制逻辑稍简单些。)

表 4.3　两位乘法算法,第 N 个时钟周期

进位输入	乘　数	移　位	ALU	进位输出
0	×0	LSL ＃2N	A+0	0
	×1	LSL ＝2N	A+B	0
	×2	LSL ＃(2N+1)	A−B	1
	×3	LSL ＃2N	A−B	1
1	×0	LSL ＝2N	A+B	0
	×1	LSL ＃(2N+1)	A+B	0
	×2	LSL ＃2N	A−B	1
	×3	LSL ＝2N	A+0	1

因为这种乘法使用已有的移位器和 ALU,所以,它所需要的额外硬件只限于乘法器专用的、每周期两位的移位寄存器,以及用于 Booth 算法控制逻辑的少许门。总起来

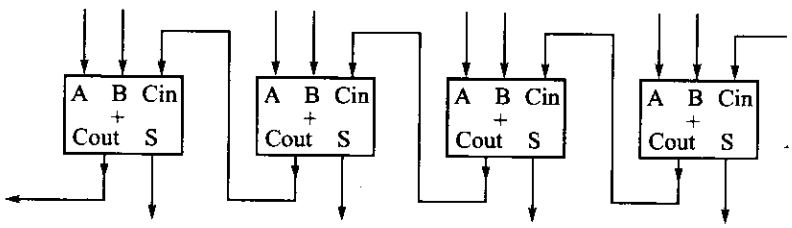
这等于 ARM 核面积百分之几的开销。

高速乘法器

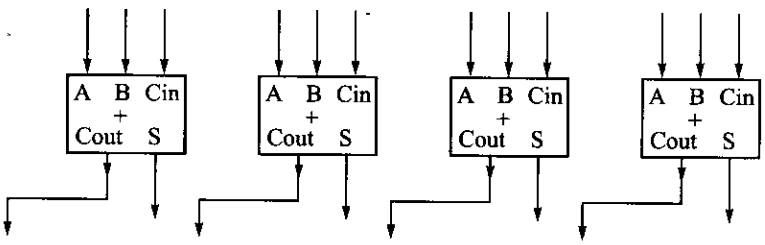
当乘法性能非常重要时，必须使用较多的硬件资源。在一些嵌入式系统中，ARM 核除了执行一般控制功能外，还要进行实时数字信号处理(DSP)。DSP 程序的特点是大量使用乘法。乘法硬件的性能可能是满足实时制约的关键。

在一些 ARM 核中，高性能乘法采用的是广泛应用的冗余二进制表示，以避免与部分积相加有关的进位传递延迟。中间结果被保存为部分和及部分进位，再使用像 ALU 中加法器那样的进位传递加法器把这二者加起来，得到真正的二进制结果。但这仅在乘法结束时做一次。在乘法期间，部分和及进位在**保留进位**加法器中结合，在每级加法中进位仅传递 1 位。这就使保留进位加法器的逻辑路径比进位传递加法器短得多，而后者可能要将进位传递 32 位。因此，可以在单时钟周期完成几个保留进位操作，而这段时间仅能容纳 1 个进位传递操作。

有许多方法可构造保留进位加法器，但最简单的是 3 输入和 2 输出的形式。它把部分和、部分进位和部分积作为输入，3 者有相同的二进制权重，产生新的部分和及部分进位以作为输出，进位具有和的 2 倍权重。每位的逻辑功能与传统的行波进位传递加法器(见图 4.10)中的全加器一致，但结构是不同的。图 4.16 说明了这两种结构。进位传递加法器取两个传统的(非冗余)二进制数作为输入，产生二进制和。保留进位加法器取 1 个二进制和 1 个冗余(部分和与部分进位)输入，产生以冗余二进制表示的和。



(a) 进位传递



(b) 保留进位

图 4.16 进位传递和保留进位加法器结构

在反复进行的乘法操作中，每一级产生的和被反馈回去，并与一个新的部分积相加。当所有的部分积都加过后，把部分和及部分进位在 ALU 的进位传递加法器中加起来，把冗余表示转化为传统的二进制数。

高速乘法器具有几层相互串联的保留进位加法器，每层处理 1 个部分积。如果部分积是按照类似表 4.3 描述的一种改进的 Booth 算法产生的，那么每一级保留进位加法器在每个周期处理两位乘数。

用于一些 ARM 核的高性能乘法器的整体结构如图 4.17 所示。寄存器的名称指的是将在 5.8 节描述的指令域。保留进位阵列有 4 层加法器，每一层处理两位乘数，因此，阵列每个周期能乘 8 位。部分和及进位寄存器在指令的开始被清 0，或者可以将部分和寄存器初始化为累加值。因为乘数在 Rs 寄存器中每周期右移 8 位，所以部分和及进位每周期循环右移 8 位。阵列最多循环 4 次。当乘数在高位有很多 0 时，可提前终止以便在较少的周期内完成指令，部分和及进位被一次合成为 32 位，并写回到寄存器堆。（当乘法提前结束时，部分和及进位需要重新对准，图 4.17 没有表示这一点。）

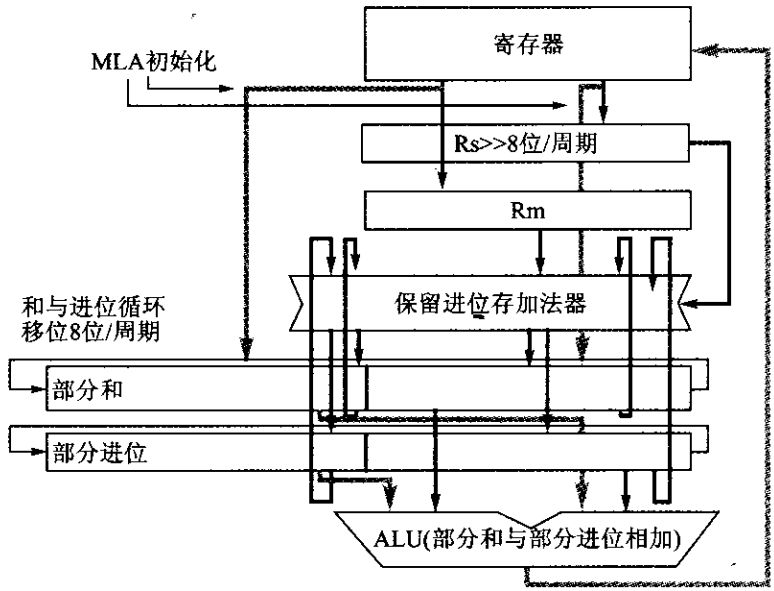


图 4.17　ARM 高速乘法器组织结构

高速乘法器比用于其他 ARM 核的低成本解决方案需要更多的硬件。有 160 位移位寄存器和 128 位保留进位加法器逻辑。增加的面积成本大约是较简单处理器核的 10%，在 ARM8 和 StrongARM 等高性能核中占的比例就更小了。获得的好处是乘法大约加速 3 倍，而且支持 64 位结果形式的乘法指令。

寄存器堆

ARM 数据通路最后一个主要模块是寄存器堆。在这里，所有的用户可见状态被保存在 31 个通用 32 位寄存器里，总共有约 1K 位数据。因为在设计中 1 位寄存器单元重复许多次，因此，值得下大力气来减小它的尺寸。

到 ARM6 为止，ARM 核使用的寄存器单元的晶体管电路如图 4.18 所示。存储单元是 CMOS 反相器的非对称交叉偶合对。当寄存器内容变化时，ALU 总线的强信号对其过载驱动。为了减小单元对新值的阻挡，反馈反相器做得较弱。A、B 读总线在时钟周期的第 2 相预充到 Vdd。因此，寄存器单元仅需要在读总线上放电。当读线使能

时，通过 n 型门管放电。

这些寄存器单元被设计为适于在 5 V 电源电压下工作。但是，在低电源电压下，通过 n 传输管写“1”就很困难。因为低电压可以得到好的电源效率，从 ARM6 开始，ARM 核或者使用全 CMOS 传输门(在写电路的 N 管上并联一个 P 管，需要互补的写使能控制线)，或者使用更复杂的寄存器电路。

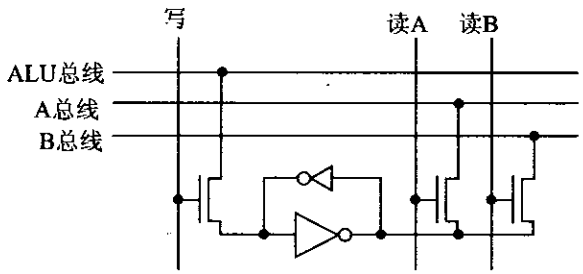


图 4.18 ARM6 的寄存器单元电路

这些寄存器单元被按列组织成 32 位寄存器。各列组合在一起形成整个寄存器堆。然后将读写使能线的译码器排列在单元列的上面，如图 4.19 所示，因而使能信号垂直布线，数据总线水平跨越寄存器单元阵列。由于译码器在逻辑上比寄存器单元本身复杂，但又要按照单元来选择水平间距，所以，译码器版图变得非常紧凑，译码器本身必须又高又窄。

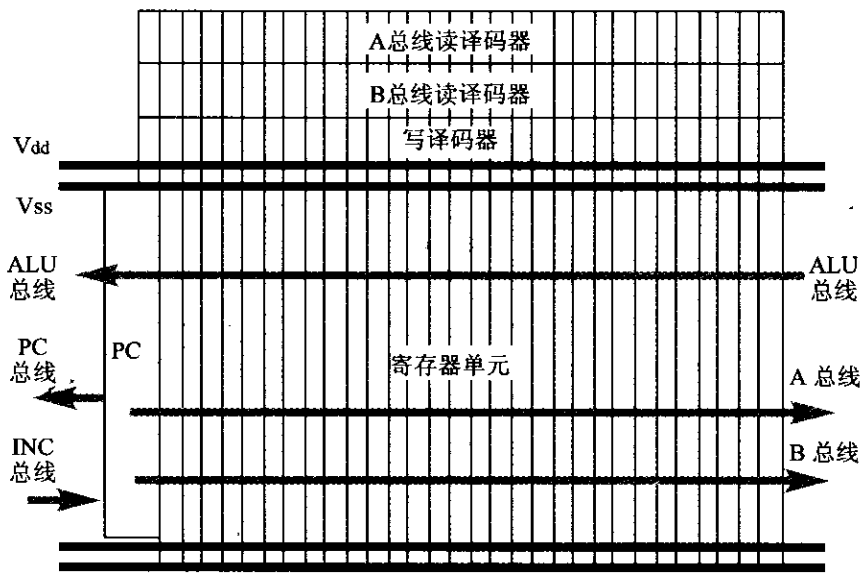


图 4.19 ARM 寄存器堆的版图规划

在较简单的核中，ARM 程序计数寄存器在物理上是寄存器堆的一部分，但它有 2 个写端口及 3 个读端口，而其他的寄存器有 1 个写端口和 2 个读端口。将 PC 安排在一端，使它可以靠近额外的端口而且可以作得较“胖”，寄存器阵列的对称因而得以保持。

在简单的 ARM 核中，寄存器堆的晶体管数占总数的 1/3。但是，由于其类似存储

器一样的紧密结构，占用的芯片面积相比之下要小得多。它不能同 SRAM 块的晶体管密度相比，因为它有两个读端口，并且要与数据通路的间距匹配。间距由较复杂的逻辑功能（如 ALU）决定。但由于较高的规则性，它的密度还是比其他逻辑功能要大。

数据通路的版图

ARM 数据通路中的每一位都按等间距布图。复杂的功能（例如 ALU）适于大间距布图；而简单的功能（例如移位寄存器）按小间距布图最有效。实际的间距将是这两者的折衷。

每个功能块都按这个间距布图。要记住，还会有总线穿越功能块（例如 B 总线穿越 ALU，但 ALU 没有使用），因此，必须为此留出空间。为数据通路制定一种如图 4.20所示的版图规划，记下穿越每一个模块的“过客”总线，这是一个好的想法。为功能模块确定排列顺序，以便减少穿越复杂功能模块的额外总线的数量。

现代 CMOS 工艺允许用多个金属层布线（早期的 ARM 核使用两层金属）。必须仔细选择哪层布线用做电源和地，哪层布线用做数据通路中的总线信号，以及哪层布线用做跨越数据通路的控制信号（例如在 ARM2 中，Vdd 和 Vss 用第 2 层金属走在数据通路两边，跨越数据通路的控制线使用第 1 层金属，而总线总是沿第 2 层金属走线）。

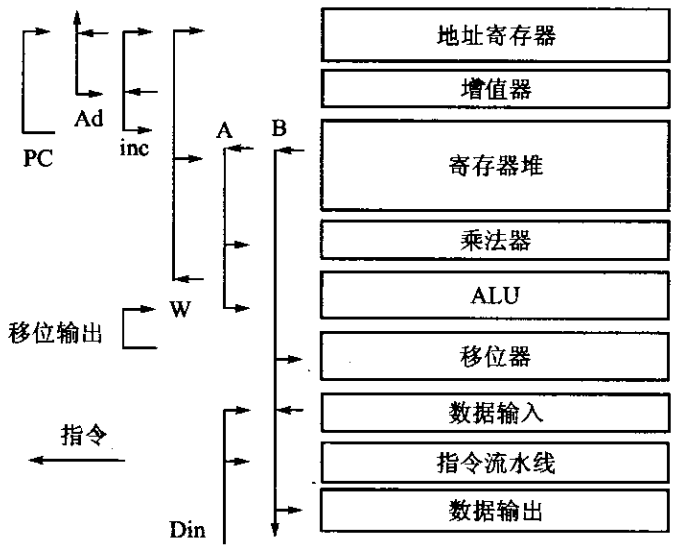


图 4.20　ARM 核的数据通路总线

控制结构

在较简单的 ARM 核中，控制逻辑有 3 种彼此相关的结构元件，如图 4.21 所示。

1）指令译码器 PLA（Programmable Logic Array，可编程逻辑阵列）。这个单元使用指令中的某些位和内部的周期计数器来定义下一个周期在数据通路上完成的操作类型。

2）分散且与每一个主要数据通路功能块相关的二级控制单元。该逻辑使用来自从主译码器 PLA 的类信息，选择其他指令位和/或处理器状态信息，以便控制

数据通路。

3) 分散的控制单元。用于周期数可变的特殊指令(多寄存器 Load 和 Store、乘法和协处理器操作)。这时主译码器 PLA 锁定在一个固定状态,直到远程控制单元指明操作完成为止。

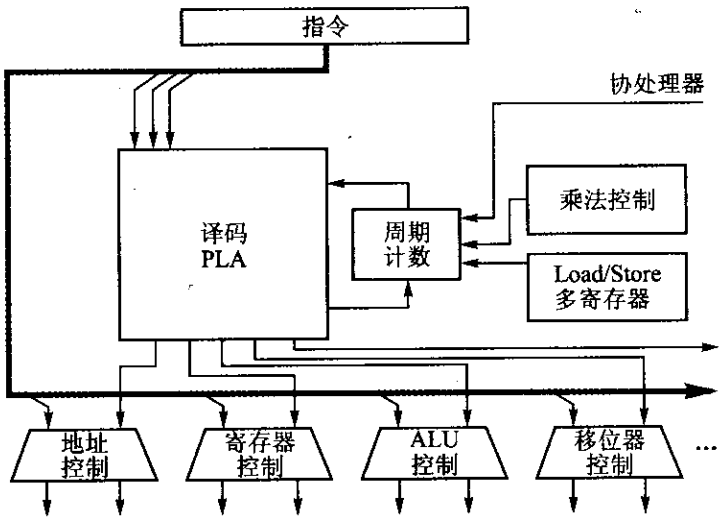


图 4.21 ARM 控制逻辑的结构

主译码器 PLA 有约 14 个输入、40 个乘积项和 40 个输出,对于不同的核,精确的数目稍有不同。在最近的 ARM 核中,它以两个 PLA 实现:较小的、快的 PLA 产生时间关键的输出;较大的、慢的 PLA 产生所有其他输出。然而从功能上它们可被视为同一个 PLA 单元。

周期计数模块可以标识多周期指令的不同周期,以便将 PLA 译码为每一个周期产生不同的控制输出。事实上它不是简单的计数器,而是一个更一般的有限状态机。它能跳过不需要的周期及锁定在一个固定的状态。它确定何时当前指令将要完成,何时开始从指令流水线中传送下一条指令,包括当指令的条件测试未满足时在它的第 1 个周期末尾中止指令。但是,它的行为在很多时候像一个简单的计数器,因此,把它看成指令周期计数器也不算过于误导。

物理设计

到此为止我们主要关心 ARM 核的逻辑设计,很少谈及它在特定的 CMOS 工艺中的物理实现。

在特定工艺中实现 ARM 处理器核(就此而言,或其他的核)有两种主要机制:

- 交付硬宏单元:可以集成到最终设计中的物理版图。
- 交付软宏单元:以 VHDL 等硬件描述语言表述的、可综合的设计。

硬宏单元在目标工艺的全部特性都已确定,具有全定制手工布图的面积优势,但只能应用于设计所采用的特定工艺。每次改变工艺都必须修改版图和重新确定特性。软宏单元可以容易地转向新的工艺技术,但每一次工艺变化后,也必须重新确定它的

特性。

早期的 ARM 核仅以硬宏单元交付。它们的数据通路采用全定制设计，在逻辑原理图级设计控制逻辑，使用自动布局布线工具和标准单元库将逻辑转化为版图。为便于工艺移植，使用普通的设计规则（单元库和全定制数据通路都是如此）来设计。这样就可以对上述物理布图进行几何转换，将其映射到一些设计规则类似但不完全一致的其他工艺。

最近的 ARM 核有了软、硬核两种形式。硬宏单元在保留手工全定制数据通路的同时，其控制逻辑用综合实现。软宏单元为寄存器传输级（RTL）描述，是完全可综合的。

一些 ARM 合作伙伴采用中间路线，使用 ARM 核的门级网表描述作为基础，向新工艺移植。在移植过程中不进行再综合，只是将上述网表（使用自动布局布线工具）映射到以新工艺实现的标准单元库。

在软、硬宏单元（或门级网表）间选择是一个复杂的决定。硬宏单元显然能给出在特定工艺下最佳的面积、性能和电源效率，但是要移植到其他工艺则需花费大量时间、劳动和成本。软宏单元和可移植的网表则更灵活，而且现在自动化工具的质量已经很高。也就是说，在性能上它们正接近手工布图。软宏单元的可移植性则意味着我们要在最新工艺的软宏单元和老工艺的硬宏单元之间进行选择，而工艺技术的先进性可轻易地胜过优化方面的微小损失。

4.5　ARM 协处理器接口

ARM 通过增加硬件协处理器来支持对其指令集的通用扩展，通过未定义指令陷阱支持这些协处理器的软件仿真。

协处理器的体系结构

协处理器的体系结构将在 5.16 节说明。它最重要的特征如下：

- 支持多达 16 个逻辑协处理器。
- 每个协处理器可以有多达 16 个专用的寄存器，其大小不限于 32 位，可以是任何合理的位数。
- 协处理器使用 Load-Store 体系结构，有对内部寄存器操作的指令，有从存储器读取数据装入寄存器以及将寄存器数据存入存储器的指令，以及与 ARM 寄存器传送数据的指令。

简单的 ARM 核提供板级协处理器接口，因此，协处理器可以作为一个独立的元件接入。高速时钟使得板级接口非常困难，因此，高性能的 ARM 协处理器接口仅限于片上使用，特别是 Cache 和存储器管理控制功能，但是也可以支持其他的片上协处理器。

ARM7TDMI 协处理器接口

ARM7TDMI 协处理器接口是基于总线监视的技术（其他 ARM 核使用不同的技

术)。协处理器与一个用于 ARM 指令流流入 ARM 的总线相连。协处理器将指令拷贝到内部的流水线,并在内部流水线中模仿 ARM 指令流水线的行为。

当每一个协处理器指令开始执行时,在 ARM 和协处理器之间有一个"握手",以确认它们两者都准备执行它。握手使用 3 种信号:

1) $\overline{\text{cpi}}$(从 ARM 到所有的协处理器):该信号代表"协处理器指令",指示 ARM 已识别到一个协处理器指令,希望执行它。
2) cpa(从协处理器到 ARM):"协处理器不存在"信号,告诉 ARM 没有能执行当前指令的协处理器。
3) cpb(从协处理器到 ARM):"协处理器忙"信号,告诉 ARM 协处理器还不能开始执行指令。

在时序上,ARM 和协处理器两者都必须独立地产生它们各自的信号。协处理器不可能在产生 cpa 和 cpb 之前一直等到看到$\overline{\text{cpi}}$信号。

握手的结果

一旦协处理器指令进入 ARM7TDMI 和协处理器流水线,依靠握手信号的不同,可能有 4 种方式处理这条指令,即

1) ARM 可以确定不执行它,或者由于它处于转移阴影中,或者由于未能通过条件码测试。(所有的 ARM 指令,包括协处理器指令,都是条件执行。)ARM 将不会声明$\overline{\text{cpi}}$,指令将被完全放弃。
2) ARM 可能确定执行它(通过声明$\overline{\text{cpi}}$来发出信号),但是没有协处理器可以接收它,因此 cpa 保持有效。ARM 将采用未定义的指令陷阱,并使用软件来恢复(可能通过仿真被捕获的指令来恢复)。
3) ARM 确定执行指令而且有协处理器接收它,但还不能执行它。协处理器使 cpa 变低,但使 cpb 仍保持为高。ARM 将在此停止指令流,遇忙等待,直到协处理器将 cpb 变低为止。如果在协处理器忙时有中断请求到达,则 ARM 将暂停以处理中断,稍后返回以重试协处理器指令。
4) ARM 确定执行指令,而且协处理器接收并立即执行它。cpi、cpa 和 cpb 都变低,双边按约定完成指令。

数据传送

如果指令是协处理器数据传送指令,则 ARM 负责产生初始存储器地址(协处理器不需要同地址总线有任何连接),但协处理器需要确定传送的长度。ARM 将连续增加地址,直到协处理器发出信号表示传送已经完成为止。握手信号 cpa 和 cpb 也用于这个用途。

因为数据传送一旦开始便不可中断,所以,协处理器应将最大的传送长度限制为 16 个字(同多寄存器 Load 和 Store 指令的最大长度一样),以至不会危及 ARM 中断响应。

优先执行

如果握手没有最终完成,那么只要它能恢复状态,一旦指令进入流水线,协处理器

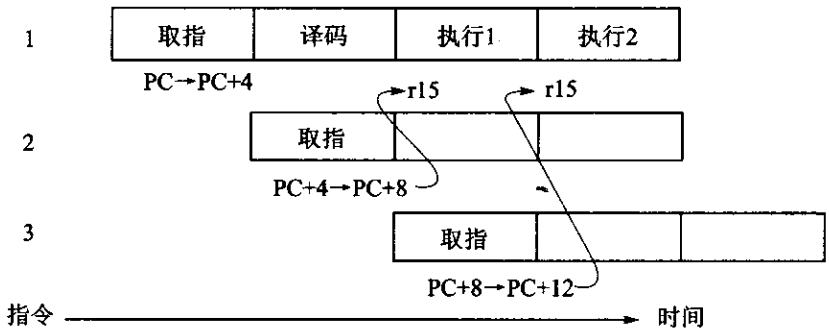
就可以开始执行。直到提交时，所有的活动都必须是等幂的(可重复为同样的结果)。

4.6　例题与练习

例题　4.1

为什么 ARM7 中的 r15 在指令的第 1 周期给出 PC+8，而在后续的时钟周期给出 PC+12?

这是展示在程序员面前的 ARM 流水线。参照图 4.2，可以看到，在当前指令(在下图中的指令 1)取指时以及它的后续指令(指令 2)取指时，PC 值都要增加，在第 1 个执行周期开始时得到 PC+8。在第 1 个执行周期，第 3 个指令(指令 3)取指，在所有的后续执行周期给出 PC+12。



当多周期指令中断流水线的流程时，它们不影响这方面的行为。一条指令总是在它的第 1 个执行周期进行下下条指令的取指，因此，r15 总是从第 1 个执行周期开始时的 PC+8 变为第 2 个(及后续)执行周期开始时的 PC+12。

(注意其他 ARM 处理器不具有这一行为，因此，当写 ARM 程序时，不要依靠它。)

练习　4.1.1

接续上例画一个流水线的流程图，举例说明 ARM 转移指令的时序。(转移目标在指令的第 1 执行周期计算，在随后的周期发送到存储器。)

练习　4.1.2

在转移目标计算之后和转移目标处的指令准备执行之前有多少执行周期?

处理器使用这些执行周期做什么?

例题　4.2

完成图 4.11 和图 4.12 中 ARM2 的 4 位进位逻辑电路。

4 位超前进位(look-ahead)的设计使用各位的进位产生和进位传递信号。这些信号由如图 4.12 所示的逻辑产生。这些位由 G[3：0]和 P[3：0]表示，从 4 位组的最高位给出的进位输出由下式给出：

$$Cout=G[3]+P[3]\cdot(G[2]+P[2]\cdot(G[1]+P[1]\cdot(G[0]+P[0]\cdot Cin)))$$

因此，在图 4.11 使用的块进位产生和传递信号 G4 和 P4，则由下式给出：

$$G4=G[3]+P[3]\cdot(G[2]+P[2]\cdot(G[1]+P[1]\cdot G[0]))$$

$$P4=P[3]\cdot P[2]\cdot P[1]\cdot P[0]$$

这两个信号独立于进位输入信号，因此，能在它到来之前被建立，使得进位仅用 1 个“与-或-非”门延迟时间就传递通过 4 位组。

练习 4.2.1

使用输入和功能选择信号，写出图 4.12 所示电路产生的一位 ALU 输出的逻辑表达式，并由此表示出表 4.1 列举的全部 ALU 功能是如何产生的。

练习 4.2.2

估计行波进位加法器和 4 位超前进位加法器的门数。两个设计都基于图 4.12 所示的电路，不同的只是进位方案。

超前进位方案的额外速度以多少门为代价？它是怎样影响加法器的规则性，进而影响了设计代价的？

第 5 章　ARM 指令集

本章内容综述

在第 3 章我们着重于用户级 ARM 汇编语言编程，对 ARM 指令集有了初步的了解。在本章我们将关注指令集更详尽的细节，介绍标准 ARM 指令集中的全部指令。

一些 ARM 核同时可以执行压缩形式的指令集，它将 ARM 指令集的一个子集编码为 16 位指令。这就是将在第 7 章中讨论的 Thumb 指令。在本章中将看到的有关 Thumb 体系的惟一内容就是在 ARM 指令集中这样一条指令，它可以使处理器切换到执行 Thumb 指令。同样，一些 ARM 核支持扩展指令集以增强它们的信号处理能力。对它们的讨论推迟到 8.9 节。

像所有处理器的全部指令集一样，ARM 指令集也有一些隐藏复杂行为的角落。这些角落对程序员通常是完全没有用处的。ARM 公司没有定义角落情况下处理器的行为，相应的指令也是不能使用的。这种以特殊方式实现的 ARM 行为在将来也不一定会以同样的方式实现。程序应当仅使用定义了语义的指令！

一些 ARM 指令并非在所有的 ARM 芯片中都能使用。当出现这种情况时将会强调指出。

5.1　引　言

在图 2.1 中曾介绍了 ARM 的程序员模型。在本章将考虑监控(supervisor)和异常(exception)模式，因此会使用在阴影中的寄存器。

数据类型

ARM 处理器支持 6 种数据类型：

- 8 位有符号和无符号字节。
- 16 位有符号和无符号半字，它们以两字节的边界对准。
- 32 位有符号和无符号字，它们以 4 字节的边界对准。

(一些早期的 ARM 处理器不支持半字和有符号字节。)

ARM 指令全是 32 位的字并且必须是字对准的。Thumb 指令是半字而且必须以两字节的边界对准。

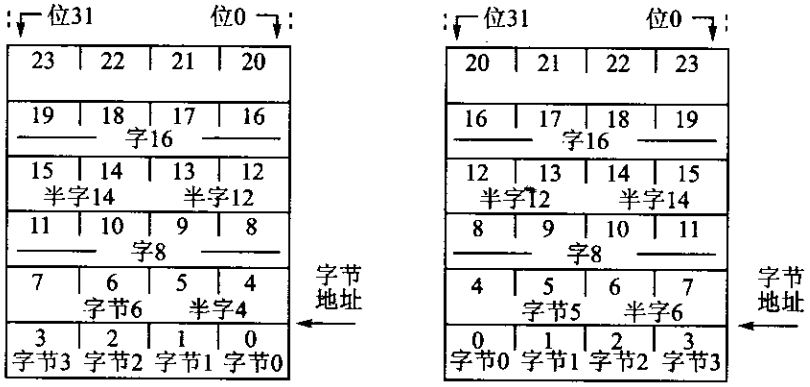
在内部，所有的 ARM 操作都面向 32 位的操作数。只有数据传送指令支持较短的数据类型。当从存储器加载一个字节时，将它 0 或符号扩展为 32 位，然后作为 32 位数据进行内部处理。

ARM 协处理器可能支持其他数据类型，特别是定义了一些表示浮点数的数据类型。在 ARM 核内没有明确地支持这些数据类型。然而在没有浮点协处理器的情况下，这些类型可由软件用上述标准类型来解释。

存储器组织

在以字节为单位寻址的存储器中，有两种方式来存储字，这根据最低有效字节与相邻较高有效字节相比是存在较低的还是较高的地址来划分。由于没有充足的理由来选择一种方式而否定另一种方式，关于哪种方式更好的争论就更像是宗教之争。

两种方式如图 5.1 所示。图中展示了在两种方式下各种类型的数据是如何存储的。（半字 12 存储于地址 12，等等。）



(a) 小端存储器组织　　(b) 大端存储器组织

图 5.1　小端和大端存储器组织

用来表示两种存储方式的术语“小端(little-endian)”和“大端(big-endian)”源于 Swift 的《格利佛游记》。小人国的居民以矮小而著名，法律强迫他们在打鸡蛋时只能打小端。当这一法律实施时，那些喜欢在大端打鸡蛋的公民对新的规则产生异议，内战爆发。最后大端派在邻近的岛屿避难，这就是 Blefuscu 王国。内战导致了许多伤亡。

用“大端”和“小端”这两个词来表示计算机存储器的两种组织方法源于 Danny Cohen于 1981 年 10 月在《Computer》上发表的“论圣战与恳求和平”一文。

就我所知，在字节排序的争论中还没有人受到致命的伤害。但是，这个问题造成了在不同排序规则的机器之间传送数据时的重大实际困难。

大多数 ARM 芯片在争论中保持严格的中立，并能配置为使用任何一种存储器管理方式来工作，但它们的缺省设置为小端格式。在本书中将通篇采用小端格式，即较高的有效字节存放在较高的存储器地址。ARM 可以是中立的，但我不是！

特权模式

正如第 3 章所述，绝大多数程序都在用户模式下操作。但是，ARM 还有一种用于

处理异常和监控调用(有时称为**软件中断**)的**特权**操作模式。

CPSR(Current Program Status Register，当前程序状态寄存器，见图 2.2)的低 5 位用于定义当前操作模式。在表 5.1 中给出了对这些位的解释。如果寄存器组不是用户寄存器，则图 2.1 中所示的相关的阴影寄存器取代相应的用户寄存器，而且当前 **SPSR**(Saved Program Status Register，程序状态保存寄存器，见后面)也变为可以访问的。

表 5.1　ARM 操作模式和寄存器使用

CPSR[4：0]	模　式	用　途	寄存器
10000	用户	正常用户模式	用户
10001	FIQ	处理快速中断	_fiq
10010	IRQ	处理标准中断	_irq
10011	SVC	处理软件中断(SWI)	_svc
10111	中止	处理存储器故障	_abt
11011	未定义	处理未定义的指令陷阱	_und
11111	系统	运行特权操作系统任务	用户

有些 ARM 处理器不能支持上面的所有操作模式，有些还支持 26 位模式以便同较早的 ARM 兼容。这些将在 5.23 节作进一步的讨论。

进入特权模式只能通过受控机构。当采取适当的存储器保护后，这些模式可以建立全面保护的操作系统。这些问题将在第 13 章作进一步的讨论。

大多数 ARM 用于嵌入式系统。在这类系统中此类保护是不适当的。但是，仍然可以使用特权模式给出较弱的保护，这对中断错误软件是有用的。

SPSR

每一种特权模式(系统模式除外)都有一个与之相关的程序状态保存寄存器 SPSR。这个寄存器在进入特权模式时保存 CPSR 的状态，以便当重新开始用户过程时能全部恢复用户状态。从进入特权模式的时候开始，到用 SPSR 来恢复 CPSR 的时候为止，在这段时间 SPSR 通常不会改变。但是如果特权软件重新进入(例如，监控代码使得监控调用自己)，那么必须把 SPSR 拷贝到一个通用寄存器并保存起来。

5.2　异　常

通常用异常来处理在执行程序时发生的意外事件，如中断、存储器故障。在 ARM 体系结构中，异常也用来指软件中断、未定义指令陷阱(它不是真正的“意外”事件)及系统复位功能(它在逻辑上发生在程序执行前而不是在程序执行中，尽管处理器在运行中可能再次复位)。这些事件都被划归“异常”，因为在处理器中它们都使用同样的基本

机制。

ARM 异常可以分为 3 类：

1) 指令执行引起的直接异常。软件中断，未定义指令(包括所要求的协处理器不存在时的协处理器指令)和预取指中止(因为取指过程中的存储器故障导致的无效指令)属于这一类。

2) 指令执行引起的间接异常。数据中止(在 Load 和 Store 数据访问时的存储器故障)属于这一类。

3) 外部产生的与指令流无关的异常。复位、IRQ 和 FIQ 属于这一类。

异常的进入

当发生异常时，ARM 尽量完成当前指令(除了复位异常立即中止当前指令)，然后脱离当前的指令序列去处理异常。间接或外部事件引起的异常将占据当前序列中的下一条指令。直接异常在产生时就按顺序处理。处理器将执行下列动作：

- 进入与特定的异常相应的操作模式。
- 将引起异常指令的下一条指令的地址保存到新模式的 r14 中。
- 将 CPSR 的原值保存到新模式的 SPSR 中。
- 通过设置 CPSR 的第 7 位来禁止 IRQ。如果异常为快速中断，则还要设置 CPSR 的第 6 位来禁止快速中断。
- 给 PC 强制赋值，使程序从表 5.2 给出的相应的向量地址开始执行。

表 5.2 异常的向量地址

异 常	模 式	向量地址
复位	SVC	0x00000000
未定义指令	UND	0x00000004
软件中断(SWI)	SVC	0x00000008
预取指中止(取指存储器故障)	Abort	0x0000000C
数据中止(访问数据存储器故障)	Abort	0x00000010
IRQ(正常中断)	IRQ	0x00000018
FIQ(快速中断)	FIQ	0x0000001C

一般地说，向量地址处将包含一条指向相应程序的转移指令。但 FIQ 可以立即开始执行，因为它占据最高向量地址。

每个特权模式的两个寄存器用来保存返回地址和堆栈指针。堆栈指针可以用来保存其他用户寄存器，这样异常处理程序就可以使用这些寄存器。FIQ 模式还有额外的专用寄存器，使用这些寄存器可以使大多数情况不必保存用户寄存器而得到较好的性能。

异常返回

一旦异常处理完毕，用户任务便恢复正常。这就要求异常处理程序代码精确恢复异常发生时的用户状态，即

- 所有修改过的用户寄存器必须从处理程序的堆栈中恢复。
- CPSR 必须从相应的 SPSR 中恢复。
- PC 必须变回到在用户指令流中相应的指令地址。

注意，这些步骤中的最后两步不能独立完成。如果先恢复 CPSR，则保存返回地址的当前异常模式的 r14 就不能再访问了；如果先恢复 PC，则异常处理程序将失去对指令流的控制，使得 CPSR 不能恢复。为确保指令总是按正确的操作模式读取，以保证存储器保护方案不被绕过，还有更加微妙的困难。因此，ARM 提供了两种机制，利用这些机制，可以使上述两步作为一条指令的一部分同时完成。当返回地址保存在当前异常模式的 r14 时，使用其中一种机制，当返回地址保存在堆栈时使用另一种机制。首先看一看返回地址保存在 r14 的情形。

- 从 SWI 或未定义指令陷阱返回，使用：

```
MOVS    pc, r14
```

- 从 IRQ、FIQ 或预取指中止返回，使用：

```
SUBS    pc, r14, #4
```

- 从数据中止返回并重新访问数据，使用：

```
SUBS    pc, r14, #8
```

当目的寄存器是 pc 时，操作码后面的修饰符 S 表示特殊形式的指令。注意返回指令如何在必要时对返回地址进行调整：

- IRQ 和 FIQ 必须返回前一条指令，以便执行因进入异常而被占据的指令。
- 预取指中止必须返回前一条指令，以便执行在初次请求访问时造成存储器故障的指令。
- 数据中止必须返回前面第二条指令，以便重新执行因进入异常而被占据的指令之前的数据传送指令。

如果异常处理程序把返回地址拷贝到堆栈（例如为了能够再次进入异常，但要注意，在这种情况下 SPSR 也同 PC 一样必须保存），可以使用一条如下的多寄存器传送指令来恢复用户寄存器并实现返回，即

```
LDMFD   r13!, {r0-r3, pc}^          ; 恢复和返回
```

这里，寄存器列表（其中必须包括 PC）后面的“^”表示这是一条特殊形式的指令。在从存储器中装入 PC 的同时，CPSR 也得到恢复。由于寄存器是按照升序装入的，所以 PC 是从存储器传送的最后一个数据。

这里使用的堆栈指针（r13）是属于特权操作模式的寄存器。每个特权模式都可以有它自己的堆栈指针。该堆栈指针必须在系统启动时进行初始化。

显然，只有当 r14 的值在存入堆栈之前进行过调整，才可以使用堆栈的返回机制。

异常的优先级

因为多种异常可以同时产生，需要定义优先级以便确定处理异常的顺序。对于ARM，优先级如下：

1）复位（最高优先级）；

2）数据异常中止；

3）FIQ；

4）IRQ；

5）预取指异常中止；

6）SWI，未定义指令（包括缺协处理器）。这两者是互斥的指令编码，因此不可能同时发生。

复位从确定的状态启动微处理器，使得所有其他未解决的异常都没有关系了。

最复杂的是 FIQ、IRQ 和第 3 个异常（不是复位）同时发生的情形。FIQ 比 IRQ 的优先级高并将 IRQ 屏蔽，所以 IRQ 将被忽略，直到 FIQ 处理程序明确地将 IRQ 使能或返回用户代码为止。

如果第 3 个异常是数据中止，那么因为进入数据中止异常并未将 FIQ 屏蔽，所以处理器将在进入数据中止处理程序后立即进入 FIQ 处理程序。数据中止将被“记”在返回路径中，当 FIQ 处理程序返回时对其进行处理。

如果第 3 个异常不是数据中止，将立即进入 FIQ 处理程序。当 FIQ 和 IRQ 两者都完成时，程序返回到产生第 3 个异常的指令，在余下的所有情况下异常将重现并作相应的处理。

地址异常

细心的读者会注意到，在表 5.2 中，在存储器的前 8 个字中，除了地址 0x00000014 之外，其余全部被用做异常向量地址。

在早期 26 位地址空间的 ARM 处理器中，曾使用地址 0x00000014 来捕获落在地址空间外的 Load 和 Store 地址。这些陷阱称为“地址异常”。

因为 32 位的 ARM 不会产生落在它 32 位地址空间之外的地址，所以，地址异常在当前的体系结构中没有作用，0x00000014 的向量地址也就不再使用了。

5.3 条件执行

ARM 指令集不同寻常的特征是每条指令（除了某些 v5T 指令）都是条件执行。条件转移是绝大多数指令集的标准特征，但 ARM 将条件执行扩展到所有的指令，包括监控调用和协处理器指令。条件域占据 32 位指令域的高 4 位，如图 5.2 所示。

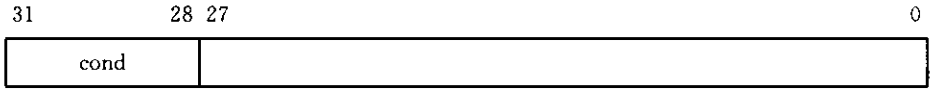


图 5.2　ARM 的条件代码域

条件域共有 16 个值，每个值都根据 CPSR 中标志位 N、Z、C 和 V 的值来确定指令是执行还是跳过。表 5.3 给出了这些条件。每条 ARM 指令助记符都可以扩展两个字母，在表中定义了这些字母。“总是(always)”条件(AL)可以省略，因为它是缺省条件。如果没有其他指定，就假定是该条件。

表 5.3　ARM 的条件码

操作码 [31：28]	助记符扩展	解　释	用于执行的标志位状态
0000	EQ	相等/等于 0	Z 置位
0001	NE	不等	Z 清 0
0010	CS/HS	进位/无符号数高于或等于	C 置位
0011	CC/LO	无进位/无符号数低于	C 清 0
0100	MI	负数	N 置位
0101	PL	正数或 0	N 清 0
0110	VS	溢出	V 置位
0111	VC	未溢出	V 清 0
1000	HI	无符号数高于	C 置位且 Z 清 0
1001	LS	无符号数低于或等于	C 清 0 或 Z 置位
1010	GE	有符号数大于或等于	N 等于 V
1011	LT	有符号数小于	N 不等于 V
1100	GT	有符号数大于	Z 清 0 且 N 等于 V
1101	LE	有符号数小于或等于	Z 置位或 N 不等于 V
1110	AL	总是(always)	任何状态
1111	NV	从不(never)(不要使用)	无

“never”条件

“never”(NV)条件是不应使用的——有很多其他方式可以在 ARM 代码中写入对处理器状态没有任何作用的指令。避免使用“never”条件的原因是 ARM 公司已指出，将来可能使用这部分指令空间作其他用途(已在 v5T 体系中使用)，因此，尽管现在的 ARM 微处理器会按照预期方式执行，但不能保证未来的机型也同样地执行。

可选用的助记符

如果在表 5.3 中的同一行有两种助记符供选用，则表示有 1 种以上的方式来解释条件域。例如在第 3 行，助记符扩展 CS 或 HS 可以使用同一个条件域的值。只要 CPSR 的位 C 置位，两者都会使指令执行。出现两种可选用的助记符是因为在不同的环境下进行的是同样的检测。如果想先将两个无符号整数相加，并想测试加法是否有进位输出，则应当使用 CS。如果想比较两个无符号整数并想测试是否第 1 个大于等于第 2 个，则应当使用 HS。可选择的助记符使程序员不需要记住无符号数比较在较高或相同时设置进位。

细心的读者将注意到，条件是成双的，意即第 2 个条件是第 1 个的反，所以，对于任一条件都有其反条件(除了“always”，因为“never”不应该使用)。因此，只要能用条件指令来实现“if …then…”，就能用带有相反条件的指令加入“…else…”。

5.4 转移及转移链接(B,BL)指令

转移和转移链接指令是改变指令执行顺序的标准方式。ARM 一般按照存储器的字地址顺序执行指令，需要时使用条件执行跳过个别指令。只要程序必须偏离顺序执行，就要使用控制流指令来修改程序计数器。尽管在特定情况下还有几种方式实现这个目的，但转移和转移链接是标准的方式。

二进制编码

转移和转移链接指令的二进制编码如图 5.3 所示。

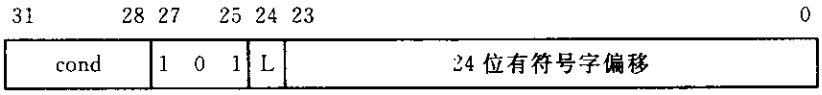


图 5.3 转移和转移链接指令的二进制编码

说 明

转移和转移链接指令使处理器开始执行来自新地址的指令。这个地址是这样计算出来的：先对指令中定义的 24 位偏移量进行符号扩展，左移两位形成字的偏移，然后将它加到程序计数器，相加前程序计数器的内容为转移指令地址加 8 个字节(参看 4.1 节中“PC 的行为”一段关于 PC 偏移的解释)。一般情况下，汇编器将会计算正确的偏移。

转移指令的范围为±32 MB。

转移指令有位 L(第 24 位)置位的链接形式，它将转移后下一条指令的地址传送到当前处理器模式下的链接寄存器(r14)。这一般用于实现子程序调用，返回时将链接寄存器的内容拷贝回 PC。

两种形式的指令都可以条件执行或无条件执行。

汇编格式

B{L}{<cond>}　　<target address>

“L”指定转移与链接属性。如果不包含 L，便产生没有链接的转移。“<cond>”应是表 5.3 给出的助记符扩展。如果省略，则假设为“AL”。“<target address>”一般是汇编代码中的标号。汇编器将产生偏移（它将是目标地址和转移指令地址加 8 的差值）。

举　例

无条件跳转：

```
        B       LABEL           ; 无条件跳转…
        …
LABEL   …                       ; …到这里
```

执行 10 次循环：

```
        MOV     r0, #10         ; 初始化循环计数器
LOOP    …
        SUBS    r0, #1          ; 计数器减 1，设置条件码
        BNE     LOOP            ; 如果计数器≠0，则重复循环…
        …                       ; …否则中止循环
```

调用子程序：

```
        …
        BL      SUB             ; 转移链接到子程序 SUB
        …                       ; 返回到这里
        …
SUB     …                       ; 子程序入口
        MOV     pc, r14         ; 返回
```

条件子程序调用：

```
        …
        CMP     r0, #5          ; 如果 r0<5
        BLLT    SUB1            ; 则调用 SUB1
        BLGE    SUB2            ; 否则调用 SUB2
        …
```

（注意，只有 SUB1 不改变条件码，本例才能正确工作，因为如果 BLLT 执行了转移，将返回到 BLGE。如果条件码被 SUB1 改变，则 SUB2 可能又会被执行。）

注意事项

1) 如果你熟悉其他 RISC 处理器，你可能会预料在上面第一个例子中 ARM 将遵循许多其他 RISC 所采用的延迟转移模式，在转移到标号 LABEL 之前会执行转移指令之后的指令。但是，这种预料将不会实现，因为 ARM 不使用转移延迟的机制。

2) 应该避免试图转移到 32 位地址空间的范围之外，因为这种转移可能导致不可预测的结果。

5.5 转移交换和转移链接交换(BX,BLX)指令

这些指令用于支持 Thumb(16 位)指令集的 ARM 芯片，是一种将处理器切换到执行 Thumb 指令或返回到 ARM 和 Thumb 调用程序的机制。一条类似的 Thumb 指令可以使处理器切换回 32 位 ARM 指令。Thumb 指令集将在第 7 章讲解。

二进制编码

转移交换(带链接选项)指令的二进制编码如图 5.4 所示。

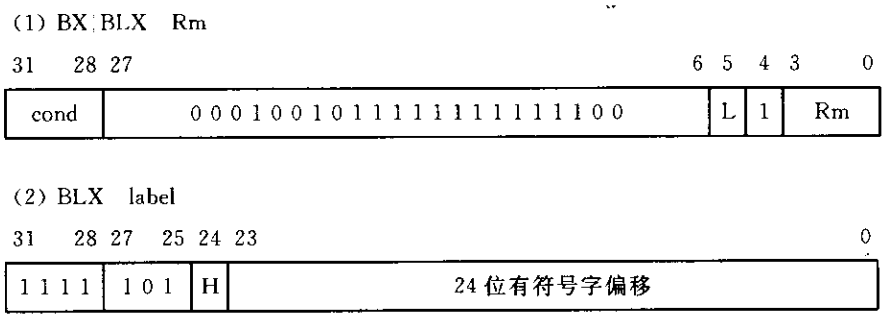


图 5.4 转移交换(带链接选项)指令的二进制编码

说 明

在图 5.4 第 1 种格式中，寄存器 Rm 指定转移目标，Rm 的第 0 位拷贝到 CPSR 中的 T 位，位[31：1]移入 PC。

- 如果 Rm[0]是 1，则处理器切换执行 Thumb 指令，并在 Rm 中的地址处开始执行。但须将最低位清 0，使之以半字的边界对准。
- 如果 Rm[0]是 0，则处理器继续执行 ARM 指令，并在 Rm 中的地址处开始执行。但须将 Rm[1]清 0，使之以字的边界对准。

在第 2 种格式中，转移目标是这样计算的：将指令中指定的 24 位偏移进行符号扩展，左移两位形成字的偏移，然后将它加到程序计数器。这时程序计数器的内容为 BX 指令地址加 8 字节(参看 4.1 节中“PC 的行为”一段关于 PC 偏移的解释)。位 H(第 24

位)也加到结果地址的第 1 位,使得可以为目标指令选择奇数的半字地址,而该目标指令将总是 Thumb 指令(BL 用做转向 ARM 指令)。汇编器将计算正常情况下的正确偏移。转移指令的范围是±32 MB。

如果在第 1 种形式中使位 L(第 5 位)置位,那么这两种形式具有链接属性的转移指令(BLX 仅用于 v5T 处理器),也将转移指令后下一条指令的地址传送到当前处理器模式的链接寄存器(r14)。当调用 Thumb 子程序时,一般用这类指令来保存返回地址。如果用 BX 作为子程序返回机制,那么调用程序的指令集能连同返回地址一起保存。因此,可使用同样的返回机制从 ARM 或 Thumb 子程序对称地返回到 ARM 或 Thumb 的调用程序。

图 5.4 第 1 种格式的指令可以条件或无条件执行,但第 2 种格式的指令是无条件执行。

汇编格式

1) B{L}X{<cond>} Rm

2) BLX <target address>

"<target address>"一般是汇编代码中的一个标号。汇编器将产生偏移(它将是目标的字地址和转移指令地址加 8 的差值),并在适当时设置位 H。

举　例

无条件跳转:

```
        BX      r0          ; 转移到 r0 中的地址
                            ; 如果 r0[0]=1,则进入 Thumb 状态
```

调用 Thumb 子程序:

```
        CODE32              ; 以下是 ARM 代码
        …
        BLX     TSUB        ; 调用 Thumb 子程序
        …
        CODE16              ; 开始 Thumb 代码
TSUB    …                   ; Thumb 子程序
        BX      r14         ; 返回到 ARM 代码
```

注意事项

1) 一些不支持 Thumb 指令集的 ARM 处理器将捕获这些指令,允许软件仿真 Thumb 指令。

2) 只有实现 v5T ARM 体系结构的处理器支持 BLX 指令的任意形式(参看 5.23 节)。

5.6 软件中断(SWI)指令

软件中断指令用于调用操作系统,常称为"监控调用"。它将处理器置于监控模式,从地址 0x08 开始执行指令。

如果存储器的这部分区域被适当保护,就有可能在 ARM 上构建一个全面防止恶意用户的操作系统。但是,由于 ARM 很少用于多用户应用环境,通常不要求这种级别的保护。

二进制编码

软件中断指令的二进制编码如图 5.5 所示。

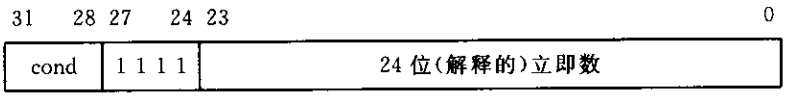


图 5.5 软件中断指令的二进制编码

说 明

24 位立即数域并不影响指令的操作,但可以由系统代码来解释。

如果条件通过,则指令使用标准的 ARM 异常入口程序进入监控模式。具体地说,处理器的行为如下:

1) 将 SWI 后面指令的地址保存到 r14_svc。

2) 将 CPSR 保存到 SPSR_svc。

3) 进入监控模式,将 CPSR[4:0]设置为 10011_2 以及将 CPSR[7]设置为 1,以便禁止 IRQ(但不是 FIQ)。

4) 将 PC 设置为 08_{16},并且开始执行那里的指令。

为了返回 SWI 后的指令,系统的程序不但必须将 r14_svc 拷贝到 PC,而且必须由 SPSR_svc 恢复 CPSR。这需要使用一种特殊形式的数据处理指令,将在下一节介绍。

汇编格式

SWI {<cond>} <24 位立即数>

举 例

输出字符"A":

```
MOV     r0, #'A'          ; 将"A"调入到 r0 中…
SWI     SWI_WriteC        ; …打印它
```

输出调用语句之后文本串的子程序:

```
        ...
        BL      STROUT                  ; 输出下列信息
        =       "Hello World", &0a, &0d, 0
        ...                             ; 返回这里
STROUT  LDRB    r0, [r14], #1           ; 取字符
        CMP     r0, #0                  ; 检查结束标志
        SWINE   SWI_WriteC              ; 如果没有结束,则打印…
        BNE     STROUT                  ; …并循环
        ADD     r14, #3                 ; 对准下一个字
        BIC     r14, #3
        MOV     pc, r14                 ; 返回
```

为结束执行用户程序,返回到监控程序:

```
        SWI     SWI_Exit                ; 返回监控
```

注意事项

1) 当处理器已经处于监控模式,只要原来的返回地址(在 r14_svc)和 SPSR_svc 已保存,就可以执行 SWI;否则当执行 SWI 时,这些寄存器将被覆盖。

2) 对 24 位立即数的解释依赖于系统。但大多数系统支持一个标准的子集用于字符输入输出及类似的基本功能。

   立即数可以指定为常数表达式,但是通常最好是在程序的开始处为所需要的调用声明名字(并设置它们的值),(或者导入一个文件,该文件为局部操作系统声明它们值)然后在代码中使用它们的名字。

   至于如何声明名字和设置它们的值,可参考第 3 章的"例题与练习"。

3) 在监控模式下执行的第 1 条指令位于 08_{16},一般是一条指向 SWI 处理程序的转移指令。而 SWI 处理程序则位于存储器内附近某处。不可能在 08_{16} 处开始写 SWI 处理程序,因为存储器中位于 $0C_{16}$ 的下一个字正是取指中止处理程序的入口。

5.7　数据处理指令

ARM 数据处理指令用于修改寄存器中数据的值。支持的操作包括各种 32 位数据类型的算术运算和逻辑运算。一个操作数在到达 ALU 之前可以移位或变换,这样就能在 1 条指令中完成像移位又相加这样的操作。

乘法指令使用不同格式,因此将在下一节单独考虑。

二进制编码

数据处理指令的二进制编码如图 5.6 所示。

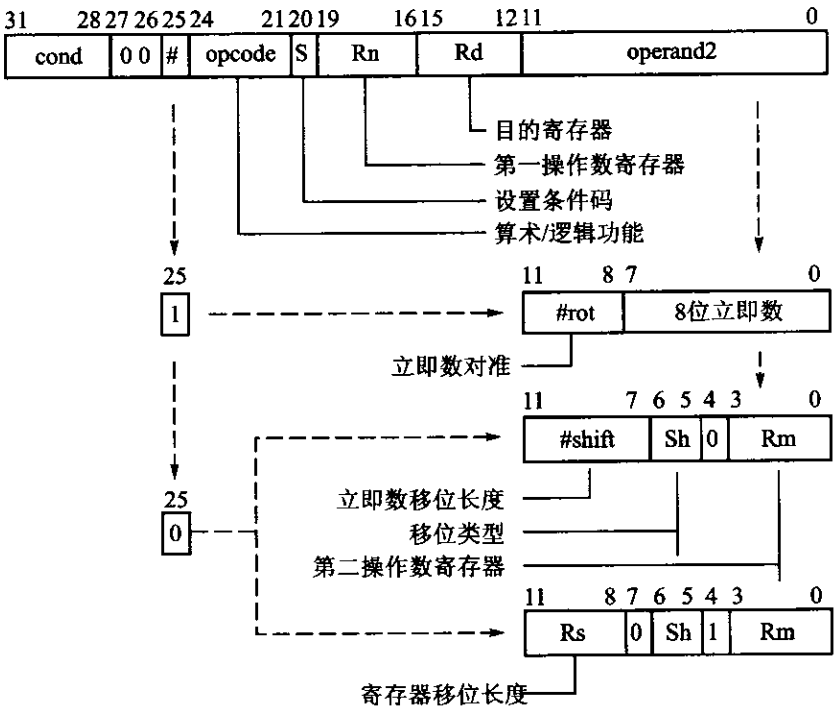


图 5.6 数据处理指令的二进制编码

说 明

ARM 数据处理指令使用 3 地址格式，这就意味着分别指定两个源操作数和一个目的寄存器。一个源操作数总是寄存器，第二个可能是寄存器、移位后的寄存器或立即数。第二操作数如果是寄存器，则应用于它的移位可能是逻辑或算术移位，或是循环移位(参见图 3.1)。移位的位数可以由立即数指定，也可以由第 4 寄存器指定。

可以指定的操作列表如表 5.4 所列。

表 5.4 ARM 数据操作指令

操作码.[24：21]	助记符	意 义	效 果
0000	AND	逻辑位与	Rd：=Rn AND Op2
0001	EOR	逻辑位异或	Rd：=Rn EOR Op2
0010	SUB	减	Rd：=Rn − Op2
0011	RSB	反向减	Rd：=Op2 − Rn
0100	ADD	加	Rd：=Rn + Op2
0101	ADC	带进位加	Rd：=Rn + Op2 + C
0110	SBC	带进位减	Rd：=Rn − Op2 + C − 1
0111	RSC	反向带进位减	Rd：=Op2 − Rn + C − 1

续表 5.4

操作码 [24：21]	助记符	意　义	效　果
1000	TST	测试	根据 Rn AND Op2 设置条件码
1001	TEQ	测试相等	根据 Rn EOR Op2 设置条件码
1010	CMP	比较	根据 Rn － Op2 设置条件码
1011	CMN	负数比较	根据 Rn ＋ Op2 设置条件码
1100	ORR	逻辑位或	Rd：＝Rn OR Op2
1101	MOV	传送	Rd：＝Op2
1110	BIC	位清 0	Rd：＝Rn AND NOT Op2
1111	MVN	求反	Rd：＝NOT Op2

当指令不需要全部的可用操作数时(例如，MOV 忽略 Rn，CMP 忽略 Rd)，不用的寄存器域应该设置为 0。汇编器将自动完成这项工作。

通过设置位 S(第 20 位)，这些指令可以直接控制处理器的条件码是否受指令执行的影响。当位 S 清 0 时，条件码不改变；当位 S 置位时(并且 Rd 不是 r15，见下面)，则有

- 如果结果为负，则标志位 N 置位；否则清 0(也就是说，N 等于结果的第 31 位)。
- 如果结果为 0，则标志位 Z 置位；否则清 0。
- 当操作定义为算术操作(ADD、ADC、SUB、SBC、RSB、RSC、CMP 或 CMN)时，标志位 C 设置为 ALU 的进位输出；否则设置为移位器的进位输出。如果不需要移位，则 C 保持。
- 在非算术的操作中，标志位 V 保持原值。在算术操作中，如果有从第 30 位到第 31 位的溢出，则置位；若不发生溢出，则清 0。仅当算术操作中操作数被认为是 2 的补码的有符号数时，这个标志位才有意义，而且指示结果超出范围。

乘以常数

用这些指令可以完成寄存器乘以小的常数，比使用下节描述的乘法指令更有效。例子在下面给出。

r15 的使用

PC 可以用做源操作数，但是使用寄存器来指定移位位数时除外。在这种情况下，3 个源操作数都不应是 r15。当 r15 用做源操作数时，提供的数值是指令的地址加 8 个字节。(8 字节的偏移显示了处理器的流水线操作，参看 4.1 节“PC 行为”一段。)

PC 还可以指定为存放结果的目的寄存器。在这种情况下，指令是某种形式的转移指令。这被用来作为一种由子程序返回的标准方法。

当指定 PC 为目的寄存器 Rd 时，位 S 仍然控制着指令对 CPSR 的作用，但却是以一种非常不同的方式。不再如前面所述的那样根据 ALU 的输出来更新标志位。如果设置了位 S，则将当前模式的 SPSR 拷贝到 CPSR。这可能影响中断使能标志位和处理器操作模式。这种机制自动恢复 PC 和 CPSR，是从异常返回的标准方式。因为在用户及系统模式没有 SPSR，在这些模式中不应该使用这种形式的指令。

汇编格式

汇编的表示法是下列格式中的一种，且当指令为一元指令(MOV、MVN)时省略 Rn，当指令为仅产生条件码输出的比较指令(CMP、CMN、TST 时 TEQ)时省略 Rd。

<op> {<cond>} {S} Rd, Rn, #<32 位立即数>
<op> {<cond>} {S} Rd, Rn, Rm, {<shift>}

这里"<shift>"指定移位类型(LSL、LSR、ASL、ASR、ROR 或 RRX)和移位位数。移位位数可以是 5 位立即数(#<# shift>)或寄存器(Rs)。只有当移位的类型为 RRX 时才不需指定移位位数。

举 例

r1 加 r3，结果放在 r5：

```
        ADD     r5, r1, r3
```

r2 递减并检查是否为 0：

```
        SUBS    r2, r2, #1          ; r2 减 1，设置条件码
        BEQ     LABEL               ; 如果 r2 为 0 则转移
        …                           ; …否则继续向下
```

r0 乘以 5：

```
        ADD     r0, r0, r0, LSL #2
```

r0 乘 10 的子程序：

```
        MOV     r0, #3
        BL      TIMES10
        …
TIMES10 MOV     r0, r0, LSL #1      ; 乘以 2
        ADD     r0, r0, r0, LSL #2  ; 乘以 5
        MOV     pc, r14             ; 返回
```

将 r0、r1 中的 64 位整数加到 r2、r3 的 64 位整数上：

```
        ADDS    r2, r2, r0          ; 加低位，保存进位位
        ADC     r3, r3, r1          ; 加高位和进位位
```

注意事项

因为立即数域必须在 32 位指令的子集内编码，所以不能表示所有的 32 位立即数值。图 5.6 所示的二进制编码表明立即数的值是如何编码的。将 8 位立即数域循环移位来产生立即数的值，而循环移位的位数为一个偶数。

5.8　乘法指令

ARM 乘法指令完成两个寄存器中数据的乘法。两个 32 位二进制数相乘的结果是 64 位积。某些指令形式将整个结果存储到两个独立指定的寄存器中。这些指令仅用于某几个版本的处理器。另外一些指令形式仅将最低有效 32 位存放到一个寄存器中。

在所有情况下，都有乘法-累加的变型，将乘积连续相加成为总和，而且有符号和无符号操作数都能使用。对于有符号和无符号操作数，结果的最低有效 32 位是一样的。因此，对于只保留 32 位结果的乘法指令，不需要区分有符号数和无符号数两种指令格式。

二进制编码

乘法指令的二进制编码如图 5.7 所示。

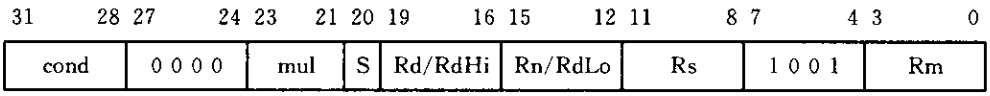


图 5.7　乘法指令的二进制编码

说　明

表 5.5 列出了各种形式乘法的功能，表中使用的符号如下：

- “RdHi：RdLo”是由 RdHi(最高有效 32 位)和 RdLo(最低有效 32 位)链接形成 64 位数，“[31：0]”只选取结果的最低有效 32 位。
- 简单的赋值由“：=”表示。
- 累加(将右边加到左边)是由“+=”表示。

同其他数据处理指令一样，位 S 控制条件码的设置。当在指令中设置了位 S 时，则

- 对于产生 32 结果的指令形式，将标志位 N 设置为 Rd 的第 31 位的值；对于产生长结果的指令形式，将其设置为 RdHi 的第 31 位的值。
- 如果 Rd 或 RdHi 和 RdLo 为 0，则标志位 Z 置位。
- 将标志位 C 设置为无意义的值。
- 标志位 V 不变。

表 5.5 乘法指令

操作码[23：21]	助记符	意 义	效 果
000	MUL	乘(32 位结果)	Rd：=(Rm * Rs)[31：0]
001	MLA	乘-累加(32 位结果)	Rd：=(Rm * Rs+Rn)[31：0]
100	UMULL	无符号数长乘	RdHi：RdLo：=Rm * Rs
101	UMLAL	无符号长乘-累加	RdHi：RdLo+=Rm * Rs
110	SMULL	有符号数长乘	RdHi：RdLo：=Rm * Rs
111	SMLAL	有符号数长乘-累加	RdHi：RdLo+=Rm * Rs

汇编格式

产生最低有效 32 位乘积的指令：

MUL{<cond>}{S} Rd，Rm，Rs
MLA{<cond>}{S} Rd，Rm，Rs，Rn

下列指令产生全部 64 位结果：

<mul>{<cond>}{S} RdHi，RdLo，Rm，Rs

在此<mul>是 64 位乘法类型(UMULL、UMLAL、SMULL、SMLAL)。

举 例

形成两个向量的标量积：

```
        MOV     r11，#20            ；初始化循环计数
        MOV     r10，#0             ；初始化总和
LOOP    LDR     r0，[r8]，#4        ；读取第一分量
        LDR     r1，[r9]，#4        ；…第二分量
        MLA     r10，r0，r1，r10    ；乘积累加
        SUBS    r11，r11，#1        ；减循环计数
        BNE     LOOP
```

注意事项

1) 应该避免 r15 定义为任一操作数或结果寄存器，否则，将产生不可预知的结果。
2) Rd、RdHi 和 RdLo 不能与 Rm 为同一寄存器，RdHi 和 RdLo 不能为同一寄存器。
3) 早期的 ARM 处理器仅支持 32 位乘法指令(MUL 和 MLA)。64 位乘法仅在名字中具有“M”的 ARM7 版本(ARM7DM、ARM7TM 等)和后续的处理器中使用。

5.9　前导 0 计数(CLZ——仅用于 v5T 体系结构)

这条指令仅用于支持 v5T 体系的 ARM 处理器。它对于数字归一化是有用的，比起使用其他 ARM 指令，它能更有效地实现其功能。

二进制编码

前导 0 计数指令的二进制编码如图 5.8 所示。

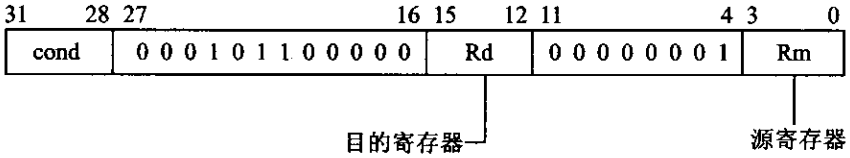


图 5.8　前导 0 计数指令的二进制编码

说　明

本指令将 Rd 设置为 Rm 中为 1 的最高有效位的位置数。如果 Rm 为 0，则 Rd 设置为 32。

汇编格式

CLZ {<cond>} Rd, Rm

举　例

```
MOV     r0, #&100
CLZ     r1, r0                  ; r1 := 23
```

注意事项

只有实现 ARM v5T 体系结构的处理器才支持 CLZ 指令(参见 5.23 节)。

5.10　单字和无符号字节的数据传送指令

这些指令是 ARM 在寄存器和存储器间传送单个字节和字的最灵活方式。通常大块数据的传送最好使用多寄存器传送指令。最近 ARM 处理器也支持传送半字和有符号字节的指令。

只要寄存器已被初始化并指向接近(通常在 4 KB 内)所需的存储器地址的某处，这些指令就可提供有效的存储器 Load/Store 机制。此机制有比较丰富的寻址模式，包括立即数和寄存器偏移、自动变址和相对 PC 的寻址。

二进制编码

单字和无符号字节数据传送指令的二进制编码如图 5.9 所示。

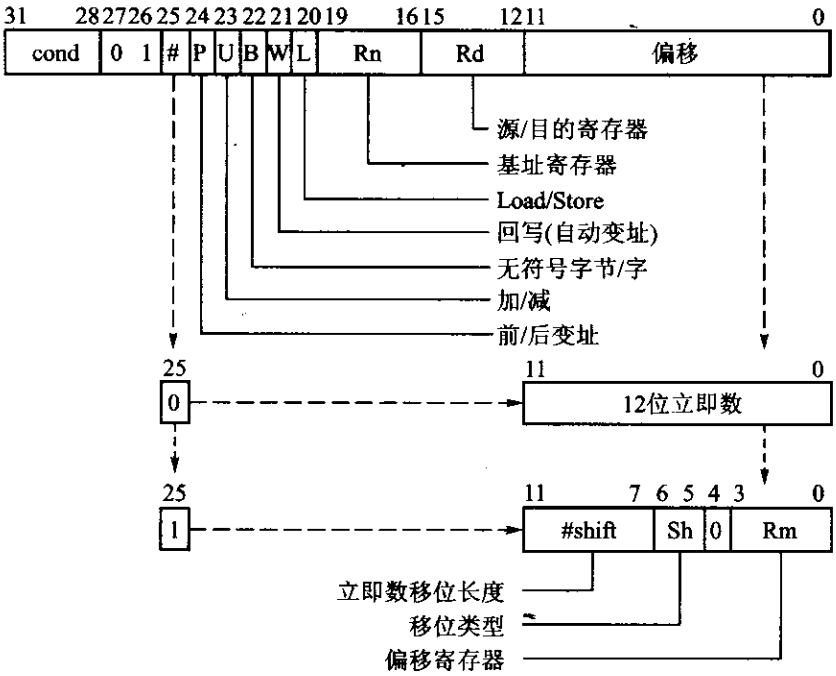


图 5.9 单字和无符号字节数据传送指令的二进制编码

说 明

这些指令构造一个地址，它从基址寄存器(Rn)开始，然后加上(U=1)或减去(U=0)一个无符号立即数或(可能是缩放的)寄存器偏移量。基址或计算出的地址用于从存储器(Rd)读取(L=1)一个无符号字节(B=1)或字(B=0)，或者向存储器(Rd)存储(L=0)一个无符号字节(B=1)或字(B=0)。当一个字节加载到寄存器时，它会 0 扩展到 32 位。当将一个字节存储到存储器时，寄存器的低 8 位写到地址指向的位置。

一种前变址(P=1)的寻址模式使用计算出的地址进行传送操作。然后当要求回写时(W=1)，将基址寄存器更新为计算出的值。

一种后变址(P=0)的寻址模式用未修改的基址寄存器来传送数据，然后将基址寄存器更新为计算出的地址，而不管位 W 如何(因为偏移除了作为基址寄存器的修改量之外已没有其他意义。如果希望不变化，则可将偏移设置为立即数 0)。由于在这种情况下位 W 是不使用的，所以它有一个替换功能(该功能与不运行在用户模式的代码有关)：设置 W=1，使处理器要求用户模式访问存储器，这样使操作系统采用用户角度(user view)来看待存储器变换和保护方案。

汇编格式

前变址的指令形式如下：

LDR|STR {<cond>} {B} Rd, [Rn, <offset>] {!}

后变址的指令形式如下：

LDR|STR {<cond>} {B} {T} Rd, [Rn],<offset>

一种有用的相对 PC 的形式(由汇编器计算所需立即数)：

LDR|STR {<cond>} {B} Rd, LABEL

LDR 是从存储器加载到寄存器，STR 是将寄存器存储到存储器；选择项 B 用于选择无符号字节传送，缺省为字；<offeset>可能是"#±<12 位立即数>"或"±Rm{, shift}"，在此移位指示符与数据处理指令的用法相同，除非寄存器指定的移位量不存在；"!"在前变址寻址的方式下选择是否回写(自动变址)。

标志位 T 用于选择用户角度的存储器变换和保护系统，它只能在非用户模式下使用。用户应当全面理解处理器工作的存储器管理环境，而这对于操作系统的专家而言实际上只是一个技巧。

举　例

将 r0 中的一个字节存到外设：

```
        LDR     r1, UARTADD          ; 将 UART 地址装入 r1 中
        STRB    r0, [r1]             ; 将数据存到 UART 中
        ...
UARTADD &       &1000000             ; 地址字符
```

汇编器将使用前变址的 PC 相对寻址模式将地址装入 r1。要做到这一点，字符必须限定在一定的范围内(即 Load 指令附近 4 KB 范围之内)。

注意事项

1) 使用 PC 作为基址时得到的传送地址为指令地址加 8 字节。它不能用做偏移寄存器，也不能用于任何自动变址寻址模式(包括任何后变址模式)。
2) 把一个字加载到 PC 将使程序转移到所加载的地址，这是一个公认的实现跳转表的方法。应当避免将一个字节加载到 PC。
3) 把 PC 存到存储器的操作在不同体系结构的处理器中产生不同的结果，因此应尽可能避免。
4) 在一般情况下，Rd、Rn 和 Rm 应当是不同的寄存器，尽管加载基址寄存器(Rd=Rn)是可以接受的，只要同一指令中没有使用自动变址。
5) 当从非字对准的地址读取一个字时，所读取的数据是包含所寻址字节的字对准的字。通过循环移位使寻址字节处于目的寄存器最低有效字节。对于这些情

况(由 CP15 寄存器 1 中第 1 位的标志位 A 控制,参见 11.2 节),一些 ARM 可能产生异常。

6) 当将一个字存入到非字对准的地址时,地址的低两位被忽略。当存入这个字时,把这两位当做 0。对于这些情况(也是由 CP15 寄存器 1 中的标志位 A 控制),一些 ARM 系统可能产生异常。

5.11 半字和有符号字节的数据传送指令

这些指令不为某些早期的 ARM 处理器支持。后来把它们加入到体系结构中。结果,正如分开的立即数域所显示的那样,它们成为某种硬塞进指令空间的东西。

这些指令使用的寻址模式是无符号字节和字的指令所用寻址模式的子集。

二进制编码

半字和有符号字节数据传送指令的二进制编码如图 5.10 所示。

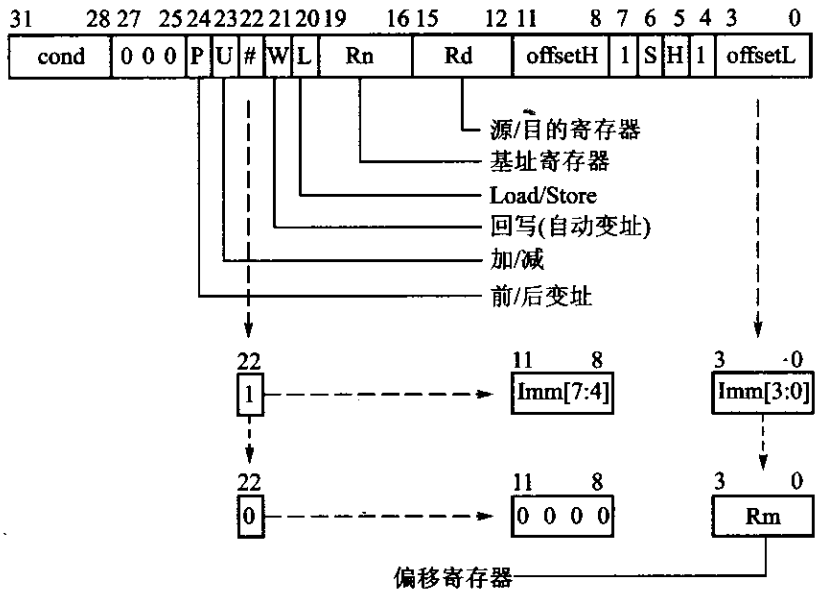


图 5.10 半字和有符号字节数据传送指令的二进制编码

说 明

这些指令与 5.10 节叙述的字和无符号字节的指令形式类似,但这里的立即数偏移限定在 8 位,也不再使用可缩放(scaled)的寄存器偏移。

位 S 和位 H 定义所传送的操作数的类型,如表 5.6 所列。注意,这些位的第 4 种组合在这种格式中没有使用,它对应于无符号字节的数据类型。无符号字节的传送应当使用前一节讲述的格式。因为在存储有符号数据和无符号数据之间没有差别,这条指令惟一的相关形式如下:

表 5.6　数据类型编码

S	H	数据类型
1	0	有符号字节
0	1	无符号半字
1	1	有符号半字

- 加载有符号字节、有符号半字或无符号半字。
- 存储半字。

在加载无符号数时用 0 扩展到 32 位；有符号数则重复其最高有效位，将其符号扩展到 32 位。

汇编格式

前变址格式：

LDR|STR{ <cond>} H|SH|SB Rd, [Rn, <offest>]{!}

后变址格式：

LDR|STR {<cond>} H|SH|SB Rd, [Rn], <offest>

其中，<offset>是"#±<8 位立即数>"或"#±Rm"；H|SH|SB 用于选择数据类型。其他部分的汇编器格式与传送字和无符号字节相同。

举　例

把一个有符号半字阵列扩展到字阵列：

```
        ADR     r1, ARRAY1          ; 半字阵列开始
        ADR     r2, ARRAY2          ; 字阵列开始
        ADR     r3, ENDARR1         ; ARRAY1 的端点+2
LOOP    LDRSH   r0, [r1], #2        ; 取符号半字
        STR     r0, [r2], #4        ; 保存字
        CMP     r1, r3              ; 检查阵列是否结束
        BLT     LOOP                ; 如果没有结束，则循环
```

注意事项

1) 如同 5.10 节描述的传送字和无符号字节的情况，也有对使用 r15 和寄存器操作数的类似限制。

2) 所有的半字传送应当使用半字对准的地址。

5.12 多寄存器传送指令

ARM 多寄存器传送指令允许当前操作模式的 16 个可见寄存器的任意子集(或全部)从存储器加载或存储到存储器中。一种指令形式还允许操作系统 Load 或 Store 用户模式寄存器来保存或恢复用户处理状态。另一种形式允许从 SPSR 恢复 CPSR 作为从异常处理返回的一部分。

这些指令用于在进入过程或返回时保存或恢复工作寄存器,对高带宽存储器块拷贝程序特别有用。

二进制编码

多寄存器数据传送指令的二进制编码如图 5.11 所示。

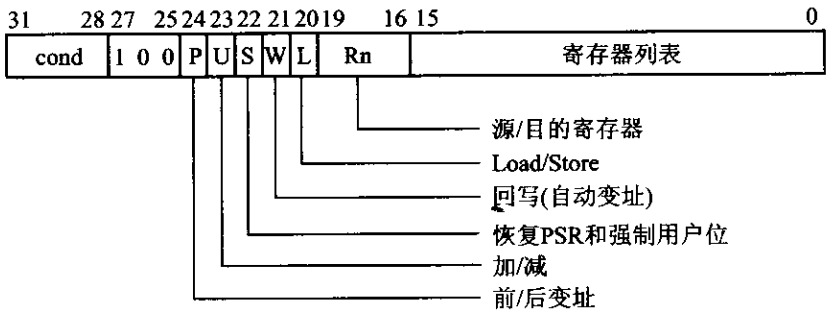


图 5.11 多寄存器数据传送指令的二进制编码

说 明

指令的低 16 位为寄存器列表,其中的每一位对应一个可见寄存器。第 0 位控制是否传送 r0,第 1 位控制 r1,依次类推到第 15 位控制传送 PC。

寄存器从存储器的连续字块加载或被存储到这些连续的字块。这些连续的字块由基址寄存器和寻址模式来定义。在传送每一个字之前(P=1)或之后(P=0),基址将增加(U=1)或减少(U=0)。支持自动变址。当指令完成时,如果 W=1,则基址寄存器将增加(U=1)或减少(U=0)所传送的字节数。

指令有一种特殊形式可以用来恢复 CPSR:如果 PC 是在多寄存器 Load 指令的寄存器列表中,而且位 S 置位,则当前模式的 SPSR 将被拷贝到 CPSR,成为一个原子的(atomic)返回和恢复状态指令。这种形式不能在用户模式的代码中使用,因为在用户模式下没有 SPSR。

如果 PC 不在寄存器列表中且位 S 置位,则在非用户模式执行的多寄存器 Load/Store 指令将传送用户模式下寄存器(虽然使用当前模式的基址寄存器)。这使得操作系统可以保存和恢复用户处理状态。

汇编格式

指令的一般形式如：

LDM|STM{<cond>}<add mode> Rn{!}, <registers>

这里<add mode>指定一种在表 3.1 中所述的寻址模式。指令中的各位同该表中的各项机械地对应，“增值(increment)”对应于 U=1，“先增”或“先减”对应于 P=1。“!”定义自动变址(W=1)，<registers>是寄存器列表，寄存器的范围括在花括弧内，例如：{r0, r3—r7, pc}。

在非用户模式下，CPSR 可以由下式恢复：

LDM {<cond>}<add mode> Rn !, <registers+pc>^

寄存器列表必须包含 PC。在非用户模式下，用户寄存器可以通过下式保存和恢复：

LDM|STM{<cond>}<add mode> Rn, <registers—pc>^

在这里寄存器列表不得包含 PC，并且不允许回写。

举　例

在进入子程序之前，保存 3 个工作寄存器和返回地址：

```
STMFD      r13!, {r0—r2, r14}
```

这里假设 r13 已被初始化用做堆栈指针。恢复工作寄存器和返回：

```
LDMFD      r13!, {r0—r2, pc}
```

注意事项

1) 如果在多寄存器 Store 指令的寄存器列表里指定了 PC，则保存的值与体系结构的实现方式有关。因此，一般应当避免在 STM 指令中指定 PC。(向 PC 加载会得到预期的结果，这是从过程返回的标准方法。)

2) 可以在多寄存器 Load/Store 指令的传送列表中指定基址寄存器，但不应在同一指令中指定回写，因为这样做的结果是不可预测的。

3) 如果基址寄存器包含的地址不是字对准的，则忽略最低两位。一些 ARM 系统可能产生异常。

4) 只有在 v5T 体系结构中，加载到 PC 的最低位才会更新 Thumb 位。

5.13　存储器和寄存器交换指令(SWP)

交换指令把字或无符号字节的 Load 和 Store 组合在一条指令中。通常都把这两种传送结合成为一个不能被外部存储器访问(例如来自 DMA 控制器的访问)分隔开的

基本的存储器操作，因此，该指令可以作为一种信号标机制的基础。这种机制可以对多处理器之间、处理器之间或处理器与 DMA 控制器之间共享的数据结构进行互斥的访问。这些指令的使用很少超出其在信号结构方面的作用。

二进制编码

存储器与寄存器交换指令的二进制编码如图 5.12 所示。

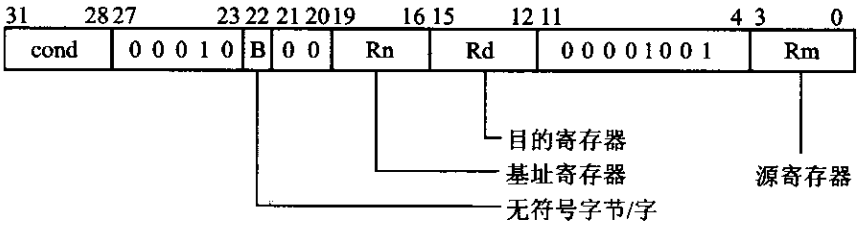


图 5.12 存储器与寄存器交换指令的二进制编码

说 明

本指令将存储器中地址为寄存器 Rn 处的字(B=0)或无符号字节(B=1)加载寄存器 Rd，又将 Rm 中同样类型的数据存入存储器中同样的地址。Rd 和 Rm 可以是同一寄存器(但两者应与 Rn 不同)。在这种情况下，寄存器与存储器中的值交换。ARM 对存储器的读写周期是分开的，但发出一个“锁”信号以向存储器系统指明两个周期不应分离。

汇编格式

SWP{<cond>} {B} Rd, Rm, [Rn]

举 例

```
ADR     r0, SEMAPHORE
SWPB    r1, r1, [r0]            ; 交换字节
```

注意事项

1) PC 不能用做指令中的任何寄存器。

2) 基址寄存器(Rn)不应与源寄存器(Rm)或目的寄存器(Rd)相同。

5.14 状态寄存器到通用寄存器的传送指令

当需要保存或修改当前模式下 CPSR 或 SPSR 的内容时，首先必须将这些内容传送到一般寄存器中，对选择的位进行修改，然后将数据回写到状态寄存器。本节讲述的指令完成这一过程的第一步。

二进制编码

状态寄存器向通用寄存器传送指令的二进制编码如图 5.13 所示。

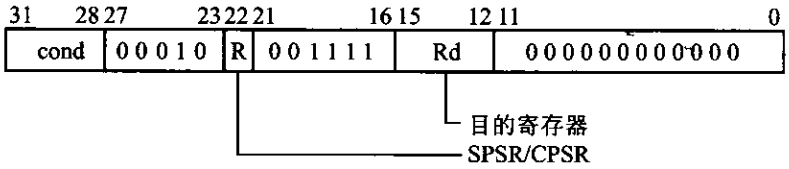


图 5.13　状态寄存器向通用寄存器传送指令的二进制编码

说　明

将 CPSR(R=0)或当前模式的 SPSR(R=1)拷贝到目的寄存器(Rd)，全部 32 位都被拷贝。

汇编格式

MRS{<cond>} Rd, CPSR|SPSR

举　例

```
MRS     r0, CPSR          ; 将 CPSR 传送到 r0
MRS     r3, SPSR          ; 将 SPSR 传送到 r3
```

注意事项

1) SPSR 形式不能用在用户或系统模式，因为在这些模式下没有可访问的 SPSR。

2) 当修改 CPSR 或 SPSR 时，必须注意保存所有未使用位的值。这将使这些位在将来使用时兼容的可能性最大。使用这些指令将状态寄存器传送到通用寄存器，只修改必要的位，再将结果传送回状态寄存器。这样做可以最好地完成对 CPSR 或 SPSR 的修改。

5.15　通用寄存器到状态寄存器的传送指令

当需要保存或修改当前模式下 CPSR 或 SPSR 的内容时，首先必须将这些内容传送到通用寄存器中，对选择的位进行修改，然后将数据回写到状态寄存器。这里讲述的指令完成这一过程的最后一步。

二进制编码

通用寄存器到状态寄存器传送指令的二进制编码如图 15.14 所示。

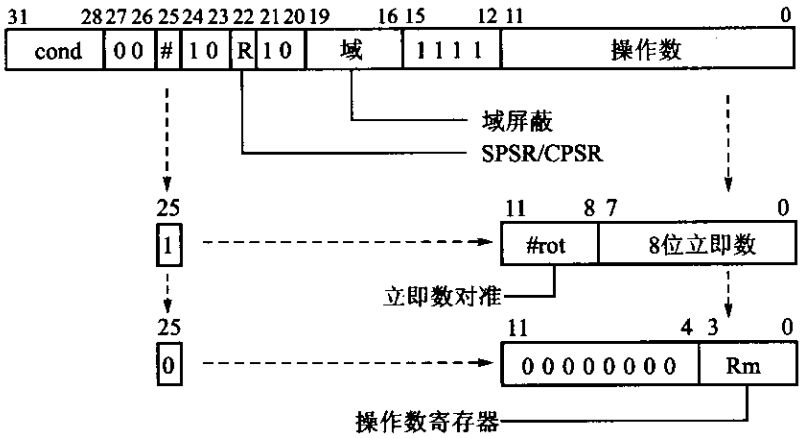


图 5.14 通用寄存器到状态寄存器传送指令的二进制编码

说 明

操作数，可能是一个寄存器(Rm)或循环移位的 8 位立即数(指定的方式与数据处理指令中第二操作数的立即数相同)，在域屏蔽(field mask)控制下被传送到 CPSR(R=0)或当前模式的 SPSR(R=1)。

域屏蔽控制 PSR 寄存器内 4 个字节的更新。指令的第 16 位决定 PSR[7：0]是否更新，第 17 位控制 PSR[15：8]，第 18 位控制 PSR[23：16]，第 19 位控制 PSR[31：24]。

当使用立即数操作数时，只有标志位(PSR[31：24])可以选择更新。(只有这些位可以由用户模式代码来更新。)

汇编格式

MSR{<cond>} CPSR_f | SPSR_f, #<32-bit immediate>

MSR{<cond>} CPSR_<field> | SPSR_<field>, Rm

这里<field>表示下列情况之一：

- c：控制域，PSR[7：0]。
- x：扩展域，PSR[15：8](在当前 ARM 中未使用)。
- s：状态域，PSR[23：16](在当前 ARM 中未使用)。
- f：标志位域，PSR[31：24]。

举 例

设置标志位 N、Z、C 和 V：

```
MSR     CPSR_f, #&f0000000      ; 设置所有的标志位
```

仅设置标志位 C，保存 N、Z 和 V：

```
MRS     r0, CPSR                ; 将 CPSR 传送到 r0
ORR     r0, r0, #&20000000      ; 设置 r0 的第 29 位
MSR     CPSR_f, r0              ; 传送回 CPSR
```

从监控模式切换到 IRQ 模式(例如,启动时初始化 IRQ 堆栈指针):

```
MRS     r0, CPSR                ; 将 CPSR 传送到 r0
BIC     r0, r0, #&1f            ; 低 5 位清 0
ORR     r0, r0, #&12            ; 将位设置为 IRQ 模式
MSR     CPSR_c, r0              ; 传送回 CPSR
```

在这种情况下,需要拷贝原来 CPSR 的值以便不改变中断使能设置。在例举的特殊情况下可以简化代码,因为从监控模式切换到 IRQ 模式(参见表 5.1)仅需要将 1 位清 0。但以上代码可以用来在任何两个非用户模式之间或从非用户模式到用户模式之间进行切换。

只有在 MSR 完成后,模式的改变才起作用。在将结果拷贝回 CPSR 之前,中间的工作对模式没有影响。

注意事项

1) 试图在用户模式下对 CPSR[23：0]进行任何修改都是无效的。
2) 尽量避免在用户或系统模式下访问 SPSR,因为在这些模式下没有 SPSR,访问会产生无法预想的结果。

5.16　协处理器指令

ARM 体系结构支持通过增加协处理器来扩展指令集的机制。最常使用的协处理器是用于控制片上功能的系统协处理器,例如控制 ARM720 上的 Cache 和存储器管理单元等。也开发了浮点 ARM 协处理器,还可以开发专用的协处理器。

协处理器寄存器

ARM 协处理器具有它们自己专用的寄存器组,它们的状态由控制 ARM 寄存器的指令的镜像指令来控制。

控制流指令由 ARM 负责处理,所以协处理器指令只同数据处理和数据传送有关。按照 RISC 的 Load/Store 体系原则,这些指令类别是被清楚区分的。指令的格式反映了这种情况。

协处理器数据操作

协处理器数据操作完全是协处理器内部的操作,它完成协处理器寄存器的状态改变。一个例子是浮点加法。在浮点协处理器中两个寄存器相加,结果放在第 3 个寄存器。

协处理器数据传送

协处理器数据传送指令从存储器读取数据装入协处理器寄存器,或将协处理器寄存器的数据存入存储器。因为协处理器可以支持它自己的数据类型,所以每个寄存器传送的字数与协处理器有关。ARM 产生存储器地址,但协处理器控制传送的字数。

协处理器可能执行一些类型转换作为传送的一部分(例如浮点协处理器将所有加载的值转换成它的 80 位内部表示形式)。

协处理器寄存器传送

除了以上情况,在 ARM 和协处理器寄存器之间传送数据有时是有用的。再以使用浮点协处理器为例,FIX 指令从协处理器寄存器取得浮点数据,将它转换为整数,并将整数传送到 ARM 寄存器中。经常需要用浮点比较产生的结果来影响控制流,因此,比较的结果必须传送到 ARM 的 CPSR。

这些指令合起来即可支持 ARM 指令集的扩展,以支持专用的数据类型和功能。

5.17 协处理器的数据操作

这些指令用于控制数据在协处理器寄存器内部的操作。标准格式遵循 ARM 整数数据处理指令的 3 地址形式,但是对所有协处理器域(field)可能会有其他的解释。

二进制编码

协处理器数据处理指令的二进制编码如图 5.15 所示。

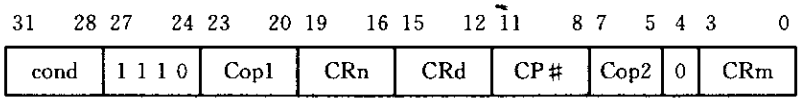


图 5.15 协处理器数据处理指令的二进制编码

说 明

ARM 对可能存在的任何协处理器提供这条指令。如果它被一个协处理器接受,则 ARM 继续执行下一条指令;如果它没有被接受,则 ARM 将产生未定义指令的陷阱(可以用来实现"协处理器丢失"的软件仿真)。

通常,与协处理器编号"CP#"一致的协处理器将接受指令,执行由 Cop1 和 Cop2 域定义的操作,使用 CRn 和 CRm 作为源操作数,并将结果放到 CRd。

汇编格式

CDP{<cond>} <CP#>, <Cop1>, CRd, CRn, CRm{, <Cop2>}

举 例

```
CDP     p2, 3, C0, C1, C2
CDPEQ   p3, 6, C1, C5, C7, 4
```

注意事项

对 Cop1、Crn、CRd、Cop2 和 CRm 域的解释与协处理器有关。以上的解释是推荐

的用法，它最大程度地与 ARM 开发工具兼容。

5.18　协处理器的数据传送

协处理器数据传送指令类似于前面讲述的字和无符号字节数据传送指令的立即数偏移格式，但偏移量限于 8 位而不是 12 位。

可使用自动变址，以及前变址和后变址寻址。

二进制编码

协处理器数据传送指令的二进制编码如图 5.16 所示。

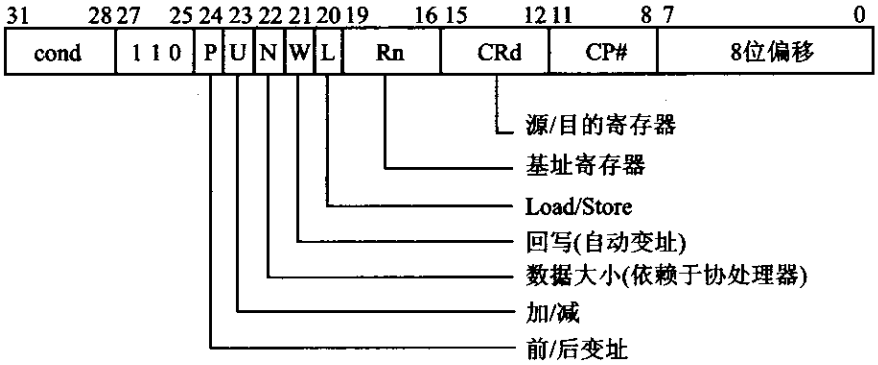


图 5.16　协处理器数据传送指令的二进制编码

说　明

该指令可用于任何可能存在的协处理器。如果没有协处理器接受它，则 ARM 将产生未定义指令陷阱，可以使用软件仿真协处理器。一般情况下，具有协处理器编号“CP#”的协处理器（如果存在）将接受这条指令。

地址计算将在 ARM 内进行。使用 ARM 基址寄存器（Rn）和 8 位立即数偏移量进行计算。8 位立即数偏移应左移两位产生字偏移。寻址模式和自动变址则以与 ARM 字和无符号字节传送指令相同的方式来控制。这样定义了第一个传送地址，随后的字则传送到递增的字地址或从递增的字地址读取。

数据由协处理器寄存器（CRd）提供或由协处理器寄存器接收，由协处理器来控制传送的字数，位 N 从两种可能的长度中选择一种。

汇编格式

前变址的格式如下：

LDC|STC{<cond>}{L} <CP#>, CRd, [Rn, <offset>]{!}

后变址的格式如下：

LDC|STC{<cond>}{L} <CP#>, CRd, [Rn,<offset>]

在这两种情况下,LDC 选择从存储器中读取数据装入协处理器寄存器,STC 选择将协处理器寄存器的数据存储到存储器。标志 L 如果存在,则选择长数据类型(N=1)。<offset>是"#±<8 位立即数>"。

举 例

```
LDC     p6, C0, [r1]
STCEQL  p5, C1, [r0], #4
```

注意事项

1) 对 N 和 CRd 域的解释与协处理器有关。以上用法是推荐的用法,且最大限度地与 ARM 开发工具兼容。
2) 如果地址不是字对准的,则最低两位有效位将被忽略,但是一些 ARM 系统可能产生异常。
3) 字的传送数目由协处理器控制。ARM 将连续产生后续地址,直到协处理器指示传送应该结束(参看 4.5 节中"数据传送"一段)。在数据传送过程中,ARM 将不响应中断请求,所以,协处理器设计者应该注意不应因为传送非常长的数据而损害系统的中断响应时间。

将最大传送长度限制到 16 个字将确保协处理器数据传送的时间不会长于多寄存器 Load/Store 指令的最坏情况。

5.19 协处理器的寄存器传送

这些指令使得协处理器中产生的整数能直接被传送到 ARM 寄存器和 ARM 条件码标志位。

典型的使用如下:

- 浮点 FIX 操作:它把整数返回到 ARM 的一个寄存器。
- 浮点比较:它把比较的结果直接返回到 ARM 条件码标志位。此标志位将确定控制流。
- FLOAT 操作:它从 ARM 寄存器中取得一个整数,并传送给协处理器,在那里整数被转换成浮点表示并装入协处理器寄存器。

在一些较复杂的 ARM CPU(中央处理单元)中,常使用系统控制协处理器来控制 Cache 和存储器管理功能。这类协处理器一般使用这些指令来访问和修改片上的控制寄存器。

二进制编码

协处理器寄存器传输指令的二进制编码如图 5.17 所示。

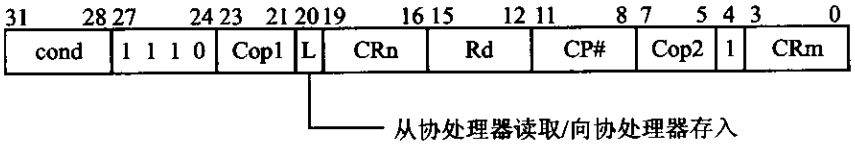


图 5.17　协处理器寄存器传送指令的二进制编码

说　明

本指令可用于任何可能存在的协处理器。通常，具有协处理器编号“CP#”的协处理器将接受这条指令。如果没有一个协处理器接受这条指令，ARM 将产生未定义指令的陷阱。

如果协处理器接受了从协处理器中读取数据的指令，那么一般它将执行由 Cop1 和 Cop2 定义的、对于源操作数 CRn 和 CRm 的操作，并将 32 位整数结果返回到 ARM，ARM 再把它装入 CRd。

如果协处理器接受了向协处理器存入数据的指令，那么它将接受一个来自 ARM 寄存器 Rd 的 32 位整数，并对它进行一些操作。

如果在从协处理器读取数据的指令中将 PC 定义为目的寄存器 Rd，则由协处理器产生 32 位整数的最高 4 位将被放在 CPSR 中的标志位 N、Z、C 和 V 中。

汇编格式

从协处理器传送到 ARM 寄存器：

MRC{<cond>} <CP#>, <Cop1>, Rd, CRn, CRm{, <Cop2>}

从 ARM 寄存器传送到协处理器：

MCR{<cond>} <CP#>, <Cop1>, Rd, CRn, CRm{, <Cop2>}

举　例

```
MCR     p14, 3, r0, C1, C2
MRCCS   p2, 4, r3, C3, C4, 6
```

注意事项

1) Cop1、CRn、Cop2 和 CRm 域由协处理器解释，推荐使用以上解释以最大限度同 ARM 开发工具兼容。

2) 若协处理器必须完成一些内部工作来准备一个 32 位数据向 ARM 传送（例如，浮点 FIX 操作必须将浮点值转换为等效的定点值），那么这些工作必须在协处理器提交传送前进行。因此，在准备数据时经常需要协处理器握手信号处于“忙-等待”状态。ARM 可以在忙-等待时间内产生中断。如果它确实得以中断，那么它将暂停握手以服务中断。当它从中断服务程序返回时，将可能重试协处理器指令，但也可能不重试。例如，中断可使任务切换。在任一情况下，协

处理器必须给出一致的结果，因此，在握手提交阶段之前进行的准备工作不许改变处理器的可见状态。

3) 从 ARM 到协处理器的传送一般比较简单，因为任何数据转换工作都可以在传送完成后在协处理器中进行。

5.20 断点指令(BKPT——仅用于 v5T 体系结构)

断点指令用于软件调试。它使处理器停止执行正常指令而进入相应的调试程序。

二进制编码

断点指令的二进制编码如图 5.18 所示。

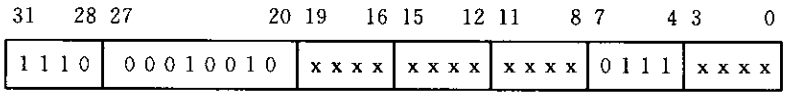


图 5.18 断点指令的二进制编码

说 明

当调试的硬件单元被适当配置时，本指令使处理器中止预取指。

汇编格式

BKPT

举 例

```
BKPT        ; !
```

注意事项

1) 仅实现 v5T 体系结构的微处理器支持 BKPT 指令(参见 5.23 节)。

2) BKPT 指令是无条件的——条件域必须包含“always”代码。

5.21 未使用的指令空间

并非全部 2^{32} 种指令位编码都指定了含义。迄今为止还未使用的编码可用于未来指令集的扩展。

每个未使用的指令编码都处于使用的编码所留下的特定间隙中，可以从它们所处的位置来推断它们未来可能的用途。

未使用的算术指令

这些指令看起来非常像 5.8 节描述的乘法指令。这将是一种可能的编码，例如对于整数除法指令就是这样。算术指令扩展空间如图 5.19 所示。

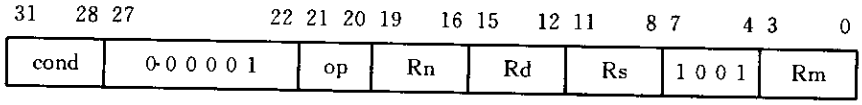


图 5.19　算术指令扩展空间

未使用的控制指令

这些指令包括 5.5 节描述的转移和交换指令，以及 5.14 和 5.15 节描述的状态寄存器传送指令。这里的间隙可以用于影响处理器操作模式的其他指令编码。控制指令扩展空间如图 5.20 所示。

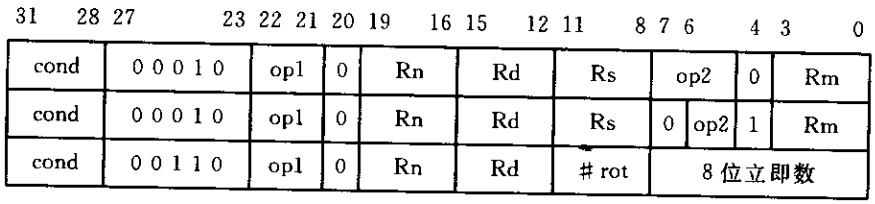


图 5.20　控制指令扩展空间

未使用的 Load/Store 指令

这些是由 5.13 节描述的 SWAP 指令以及 5.11 节描述的 Load 和 Store 半字和有符号字节指令占据的区域中未使用的编码。如果将来需要增加数据传送指令，就可以使用这些指令。数据传送指令扩展空间如图 5.21 所示。

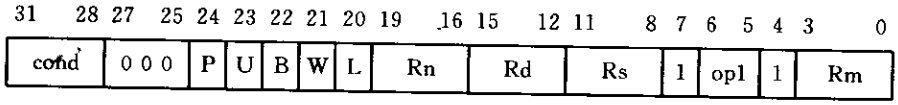


图 5.21　数据传送指令扩展空间

未使用的协处理器指令

下面的指令格式类似于 5.18 节描述的数据传送指令，可能用来支持所有可能需要增加的协处理器指令。协处理器指令扩展空间如图 5.22 所示。

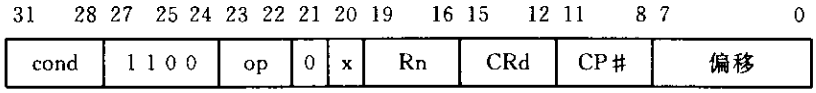


图 5.22 协处理器指令扩展空间

未定义的指令空间

最大未定义指令的区域看起来像 5.10 节描述的字和无符号字节数据传送指令，然而未来对于这一空间的选用完全保持开放。未定义的指令空间如图 5.23 所示。

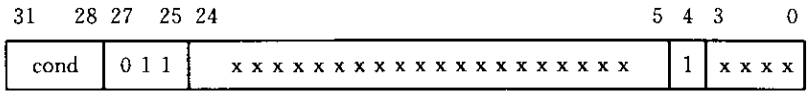


图 5.23 未定义的指令空间

未使用指令的行为

如果企图执行一条指令，它符合如图 5.23 所示的编码，即在未定义的指令空间，那么所有当前的 ARM 处理器将产生未定义指令的陷阱。

如果执行任何未使用的操作码，则最新的 ARM 处理器产生未定义指令的陷阱。但早先版本(包括 ARM6 和 ARM7)的行为无法预测，因此，应该避免这些指令。

5.22 存储器故障

ARM 处理器允许存储器系统(或者，更通常的说法是存储器管理单元)在任何存储器访问时失败。这就意味着不是从存储器中返回需要的值，而是存储器系统返回一个信号，指示存储器访问未能正确完成。处理器将进入异常处理，系统软件将试图从有问题的地方恢复。在通用机器中最普遍的存储器失败源有缺页、页保护和存储器软错误，分别解释如下：

缺 页

被寻址的存储器地址已分页到磁盘。

在虚拟存储器系统中，把很少使用的页保存在磁盘中。试图访问在这种页面上的指令或数据将会失败，使 MMU 中止访问。系统软件必须识别中止的原因，从磁盘中把所需的页读进存储器，据此修改 MMU 中的变换表，并重试被中止的访问。

由于从磁盘中读取一个页是一个缓慢的过程，操作系统常常跳出故障过程，在传送的同时安排另一个任务。

页保护

被寻址的存储器地址临时不可访问。

当将一个页面调入存储器中时，操作系统起初可能将它置为只读。试图写这个页面将出现故障，并警告操作系统：本页面已被修改，当再次将它交换到硬盘时必须保存。（如果仍有原拷贝，则不必将未修改的页面再次写到磁盘。）

一些操作系统间隔一段时间就将页面置为不可访问的，以便产生有关它们用于页面调度算法的统计。

存储器的软错误

在存储器中检测到软错误。

由于 α-粒子的辐射改变动态 RAM 存储单元的状态，会使大型存储器系统产生不可忽略的错误率。如果存储器系统只有简单的检错能力（诸如奇偶校验），则错误不可恢复，因此必须终止错误处理程序。如果存储器系统具有全面的检错和纠错（ECC）硬件，那么处理器一般将觉察不到错误，尽管还会产生故障以便操作系统能够累加对存储器故障率的统计。在中间情况下，存储器具有硬件检错，但依靠软件来纠正错误。这时将顺序执行故障、纠错和重新访问存储器。

嵌入式系统

在典型的小型嵌入式 ARM 应用中，通常不使用硬盘。在任何情况下，对磁盘划分页面通常与系统必须满足的实时性约束不相容。另外，存储器系统通常较小（最多几个兆字节，包括几块存储器芯片），所以，软错误率可以忽略不计，很少包含错误检测。因此，许多嵌入式系统根本不使用存储器故障。

在嵌入式系统中，典型的应用可能是将子程序库以压缩形式存入 ROM 中，使用虚拟存储技术捕获对单个子程序的调用，根据需要将其扩展到 RAM 中来执行。以压缩的方式保存的好处是可减小 ROM 的容量和降低其成本；代价是解压缩需要的时间加长。

在嵌入式系统中另一个用途可能是为在实时操作系统中运行的进程提供一些保护。

存储器故障

ARM 分别处理在取指过程中检测到的存储器故障（prefetch abort，预取指中止）和在数据传送过程中检测到的存储器故障（date abort，数据中止）。

预取指中止

如果发生取指故障，则存储器系统产生一个异常中止信号（一个给处理器的专用输入信号）并返回一个没有意义的指令字。在内部，ARM 将没有意义的指令字和中止标志一同放到指令流水线中，然后继续进行通常的操作，直到指令进入译码级为止。在那里中止标志取代指令，并使译码器使用预取指故障向量产生一个异常进入程序。

如果中止的指令没有得到执行，例如因为它是紧跟在转移指令后取指，而且转移指令最终执行了，那么这时将不产生异常，故障被忽略。

数据中止

对在访问存储器读取数据的过程中产生的存储器故障，处理起来要复杂得多。存储器系统不需要区分是读取指令还是读取数据，当它看到一个无法处理的地址时，只是产生一个中止的输入信号。但是，处理器在响应数据中止时更为困难，因为这是与当前正在执行的指令有关的问题，而预取中止是与还没有进入译码的指令有关的问题。

因为在某些情况下，在故障的原因解决后需要重新执行指令，指令应尽力确保它在中止后的状态(亦即寄存器中的值)与它开始执行前的状态相同。若做不到这一点，那么它应至少确保恢复足够的状态，以便指令在第二次执行后，它的状态与假如它第一次就被完全执行的情形相同。

LDM 数据中止

为了看一看这有多么困难，考虑一个多寄存器 Load 指令，其中寄存器列表中有 16 个寄存器，使用 r0 为基址寄存器。最初地址是好的，加载开始。加载的第一个数据覆盖 r0，以后的寄存器相继被覆盖，直到最后一个地址(预定为 PC)跨越页面边界并发生故障为止。多数的处理器状态已经丢失，处理器怎能恢复呢?

中止信号恰好能及时地防止 PC 被覆盖，所以，我们至少有一个用于这个指令的地址，而它就是造成故障的地址。看起来已经把基址寄存器丢失了很长时间，如何能去重新执行指令? 有幸的是，处理器在执行指令的同时，在一个黑角落保存了基址寄存器(可能是自动变址以后的)的一个拷贝。因此，假若访问 PC 没有发生故障，那么在应该将 PC 的值变为新值时，指令的最后一个动作就是把这个保存的值拷贝回基址寄存器。

我们保存了 PC 和(修改过的)基址寄存器，但是我们同时也覆盖了若干其他寄存器。对基址寄存器的修改可以用软件改回来，因为可以检查指令并确定寄存器列表中寄存器的数目及寻址模式。而被覆盖的寄存器正是重新执行指令时将用正确的数据再次置数的，所以万事大吉(恰好!)。

历史的记录

在开发第一片 ARM 处理器的过程中，很晚的时候才提出从中止的数据进行恢复的要求。直到这时，各种多寄存器 Load 和 Store 寻址模式都是根据模式从基址开始向上增加或向下减少存储器的地址。因此，芯片设计中使用了地址增/减单元。

当明显看出必须支持虚拟存储器时，人们很快看到递减模式使异常中止更难恢复，因为在中止被告知前，PC 可能被覆盖。因此，就将设计改变为总是增加地址。使用的存储器地址也是这样，寄存器到存储器的映射没有改变(见图 3.2)，只是传送的顺序改变为最低地址优先，最后是 PC。

这些变化实现得太晚，以致于没能影响地址产生逻辑的版图，因此，在第一个 ARM 硅片中地址增/减的硬连线总是使用增加。无需多说，这些冗余没有在后续的实现中遗传下来。

中止时序

处理器越早地从存储器系统得到故障的指示，就越容易保留状态；而处理器要求故障信号越早，存储器系统就越难设计。因此，在处理器失败处理的结构简化和存储器系统的工程效率之间有一种紧张关系。

这些紧张关系不影响存储器管理单元。如果处理器具有 Cache 存储器，则 Cache 的设计也会受到异常中止时序的影响。

早期的 ARM 处理器需要在时钟周期的第一相结束且在存储器访问周期的一半之前，(参见图 4.8)就得到故障信号。具有一个 CAM-RAM(CAM 是按内容访问存储器)组织的全相联 Cache 可以设计为仅从 CAM 访问中产生它的命中/未命中(hit/miss)信号，这样就能及时确认良好的访问，或在第一相停止处理器。一旦处理器停止，MMU 就有时间来控制产生中止信号。另一方面，组相联(set-associative)Cache 通常在时钟周期末产生它的命中/未命中信号，这太晚了，以致不能把未命中推迟到 MMU(它可能产生一个异常中止)而没有明显的性能损失。(ARM710 具有组相联 RAM-RAM Cache，它没有遵从这一规则，Cache 仍然在第一相结束之前产生它的命中/未命中信号和 MMU 的保护信息。)

为了更容易对 Cache 和 MMU 的设计进行约束，后期的 ARM 被重新设计为使中止在周期的结束处标记，与读数据具有同样的时序。必须接受的妥协是现在处理器状态改变得更多了，因此，在中止的恢复软件方面有许多工作要做。

一些 ARM 处理器可以配置(通过外部硬连线或使用位 L，CP15 寄存器 1 的第 6 位，可参见 11.2 节)为使用早期或晚期的中止时序来工作。

ARM 数据中止

在数据中止后 ARM 的状态与处理器型号有关。在一些采用早期/晚期的中止配置的处理器中，则有

- 在所有的情况下 PC 被保护(因此，在数据中止时，异常入口 r14_abt 的内容为故障指令的地址加 8 个字节)。
- 基址寄存器不会被修改，或者包含一个由自动变址修改的值(它不会被调入的值覆盖)。
- 其他目的寄存器可能被覆盖。当指令重执行时，正确的值将被装入。

因为基址寄存器可能被自动变址修改，所以应该避免使用某些(不是非常有用的)自动变址模式，例如：

```
LDR        r0, [r1], r1
```

这条指令使用 r1 作为 Load 地址，然后使用后变址将 r1 自身相加，在过程中丢失了最高位。如果跟着就是数据中止，只留下了 r1 的修改后的值，则不可能恢复原先的传送地址。一般来说，在寻址模式中应该避免使用同一寄存器作为基址和变址。

5.23 ARM 体系结构的各种版本

ARM 体系结构在其发展过程中经历了多次修订。对各种体系结构的版本介绍如下：

版本 1

ARM 体系结构版本 1 描述的是第一个 ARM 处理器，由 Acorn Computer 公司在 1983—1985 年间开发。第一批 ARM 芯片仅支持 26 位寻址，不支持乘法或协处理器。在附属于 BBC 微计算机的 ARM 第二个处理器中，它得到了惟一的应用。这种微计算机制造得很少，但是使 ARM 成为第一个商用单片 RISC 微处理器。它们也在 Acorn 内部用于 Archimedes(阿基米德)个人工作站的样机。

版本 2

ARM2 芯片在 Acorn 的 Archimedes 和 A3000 产品中批量销售。它仍然是 26 位地址的机器，但包含了对 32 位结果的乘法指令和协处理器的支持。ARM2 使用了 ARM 公司现在称为 ARM 体系结构版本 2 的体系结构。

版本 2a

ARM3 芯片是第一片具有片上 Cache 的 ARM。这一体系结构非常类似于版本 2，但是增加了合并的 Load 和 Store(SWP)指令，并引入了使用协处理器 15 作为系统控制协处理器来管理 Cache。

版本 3

ARM 公司成为独立的公司后，在 1990 年设计的第一个微处理器是 ARM6。它作为宏单元，作为独立的处理器(ARM60)和作为具有片上 Cache、MMU 和写缓冲(用于 Apple Newton 的 ARM600 和 ARM610)的集成 CPU 来出售。ARM6 引入 ARM 体系结构版本 3，它具有 32 位地址以及分开的 CPSR 和 SPSR，并增加了未定义和异常中止模式，以便在监控模式下支持协处理器仿真和虚拟存储器。

ARM 体系结构版本 3 向后同版本 2a 兼容，允许硬连线 26 位操作或在过程之间 26 位和 32 位混合操作。

版本 3G

ARM 体系结构 3G 是不与版本 2a 向后兼容的版本 3。

版本 3M

ARM 体系结构版本 3M 引入了有符号和无符号数乘法和乘-加指令。这些指令产生全部 64 位结果。

版本 4

体系结构版本 4 增加了有符号和无符号半字和有符号字节 Load 和 Store 指令，并为结构定义的操作预留了一些 SWI 空间。引入了系统模式(使用用户寄存器的特权模式)，将几个未使用指令空间的角落作为未定义指令使用。

在这一级，在早期 ARM 中用 r15 产生“PC+12”的用法将给出无法预测的结果(因此，体系结构版本 4 所依从的体系结构不需要复制“PC+12”的行为)。这是第一个具有全部正式定义的体系结构版本。

版本 4T

在体系结构版本 4T 中引入了 16 位 Thumb 压缩形式的指令集。

版本 5T

最近已经引入 ARM 体系结构版本 5T。在写本书时只有 ARM10 处理器支持它(很快也会支持 5TE 版本)。它是体系结构版本 4T 的扩展集，加入了 BLX、CLZ 和 BKPT 指令。

版本 5TE

版本 5TE 在体系结构版本 5T 的基础上增加了 8.9 节介绍的信号处理指令集。

总　结

表 5.7 总结了每个核使用的 ARM 体系结构的版本。

表 5.7　ARM 体系结构总结

核	体系结构
ARM1	v1
ARM2	v2
ARM2aS，ARM3	v2a
ARM6，ARM600，ARM610	v3
ARM7，ARM700，ARM710	v3
ARM7TDMI，ARM710T，ARM720T，ARM740T	v4T
StrongARM，ARM8，ARM810	v4
ARM9TDMI，ARM920T，ARM940T	v4T
ARM9E-S	v5TE
ARM10TDMI，ARM1020E	v5TE

5.24 例题与练习

(还可参阅 3.4 节和 3.5 节。)

例题 5.1

写一个标量乘法的程序,要明显地快于 5.8 节给出的程序。

在原程序中,每对数据花费 10 个时钟周期再加上(与数据相关的)乘的时间。每个数据的读入花费 3 个时钟周期,每次循环的转移花费 3 个时钟周期。

可以使用多寄存器 Load 指令来减少加载所用的时间,还可以用循环展开来减少转移所用的时间。将这两种技巧结合起来,得到如下程序:

```
        MOV     r11, #20            ; 初始化循环计数器
        MOV     r10, #0             ; 初始化总和
LOOP    LDMIA   r8!, {r0-r3}        ; 加载 4 个第 1 个向量值…
        LDMIA   r9!, {r4-r7}        ; …和 4 个第 2 个向量值
        MLA     r10, r0, r4, r10    ; 累加第 1 次乘积
        MLA     r10, r1, r5, r10    ; 累加第 2 次乘积
        MLA     r10, r2, r6, r10    ; 累加第 3 次乘积
        MLA     r10, r3, r7, r10    ; 累加第 4 次乘积
        SUBS    r11, r11, #4        ; 循环计数减 4
        BNE     LOOP                ; 如果没有结束,则循环
```

现在循环的开销为 16 个周期加上乘的时间,完成 4 对数值的累加,或者说每对数据用 4 个周期。

练习 5.1.1

编写一个子程序,从存储器某处拷贝一个字节串到存储器的另一处。源字节串的开始地址放入 r1,长度(以字节为单位)放入 r2 中,目的字节串的开始地址放入 r3。

练习 5.1.2

使用以上例题中示范的技巧重复前一个练习以改善性能。假设源和目的字节串是字对准的,而且字节串的长度是 16 字节的倍数。

练习 5.1.3

现在假设源字符串是字对准的,但目的字符串可以任意字节对准。字节串的长度仍是 16 字节的倍数。编写一个程序,一次处理 16 字节,使用多寄存器传送指令进行批量存储,使用字节存储指令处理两端的情形。

第6章　体系结构对高级语言的支持

本章内容综述

高级语言使程序能以抽象的方式来表述，例如数据类型、结构、进程、函数等等。有些指令集努力去支持这些高层级的概念，而 RISC 方法则表现为偏离这样的指令集。所以我们必须了解，虽然 RISC 指令集是更加基本的，它仍然能提供可以汇编的积木块，对高级语言给予必要的支持。

在本章中，我们将考察高级语言对体系结构的要求，以及怎样可以满足这些要求。我们将使用 C 作为高级语言的例子（尽管有些人对它充当这个角色的资格有异议），把 ARM 指令集作为这种语言的编译目标。

在分析过程中，我们会清楚地看到，RISC 体系结构（例如 ARM 使用的结构）很有味道，而且若干重要的决定可以由编译器的编写者根据口味来确定。其中一些选择将影响到是否容易由不同的源语言产生的例程来构成程序。因为这是一个重要的问题，所以规定了 ARM 进程调用标准。编译器的编写者应该使用这个标准以确保调用的入口与现存条件的一致性。

另一个从编译器的一致性受益的领域是对浮点运算的支持。它使用在 ARM 硬件指令集未定义的数据类型。

6.1　软件设计中的抽象

我们已介绍过对软件设计很重要的一个抽象级别——汇编级。ARM 处理器的精髓是在其指令集中，这已在第 5 章介绍过了。在第 9 章中我们将看到有很多方式实现 ARM 体系结构，但是体系结构的全部要点就是要确保程序员不必关心详细的实现细节。如果一个程序在一种实现中能正确地运行，它就应该在所有实现中正确地运行（带有一定的限制）。

汇编级的抽象

在汇编级上编写程序的程序员（几乎）直接用原始的机器指令集工作，用指令、地址、寄存器、字节和字来表述程序。

当优秀的程序员面对一个不寻常的任务时，一开始就会决定用能简化程序设计的

较高级别的抽象。例如，一个图形程序可能会画大量的线，所以，画一条已知端点坐标的线的子程序将是很有用的。余下的程序就可以仅仅是处理这些端点坐标。

在汇编编程级，抽象是很重要的。但是，支持抽象以及用机器的原始语言来表述抽象则取决于程序员。因此，程序员必须很好地理解这些原始语言，并准备经常后退到在机器语言的级别上思考。

高级语言

高级语言使程序员可以在比机器语言更高的抽象层次上思考。确实，程序员可能甚至不知道程序最终会在什么机器上运行。例如，寄存器号这样的参数对不同的体系结构是不同的，所以很明显，这些参数不能反映在语言的设计中。

用于高级语言的、在目标体系结构上的、支持抽象的任务落在编译器身上。编译器本身是一个非常复杂的软件，而编译器产生的代码的效率在相当大的范围内依赖于目标体系结构对编译器提供的支持。

在一个时期内公认的明智看法是，支持编译器的最好方式是增加指令集的复杂度，来直接实现语言的高级操作。RISC 原理的引入改变了这种方式，把指令集的设计集中于灵活的基本操作，编译器可以由这些基本操作来构造它的高层级操作。

本章介绍高级语言的要求，并表明基于 RISC 原理的 ARM 体系结构如何满足这些要求。

6.2 数据类型

有可能用基本的布尔逻辑变量“真”(1)和“伪”(0)来表述计算机程序，尽管不很方便。我们可以看到这是可能的，因为在门级全部都可以用硬件来操作。

在 ARM 指令集的定义中，在用指令、字节、字和地址等表述处理器的功能时，已经引入了脱离逻辑变量的抽象概念。每一个这类术语都描述了一个以独特方式观察的逻辑变量的集合。注意，例如指令、数据字和地址都是 32 位长，而存放它们的存储器单元不能区分存放的是何种类型。区别不在于信息存储的方式，而在于它们的使用方法。计算机的数据可以用下列各项来表征，即

- 它需要的位数；
- 这些位的顺序；
- 这些位的用途。

一些数据类型，例如地址和指令，主要是供计算机使用；而其他的数据则以人可以获取的方式表示信息。用计算的术语来说，后一类中最基本的是数。

数

数，估计在最初只是用于检查是否有羊在夜间被偷的简单概念。随着岁月流逝，已经发展为非常复杂的机制。它能计算只有(1/1 000) mm 宽，每秒钟开关 1 亿次的晶体管行为。

罗马数字

下面是人类书写的一个数字：

MCMXCV

十进制数

对**罗马**数字的解释是很复杂的。符号的值不仅依赖于符号，还依赖于符号相对于近邻符号的位置。书写数字的方式在很大程度上(但不是全部，见本书开头几页的页数)被**十进制**方案所取代。用这种方式，同一个数可表示如下：

1995

我们知道，最右面的数字代表单位数，它左边的一个数字代表十，然后是百、千等等。每向左移动 1 位，数字的值就增大 10 倍。

二进制编码的十进制数

为了用一组布尔变量来表示这样一个数字，最简单的是找到一种方式来表述每一个数字，再用 4 个这样的表述来表示整个数。我们需要 4 个布尔变量才能表示从 0 到 9 的不同数字，所以，第一种易于用逻辑门处理的数字形式如下：

0001 1001 1001 0101

这就是被一些计算机支持并通常用于小型计算器的二进制编码的十进制数。

二进制计数法

多数计算机在多数时间都摈弃人为的十进制计数方法，而统统偏爱纯二进制计数法。用二进制计数法同样的数变为

11111001011

这里，最右面的数字代表单位数，它左面的代表 2，然后是 4，等等。每向左移动 1 位，数字的值就增大 1 倍。因为某一列的数值 2 可以由左面一列的数值 1 来代表，所以，只需要 0 和 1 两个数字就可以表述任何数值，并且每个二进制数字都可以用单个布尔变量来表示。

十六进制计数法

机器内部广泛地使用二进制数，尽管典型的 32 位二进制数相当难记，但是也不愿把它转换成熟悉的十进制形式(它很难处理并且易出错)。计算机用户经常用十六进制(基于 16)的计数法来表示数。这是很容易的，因为二进制数可以划分为 4 个数字一组，把每一组替换成一个十六进制数字。因为在十六进制中需要表示数字 0～15 的符号，所以，当十进制的符号用完后，就使用了字母表里靠前面的几个字母：用 0～9 表示它们自己，而用 A～F 表示 10～15。数字变为

7CB

(在一个时期,流行用八进制(基于 8)计数法达到类似的作用。该方法避免了使用字母,但是与 4 个数字一组相比,3 个一组使用起来不太方便,所以,八进制的使用在很大程度上被舍弃了。)

数的范围

在纸上书写时,我们根据我们想要写的数的需要来选用十进制数字。计算机通常预留固定数目的位来表示数,所以,如果数目太大,就不能表示了。ARM 安排了 32 位来表示数量,所以该体系结构支持的第一种数据类型就是 32 位的(无符号)整数,它的数值范围为

$0\sim4\ 294\ 967\ 295_{10} = 0\sim\mathrm{FFFFFFFF}_{16}$

(下标表示数字的基数。上述数的范围首先用十进制表示,然后用十六进制。注意十六进制的 F 代表全"1"的二进制数。)

这看来是很大的范围,对多数用途来说也确实是足够的,但是程序员必须知道它的限制。将此范围内两个接近最大值的无符号数相加,将得到错误的结果,因为正确的答案已经不能在 32 位之内表示了。只有程序状态寄存器中的标志位 C 能给出指示,指出发生了错误,答案不可信。

如果从一个小数中减掉一个大数,结果将为负数,也不能用任何无符号整数来表示。

有符号整数

在很多情况下,既能表示负数又能表示正数是很有用的。ARM 支持 2 的补码的 2 进制计数法。这时最高位成为负数。在 32 位有符号数中,除了第 31 位外,所有位都有着与在无符号数中相同的数值,而第 31 位的值为 -2^{31},而不是 $+2^{31}$。现在,数的范围为

$-2\ 147\ 483\ 648_{10}\sim+2\ 147\ 483\ 647_{10}=80000000_{16}\sim7\mathrm{FFFFFFF}_{16}$

注意,数的符号只由第 31 位决定,正整数的数值和它对应的无符号数的表示相同。

ARM 与多数处理器一样,用 2 的补码计数法表示有符号整数。因为它们的加和减与无符号整数使用同样的布尔逻辑功能,所以不需要另外的指令(例外的情况是全 64 位结果的乘法,ARM 对有符号数和无符号数有分别的指令)。

体系结构对有符号数的支持是程序状态寄存器中的标志位 V,在操作数为无符号数时该标志位无作用。但是在有符号数参加运算时表示**溢出**(超范围)错误。源操作数不会超过范围,因为它们被表示为 32 位的数值。但是当两个接近范围的极端值的数相加或相减时,结果可能跑到范围之外。它的 32 位表述将是一个在范围之内的、数值和符号错误的数。

其他数的长度

在 ARM 中数的自然表述为有符号的和无符号的 32 位整数。实际上,在处理器内

部就是这样的。但是所有 ARM 处理器还执行 8 位的 Load 和 Store，使小的正数在存储器内占据比 32 位的字更小的空间。除了最早期的处理器以外，所有处理器还都支持有符号字节及有符号和无符号 16 位半字的传送，主要是为了减少存储小数值所需的存储器空间。

如果 32 位的整数太小，可以使用多字和多寄存器来操作更大的数。可以用两个 32 位的加法来执行一个 64 位的加法，使用状态寄存器中的标志位 C 从低字向高字传送进位，即

```
; 将[r1,r0]加到[r3,r2]的 64 位加法
ADDS    r2, r2, r0          ; 加低位，保存进位
ADC     r3, r3, r1          ; 加高位及进位
```

实　数

至此，我们仅仅考虑了整数。实数用来表示分数和超越(transcendental)数，在处理物理量时它们是很有用的。

实数在计算机中的表述是个大问题，推到下一节讲述。ARM 核不支持实数类型，尽管 ARM 公司定义了一系列有关实数的类型和指令。这些指令或者在浮点协处理器中执行，或者用软件来仿真。

可打印的字符

除了数以外，下一种最基本的数据类型是可打印的字符。为控制标准的打印机，需要一种方法来表示一般的字符，例如大写和小写的字母、1～9 的十进制数字、标点符号和若干特殊字符(如£、$、%等)。

ASCII 码

数一数这些不同的字符，总数很快就达到一百来个。某些时候之前，这些字符的二进制表述被标准化为 7 位 ASCII(American Standard for Computer Information Interchange，美国计算机信息交换标准)码，其中包括这些可打印字符和一些控制码。控制码的命名参照了这些码原来在电传打字机上的名字，例如“回车”、“换行”和“铃”等。

在计算机中存储 ASCII 字符的一般方法是将 7 位二进制代码放入 8 位的字节。许多系统扩展了这些代码，例如使用 8 位的 ISO 字符集，其中用字节所表示的另外 128 个代码表示特殊的字符(例如带重音号的字符)。表示字符最灵活的方法是 16 位的 Unicode，它包含很多单字节编码的 8 位字符集。

按照 8 位可打印字符编码，“1995”是：

$$00110001001110010011100100110101 = 31\ 39\ 39\ 35_{16}$$

ARM 对字符的支持

ARM 体系结构中支持字符操作的是无符号字节的 Load 和 Store 指令。前面已经提到过，这些指令可用来支持小的无符号整数。但是与它们频繁地用来传送 ASCII 字

符相比，前面讲到的作用就小得多了。

ARM 体系结构中没有任何东西反映 ASCII 定义的特殊编码。如果其他编码使用的位数不多于 8 位，那么也会得到同样好的支持。但是，目前如果没有极好的理由而选用其他代码来表示字符，将是荒谬的。

字节顺序

上面 ASCII 码的例子向我们提示了一个潜在的困难。它是从左向右书写以备读出的。但是如果作为 32 位的字来读，最低有效的字节在最右面。一个输出字符的例程可以顺序地按递增的字节地址打印字符。在这种情况下，用小端(little-endian)寻址，将打印出“5991”。显然需要小心对待存储器中一个字内字节的顺序。

高级语言

高级语言定义其规范中需要的数据类型，通常不会参照任何特定的体系结构。有时用于表示特定数据类型的位数依赖于体系结构，以便机器使用其最有效的宽度。

ANSI C 的基本数据类型

由美国国家标准学会(ANSI)定义的一种 C 语言，被称为“ANSI 标准 C”或简称“ANSI C”。它定义了以下基本数据类型：

- 至少 8 位的有符号和无符号的**字符**。
- 至少 16 位的有符号和无符号的**短整数**。
- 至少 16 位的有符号和无符号的**整数**。
- 至少 32 位的有符号和无符号的**长整数**。
- 浮点数，双精度和扩展双精度浮点数。
- **枚举**类型。
- 位域。

除了标准的整数外，ARM C 编译器接受以上各种数据类型的最小宽度。它使用 32 位整数，因为这是最经常使用的数据类型，而且与 16 位操作相比，ARM 能更有效地支持 32 位操作。

枚举类型(变量在指定的一系列数值中选取 1 个值)是用所需取值范围内最小的整数类型来实现的。位域型(布尔变量的集合)在整数内实现。可能几个变量共用一个整数，第一个声明的变量占据最低位，但不可跨越字的边界。

ANSI C 延伸出的数据类型

此外，ANSI C 标准还定义了延伸的数据类型：

- 由同一数据类型的几个对象组成的**数组**。
- 返回特定类型对象的**函数**。
- 包含一系列不同类型的对象的**结构**。
- 指向特定类型对象的**指针**(它通常为机器地址)。
- 允许不同类型对象在不同时间占据同一空间的**联合数据类型**。

ARM 的指针长度为 32 位(ARM 本身的地址宽度),类似于无符号整数,但是它遵从不同的算术规则。

ARM C 编译器把字符对准字节的边界(也就是说在下一个可用的地址),短整数在偶地址,所有其他类型在字的边界。结构总是在字的边界开始它的第一个分量,然后按照这些对准规则尽量紧凑地排列其他的分量。(压缩的(packed)结构违反对准规则。存储器的使用将在 6.9 节更广泛地讨论。)

ARM 体系结构对 C 数据类型的支持

上面我们已经看到,对有符号和无符号的 32 位整数以及无符号的字节,包括 C 整数(用 32 位数值来实现)、长整数和无符号的字符,ARM 整数核都提供自然的支持。指针是用 ARM 本身的地址实现的,因而正好可以支持。

ARM 的寻址模式对数组和结构提供了一定的支持。使用基址比例变址的寻址模式,可以搜索对象宽度为 2^n 字节的数组。搜索时用指针指向数组的开始处,将循环变量作为变址。使用基址加立即数偏移寻址模式,可以访问结构中的对象。然而,要执行更复杂的访问,必须有另外的地址计算指令。

ARM 的现行版本包括有符号字节以及有符号和无符号 16 位数据的 Load 和 Store,自然对短字节和有符号字符类型提供支持。

下一节将讨论浮点类型。然而我们在这里可以说,基本的 ARM 核几乎不直接支持它们。在没有专门支持浮点数的硬件的情况下,这种类型(以及处理它们的指令)是靠复杂的软件仿真程序来处理的。

6.3　浮点数据类型

浮点数试图以相同的精度表示实数。表示一个实数的一般方法如下:

$$R=a\times b^n \tag{13}$$

其中 n 的选取要使 a 落在一个定义的数值范围之内。b 通常隐含在数据类型中且常常等于 2。

IEEE 754

在计算机中操作浮点数,要确保在不同机器上运行同一个问题时结果的一致性,则有很多复杂的问题要解决。1985 年发布的关于二进制浮点数运算的 IEEE 标准(ANSI/IEEE标准 754-1985,有时简称为 IEEE 754)对解决一致性问题有极大的帮助。标准中相当详尽地定义了浮点数应如何表示,计算的精度应如何执行,误差应如何检测并返回等等。

根据 IEEE 754,浮点数最紧凑的表示法是 32 位单精度格式。

单精度

IEE 754 单精度浮点数格式如图 6.1 所示。

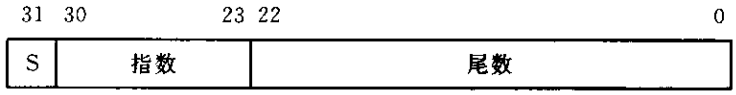


图 6.1 IEEE 754 单精度浮点数格式

数由 3 部分组成，即**符号位**（“S”）、**指数**和**尾数**。指数是有＋127 偏置（bias）（用于规格化）的无符号整数。对前一句话中的一些术语可能不熟悉。为了解释它们，让我们看一看“1995”这个我们认识的数是如何转换成这种格式的。

我们先由 1995 的二进制表示开始，它已经被表示为

11111001011

这是一个正数，所以位 S 为 0。

规格化的数

第一步是将数**规格化**，这就是将它转换为式（13）所示的格式，其中 $1 \leqslant a < 2$ 且 $b=2$。看一下该数的二进制形式，在第一个“1”后面插入一个二进制小数点（其意义类似于熟悉的十进制小数点）就可将 a 限制在这个范围之内。在二进制整数的表示中，二进制小数点的隐含位置就是最右一个数字的右面，所以，在此把它向左移动了 10 位。这样，1995 的规格化表述成为

$$1995 = 1.111100\uparrow 011 \times 2^{1010} \tag{14}$$

式中 a 和 n 都采用了二进制计数法。

任何数被规格化后，a 中二进制小数点前面的一位将为“1”（否则该数未被规格化）。因此，不需要存储这一位。

指数偏置

最后，由于需要格式表示很小的数和很大的数，所以有些数在其规格化格式中可能需要负的指数。标准没有使用有符号的指数，而是规定了偏置。将偏置（在单精度规格化数中为＋127）加到指数中。在此 1995 表示为如图 6.2 所示。

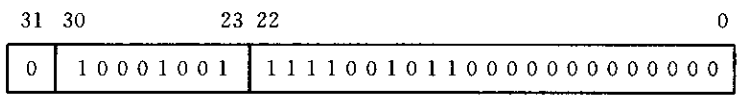


图 6.2 1995 的 IEEE 754 单精度表示

指数为 127＋10＝137；尾数的右面用 0 扩充以填满 23 位的域。

规格化数值

一般来说，32 位规格化数的值由下式给出：

$$\text{值(规格化)} = (-1)^S \times 1.\text{尾数} \times 2^{(\text{指数}-127)} \tag{15}$$

尽管这个格式有效地代表一个很广的数值范围，但它有一个相当明显的问题：无法表示 0。因而 IEEE 754 标准保留了指数为 0 或 255 的数，用来表示特定的数，即

- 用 0 指数和尾数来表示 0(符号为任何值,所以正 0 和负 0 都可以表示)。
- 用最大的指数和 0 尾数以及适当的符号位来表示正负无穷。
- 用最大的指数和非 0 的尾数表示 NaN(Not a Number, **非数**);"静"NaN 在最高有效尾数位的位置为"1",而"信号"NaN 在最高有效尾数位的位置为"0"(但是其他位应为"1",否则就与无穷一样了)。
- **非规格化数**,即一些仅仅因为太小以致于不能按此格式规格化的数。它们指数为 0,尾数为非 0,值由下式给出,即

$$\text{值(非规格化)} = (-1)^S \times 0.\text{尾数} \times 2^{(-126)} \tag{16}$$

NaN 格式用来表示浮点数无效操作的结果,例如取负数的对数;还用于防止在一系列操作中,当中间的错误条件未被检查出时产生表面上合法的结果。

双精度

在许多应用中单精度格式提供的精确度是不够的。使用双精度格式,即用 64 位存储每个浮点数,可以大大提高精确度。

对格式的解释与单精度数类似,但是这里规格化数的指数偏置为+1 023。IEEE 754 双精度浮点数格式如图 6.3 所示。

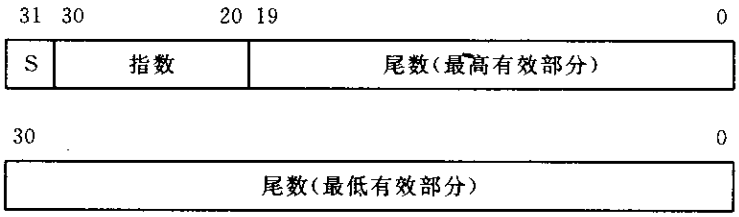


图 6.3　IEEE 754 双精度浮点数格式

双扩展精度

双扩展精度格式提供了更高的精确度。它使用扩展到 3 个字的 80 个信息位。指数偏值为 16 383,而位 J 是二进制小数点左面的一位(对于所有规格化数,该位为"1")。IEEE 754 双扩展精度浮点数格式如图 6.4 所示。

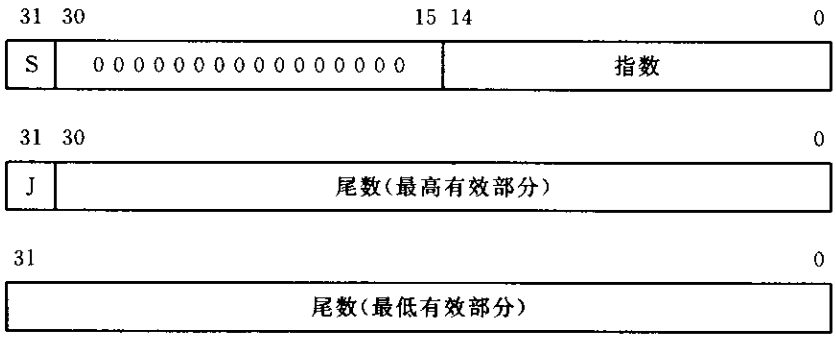


图 6.4　IEEE 754 双扩展精度浮点数格式

压缩的十进制数

除了上面所述的二进制浮点数表示法之外，IEEE 754 标准还规定了压缩(packed)的十进制格式。对于前面的式(13)，在压缩的十进制格式中，b 是 10，而 a 与 n 则以二进制编码的十进制格式存储。这种格式曾在 6.2 节“二进制编码的十进制数”一段中说明过。这种数在规格化时要使 $1 \leqslant a < 10$。压缩的十进制如图 6.5 所示。

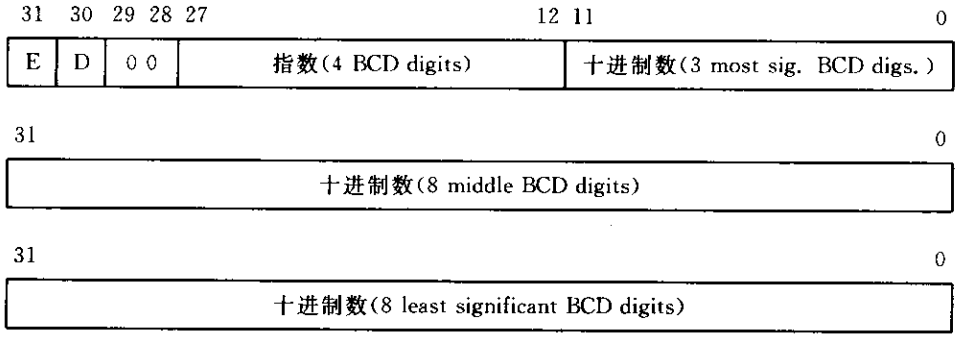


图 6.5 IEEE 754 压缩的十进制浮点数格式

指数的符号位于第一个字的第 31 位(“E”)，十进制数的符号在第 30 位(“D”)。数的值为

$$\text{值(压缩)} = (-1)^{D} \times \text{十进制数} \times 10^{((-1)^{E} \times \text{指数})} \tag{17}$$

扩展的压缩十进制数

扩展的压缩十进制数占据 4 个字以得到更高的精度。其数值依然如式(17)所示，其格式如图 6.6 所示。

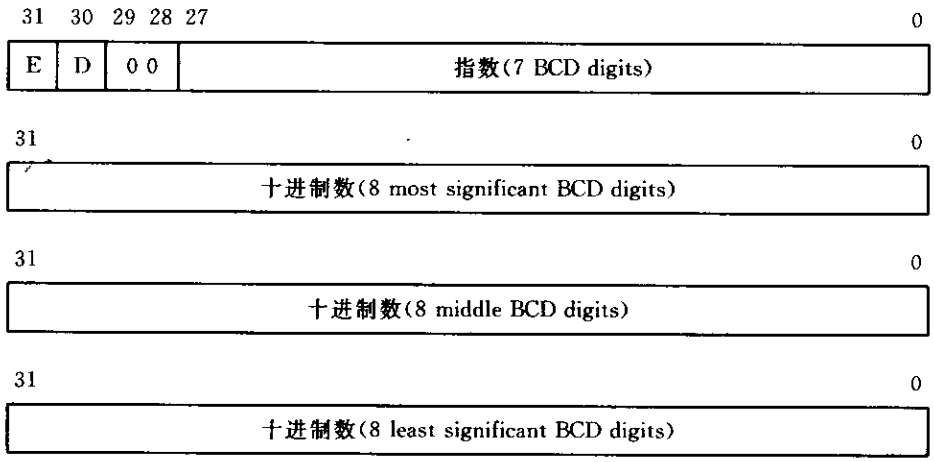


图 6.6 IEEE 754 扩展的压缩十进制浮点数格式

ARM 浮点指令

尽管在 ARM 整数核中没有直接支持任何浮点数据类型，但 ARM 公司在协处理器指令空间定义了一系列浮点指令。通常这些指令全部通过未定义指令陷阱（它收集所有硬件协处理器不接受的协处理器指令）在软件中实现，但是一个子集可由 FPA10 浮点协处理器以硬件操作。

ARM 浮点库

作为 ARM 浮点指令集（对于 Thumb 代码是惟一的选择）的替代方法，ARM 公司还提供了 C 浮点库。该库支持 IEEE 单精度和双精度格式。C 编译器有一个标志来选择这个例程。它产生的代码与软件仿真相比既快（通过避免中断、译码和浮点指令仿真）又紧凑（仅由于所使用的函数必须包括在映射中）。

6.4　ARM 浮点体系结构

对于需要充分浮点支持的情况，ARM 浮点体系结构对上节所述的各数据类型提供广泛的支持。这种支持或者是纯软件的，或者是使用基于 FPA10 浮点加速器的软件/硬件联合的解决方案。

ARM 浮点体系结构提供：

- 当协处理器号为 1 或 2 时（浮点系统使用两个逻辑协处理器号），提供对协处理器指令集的解释。
- 在协处理器 1 和 2 中的 8 个 80 位的浮点寄存器（同样的物理寄存器出现于两个逻辑协处理器中）。
- 用户可见的浮点状态寄存器（FPSR）。它控制各种操作选项及指示错误条件。
- 可选用的浮点控制寄存器（FPCR）。它是用户不可见的，并且只能由硬件加速器专用的支撑软件使用。

注意，ARM 协处理器体系结构使用户可以使用浮点仿真器（FPE）软件，也可以将 FPA10 与浮点加速器支持代码（FPASC）结合起来使用，或者将任何其他支持同样指令集的软硬件结合起来使用。二进制应用软件在各种支持环境下工作，尽管编译器的优化策略是不同的（FPE 软件适合于分组的浮点指令，而 FPA10/FPASC 适合于分布式指令）。

FPA10 数据类型

ARM FPA10 硬件浮点加速器支持单精度、双精度和扩展双精度格式。压缩的十进制格式仅由软件支持。

协处理器中的寄存器全都是扩展的双精度，除了某些指令的快速版本外，所有内部的计算也都是在这种最高的精度格式下进行的。这些快速指令不产生满 80 位的精确结果。但是，在存储器和这些寄存器之间的 Load 和 Store 可以按照需要来转换数据的

精度。

这类似于 ARM 的整数体系结构中的整数处理：所有的内部操作都是基于 32 位的数值，但是存储器传送可以指定为字节和半字。

Load 和 Store 浮点指令

因为只有 8 个浮点寄存器，在协处理器数据传送指令（见图 5.16）中的寄存器指示符（specifier）域就有一个多余的位。该位被用做附加的数据大小指示符。Load 和 Store 浮点数的二进制编码如图 6.7 所示。

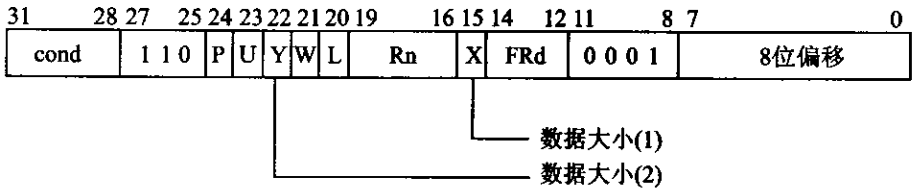


图 6.7 Load 和 Store 浮点数的二进制编码

该格式中的其他域已在 5.18 节中说明。X 和 Y 两位用于指定 4 种精度之一，即从单精度、双精度、扩展的双精度和压缩的十进制这 4 种精度中选择一种。（在压缩十进制和扩展压缩十进制之间如何选择，则由 SPSR 中的某一位控制。）

Load 和 Store 多浮点数

Load 和 Store 多浮点寄存器指令用于存储和恢复浮点寄存器的状态。每个寄存器用 3 个存储器字来存储，没有规定精确的格式。所存储数据的惟一作用就是使用相应的 Load 多浮点指令重新读取它们以恢复寄存器内容。FRd 用于指定第一个要传送的寄存器；X 和 Y 指定要传送的寄存器数目，可以是 1～4。注意，这些指令使用 2 号协处理器，而其他浮点数指令使用 1 号协处理器。Load 和 Store 多浮点数的二进制编码如图 6.8 所示。

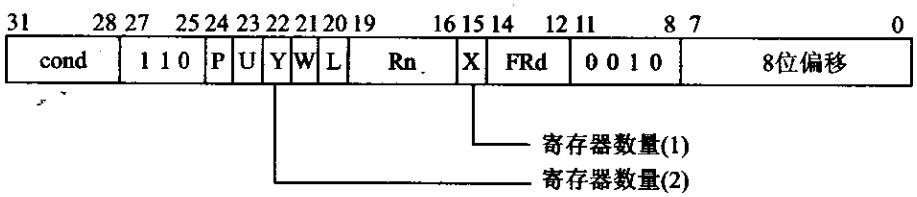


图 6.8 Load 和 Store 多浮点数的二进制编码

浮点数据的操作

浮点数据操作执行浮点寄存器中数据的运算功能。这些操作与外部世界惟一的交互就是通过 ARM 协处理器握手信号确认操作应当完成。（实际上，浮点协处理器只需要在完成状态改变之前等待来自 ARM 的确认，它可以在握手信号开始之前就开始执行一条这类指令。）浮点数据操作的二进制编码如图 6.9 所示。

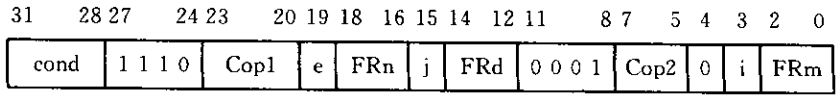


图 6.9　浮点数据操作的二进制编码

由于在 8 个浮点寄存器中指定 1 个只需要 3 位，因此，可以将指令格式中 3 个寄存器指示符域中多余的 3 位扩充为操作码位：

- i：选择第二操作数是寄存器（"FRm"）还是 8 个常数之一。
- e 和 Cop2：控制目标的大小和舍入模式。
- j：选择一元（单操作数）还是二元（双操作数）操作。

指令包括简单算术操作（加、减、乘、除、余数和幂）、超越函数（对数、指数、正弦、余弦、正切、反正弦、反余弦和反正切）以及其他多种操作（平方根、传送、绝对值和舍入）。

浮点寄存器传送

寄存器传送指令从 ARM 寄存器接收数据或向其返回数据。这一般都伴随着浮点数处理功能。浮点寄存器传送二进制编码如图 6.10 所示。

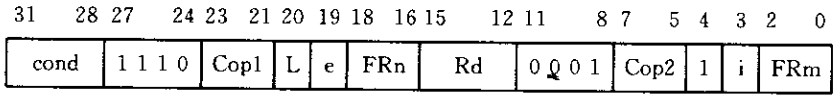


图 6.10　浮点寄存器传送二进制编码

从 ARM 向浮点单元的传送包括"浮动"（将 ARM 寄存器中的整数转换为浮点寄存器中的实数）以及向浮点状态和控制寄存器写入；反方向传送则包括"固定"（将浮点寄存器中的实数转换为 ARM 寄存器中的整数）以及读取状态和控制寄存器。

浮点比较指令是这类指令中的特例。该指令中的 Rd 是 r15。当两个浮点数比较时，比较的结果返回 ARM CPSR 中的标志位 N、Z、C 和 V。这些标志位可以直接控制条件指令的执行。

- N 表示小于。
- Z 表示相等。
- C 和 V 表示更复杂的条件，包括非正常的比较结果。当一个操作数是 NaN（非数）时，就会出现这个结果。

浮点指令的频率

FPA10 是按照各种浮点指令的典型使用频率设计的。这些频率是使用浮点仿真器软件运行编译过的程序来测定的，综述于表 6.1。

统计后发现，Load 和 Store 操作的数量占优势，因此，在 FPA10 的设计中，容许它们与内部算术指令并行操作。

表 6.1 浮点指令的频率

指 令	频率/%
Load/Store	67
加	13
乘	10.5
比较	3
定点和浮点	2
除	1.5
其他	3

FPA10 的组织

FPA10 的内部组织如图 6.11 所示。它的外部接口连接 ARM 的数据总线和协处理器的握手信号，所以它需要的引脚数目适中。主要的部件如下：

- 协处理器流水线跟随器(见 4.5 节)。
- Load/Store 单元。当从存储器读取或向存储器写入浮点数据时，它对浮点数据实行格式转换。
- 寄存器堆。它可以存储 8 个 80 位的扩展双精度浮点操作数。
- 算术单元。它包含加法器、乘法器和除法器，加上舍入及规格化硬件。

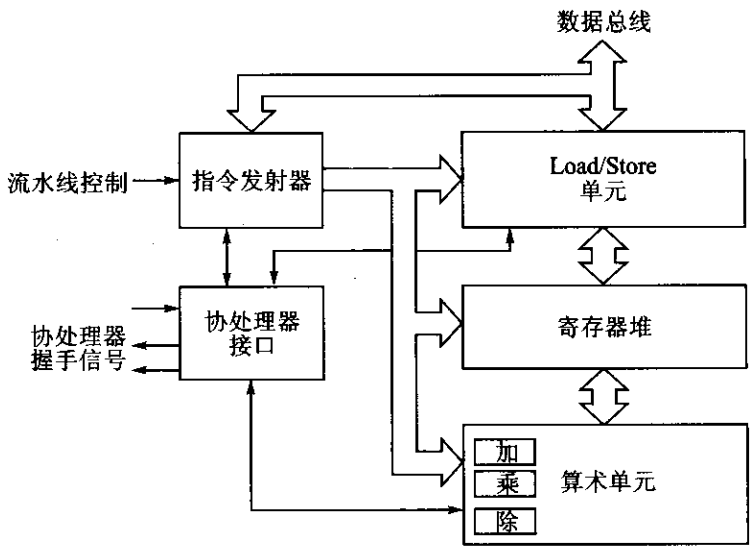


图 6.11 FPA10 的内部组织

Load/Store 单元和算术单元并行工作，使得从存储器读取新操作数时，可以同时处理以前读取的操作数。硬件互锁机构可以防止数据冒险。

FPA10 流水线

FPA10 的算术单元的操作分为 4 级流水线：

1）准备：操作数对准。

2）计算：加、乘或除。

3）对准：将结果规格化。

4）舍入：对结果进行适当的舍入。

只要在指令流水线中检测到浮点操作，就可以开始执行（也就是说，在 ARM 握手信号出现之前），但是结果的回写必须等待握手信号。

浮点上下文切换

FPA 的寄存器代表附加的处理状态，必须通过上下文切换来保存和恢复。但是，在典型情况下只有少数的处理过程使用浮点指令。因此，在每次切换时都保存和恢复 FPA 寄存器是一项不必要的开销。实际中采用下述算法通过软件来降低保存和恢复的次数，即

- 当切断一个使用 FPA 的过程时，不保存 FPA 寄存器，但是将其关断。
- 如果随后的过程执行浮点指令，将引起中断，中断代码将保存 FPA 的状态并启动 FPA。

只有使用 FPA 的那些过程才会产生 FPA 状态保存和恢复的开销。若一个典型的系统只有一个使用 FPA 的过程，则完全可以避免开销。

FPA10 的应用

FPA10 被用做 ARM7500FE 芯片（见 13.5 节）上的宏单元。

VFP10

一个性能高得多的浮点单元 VFP10 用来与 ARM10TDMI 处理器核（见 12.6 节）协同工作。VFP10 与 FPA10 支持不同的浮点指令集。VFP10 支持向量浮点操作。

6.5　表达式

在 ANSI C 中规定 n 位整数的无符号运算结果对 2^n 取模，所以不会出现溢出。这样，基本的 ARM 整数数据处理指令直接实现了大多数的 C 的整数运算、位操作和移位操作。例外的情形是除法和余数运算，它们需要几条 ARM 指令。

寄存器的使用

因为所有数据处理指令都是对整数数据操作的，所以复杂表达式有效求值的关键就是以正确的次序把所需数值放入寄存器，并确保经常使用的数值通常要驻留在寄存器。这显然需要在可以存入寄存器的数值的数量与在表达式求值过程中保留中间结果

的寄存器的数量之间(记住还需要寄存器来保存那些必须从存储器读取的操作数的地址)进行折衷。优化这项折衷是编译器的一个主要任务,这就是要排列出正确的次序,以便按照这个次序读取数据,并把它们组合起来,得到符合语言中规定的操作符顺序的结果。

ARM 的支持

ARM 使用的 3 地址指令格式使得编译器在表达式求值过程中如何保留和复用寄存器方面有最大的灵活性。

Thumb 指令一般是 2 地址指令,它在一定程度上限制了编译器的自由。而且,由于通用寄存器数目较少,也使得编译器的工作更为困难(并导致代码效率较低)。

访问操作数

过程使用的操作数一般都以下列所述方式之一来给出,并可按所指出的方式访问,即

1. 通过寄存器传送的参数

这个数值已经在寄存器中,不需要进一步的操作。

2. 通过堆栈传递的参数

在编译时已确定立即偏移量的堆栈指针(r13)相对寻址使操作数能用单个 LDR 来收集。

3. 过程文字库(literal pool)中的常量

在编译时已确定立即偏移量的 PC 相对寻址使其能用单个 LDR 来访问。

4. 局部变量

局部变量在堆栈分配空间,用相对堆栈指针的 LDR 指令来访问。

5. 全局变量

全局(和静态)变量在静态区域分配空间,用静态基址的相对寻址来访问。静态基址通常存于 r9(见 6.8 节中"ARM 过程调用标准"一段)。

如果传送的数值是指针,则可能需要一个额外的 LDR(带偏移量)来访问指针指向的结构中的操作数。

指针运算

指针的运算依赖于该指针指向的数据类型的宽度。如果指针是递增的,它每次变化的字节数为数据项的宽度。

因而:

```
int *p;
p=p-1;
```

将 p 的值增加 4 个字节。由于数据类型的宽度在编译时已确定,编译器可以将常量以适当的尺度缩放。如果偏移量是一个变量,则它必须在运行时缩放:

```
int i = 4;
p=p + i;
```

如果 p 存放于 r0 而 i 存放于 r1，则 p 的变化可以汇编为

```
ADD        r0, r0, r1, LSL #2          ;将 r1 放大到整数(字)
```

若数据的类型为结构，且其字节数不是 2 的幂时，需要乘上一个小的常数。使用移位和加法指令，需要时使用临时寄存器，通常可以经过少量的操作得到所需的乘积。

数　组

C 中的数组与指针操作的速记符没多大差别，所以以上的说明在这里也同样适用。声明：

```
int a[10];
```

数组的名字 a 只是指向数组第一个元素的指针，引用 a[i]则相当于指针加偏移量的形式*(a+i)。这两种方式可互换使用。

6.6　条件语句

如果检测到布尔运算结果为真(或假)，则条件语句执行；在 C 中，这包括 if…else 语句和开关(C 中的 case 语句)。

if…else

如果条件执行语句很小，则 ARM 体系结构对条件表达式提供非常有效的支持。例如，下面是一个在两个整数中选择最大值的 C 语句：

```
if (a>b) c=a; else c=b;
```

如果变量 a、b 和 c 在寄存器 r0、r1 和 r2 中，则汇编后的代码会如下简单：

```
CMP        r0, r1                      ; if(a>b)…
MOVGT      r2, r0                      ; …c=a…
MOVLE      r2, r1                      ; …else c=b
```

(在这个特例中，C 编译器可以产生一个更清楚的代码序列：

```
MOV        r2, r0                      ; c=a…
CMP        r0, r1                      ; if (a>b)…
MOVLE      r2, r1                      ; …c=b
```

但是这些并不能扩展到更复杂的 if 语句，所以，我们在这个一般性的讨论中忽略这种情况。)

if 与 else 序列的长度可以为少数几条指令，每条指令的条件是相同的(只要条件执行的指令没有改变条件码)。但是，如果超过 2 或 3 条指令，一般最好使用更常规的解

决方法：

```
        CMP     r0, r1          ; if (a>b)…
        BLE     ELSE            ; 如果为伪，则跳过子句
        MOV     r2, r0          ; …c=a…
        B       ENDIF           ; 跳过 else 子句
ELSE    MOV     r2, r1          ; …else c=b
ENDIF   …
```

这里 if 和 else 序列可以任意长，可以自由地使用条件代码（例如包括嵌套的 if 语句），因为转移指令不需要紧跟着比较指令。

但应注意，对于这个简单的例子，无论是否执行了转移，第二个代码序列的执行时间大约是第一个代码序列的两倍。在 ARM 中转移是昂贵的，所以没有使用转移的第一个代码序列非常有效率。

switch

switch，或 case 语句将 if…else 语句的两路判定扩展到多路。switch 语句的标准 C 格式为

```
switch (expression) {
    case constant-expression1 : statements1
    case constant-expression2 : statements2
    …
    case constant-expressionN : statementsN
    default : statementsD
}
```

通常每一组语句（statement）都以 break（或 return）结束，以使开关语句终止；否则按照 C 的语法将执行到下一组语句（statement）。带有 break 的开关语句总是可以解释为等效的 if…else 语句：

```
temp = expression;
if (temp == constant-expression1) {statements1}
else
else if (temp == constant-expressionN) {statementsN}
else {statementD}
```

但是，如果开关语句中有很多 case，代码就会执行得很慢。一种替代的方法是使用**跳转表**。形式最简单的跳转表中包含开关表达式的每一个可能数值对应的目标地址：

```
                ; r0 包含表达式的值
ADR     r1, JUMPTABLE       ; 得到跳转表的基址
CMP     r0, #TABLEMAX       ; 检查是否超限
LDRLS   pc, [r1, r0, LSL #2] ; …如果不超限，则得到 pc
                ; statementsD ; …否则，省缺值
```

```
        B       EXIT                    ; 中断
L1      …       ; statements1
        B       EXIT                    ; 中断
        …
LN      …       ; statementsN
EXIT    …
```

显然，跳转表不可能包含每一个可能的 32 位整数值对应的地址。同样，确保跳转表的查找范围不超过表的末端(超限)是至关重要的，所以必须进行检查。

switch 语句的另一种编译方法可用 Dhrystone 基准程序中的过程来说明。它是这样结束的：

```
 switch (a) {
   case 0: *b = 0; break;
   case 1: if ( c > 100 ) *b = 0; else *b = 3; break;
   case 2: *b = 1; break;
   case 3: break;
   case 4: *b = 2; break;
 } / * end of switch * /
} / * end of procedure * /
```

产生的代码突出显示了 ARM 指令集的若干特点。switch 语句的实现方式是将表达式的值(存于 v2，寄存器的命名习惯遵从在 6.8 节将会讲到的 ARM 过程调用标准)进行字偏移后再加到 PC。在超限的情况下，由加法指令设定 PC，使执行程序落入到由字偏移产生的指令行。只需要 1 条指令的任何 case(例如 case 3)以及最后的 case(无论长度如何)，都可以编译为一行；其他的 case 则需要转移。这个例子还可以说明 if…then…else 的使用条件指令的实现方法。

```
        ; 在入口，a1 = 0，a2 = 3，v2 = 开关表达式
        CMP     v2, #4                  ; 检查是否超限
        ADDLS   pc, pc, v2, LSL #2      ; …如果不超限，则加到 pc(+8)
        LDMDB   fp, {v1, v2, fp, sp, pc}; …如果超限，则返回
        B       L0                      ; case 0
        B       L1                      ; case 1
        B       L2                      ; case 2
        LDMDB   fp, {v1, v2, fp, sp, pc}; case 3(返回)
        MOV     a1, #2                  ; case 4
        STR     a1, [v1]
        LDMDB   fp, {v1, v2, fp, sp, pc}; 返回
        STR     a1, [v1]
        LDMDB   fp, {v1, v2, fp, sp, pc}; 返回
        LDR     a3, c_ADDR              ; 得到 c 的地址
        LDR     a3, [a3]                ; 得到 c
        CMP     a3, #&64                ; c > 100 ? …
```

```
        STRLE   a2, [v1]                ; …否：*b = 3
        STRGT   a1, [v1]                ; …是：*b = 0
        LDMDB   fp, {v1, v2, fp, sp, pc}  ; 返回
c_ADDR  DCD     <address of c>
L2      MOV     a1, #1
        STR     a1, [v1]                ; *b = 1
        LDMDB   fp, {v1, v2, fp, sp, pc}  ; 返回
```

6.7 循 环

C 语言支持 3 种形式的循环控制结构：

- for (e1; e2; e3) {…}
- while (e1) {…}
- do {…} while (e1)

这里，e1、e1 和 e3 是表达式，它们取值为“真”或“假”；{…}是循环体，它执行的次数取决于控制结构。循环体通常很简单，它将负担给予编译器，以减小控制结构的开销。每个循环必须至少有 1 条转移指令，但是再多也会是浪费。

for 循环

典型的 for 循环使用控制表达式来管理索引。

```
for ( i = 0; i < 10; i++ )  { a(i) = 0 }
```

第 1 个控制表达式在循环开始前执行一次；第 2 个在每次进入循环时进行检测并控制循环是否执行；第 3 个在每次循环的结束时执行，准备下一次循环。这个循环可以编译成：

```
        MOV     r1, #0                  ; 数值存入 a[i]
        ADR     r2, a[0]                ; r2 指向 a[0]
        MOV     r0, #0                  ; i = 0
LOOP    CMP     r0, #10                 ; i < 10 ?
        BGE     EXIT                    ; 如果 i >= 10,则结束
        STR     r1, [r2, r0, LSL #2]    ; a[i] = 0
        ADD     r0, r0, #1              ; i++
        B       LOOP
EXIT    …
```

（这个例子中可以看到称为**简化强度**的优化技术，也就是说，把固定的操作移到循环的外部。最前面两行指令逻辑上是在 C 语言的循环之内的，但是，因为它们在每一次循环中都把寄存器初始化到同一个常数，所以被移到了循环外部。）

可以改善上述代码，即把到 EXIT 的条件转移省略，在随后的指令上施加相反的条件，使得在循环的终止条件满足时程序继续向下执行而不是通过转移来绕过它们。然

而，代码还可以继续改善，即把检测移到循环的末尾(这里之所以可以这样作，是因为初始化和检测都是针对常数的，所以编译器可以确信循环至少会执行一次)。

while 循环

while 循环结构简单。一般来说，当迭代的数不是常数，或者明确地在运行时由变量来定义的情况下，使用它来控制循环。在概念上 while 循环的标准排列如下：

```
LOOP    …                          ; 求值表达式
        BEQ     EXIT
        …                          ; 循环体
        B       LOOP
EXIT    …
```

看来需要两个转移指令，因为有时循环体完全不执行，代码必须容许这种情况。稍微改变以下顺序会得到更有效的代码，即

```
        B       TEST
LOOP    …                          ; 循环体
TEST    …                          ; 求值表达式
        BNE     LOOP
EXIT    …
```

与原来的代码相比，第二种代码以完全相同的方式执行循环体和对控制表达式求值，而且代码的规模也相同。但是它每次迭代都少执行一次转移，所以，第二个转移从循环的开销中去除了。编译器还可以产生一种更有效的代码：

```
        …                          ; 表达式取值
        BEQ     EXIT               ; 需要时跳过循环
LOOP    …                          ; 循环体
TEST    …                          ; 表达式取值
        BNE     LOOP
EXIT    …
```

这样当遇到整个 while 结构时少执行一次转移(假设循环体至少执行一次)。这种代码获益适度但多用一条指令，所以仅当性能比代码规模重要很多时它才值得使用。

do…while 循环

do…while 循环在概念上的排列类似于上述改善了的 while 循环。但是因为循环体在检测之前执行(因而总是至少执行一次)，所以没有初始的转移。

```
LOOP    …                          ; 循环体
        …                          ; 表达式取值
        BNE     LOOP
EXIT    …
```

6.8 函数与过程

程序设计

在实际的编程中需要将大的程序分解为足够小的模块以便充分地测试。大的单块程序太复杂，不能充分测试，容易在角落中隐藏 bug。而这些 bug 在程序编写完成后并不会很早地暴露出来，因而不能在程序提交给用户之前得到纠正。

每个小的软件模块应有定义良好的接口并执行特定的操作。它实现这个操作的方式对程序的其他部分应该是不重要的(这是**抽象**的原理，见 6.1 节)。

程序的层次

此外，应把整个程序设计为模块的**层次结构**，而不是简单的并列。

图 6.12 表示了一种典型的层次结构。顶层是称为 main 的主程序。其他的层次是相当随意的。低层程序可以被多个高层程序共用。调用可以跨越层次，而且在整个层次结构中深度可以是不同的。

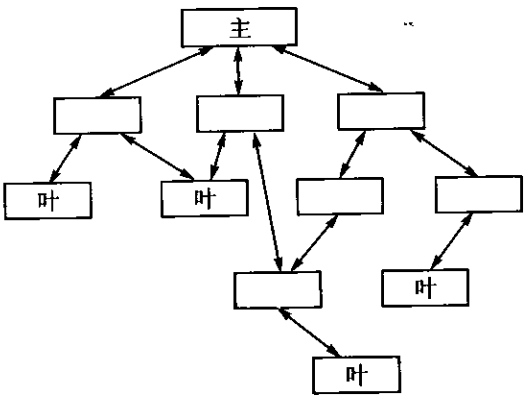


图 6.12 典型的层次化程序结构

叶程序

在层次化结构的最低层的是叶程序。这些程序本身不再调用更低层的程序。在典型的程序中，一些底层的程序是**库**或**系统函数**。它们是预先定义的操作，可以是叶程序，也可以不是叶程序(这就是说，它们可以有内部结构，也可以没有)。

术 语

有一些术语用于描述程序结构中的模块，它们常常是不严密的。我们将使用以下术语：

- 子程序：这个术语一般是指被高层程序调用的程序，特别是以汇编语言级看一

个程序时。

- 函数：一种子程序，它通过它的名字返回一个值。典型的调用方式为

```
c = max ( a, b );
```

- 过程：一种子程序，调用它对特定的数据项执行某种操作。典型的调用方式为

```
printf ( "Hello World\n" );
```

C 函数

一些程序语言明确地区分函数和过程，但是 C 没有这样。在 C 中，所有的子程序都是函数。但是除了返回值以外，它们还可以有其他作用。当返回值是 void 类型时，返回值将被有效地抑制，只留下其他作用，表现出像过程一样的行为。

变元与参数

变元是传递给函数调用的表达式。函数收到的值为**参数**。C 使用严格的“值调用”的语义规则。因此，当调用一个函数时，对每个变元进行复制，尽管函数可能会改变其参数的值。但是，因为它们只是变元的复制，变元本身不会受到影响。

（引用调用语义可使函数内参数的任何改变传递到调用程序。显然，只有当变元是简单变量时才有意义。但是 C 不支持这一点。）

若 C 函数要改变调用程序中的数据，除了返回单个数值外，还可以将数据的指针作为变元。这样函数可以使用指针来访问和修改数据结构。

ARM 过程调用标准

为了使不同编译器产生的程序、汇编语言编写的程序能灵活地混合，ARM 公司定义了一系列过程进入和退出规则。ARM C 编译器使用 ARM 过程调用标准（APCS，ARM Procedure Call Standard）。但只有在必须详细理解汇编级输出的时候，这对于 C 程序员才是重要的。

APCS 在 ARM 体系结构的完美风格中施加了若干约定：

- 规定了通用寄存器的特殊用法。
- 规定了使用哪一种 ARM 指令集支持的满栈/空栈、递增/递减各种堆栈形式。
- 规定了在调试程序时用于回溯（back-tracing）的基于堆栈的数据结构的格式。
- 规定了变元和结果的传递机制，以供所有外部可见函数及过程使用。（“外部可见”的意思是过程的接口在当前程序模块的外部提供。仅在当前模块内部使用的函数可以不遵循这个约定从而得到优化。）
- 支持 ARM 的共享库机制，也就是说它支持一种共享（在进入）代码访问静态数据的标准方式。

APCS 寄存器的使用

16 个当前可见 ARM 寄存器用法的相关约定如表 6.2 所列。寄存器分为 3 组：

1. 4 个变元寄存器，它们将数值传递给函数

函数不需要保留这些寄存器。一旦函数使用或保存其参数值，可以把它们用做临时寄存器。如果这个函数调用了另一个函数，那么由于寄存器将不保留，所以，若寄存器中的数值还有用，就必须在调用前保存。因此，当这样使用时，这些数值是**调用者保存的**寄存器变量。

2. 5 个(到 7 个)寄存器变量，函数不能影响这些寄存器中的数值

这些是**被调用者保存**的寄存器变量。如果函数想使用这些寄存器，就必须先保存它们。但是它可以依赖它调用的函数而不改变它们。

3. 7 个(到 5 个)至少在部分时间内具有专门用途的寄存器

例如，链接寄存器(LR，Link Register)在函数进入时装入返回地址。但是如果进行了保存(正如当函数调用子函数时它必须作的那样)，就可以把它当做临时寄存器使用。

表 6.2 APCS 寄存器的使用约定

寄存器	APCS 名称	APCS 作用
0	a1	变元 1/整数结果/临时寄存器
1	a2	变元 2/临时寄存器
2	a3	变元 3/临时寄存器
3	a4	变元 4/临时寄存器
4	v1	寄存器变量 1
5	v2	寄存器变量 2
6	v3	寄存器变量 3
7	v4	寄存器变量 4
8	v5	寄存器变量 5
9	sb/v6	静态基/寄存器变量 6
10	sl/v7	堆栈限/寄存器变量 7
11	fp	帧指针
12	ip	临时寄存器/新 sb 内部链接单元调用
13	sp	当前堆栈帧的低端
14	lr	链接地址/临时寄存器
15	pc	程序计数器

APCS 变量

有一些(16)不同的 APCS 变量，用于为不同的系统产生代码。它们支持：

1. 32 位或 26 位 PC

老的 ARM 处理器工作于 26 位的地址空间。一些较晚的版本为了向后兼容而继续支持它。

2. 隐式与显式的堆栈超限检查

若要代码可靠运行，则必须检查堆栈的溢出。编译器可以插入指令来执行显式的溢出检查。

若存在存储器管理硬件，则 ARM 系统可以将存储器以页为单位分配给堆栈。如果下一个逻辑页映射完了，则堆栈溢出将会造成数据的异常终止，并被检测出来。因而，存储器管理单元可以执行堆栈超限检查，编译器不需要插入指令进行显式检查。

3. 传递浮点变元的两种方式

ARM 浮点体系结构(见 6.4 节)规定了 8 个浮点寄存器。APCS 可以使用它们向函数传递浮点变元，而且当系统大量使用浮点变量时，这是最有效的方法。但是，如果系统很少使用或不用浮点型数据(很多 ARM 系统不使用)，那么这个方法招致很少的开销，因为避免了将浮点变元传递到整数寄存器和/或堆栈中。

4. 可重入和非可重入代码

可重入的代码是与位置无关的，并且通过静态基址寄存器(SB)间接地对所有数据寻址。该代码可以放入 ROM，还可以被几个客户程序共享。一般来说，放在 ROM 或共享库的代码应是可重入代码，而应用程序代码则不是。

变元的传递

一个 C 函数可能有很多(甚至可变数量)的变元。APCS 按如下方式组织变元：

1) 如果通过浮点寄存器传递浮点数值，则前面的 4 个浮点变元装入前 4 个浮点寄存器中。
2) 其余所有的变元组织为一系列字。前面的 4 个字装入 a1～a4，其余的字按倒序压入堆栈。

注意，多字的变元，包括双精度浮点数，可以在整数寄存器、堆栈，甚至分割开在寄存器和堆栈传递。

结果的返回

简单的结果(例如整数)通过 a1 返回。更复杂的结果通过存储器返回。指定存储器位置的地址通过 a1 作为附加的第一变元有效地传递到函数。

函数的进入和退出

如果一个简单的叶函数只使用 a1～a4 即可完成它的全部功能，那么它可以编译为调用开销最少的代码：

```
        BL      leaf1
        ...
leaf1   ...
        MOV     pc, lr                  ; 返回
```

在一些典型的程序中，大约 50%的函数调用是调用叶函数，而这些调用通常是很简单的。

当必须保存寄存器时，函数必须创建一个堆栈帧。这可以有效地利用 ARM 的多

寄存器 Load 和 Store 指令来编译，即

```
        BL      leaf2
        ...
leaf2   STMFD   sp!, {regs, lr }        ; 保存寄存器
        ...
        LDMFD   sp!, {regs, pc}         ; 恢复和返回
```

这里，保存和恢复的寄存器的数量将是实现函数所需要的最小数量。注意，从链接寄存器(LR)保存的数值直接返回到程序计数器(PC)，因而 LR 可以作为函数体中的临时寄存器使用(这个函数不是上面那种简单的形式)。

在以下情况下使用更复杂的函数调用程序：需要创建堆栈回溯数据结构，处理浮点寄存器传递的浮点变元，检查堆栈溢出等等。

尾随函数

在临近返回之前立即调用其他函数的简单函数通常并不会导致任何明显的调用开销。编译器将让代码直接从后续的函数返回。这使得饰面(veneer)函数(指一些函数，它们只是对变元简单地重排序、改变其类型或增加额外变元)特别有效。

ARM 的效率

总的来说，ARM 可以高效而灵活地支持函数和过程。过程调用标准的各种风格能很好地与不同的应用要求相匹配，而且都能得到高效的代码。编译器使用多寄存器 Load 和 Store 指令以得到良好功效。没有这些指令，调用非叶(non-leaf)函数的花费将会大得多。

编译器还能对叶函数及尾随函数进行有效的优化，从而鼓励好的编程风格。当叶函数调用效率很高时，程序员就可以使用较好的结构化设计；当饰面函数效率很高时，程序员就可以更好地使用抽象。

6.9 使用存储器

像多数计算机系统一样，ARM 系统的存储器也以线性逻辑地址进行组织。C 程序希望访问的程序存储器是固定区域(应用程序的映射驻留在该区域)，存储器还会支持两个数据区，这些数据区的大小动态地变化，且编译器使用数据区时通常不能超出数据区的最大范围。这两个动态数据区为

1. 堆　栈

只要有一个(non-trivial)函数被调用，在堆栈中就产生一个新的活动帧，其中包含回溯记录、局部(非静态)变量等等。

2. 堆

堆(cheap)是存储器中一个区域，用于满足程序中新的数据结构对更多存储空间的要求(malloc())。如果程序在一段长时间内连续地要求存储器，那么它就应该注意释

放不再需要的所有存储空间；否则堆会涨大到将存储器用完。

地址空间模型

存储器的通常用法如图 6.13 所示。图中应用程序可以使用整个存储空间(或者说存储器管理单元可以让应用程序认为它拥有整个存储器空间)。应用程序的映射被装入最低的地址。堆从应用程序的顶端向上生长，堆栈从存储器的顶端向下生长。位于堆顶和堆栈底之间未使用的存储器会按堆或堆栈的需要来分配，如果这部分存储器用完了，那么程序将因缺乏存储器而停止。

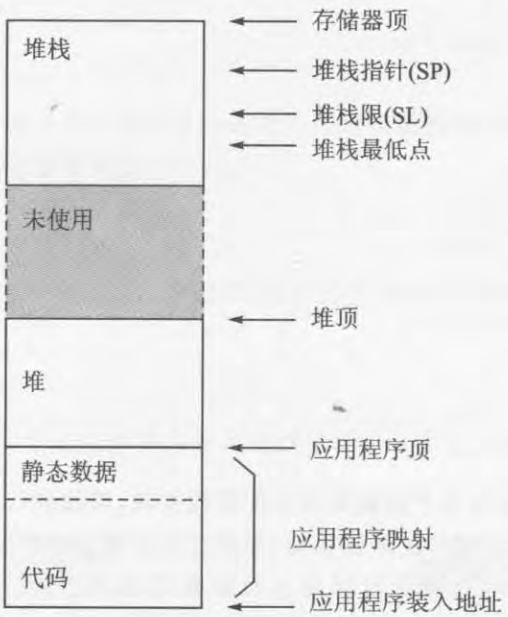


图 6.13　标准的 ARM C 程序地址空间模型

在一个典型具有存储器管理的 ARM 系统中，分配给单个应用程序的逻辑空间是非常大的，为 1～4 GB。存储器管理单元将根据需要向堆或堆栈分配附加的页，直到把所有可分配的页都用完为止(分配完所有的物理存储器页，或者在有虚拟存储器的系统中用完硬盘上的交换(swap)空间)。通常需要很长时间堆顶才能与堆栈底相遇。

在没有存储器管理支持的系统中，一旦操作系统的需要得到满足，应用程序会分配到全部(如果它是当时运行的惟一应用程序)或部分(如果运行 1 个以上应用程序)剩余的物理存储地址空间。然后，当堆顶与堆栈底相遇时，应用程序正好用完存储器。

组块堆栈模型

还可能有其他的地址空间模型，包括实现“组块(chunked)”堆栈。这时的堆栈是堆中一系列连接的组块。这使应用程序占据存储器中一块连续的区域。该区域按需要向一个方向生长，在存储器非常小的情况下可能更加方便。

堆栈的行为

了解在程序运行过程中堆栈的动态行为是很重要的，因为它显示出局部变量(在堆栈中被分配空间)的范围规则。

考虑下面这个简单的程序结构：

```
main ( ) {
            …                                   /* t1 */
            func1 ( );
            …                                   /* t5 */
            func2 ( );
            …                                   /* t7 */
} /* end of main */
func1 ( ) {
            …                                   /* t2 */
            func2 ( );
            …                                   /* t4 */
} /* end of func1 */
func2 ( ) {
            …                                   /* t3, t6 */
} /* end of func2 */
```

假设编译器为每个函数调用分配堆栈空间，堆栈的行为将如图 6.14 所示。每次调用函数时，都为变元分配堆栈空间(如果它们不能全部传递到寄存器)，用来保存寄存器以便在函数内使用，保存返回地址和原来的堆栈指针，并为局部变量分配堆栈的存储器。

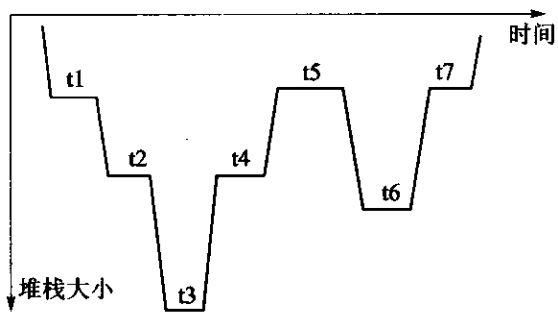


图 6.14 堆栈行为示例

注意当函数退出时堆栈的空间是如何恢复的，随后的调用又是如何再使用的，两次调用 func2( )时如何分配了不同的存储器区域(图 6.14 中的时间 t3 和 t6)。因而即使第一次调用时局部变量所用的存储器在插入调用其他函数时没有被覆盖，在第二次调用时也不能读取原有的数据，因为不知道它的地址。一旦过程退出，局部变量的数值就会永远地丢失。

数据存储

C 语言支持的各种数据类型的二进制表示需要不同数目的存储器来存储。基本的数据类型占据一个字节(字符)、半字(短整数)、字(整数和单精度浮点数)或多字(双精度浮点数)。衍生的数据类型(结构、数组、共用等)由多个基本数据类型来定义。

当数据在存储器中适当对准时,ARM 指令集与许多其他 RISC 处理器一样,在存取数据项方面是最有效率的。以任何字节地址访问一个字节,其效率是相同的。但是,向非字对准的地址存储一个字将使用多达 7 条 ARM 指令,并需要临时工作寄存器,效率很低。

数据对准

因此,ARM 的 C 编译器通常按适当的边界来对准数据。

- 字节存储于任何字节地址。
- 半字存储于偶数的字节地址。
- 字存储于 4 字节的边界。

如果同时声明几个不同类型的数据项,那么为实现对准,编译器会在需要的地方进行**填充**,即

```
struct S1 { char c; int x; short s; } example1;
```

如图 6.15 所示,这个结构体将占据存储器中的 3 个字。(注意,结构体也是填充到字的边界。)

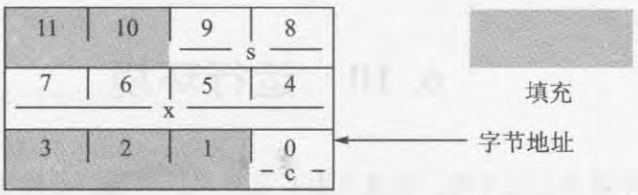


图 6.15　一般结构体存储器分配的例子

数组在存储器中将相应的基本数据项重复排列,每个数据项都要遵从对准规则。

存储器效率

根据上述数据对准规则,程序员可以通过适当组织结构体来帮助编译器减少存储器的浪费。一个与上例同样内容的结构体,按下述方式重新排序,就只占据存储器的两个字,而不是原来的 3 个字。

```
struct S2 { char c; short s; int x; } example2;
```

其存储器的占用方式如图 6.16 所示。一般来说,编译器必须插入一些填充的字节以保持有效对准。如果排列结构体的元素,那么使小于 1 个字长的数据类型在 1 个字内组合起来,会使填充部分最小化。

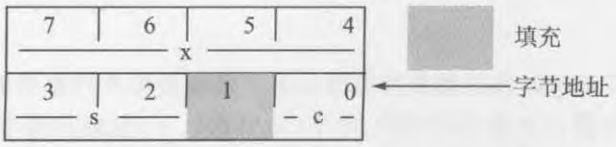


图 6.16 更有效的结构体存储器分配的例子

压缩的结构体

有时需要与其他计算机交换数据，而那个计算机遵循另一种对准规定；或者需要压缩数据以减少存储器的使用，即使这样做会降低性能。为此目的，ARM 的 C 编译器可以产生使用压缩数据结构的代码。在这种数据结构中去除了所有的填充字节。

```
__packed struct S3 { char C; int x; short s; } example3;
```

压缩的结构体对结构中所有域进行精确的控制，但招致的开销是 ARM 对非对准操作数访问的效率较低，因而应在确实需要时才使用。上面声明的压缩结构体占用存储器的情况如图 6.17 所示。

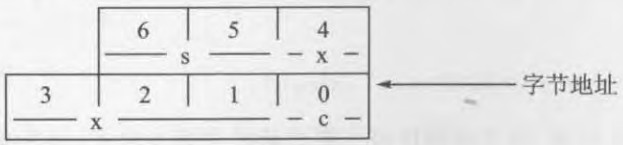


图 6.17 压缩结构体存储器分配的例子

6.10 运行环境

C 程序需要运行环境。这通常由 C 程序可以调用的函数库来提供。C 程序员可以在 PC 或工作站上找到完整的 ANSI C 库。其中有多种函数，例如文件管理、输入输出(printf())和实时时钟等等。

最小运行时间库

在如移动电话这样的小型嵌入式系统中，这些函数中的多数是不相关的。ARM 公司提供一个最小的独立运行库。一旦把它装入目标环境，就可以运行基本的 C 程序。因此，这个库反映出 C 程序的最低要求。它包括：

1. 除法和余数函数

因为 ARM 指令集不包括除法指令，所以除法要作为库函数来实现。

2. 堆栈限制检查函数

小型嵌入式系统不太可能配备存储器管理硬件来检测堆栈的溢出，因而需要这些库函数确保程序安全运行。

3. 堆栈和堆的管理

所有 C 程序都使用堆栈来处理(多)函数调用,而且除了最无关紧要的之外,所有函数调用都会在堆中创建数据结构。

4. 程序启动

一旦堆栈和堆被初始化,则程序通过调用 main()函数来启动。

5. 程序终止

多数程序通过调用_exit()函数来终止。如果检测到错误,即使永久运行的程序也应终止。

为这个最小的库所产生的代码的总数为 736 字节。而且在实现时,它允许链接程序忽略任何未引用的部分,以便在很多情况下把库的映射减小到 500 字节。这比整个 ANSI C 库要小得多。

6.11　例题与练习

例题　6.1

编写、编译并执行 C 语言的 Hello World 程序。

下面的程序实现了所要求的功能:

```
/* Hello World in C */
#include <stdio.h>

int main( )
{
  printf ( "Hello World\n" );
  return ( 0 );
}
```

在这个例子中需注意的主要事情为

- #include 命令,它使这个程序使用 C 中所有的标准输入输出函数。
- main 过程的声明,每个 C 程序必须有这个过程,程序的执行是通过调用它才开始的。
- printf (…)语句调用 stdio 中的一个函数,它将输出送到标准输出设备。缺省时为显示终端。

同汇编语言编程的练习一样,主要的挑战是建立从编辑文本到编译、链接和运行程序的使用工具的流程。一旦这个程序工作了,学习更复杂的程序就相当简单了(至少在程序的复杂度使得程序设计变为一个真正挑战之前)。

使用 ARM 软件开发工具,上面的程序应保存为 HelloW.c。然后应使用过程管理器创建一个新的工程(project),并加入这个文件(作为工程中的惟一文件)。点击 Build 按钮使程序编译和链接,再点击 Go 按钮使它在 ARMulator 上运行,在终端窗口有希望得到预期的输出。

练习 6.1.1

用编译器(使用"-s"选项)产生汇编程序,并查看产生的代码。

练习 6.1.2

在调试状态下运行程序,使用单步执行以便观察处理器执行代码的过程。

例题 6.2

使用 32 位二进制计数法、二进制编码的十进制计数法、ASCII 和单精度浮点计数法写出数字 2001。

二进制: 2001 = 1024+512+256+128+64+16+1
= $00000000000000000000011111010001_2$

BCD: 2001 = 0010 0000 0000 0001

ASCII: 2001 = 00110010 00110000 00110000 00110001

F-P: 2001 = $1.1111010001\times2^{1010}$
= 01001001 11110100 01000000 00000000

练习 6.2.1

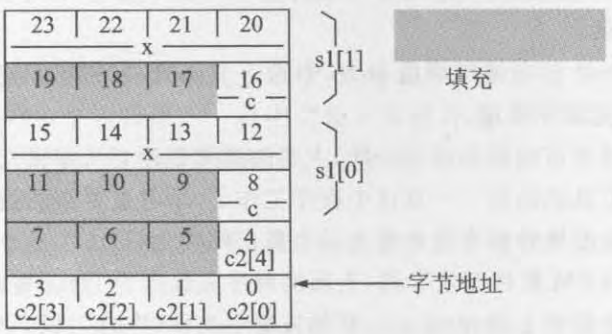
写一段 C 程序,将罗马数字表示的数据转换成十进制格式。

例题 6.3

画出下列数据在存储器中是如何组织的。

```
struct S1 { char c; int x;};
struct S2 {
  char c2[5];
  S1   s1[2];
} example;
```

第一个结构语句只是声明了一个类型,所以不用分配存储器。第二个结构语句建立了一个叫做 example 的结构,其中包含由 5 个字符组成的数组,随后是由两个 S1 类型的结构组成的数组。结构必须从字边界开始,所以,字符数组将被填充以充满两个字,每个 S1 结构也将占据两个字。因此,存储器的组织如下图所示。



练习 6.3.1

画出同样的结构压缩时在存储器中是如何组织的。

第 7 章　Thumb 指令集

本章内容综述

Thumb 指令集是针对代码密度的问题而提出的。它可以看作是 ARM 指令压缩形式的子集。所有的 Thumb 指令都有相对应的 ARM 指令，而 Thumb 的编程模型也对应于 ARM 的编程模型。在 ARM 指令流水线中实现 Thumb 指令须先进行动态解压缩，然后再把它作为处理器内的标准 ARM 指令来执行。

Thumb 是一个不完整的体系结构。不能指望微处理器只执行 Thumb 指令而不支持 ARM 指令集。因此，Thumb 只需支持通用功能，必要时可以借助于完善的 ARM 指令集（例如，所有异常自动进入 ARM 模式）。

ARM 开发工具完全支持 Thumb 指令，应用程序可以灵活地将 ARM 和 Thumb 子程序混合编程，以便在例程的基础上提高性能或代码密度。

本章将介绍 Thumb 的结构及实现，并提出适于采用 Thumb 指令的应用程序所具有的特征。当应用得当时，使用 Thumb 指令集可以同时达到降低功耗、节约成本和提高性能的目的。

7.1　CPSR 中的 Thumb 指示位

支持 Thumb 指令的 ARM 微处理器也可以执行标准的 32 位 ARM 指令集。在任何时刻，对指令流的解释取决于 CPSR 的第 5 位，即位 T（参见图 2.2）。若 T 置位，则认为指令流为 16 位的 Thumb 指令；否则为标准的 ARM 指令。

并不是所有的 ARM 处理器都支持 Thumb 指令。只有在命名中有字母 T 的才支持，例如将在 9.1 节中介绍的 ARM7TDMI。

进入 Thumb 模式

复位后，ARM 启动并执行 ARM 指令。转向执行 Thumb 指令的通常方法是执行一条交换转移指令 BX（Branch and Exchange，参见 5.5 节）。若 BX 指令指定的寄存器的最低位为 1，则将 T 置位，并将程序计数器切换为寄存器其他位给出的地址。注意，由于该指令引起了转移，它将刷新指令流水线，消除已在流水线中的所有指令在解释上的所有不确定性（不执行这些指令）。

异常返回也可以将微处理器从 ARM 状态转换为 Thumb 状态。返回时可使用一种特殊形式的数据处理指令，或使用一种特殊形式的多寄存器 Load 指令（参见 5.2 节的“异常返回”）。通常这两种指令用于返回到进入异常前所执行的指令流，而不是特地用于转换到 Thumb 模式。同 BX 指令一样，这两种指令也改变程序计数器并因而刷新指令流水线。

退出 Thumb 模式

执行 Thumb BX 指令（将在 7.3 节中介绍）可以显式地返回到 ARM 指令流。

由于进入异常总是在 ARM 模式进行，因此，任何时候发生异常都能隐含地返回到 ARM 指令流。

Thumb 系统

从上面可知，如果只处理初始化和进入异常，则 Thumb 系统只需要包含部分 ARM 指令。

但是大多数 Thumb 应用不会只包括这个 ARM 指令最小集。一个典型的嵌入式系统会在 ARM 核所在的芯片上集成一个小的、高速的 32 位存储器，把有速度要求的关键子程序（如数字信号处理算法）以 ARM 代码形式保存在这个存储器中。大多数对速度没有要求的程序存储在片外 16 位 ROM 中。在本章的最后将对此进一步讨论。

7.2 Thumb 编程模型

Thumb 指令集是 ARM 指令集的一个子集，并只能对限定的 ARM 寄存器进行操作。其编程模型如图 7.1 所示。Thumb 指令集对低（Lo）8 个通用寄存器 r0～r7 具有全

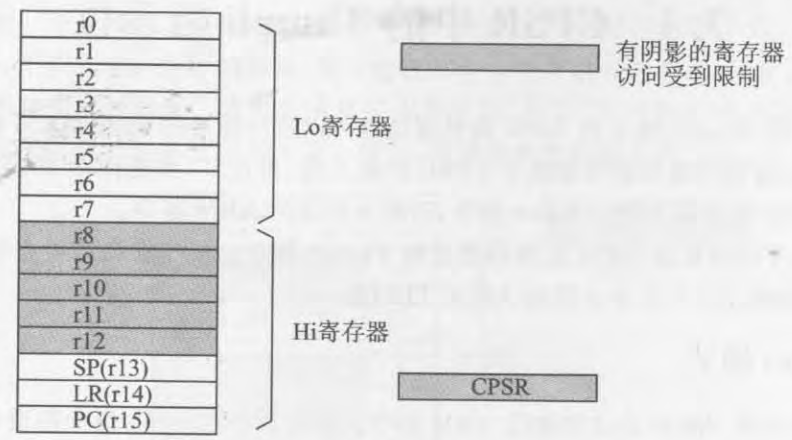


图 7.1 Thumb 指令可访问的寄存器

部访问权限，对寄存器 r13～r15 进行扩展以作特殊应用：

- r13 用做堆栈指针。
- r14 用做链接寄存器。
- r15 用做程序计数器 PC。

这些用法与 ARM 指令集对这些寄存器的用法非常相似，尽管作为堆栈指针的 r13 在 ARM 代码中是纯粹的软约定，而在 Thumb 代码是某种硬连接。其他寄存器(r8～r12 及 CPSR)只能作有限访问：

- 少数指令可以使用高寄存器(Hi 寄存器，r8～r15)。
- CPSR 的条件标志位由算术和逻辑操作设置并控制条件转移。

Thumb-ARM 相似处

所有的 Thumb 指令都是 16 位的。它们都有相对应的 ARM 指令，因此，继承了 ARM 指令集的许多特点：

- Load-Store 结构，有数据处理、数据传送及流控制指令。
- 支持 8 位字节、16 位半字和 32 位字数据类型，半字以两字节边界对准，字以 4 字节边界对准。
- 32 位无分段存储器。

Thumb-ARM 差异处

为了实现 16 位指令长度，丢弃了 ARM 指令集一些特性：

- 大多数 Thumb 指令是无条件执行的(所有 ARM 指令都是条件执行的)。
- 许多 Thumb 数据处理指令采用 2 地址格式(目的寄存器与一个源寄存器相同)。

  (除 64 位乘法指令外，ARM 数据处理指令采用 3 地址格式。)
- 由于采用高密度编码，Thumb 指令格式没有 ARM 指令格式规则。

Thumb 异常

所有异常都使微处理器返回 ARM 执行状态，并在 ARM 的编程模型中进行处理。由于位 T 驻留在 CPSR 中，它在进入异常时被保存到相应的 SPSR 中。从异常指令返回时将恢复微处理器状态，并按照发生异常时处理器的状态继续执行 ARM 或 Thumb 指令。

应该注意到，ARM 异常返回指令(见 5.2 节的“异常返回”)需要根据 ARM 流水线的行为对返回地址进行调整。由于 Thumb 指令是 2 个字节长，而不是 4 个字节，所以，由 Thumb 执行状态进入异常时其自然偏移应与 ARM 不同。这是因为拷贝到异常模式链接寄存器的 PC 地址值将以 2 个字节的倍数而不是以 4 个字节的倍数增加。但是，Thumb 结构要求链接寄存器的值能自动调整以与 ARM 返回偏移匹配，使得在两种模式下可以使用同样的返回指令，而不是使返回过程复杂化。

7.3 Thumb 转移指令

这类控制流指令包括在 ARM 指令集中已介绍过的、多种形式的、相对于 PC 的转移指令和转移链接指令，以及用于 ARM 和 Thumb 指令集转换的转移交换指令。

ARM 指令有一个大的(24 位)偏移域(offset field)，这不可能在 16 位 Thumb 指令格式中表示。为此，Thumb 指令集有多种方法实现其子功能。

二进制编码

Thumb 转移指令二进制编码如图 7.2 所示。

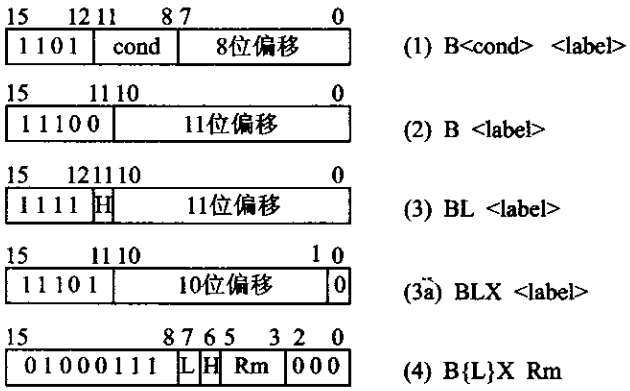


图 7.2 Thumb 转移指令二进制编码

说 明

转移指令的典型用法如下：

1) 短距离条件转移指令可用于控制循环的退出。

2) 中等距离的无条件转移指令用于实现 goto 功能。

3) 长距离子程序调用。

ARM 指令集用同一条指令处理所有这些情况。在前两种情况下浪费了 24 位偏移的很多位。Thumb 指令集对每种情况采用不同的指令模式，分别如图 7.2 所示，因而更有效。

前两种转移格式是条件域和偏移长度的折衷。第 1 种格式中条件域与 ARM 指令相同(参见 5.3 节)。前两种格式的偏移值都左移 1 位(以实现半字对准)，并符号扩展到 32 位。

第 3 种格式更为精妙。转移链接子程序通常需要一个大的范围，很难用 16 位指令格式实现。为此，Thumb 采用两条这样格式的指令组合成 22 位半字偏移(符号扩展为 32 位)，使指令转移范围为 ±4 MB。为了使这两条转移指令相互独立，以致使它们之间也能响应中断等，将链接寄存器 LR 作为暂存器使用。LR 在这两条指令执行完成后

会被覆盖，因此，LR 中不能装有有效内容。这个指令对的操作如下：

1)（H=0）　LR:=PC +（偏移量左移 12 后符号扩展至 32 位）

2)（H=1）　PC:=LR +（偏移量左移 1 位）

LR:=oldPC + 3

这里，oldPC 是第 2 条指令的地址；加 3 使产生的地址指向下一条指令，并且使最低位置位以指示这是一个 Thumb 程序。

用 3a 格式的指令代替上面的第 2 步就可以实现 BLX 指令。格式 3a 只在 v5T 结构中有效。它使用与上面 BL 指令同样的第一步，即

1)（BL，H=0）　LR:=PC +（偏移量左移 12 后符号扩展至 32 位）

2)（BLX）　PC:=LR +（偏移量左移 1 位）&0xffff_fffc

LR:=oldPC + 3

Thumb 位清 0。

应注意转移的目标是 ARM 指令，偏移地址只需要 10 位，而且 PC 值的位[1]（PC[1]）可能为 1，因此，必须进行清 0 操作。

第 4 种格式直接对应 ARM 指令 B{L}X（参见 5.5 节，不同之处是 BLX（仅在 v5T 结构中有效）指令中 r14 值为后续指令地址加 1，以指示是被 Thumb 代码调用）。当指令中 H 置 1 时，选择高 8 个寄存器（r8～r15）。

汇编格式

```
B<cond>   <label>     ; 格式 1：目标为 Thumb 代码
B         <label>     ; 格式 2：目标为 Thumb 代码
BL        <label>     ; 格式 3：目标为 Thumb 代码
BLX       <label>     ; 格式 3a:目标为 ARM 代码
B{L}X     Rm          ; 格式 4：目标为 ARM 或 Thumb 代码
```

转移链接产生两条格式 3 指令。格式 3 指令必须成对出现而不能单独使用。同样 BLX 产生一条格式 3 指令和一条格式 3a 指令。

汇编器根据当前指令地址、目标指令标号的地址以及对流水线行为的微调计算出应插入指令中相应的偏移量。若转移目标不在寻址范围内，则给出错误信息。

等效 ARM 指令

尽管格式 1～3 与 ARM 的转移和转移链接指令非常相似，但 ARM 指令只支持字（4 字节）偏移，而 Thumb 指令要求半字（2 字节）偏移，因此，这些 Thumb 指令不能直接映射为 ARM 指令。对支持 Thumb 的 ARM 核进行了轻微修改以支持半字转移偏移，使 ARM 转移指令支持半字偏移。

格式 4 与 ARM 指令在汇编语言级是等价的。BLX 指令只有 v5T 结构的 ARM 微处理器支持。

子程序调用及返回

上面的指令及等效的 ARM 指令所调用的函数可以用与调用程序相同或不相同的

指令集编写。

如果函数只由相同的指令集调用，则它可以用传统的 BL 调用，用“MOV pc，r14”或“LDMFD sp!，{…，pc}”（在 Thumb 代码中为 POP “{…，pc}”）返回。

如果函数可以由不相同的指令集调用，或可以由相同与不相同指令集调用，则可以用“BX lr”或“LDMFD sp!，{…，rN}；BX rN”（在 Thumb 代码中为“POP {…，rN}；BX rN”）返回。

支持 v5T 结构的 ARM 微处理器也可以用“LDMFD sp!，{…，pc}”（在 Thumb 代码中为“POP {…，pc}”）返回，因为这些指令采用加载的 PC 值的最低位来更新 Thumb 标志位。早于 v5T 结构的微处理器不支持这样的用法。

7.4 Thumb 软中断指令

Thumb 软中断指令的行为与 ARM 等价指令完全相同。进入异常的指令使微处理器进入 ARM 执行状态。

二进制编码

Thumb 软中断指令的二进制编码如图 7.3 所示。

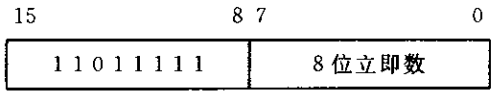


图 7.3 Thumb 软中断指令的二进制编码

说 明

这个指令将引起下列动作：

- 将下一条 Thumb 指令的地址保存到 r14_svc；
- 将 CPSR 寄存器保存到 SPSR_svc；
- 微处理器关闭 IRQ，将 Thumb 位清 0，并通过修改 CPSR 的相关位进入监控模式；
- 强制将 PC 值置为地址 0x08。

然后进入 ARM 指令 SWI 的处理程序。正常的返回指令将恢复 Thumb 的执行状态。

汇编格式

SWI　　<8 位立即数>

等价 ARM 指令

等价的 ARM 指令有相同的汇编语法。8 位立即数以 0 扩展来填满 ARM 指令的 24 位

立即数域。

很明显，这将会把 Thumb 代码的 SWI 指令限制到前 256 种，而 ARM 的 SWI 指令可以达到 1 600 万种。

7.5　Thumb 数据处理指令

Thumb 数据处理指令包括一组高度优化且相当复杂的指令，范围涵盖编译器通常需要的大多数操作。

这些指令的功能足够清楚。应选择哪些指令和不应选择哪些指令远不是显而易见的。但是这些选择是基于对典型应用程序需求的详细分析而做出的。

二进制编码

Thumb 数据处理指令的二进制编码如图 7.4 所示。

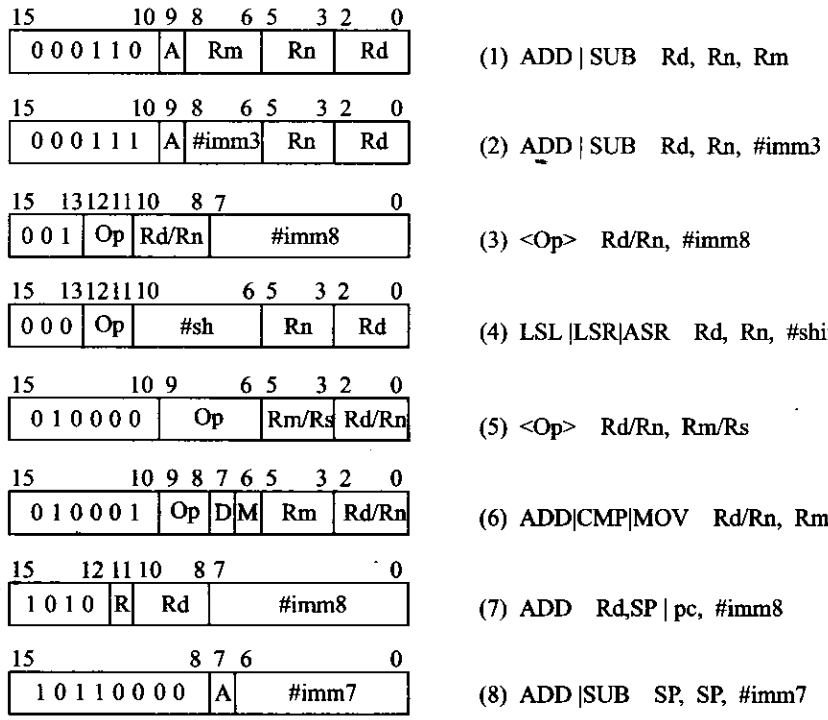


图 7.4　Thumb 数据处理指令的二进制编码

说　明

这些指令都能够映射到相应的 ARM 数据处理指令(包括乘法指令)。尽管 ARM 指令支持在单条指令中完成一个操作数的移位及一个 ALU 操作，但 Thumb 指令集将移位操作和 ALU 操作分离为不同的指令。因此，Thumb 指令集中移位操作是作为操作符出现的，而不是作为操作数的修改量出现。

汇编格式

指令的各种格式如下：

```
1) <op>   Rd, Rn, Rm                ; <op> = ADD|SUB
2) <op>   Rd, Rn, #<#imm3>          ; <op> = ADD|SUB
3) <op>   Rd|Rn, #<#imm8>           ; <op> = ADD|SUB|MOV|CMP
4) <op>   Rd, Rn, #<#sh>            ; <op> = LSL|LSR|ASR
5) <op>   Rd|Rn, Rm|Rs              ; <op> = MVN|CMP|CMN
   ; …TST|ADC|SBC|NEG|MUL|LSL|LSR|ASR|ROR|AND|EOR|ORR|BIC
6) <op>   Rd|Rn, Rm                 ; <op> = ADD|CMP|MOV
   ;                                         (Hi regs)
7) ADD    Rd, sp|pc, #<#imm8>
8) <op>   sp, sp, #<#imm7>          ; <op> = ADD|SUB
```

等价的 ARM 指令

在 Thumb 指令集中有等价指令的 ARM 数据处理指令如下所列。等价的 Thumb 指令列在注释部分。

使用低 8 个通用寄存器(r0～r7)的指令：

ARM 指令		Thumb 指令	
MOVS	Rd, #<#imm8>	MOV	Rd, #<#imm8>
MVNS	Rd, Rm	MVN	Rd, Rm
CMP	Rn, #<#imm8>	CMP	Rn, #<#imm8>
CMP	Rn, Rm	CMP	Rn, Rm
CMN	Rn, Rm	CMN	Rn, Rm
TST	Rn, Rm	TST	Rn, Rm
ADDS	Rd, Rn, #<#imm3>	ADD	Rd, Rn, #<#imm3>
ADDS	Rd, Rd, #<#imm8>	ADD	Rd, #<#imm8>
ADDS	Rd, Rn, Rm	ADD	Rd, Rn, Rm
ADCS	Rd, Rd, Rm	ADC	Rd, Rm
SUBS	Rd, Rn, #<#imm3>	SUB	Rd, Rn, #<#imm3>
SUBS	Rd, Rd, #<#imm8>	SUB	Rd, #<#imm8>
SUBS	Rd, Rn, Rm	SUB	Rd, Rn, Rm
SBCS	Rd, Rd, Rm	SBC	Rd, Rm
RSBS	Rd, Rn, #0	NEG	Rd, Rn
MOVS	Rd, Rm, LSL #<#sh>	LSL	Rd, Rm, #<#sh>
MOVS	Rd, Rd, LSL Rs	LSL	Rd, Rs
MOVS	Rd, Rm, LSR #<#sh>	LSR	Rd, Rm, #<#sh>
MOVS	Rd, Rd, LSR Rs	LSR	Rd, Rs
MOVS	Rd, Rm, ASR #<#sh>	ASR	Rd, Rm, #<#sh>
MOVS	Rd, Rd, ROR Rs	ASR	Rd, Rs

MOVS	Rd, Rd, ROR Rs	ROR	Rd, Rs
ANDS	Rd, Rd, Rm	AND	Rd, Rm
EORS	Rd, Rd, Rm	EOR	Rd, Rm
ORRS	Rd, Rd, Rm	ORR	Rd, Rm
BICS	Rd, Rd, Rm	BIC	Rd, Rm
MULS	Rd, Rm, Rd	MUL	Rd, Rm

使用高 8 个寄存器(r8～r15)的指令。在有些情况下结合低 8 个寄存器使用。

ARM 指令		Thumb 指令	
ADD	Rd, Rd, Rm	ADD	Rd, Rm (1/2 Hi regs)
CMP	Rn, Rm	CMP	Rn, Rm (1/2 Hi regs)
MOV	Rd, Rm	ADD	Rd, Rm (1/2 Hi regs)
ADD	Rd, pc, #＜#imm8＞	ADD	Rd, pc, #＜#imm8＞
ADD	Rd, sp, #＜#imm8＞	ADD	Rd, sp, #＜#imm8＞
ADD	sp, sp, #＜#imm7＞	ADD	sp, sp, #＜#imm7＞
SUB	sp, sp, #＜#imm7＞	SUB	sp, sp, #＜#imm7＞

注意事项

1) 所有对低 8 个寄存器操作的数据处理指令都更新条件码位(等价的 ARM 指令位 S 置位)。
2) 对高 8 个寄存器操作的指令不改变条件码位。CMP 指令除外,它只改变条件码。
3) 上面的指令中"1/2 Hi regs"表示至少有 1 个寄存器操作数是高 8 位寄存器。
4) #imm3、#imm7、#imm8 分别表示 3 位、7 位和 8 位立即数域。#sh 表示 5 位的移位数域。

7.6 Thumb 单寄存器数据传送指令

选择哪些 ARM 指令并将其重新表示为 Thumb 指令是很复杂的事情。这主要是基于编译器的行为来选择的。

注意到对文字库(literal pool)(相对于 PC)和堆栈(相对于 SP)的访问有较大的偏移。与对无符号操作数的支持(基址偏移寻址或基址变址寻址)相比,对有符号操作数的支持(仅有基址变址寻址)有一定的限制。

二进制编码

Thumb 单寄存器数据传送指令二进制编码如图 7.5 所示。

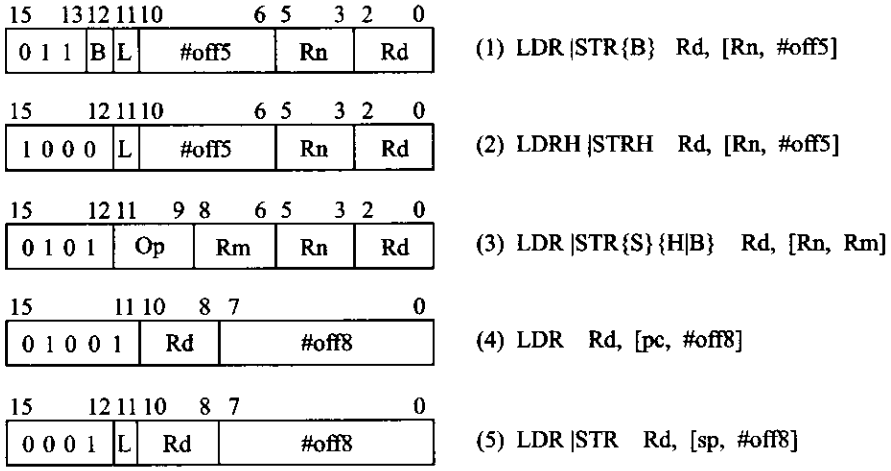


图 7.5 Thumb 单寄存器数据传送指令二进制编码

说 明

这些指令是从 ARM 单寄存器传送指令中精心导出的子集，并且与等价的 ARM 指令有严格相同的语义。

在所有的指令中，对偏移量需要根据数据类型按比例调整。例如，5 位偏移量的范围在字节 Load 和 Store 指令中是 32 字节，在半字 Load 和 Store 指令中是 64 字节，在字 Load 和 Store 指令中是 128 字节。

汇编格式

各种汇编格式如下：

```
1) <op>   Rd, [Rn, #<#off5>]   ; <op> = LDR|LDRB|STR|STRB
2) <op>   Rd, [Rn, #<#off5>]   ; <op> = LDRH|STRH
3) <op>   Rd, [Rn, Rm]         ; <op> =…
          ; …LDR|LDRH|LDRSH|LDRB|LDRSB|STR|STRH|STRB
4) LDR    Rd, [pc, #<#off8>]
5) <op>   Rd, [sp, #<#off8>]   ; <op> = LDR|STR
```

等价 ARM 指令

与这些 Thumb 指令等价的 ARM 指令有完全相同的汇编格式。

注意事项

1) #off5 和 #off8 分别表示 5 位和 8 位的立即数偏移。在所有情况下，汇编格式用字节表示偏移。在指令二进制编码中的 5 位和 8 位偏移需要根据存取的数据类型进行比例调整。

2) 与 ARM 指令相同，只有 Load 指令支持符号数。对于 Store 指令，有符号和无

符号存储有相同的结果。

7.7　Thumb 多寄存器数据传送指令

同 ARM 指令一样，Thumb 多寄存器数据传送指令可以用于过程调用与返回以及存储器块拷贝。但为了编码的紧凑性，这两种用法由分开的指令实现，其寻址方式的数量也有所限制。在其他方面，这些指令的性质与等价的 ARM 指令相同。

二进制编码

Thumb 多寄存器传送指令的二进制编码如图 7.6 所示。

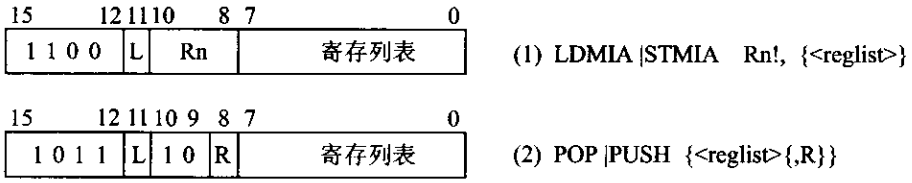


图 7.6　Thumb 多寄存器传送指令的二进制编码

说　明

指令的块拷贝形式使用 LDMIA 和 STMIA 寻址模式(参见图 3.2)。Lo 寄存器(低 8 个寄存器 r0～r7)中的任何一个可以作为基址寄存器。寄存器列表可以是这些寄存器的任意子集，但不应包括基址寄存器，因为总是选择回写(若 Load 和 Store 多寄存器时基址寄存器在寄存器列表中，同时又选择回写，将使结果不确定)。

堆栈形式使用 SP(r13)作为基址寄存器，并且也总是使用回写。堆栈的模式也固定为满栈递减。寄存器列表除了可以是 8 个 Lo 寄存器外，链接寄存器 LR(r14)可以出现在 PUSH 指令中，PC(r15)可以出现在 POP 指令中，以优化过程调用及返回程序，正如 ARM 代码经常做的那样。

汇编格式

<reg list>是寄存器的列表，寄存器范围是 r0～r7。

```
LDMIA    Rn!, {<reg list>}
STMIA    Rn!, {<reg list>}
POP      {<reg list>{, pc}}
PUSH     {<reg list>{, lr}}
```

等价 ARM 指令

对于两种块格式，等价的 ARM 指令有相同的汇编格式。在后两种指令格式中要以合适的寻址模式来代替 POP 和 PUSH。

块拷贝：

```
        LDMIA    Rn!, {<reg list>}
        STMIA    Rn!, {<reg list>}
Pop:
        LDMFD    SP!, {<reg list>{, pc}}
Push:
        STMFD    SP!, {<reg list>{, lr}}
```

注意事项

1) 基址寄存器必须是字对准的，否则，一些系统将忽略地址值的低 2 位，而另一些系统会产生未对准访问异常。
2) 由于所有这些指令都采用基址回写，因此，基址寄存器不应出现在寄存器列表中。
3) 编码后寄存器列表的每一位对应一个寄存器。位[0]指示 r0 寄存器是否传送；位[1]控制 r1 等等。在 POP 和 PUSH 指令中，R 位控制 PC 和 LR。
4) 在 v5T 结构中，加载的 PC 的最低位更新 Thumb 位，因此，可以直接返回到 Thumb 或 ARM 调用程序。

7.8 Thumb 断点指令

Thumb 断点指令的行为与等价的 ARM 指令完全相同。断点指令用于软件调试，可以使微处理器中断正常指令执行，进入相应的调试程序。

二进制编码

Thumb 断点指令二进制编码如图 7.7 所示。

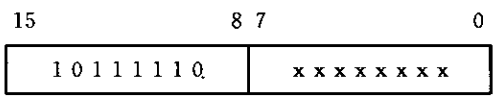


图 7.7 Thumb 断点指令二进制编码

说 明

当硬件调试单元作适当配置时，断点指令会使微处理器放弃指令预取。

汇编格式

BKPT

等价 ARM 指令

等价的 ARM 指令有完全相同的汇编语法。只有实现了 v5T 结构的 ARM 处理器

支持 BKPT 指令。

7.9　Thumb 的实现

对 3 级流水线 ARM 处理器的大部分逻辑作相对较小的改动就可以实现 Thumb 指令集(5 级流水线的实现要复杂些)。增加的最大逻辑是指令流水线中的 Thumb 指令扩展逻辑。这部分逻辑将 Thumb 指令转化为对应的等价 ARM 指令,其逻辑组织如图 7.8 所示。

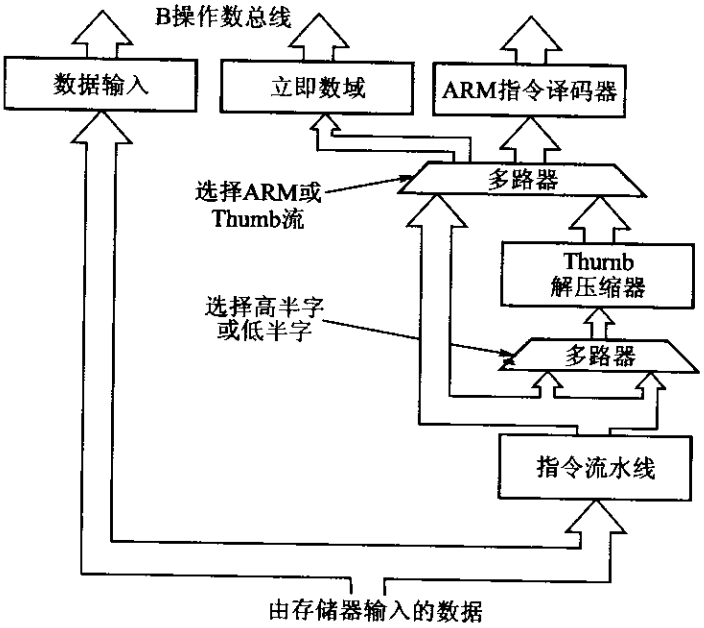


图 7.8　Thumb 指令扩展逻辑组织

增加的指令扩展逻辑与指令译码器串联,可能会增加译码时间。但是实际上 ARM7 的流水线在译码周期的第一相只做了很少的工作。因此,可以把扩展逻辑安排在这里而不会影响周期时间或增加流水线延迟。ARM7TDMI 的 Thumb 流水线操作与前面 4.1 节中"3 级流水线"一段所述的方式完全相同。

指令映射

Thumb 指令扩展逻辑将 16 位 Thumb 指令静态地转换为等价的 32 位 ARM 指令。这包括主操作码和次操作码的查表转换,3 位寄存器指示符(specifier)零扩展成 4 位寄存器指示符,以及所需的其他域的映射。

例如,Thumb 指令"ADD　Rd, #imm8"(参见 7.5 节)与对应的 ARM 指令"ADD Rd, Rd, #imm8"(参见 5.7 节)的映射如图 7.9 所示。

注意:

- 由于转移指令是惟一条件执行的 Thumb 指令,因此,其他 Thumb 指令在转换

时使用条件“always”。

- 在 Thumb 操作码中隐含地指定 Thumb 数据处理指令是否应修改 CPSR 中的条件码，在 ARM 指令中要明确指定。
- 总是可以通过重复寄存器指示符将 Thumb 的 2 地址指令格式转换为 ARM 的 3 地址指令格式(用其他方法一般不易完成)。

指令扩展逻辑的简单性对 Thumb 指令集的效率是非常重要的。如果 Thumb 扩展逻辑复杂、速度低并且功耗大，那么 Thumb 就没有什么价值了。

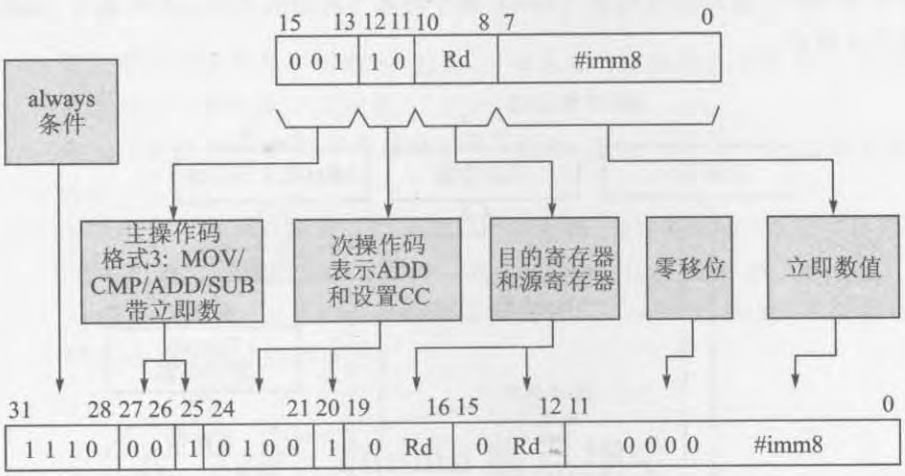


图 7.9 Thumb 到 ARM 的指令映射

7.10 Thumb 的应用

我们需要回顾一下 Thumb 的特点来看一看它更适合哪方面的应用。Thumb 指令长为 16 位，只用 ARM 指令一半的位数来实现同样的功能。但由于 Thumb 指令一般比 ARM 指令的语义内涵少，所以，实现特定的程序所需的 Thumb 指令数目较 ARM 的多。增加的比率因不同的程序而变化。在一个典型的例子中，Thumb 代码所需的空间为 ARM 代码的 70%。因此，在比较 Thumb 方案与纯 ARM 方案时，我们会发现：

Thumb 的特点

- Thumb 代码所需空间为 ARM 代码的 70%。
- Thumb 代码使用的指令数比 ARM 代码多 40%。
- 用 32 位存储器，ARM 代码比 Thumb 代码快 40%。
- 用 16 位存储器，Thumb 代码比 ARM 代码快 45%。
- 使用 Thumb 代码，外部存储器功耗比 ARM 代码少 30%。

由此，若性能最重要，则系统应使用 32 位存储器和运行 ARM 代码；若成本及功耗更重要，则最好选择 16 位存储器系统及 Thumb 代码。若两者结合使用，会在两方面取得最好的效果。

Thumb 系统

- 高端的 32 位 ARM 系统可以用 Thumb 代码实现特定的非关键程序，以节省功耗或降低对存储器的需求。
- 低端的 16 位系统可以有小规模的 32 位片上 RAM 供运行 ARM 代码的关键程序使用，所有非关键程序使用片外 Thumb 代码。

上面第二种情况或许更接近于 Thumb 所适宜的应用。在移动电话和寻呼机的应用中，需要 ARM 最大处理能力的实时信号处理(DSP)功能。但这些程序可以紧凑编码并放在小规模的片上存储器中。那些控制用户界面、电池管理系统等等复杂的比较长的代码是非实时的，可由 Thumb 代码放在片外 ROM 中，由 8 位或 16 位总线就可以得到好的性能，同时降低成本和延长电池寿命。

7.11　例题与练习

例题　7.1

用 Thumb 指令重新编写 3.4 节中的 Hello World 程序。两种实现的代码长度相比会是怎样呢？

下面是原先的 ARM 程序：

```
                AREA    HelloW, CODE, READONLY
SWI_WriteC              EQU     &0
SWI_Exit                EQU     &11
                ENTRY
START           ADR     r1, TEXT
LOOP            LDRB    r0, [r1], #1
                CMP     r0, #0
                SWINE   SWI_WriteC
                BNE     LOOP
                SWI     SWI_Exit
TEXT            = "Hello World", &0a, &0d, 0
                END
```

```
; 输出 r0 中的字符
; 结束程序
; 代码的入口
; r1 → "Hello World"
; 取下一个字节
; 检查文本终点
; 若非终点，则打印…
; …并返回 LOOP
; 执行结束

; 源程序结束
```

这些 ARM 指令的大多数有直接等价的 Thumb 指令，但有些指令没有。Load 字节指令不支持自动变址，监控调用不能条件执行。因此，需要对 Thumb 代码做稍许改动，即

```
                AREA    HelloW_Thumb, CODE, READONLY
SWI_WriteC              EQU     &0
SWI_Exit                EQU     &11
                ENTRY
```

```
; 输出 r0 中的字符
; 结束程序
; 代码的入口
```

```
                CODE32                                  ; 进入 ARM 状态
                ADR     r0, START+1                     ; 取 Thumb 入口地址
                BX      r0                              ; 进入 Thumb
                CODE16                                  ; 下面是 Thumb 代码…
START           ADR     r1, TEXT                        ; r1 → "Hello World"
LOOP            LDRB    r0, [r1]                        ; 取下一字节
                ADD     r1, r1, #1                      ; 指针加 1 "T
                CMP     r0, #0                          ; 检查文本终点
                BEQ     DONE                            ; 完成? "T
                SWI     SWI_WriteC                      ; 若非终点,则打印…
                B       LOOP                            ; …并返回 LOOP
DONE            SWI     SWI_Exit                        ; 结束执行
                ALIGN                                   ; 保证 ADR 正常执行
TEXT            DATA
                =       "Hello World", &0a, &0d, 0
                END
```

上面的代码中加入了两条指令，用“"T”标记，用于补充 Thumb 指令的不足。ARM 代码的长度为 6 条指令加上 14 字节的数据，共 38 字节。Thumb 代码的长度为 8 条指令加 14 字节的数据(不计将微处理器转入 Thumb 指令状态的部分)，共 30 字节。

这个例子给出了在编写 Thumb 代码时应记住的一些重点：

- 汇编器需要知道什么时候产生 ARM 代码，什么时候产生 Thumb 代码。用 CODE32 和 CODE16 指令提供这类信息。(这些是用于汇编的伪指令，它本身不产生任何代码。)
- 由于微处理器在调用这些代码时处于 ARM 执行状态，因此，必须做出明确的准备，指示微处理器执行 Thumb 指令。倘若将 r0 适当地初始化，就可用“BX r0”指令来完成这种转换。特别要注意，r0 寄存器的最低位被置位，以便使微处理器在转移目标处执行 Thumb 指令。
- Thumb 指令 ADR 只能产生字对准的地址。由于 Thumb 指令是 16 位的，不能保证任意数目的 Thumb 指令后面的地址是字对准的，因此，例中程序在文本字符串之前有一明确的“ALIGN”。

为了能在 ARM 软件开发工具套件中编译并执行这个程序，必须启动可以产生 Thumb 代码的汇编器，并用 ARMulator 仿真一个 Thumb-aware 的处理器核。项目管理器的缺省设置是 ARM6 核，并且只产生 32 位 ARM 代码。产生代码之前在项目管理器中的 Options 菜单下选择 Project，并在 Tools 对话框中选择 TCC/TASM，就可以改变原来的缺省设置。这会使目标微处理器自动的转换为支持 Thumb 的 ARM7t。

在其他方面，Thumb 代码的编译及执行与 ARM 代码很相似。

练习　7.1.1

将 3.4 节和 3.5 节的其他程序转换为 Thumb 代码，并与原来的 ARM 代码进行长度比较。

练习　7.1.2

使用 TCC 编译 C 源程序产生 Thumb 代码(照例从 Hello World 程序开始)。查看 C 编译器(用"-S"选项)产生的汇编码。与编译为 ARM 代码的相同程序比较代码的长度及执行时间。

第8章　体系结构对系统开发的支持

本章内容综述

设计任何计算机系统都是一个复杂的任务，设计一个嵌入式 SoC 更是如此。SoC 的开发是在一个 CAD 环境中进行的，而第一个芯片不但要正确工作，而且必须达到需要的性能并且是可制造的。修补设计缺陷的惟一机会是软件。修改芯片来改正错误不但费时，并且会显著地增加成本。

在过去的 20 年里，微处理器系统开发的主要方法是使用在线仿真器 ICE(In-Circuit Emulator)。系统本身是包括微处理器芯片、各种存储器和外围器件的印刷电路板。使用在线仿真器时，目标板上的微处理器由 ICE 的仿真头代替。ICE 仿真微处理器的功能，使用户能够观察系统内部的状态，修改处理器内部寄存器及存储器的值，设置断点等等。

现在，微处理器本身就是一个大的芯片中的单元，以前的方法完全失效了，因为不可能拔下一个芯片的一部分！到目前为止，没有任何方法能够完全取代 ICE 的作用。但是有几项技术能起到一些作用，其中一些需要在处理器体系结构上的明确支持。本章介绍基于 ARM 核的系统芯片开发技术，以及为了帮助开发而在 ARM 核中实现的结构。

8.1　ARM 存储器接口

本节介绍 ARM 处理器与标准存储器元件构成的存储器系统连接的基本原理。存储器接口的有效性是决定系统性能的重要因素，因此，开发高性能系统的设计工程师必须深入理解这些原理。最近的 ARM 核已具有 AMBA 接口(参见 8.2 节)，但这里依然对许多基本问题进行讨论。

ARM 总线信号

ARM 处理器芯片的总线接口在细节上各有不同，但是，它们基本上是相似的。存储器总线接口信号包括下列部分：

- 32 位地址总线 A[31：0]，给出被访问数据的字节地址。
- 32 位双向数据总线 D[31：0]，用于数据传送。

- 指定是否需要存储器($\overline{\text{mreq}}$)和地址是否为连续地址(seq)的信号。这些信号在前一个周期发出以使存储器控制逻辑能够做好准备。
- 指定传送方向($\overline{\text{r}}$/w)及传送位数(早期的处理器为 $\overline{\text{b}}$/w,后期的处理器为 mas[1：0])的信号。
- 总线时序及控制信号(abe、ale、ape、dbe、lock 和 bl[3：0])。

简单存储器接口

最简单的存储器接口适合于 ROM 和静态 RAM(**SRAM**)。这些器件要求在周期结束之前地址都要稳定。后期的处理器可以通过禁止地址流水来实现这一点(将 ape 固定为低),早期的处理器可以通过重新安排地址总线的时序来实现(将 ale 连接到 $\overline{\text{mclk}}$)。地址和数据总线可以直接连接到存储器,如图 8.1 所示。图中也给出了输出使能信号($\overline{\text{RAMoe}}$和$\overline{\text{ROMoe}}$)和写使能信号($\overline{\text{RAMwe}}$)。

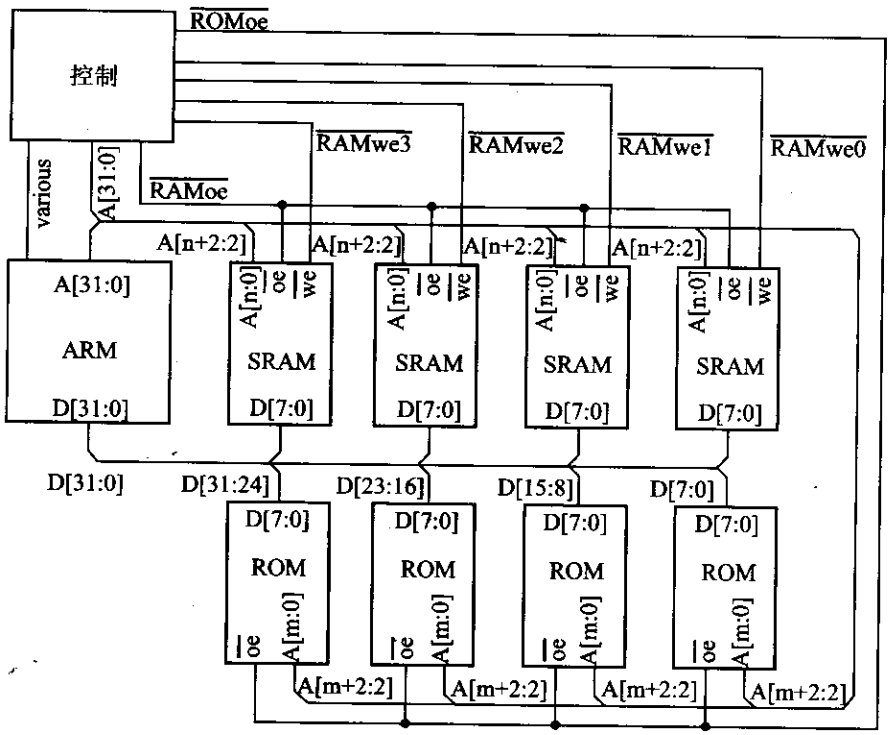


图 8.1　基本的 ARM 存储器系统

图中给出的是 8 位存储器器件的连接,这是标准的 SRAM 和 ROM 的连接结构。每类存储器都需要 4 个器件才能构成 32 位宽的存储器。每个的器件连接总线的单个字节。图中的标注给出了器件内部的器件总线标号以及它在器件外部连接的总线。比如,最靠近 ARM 处理器的 SRAM 器件的端口 D[7：0]连接到数据总线的端口 D[31：24]。D[31：24]连接到 ARM 的 D[31：24]引脚。

由于最低位的两根地址线 A[1：0]用于字节选择,它们用于控制逻辑,不与存储器相连,因此,存储器器件连接到 A[2]及更高位的地址线。使用地址线的精确数字依赖

于存储器体的大小(如 128 KB 的 ROM 使用 A[18：2])。

尽管 ARM 支持字节和字的存储器读操作,但存储器系统忽略这些差别(以浪费一些功耗为代价),总是提供一个字的数据。ARM 将抽取寻址的字节而忽略其他数据。这样 ROM 存储器不需要每块使用单独的读使能信号,使用 16 位器件也不会产生问题。但是对于字节写操作,每一块需要一个单独的写使能信号,因此,控制逻辑需要产生 4 个字节写使能控制信号。这使得宽数据的 RAM 很难使用(而且低效率),除非这些 RAM 提供字节使能,因为写一个字节将需要对存储器进行读—修改—写的操作。由于许多处理器需要支持字节写操作,因此,若 RAM 块的数据宽度大于 1 个字节,它们必须提供独立的字节使能。

控制逻辑

控制逻辑完成下列功能:

1. 决定何时激活 RAM 及何时激活 ROM

控制逻辑决定系统存储器的映射。复位后处理器从 0 地址开始。由于 RAM 还没有初始化,所以它肯定会找到 ROM。因此,最简单的存储器映射是当 A[31]为低时使能 ROM,为高时使能 RAM(大多数 ARM 系统在启动后立即改变存储器映射,将 RAM 放在存储器的低地址处以便使异常向量可以修改)。

2. 在写操作时控制字节写使能信号

当进行字写操作时,应使所有的字节写使能信号有效;在字节写操作时,仅使被寻址字节的使能信号有效;对于支持半字的 ARM,半字写操作应使 4 个使能信号中的两个有效。

3. 在处理器继续操作之前保证数据已准备好

最简单的办法是使时钟 mclk 降到足够慢,以保证所有存储器件能够在单个时钟周期内完成访问。更复杂的系统可以按照 RAM 的访问时间设置时钟。对于慢速的设备,如 ROM 及外围端口,使用等待状态访问。

完成上述功能的逻辑非常简单,如图 8.2 所示(全部逻辑可用一个单片可编程器件实现)。与双向数据总线相关的设计可能是最微妙的部分。保证在任何时间只有 1 个器件驱动数据总线是非常重要的,所以将总线转向写周期时,或者从读 ROM 切换到读 RAM 时,都需要小心。图中采用的方法是在 mclk 为高时激活相应的数据源,而在 mclk 为低时关闭所有的数据源,因此,处理器数据总线使能信号 dbe 也应与 mclk 连接。这是一个非常保守的方法,它限制了能够使用的最大时钟频率,从而损害了系统的性能。

要注意的是,这个设计假定 ARM 的输出在时钟周期末已经稳定。在较新型的处理器中,当地址流水线使能(ape)控制输入固定为低时即是如此。早期的处理器应使 ale=$\overline{\text{mclk}}$来重调整地址输出的时序。这需要一个在 mclk 为低时导通的外部透明锁存器,以重新调整的 $\overline{\text{r}}$/w 和 $\overline{\text{b}}$/w 的时序(它取代 mas[1],mas[0]固定为低)。

这个简单的存储器系统没有使用$\overline{\text{mreq}}$(或 seq)信号。它在每个周期都激活存储器。对于真正的存储器访问,ARM 将只请求写周期,因而这种方法是安全的。在所有内部及协处理器传送周期,$\overline{\text{r}}$/w 保持为低。

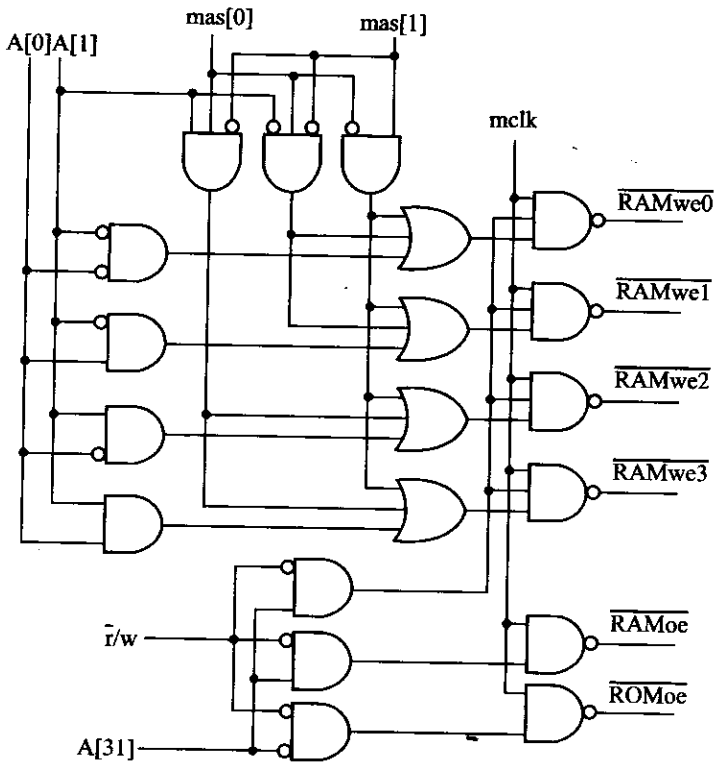


图 8.2　简单的 ARM 存储器系统控制逻辑

等待状态

在这个系统中，如果提高系统的时钟速度，那么当最慢路径失败时，系统将停止工作。最慢路径通常是 ROM 访问。如果时钟速度以 RAM 访问时间为准，那么在进行 ROM 访问时，插入等待状态则会大幅度提高系统的性能。通常 ROM 的每次访问都有固定的时钟周期数。精确的数目可以根据时钟速率和 ROM 数据手册确定。我们将假定 ROM 存取时间为 4 个时钟周期。

现在必须在存储器控制逻辑中加上一个简单的有限状态机来控制 ROM 访问。一种合适的状态转换图如图 8.3 所示。插入 ARM 的$\overline{\text{wait}}$输入后，3 个 ROM 状态将 ROM 访问时间扩展到 4 个时钟周期。这里在设计上存在的问题是，由于地址在当前周期早期已经有效，并且$\overline{\text{wait}}$必须在 mclk 上升沿之前插入，但由于没有时钟沿用于产生$\overline{\text{wait}}$信号，因此，$\overline{\text{wait}}$不能作为简单的状态机输出产生。另一个问题是要产生一个扩展的没有毛刺的$\overline{\text{ROMoe}}$信号。

图 8.4 给出了一种可采用的电路。状态机是个同步计数器，使用了两个边沿触发的触发器。当检测到一个 ROM 读取操作时，使$\overline{\text{wait}}$有效；而当 ROM3 信号有效时，则停止发出$\overline{\text{wait}}$信号。两个电平触发锁存器锁存$\overline{\text{wait}}$信号以产生一个干净、扩展的$\overline{\text{ROMoe}}$信号。图 8.5 中所示电路的时序图可以清楚地说明这个电路的逻辑操作。

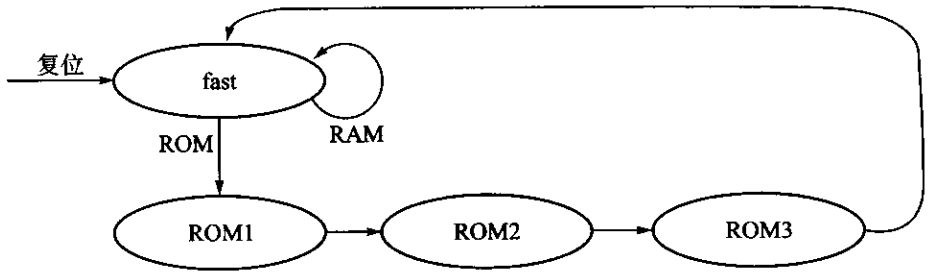


图 8.3 ROM 等待控制状态转换图

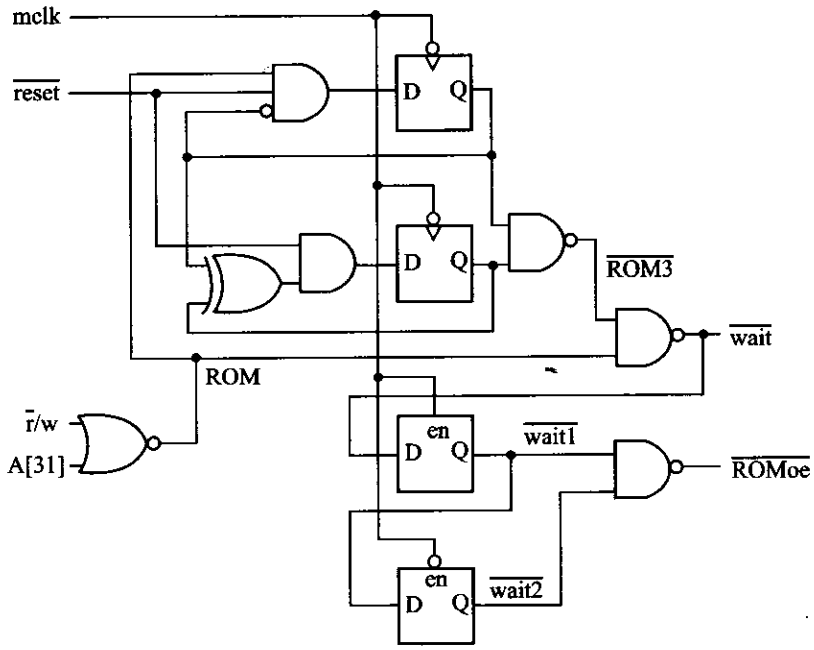


图 8.4 ROM 等待状态产生电路

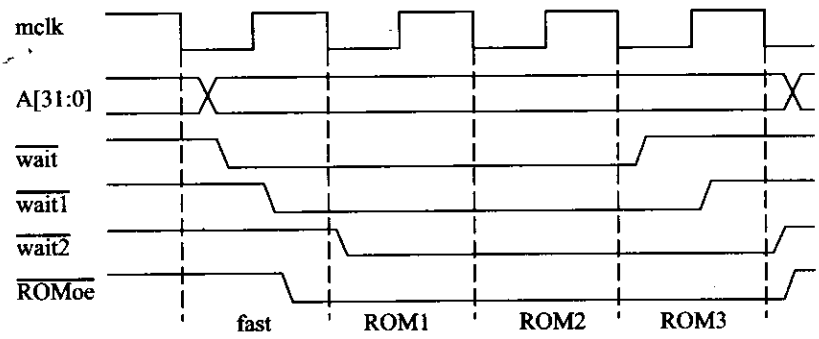


图 8.5 ROM 等待状态电路的时序

顺序访问

如果系统的工作速度更快，则有可能在单个时钟周期内不能完成一个新地址的译码及 RAM 访问操作。只要发射了一个未知地址，就插入一个额外的周期使得有足够的时间进行地址译码。非顺序地址都是未知的，但是，在一个典型的程序中，顺序地址约占全部地址的 75%。

现在我们也应该开始考虑那些不使用存储器的周期，因为它们没有理由不在全时钟速率下工作。图 8.6 给出了一个合适的状态转换图。

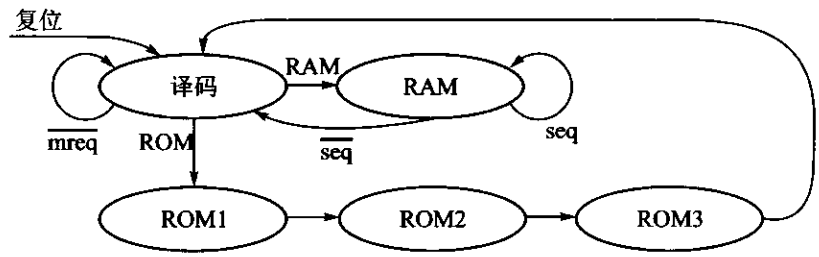


图 8.6　具有用于地址译码的等待状态的状态转换图

DRAM

动态随机访问存储器(**DRAM**)是最便宜的存储器(用每位的价格来衡量)。动态存储器在电容中注入电荷来存储信息。这些电荷逐渐漏掉(经过 1 ms 左右)。存储的数据必须在电荷泄漏前读出或重新写入(刷新)。对存储器的刷新通常由存储器控制逻辑而不是由处理器完成，因此，我们不在这里对其详细讨论。我们所关心的是存储器的内部组织，如图 8.7 所示。

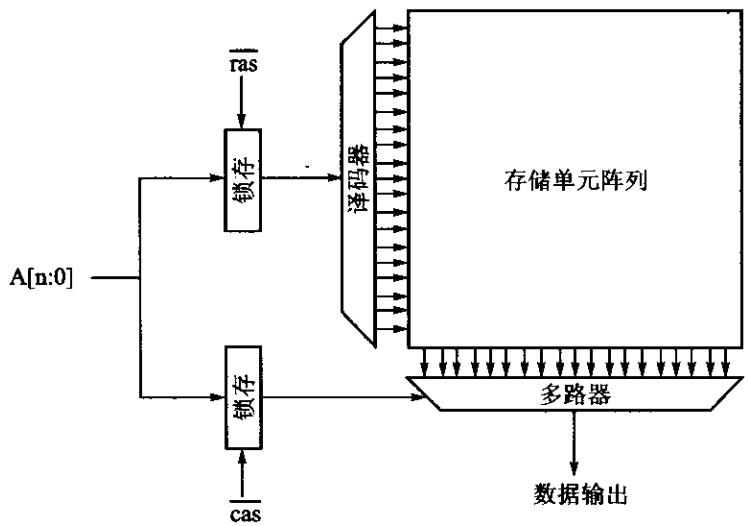


图 8.7　DRAM 存储器的组织

同大多数存储器件一样，DRAM 中的存储单元以近似矩形的阵列形式组织。与其他大多数存储器件不同的是，这种组织对用户是可见的。阵列通过行和列来寻址。DRAM 从**多路复用的地址总线**分别接收行地址和列地址。首先给出行地址，并由低有效的**行地址选通信号**$\overline{\text{ras}}$(row address strobe)控制锁存；然后给出列地址，并由低有效的**列地址选通信号**$\overline{\text{cas}}$(column address strobe)控制锁存。如果下一个访问在相同的行中，则只需要给出新的列地址。这种只锁存新的列地址的存储器访问方式(简称$\overline{\text{cas}}$访问方式)不需要使整个阵列都处于活动状态，因此，与整个行-列形式的访问方式相比，不但速度可以提高 2～3 倍，而且功耗也可以降低很多，因此，应尽可能多的使用$\overline{\text{cas}}$方式。

在存储器访问过程中，要尽可能早地检测出新的地址与前次地址在相同的行中。这是比较困难的。将新地址与前次地址的相关位进行比较往往比较慢。

ARM 地址增值器

ARM 采取的解决方案利用了这样一个事实，即大多数地址(典型为 75%)由地址增值器产生。如图 8.8 所示的 ARM 地址选择逻辑从 4 个源中选择下一周期所用的地址。增值器为 4 个源之一。当下一个地址由增值器产生时，ARM 输出一个 seq 指示信号。外部逻辑检查上一个地址是否在行的边界。若上一个地址不在行末并且 seq 信号有效，则可以进行$\overline{\text{cas}}$访问方式。

尽管这种机制不能找出所有在同一 DRAM 行的访问操作，但能找到大多数这种操作，并且非常易于实现。seq 信号和上一个地址在时钟的前半周期已经有效，给存储器控制逻辑留下充裕的时间。

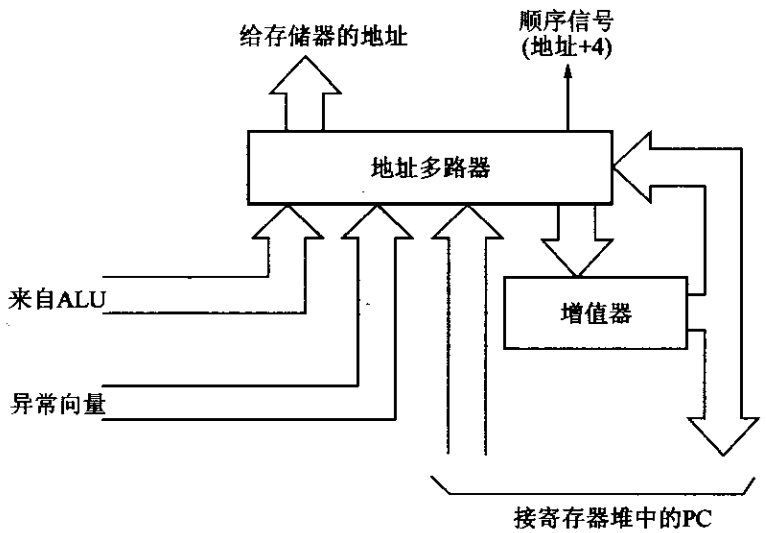


图 8.8 ARM 地址寄存器结构

一个典型的 DRAM 时序如图 8.9 所示。第一次的非顺序访问需要锁存行地址而花费两个时钟周期；而随后的顺序访问则使用$\overline{\text{cas}}$访问方式，只需要 1 个时钟周期。(这里使用的是早期的地址时序，ape 和 ale 为高电平。)

seq 信号的另一种应用是指示一个周期与之前的内部或协处理器寄存器传送周期

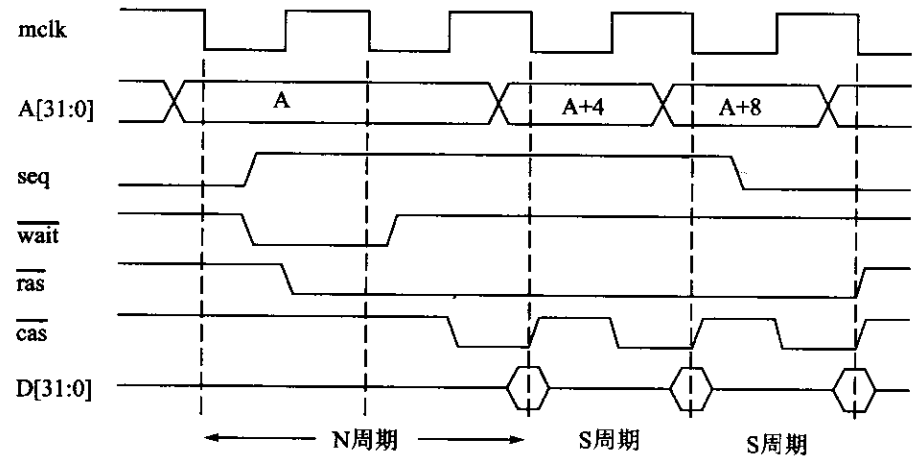


图 8.9　DRAM 时序图

使用相同的地址。这也可以用来提高 DRAM 的访问性能。只有 seq 信号有效时间足够早，才有可能使 DRAM 访问提前 1 个周期开始。典型时序如图 8.10 所示。（在这个过程中wait处于无效状态。）

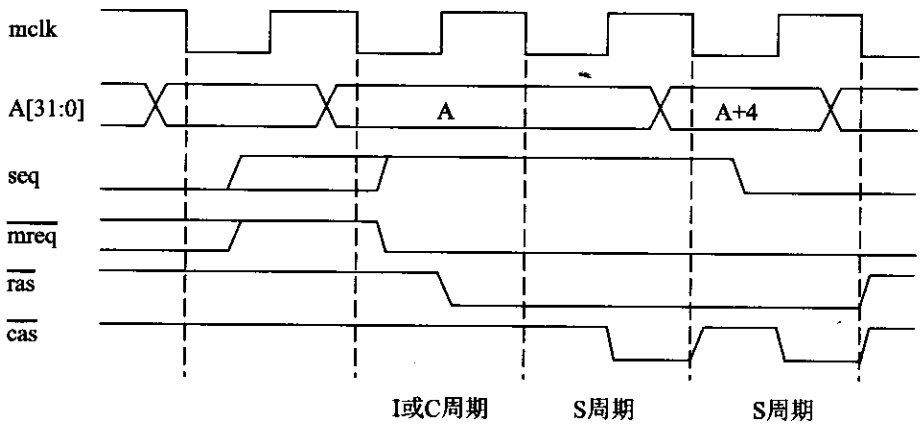


图 8.10　一个内部周期之后的 DRAM 时序

外围接口访问

大多数系统除了上述的存储器元件之外，还有一些外围接口器件。通常这些器件的访问速度较慢，可以使用与前面讲到的 ROM 访问相似的技术实现其接口。

8.2　AMBA 总线

ARM 处理器核有一个优化的总线接口用于高速 Cache。不管有没有 Cache，当 ARM 核作为一个元件集成到复杂的系统芯片上时，它需要某种接口与片上其他宏单元进行通信。

虽然这种接口并不特别难以设计，但是有多种可能的解决方案。如果在每一项设计中都专门选择一种总线结构，这将耗费设计资源，并制约外围宏单元的复用。为避免这种浪费，ARM 公司提出了 **AMBA**(Advanced Microcontroller Bus Architecture)总线，使片上不同宏单元的连接实现标准化。以这种总线接口设计的宏单元可以被看做将来系统芯片的零件箱，采用将现有的宏单元重新组合的方法设计复杂的片上系统，最终将能变成一项非常简单的任务。

AMBA 总线

AMBA 规范定义了 3 种总线：

- **AHB**(Advanced High-performance Bus)：用于连接高性能系统模块。它支持突发(burst)数据传送方式及单个数据传送方式，所有时序都以单一时钟的沿为基准。
- **ASB**(Advanced System Bus)：用于连接高性能系统模块，它支持突发数据传送模式。
- **APB**(Advance Peripheral Bus)：为低性能的外围部件提供较简单的接口。

一个典型的基于 AMBA 的微控制器将使用 AHB 或 ASB 总线，再加上 APB 总线，如图 8.11 所示。ASB 总线是旧版的系统总线；而 AHB 则较晚推出，以增强对更高性能、综合及时序验证的支持。

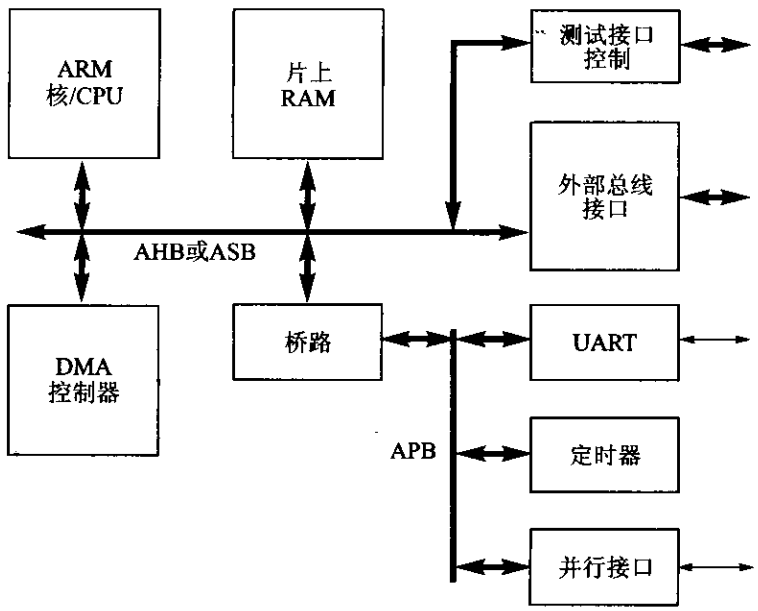


图 8.11 典型的基于 AMBA 的系统

APB 总线通常用做局部的二级总线。它像是 AHB 或 ASB 上的单个从属模块。在以下几节中，假定系统总线是 ASB。AHB 总线的细节将在本节的最后讨论。

判 决

总线主控器向中央判决器发出请求后开始一个总线传输。当有请求冲突时，由判

决器决定优先级。判决器设计是与系统相关的问题。ASB 只给出了必须遵守的协议：

- 主控器 x 向中央判决器发出请求(AREQx)。
- 当总线空闲时，判决器向主控器发出确认信号(AGNTx)。(当判决决定发出哪个确认信号时，必须考虑总线锁定信号(BLOK)，以保证不侵扰总线的原子(atomic)传输)。

总线传输

当主控器的访问总线请求被确认后，它发出地址及控制信息以指示传输的类型以及应响应的从动器件。下列信号用来定义传输时序，即

- 总线时钟 BCLK：它通常与 ARM 处理器时钟 mclk 相同。

保持确认(grant)信号的总线主控器使用下列信号进行总线传输，即

- 总线传输 BTRAN[1：0]：指示下一个总线周期为地址周期、顺序周期还是非顺序周期。它由确认信号使能，并在它涉及到的总线周期之前。
- 地址总线 BA[31：0]：(在只需适度地址空间的系统中不需要实现所有的地址线。在多路器实现中，地址由数据总线输出)。
- 总线传输方向 BWRITE。
- 总线保护信号 BPROT[1：0]：它指示是取指令还是取数据，是监控程序访问还是普通用户访问。
- 传输长度 BSIZE[1：0]：指明传输的是字节、半字还是字。
- 总线锁定 BLOK：使主控器保持总线以完成读—修改—写的原子传输。
- 数据总线 BD[31：0]：用于发送写数据及接收读数据。在一个采用地址和数据多路器的实现中，地址信号也通过这个总线传输。

从动单元可能立即处理所要求的传输，在 BD[31：0]上接收写数据或发送读数据，或发出下列响应信号之一：

- 总线等待 BWAIT：允许从动模块不能在当前周期完成传输时插入等待状态。
- 总线终止 BLAST：允许从动模块终止一次顺序突发传输，以强制总线主控器发出新的总线传输请求来继续传输。
- 总线错误 BERROR：指示传输未能完成。如果主控器是处理器，则应中止传输。

总线复位

ASB 支持多个独立的片上模块，其中许多模块能够驱动数据总线(及一些控制线)。只要所有模块都遵守总线协议，则在任何时间只有 1 个模块驱动任何总线。但是刚加电后，所有模块都进入未知状态。加电后时钟振荡器需要一些时间才能稳定，因此，可能没有可靠的时钟使所有模块顺序进入已知状态。在任何情况下，如果两个以上的模块加电后试图向相反的方向驱动总线，那么输出驱动的冲突可能会造成电源的急剧短路问题。这可能妨碍芯片正常启动。

测试接口

AMBA 的一个可能用途就是通过测试接口控制器对模块测试方法提供支持。这

种方法将外部测试仪作为 ASB 总线上的一个主控器，使 AMBA 上的每个模块都可以单独测试。

支持测试模式的惟一要求就是测试仪能够通过 32 位双向端口访问 ASB 总线。如果存在对外部存储器和外围器件的 32 位双向数据总线接口，这就足够了。如果片外数据接口只有 16 位或 8 位宽，那么就需要其他信号（如地址线）给出 32 位宽以用于测试操作。

测试接口允许 ASB 地址和数据总线的控制使用两个测试请求输入信号（TREQA 和 TREQB）上定义的协议，并且控制器中有地址锁存及增值器。一个设计合理的宏单元模块允许其一组超过 32 位的接口信号能够操作。例如，ARM 宏单元有 13 个控制和配置输入、32 位数据输入、15 位状态输出和 32 位地址和数据输出。在一个由 TREQA 和 TREQB 控制下进行状态转换的有限状态机的自动控制下，测试向量的加入及其响应遵循一定的顺序。

AMBA 宏单元测试方法与基于 JTAG 的测试方法（参见 8.6 节）相比，虽然可能缺少一般性，但由于使用并行测试接口从而降低了测试成本。

APB 总线

ASB 提供了相对较高性能的片上互连，适合于处理器、存储器和具有复杂内建接口的外围宏单元。对非常简单且性能低的外围器件，接口的开销就太高了。作为 ASB 总线的补充，APB（Advanced Peripheral Bus）是一个简单的静态总线，为非常简单的外围宏单元提供最小的接口。

APB 总线包括地址（PASSR[n：0]，通常不需要全部 32 位）；读数据和写数据（PRDATA[m：0]和 PWDATA[m：0]，其中 m 为 7、15 或 31）总线，不会宽于要连接的外围器件必需的总线宽度；一个读/写方向指示信号（PWRITE）；单独的外围选通信号（PSELx）；一个外围时间选通信号（PENABLE）。APB 传送时序基于 PCLK。所有 APB 器件由信号 PRESETn 复位。

所有地址和控制信号都根据时间选通信号建立及保持，以保证局部译码时间，当局部使能时进行选择的操作。基于简单寄存器映射并作为从器件的外围器件可用最小的逻辑开销直接接口。

AHB 总线

AHB 总线计划在高性能系统中取代 ASB 总线，如那些基于 ARM1020E 的系统（在将 12.6 节中介绍）。

AHB 总线和 ASB 总线有下列不同的特点：

- AHB 总线支持分时处理。有很长响应延迟的从机在准备传送的数据时，让出总线使其从事其他传送操作。
- 使用单一时钟沿控制所有操作，有利于综合和设计验证（通过使用静态时序分析和其他相似工具）。
- 使用中心多路器总线方案而不是三态驱动的双向总线（参见图 8.12）。
- 支持更宽的 64 位或 128 位数据总线配置。

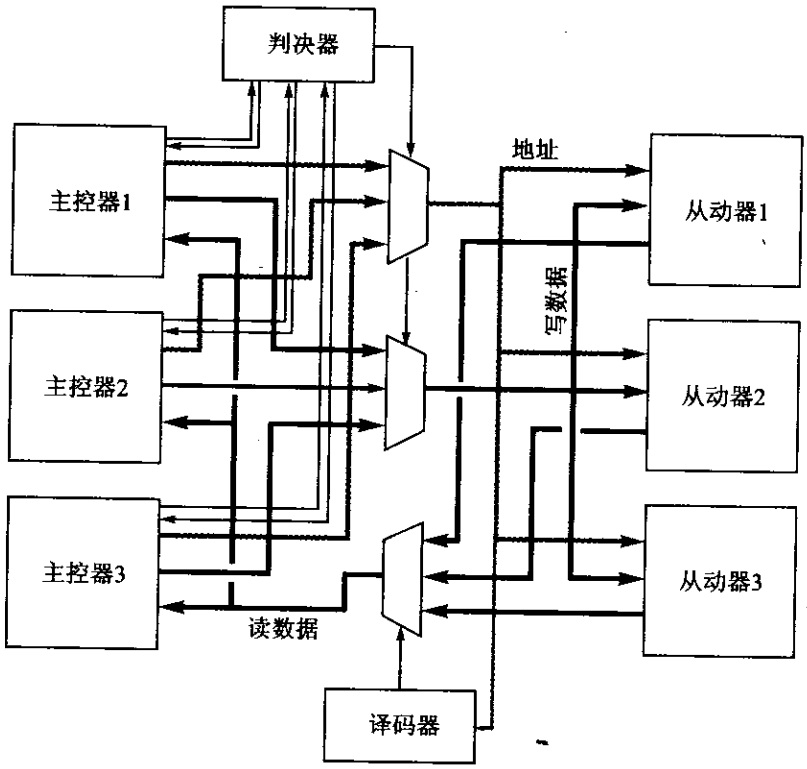


图 8.12　AHB 多路器总线方案

多路器总线方法可能会带来很多额外走线。但双向总线会带来很多设计问题，对综合系统则问题更多。例如，当芯片特征尺寸缩小时，布线延时成为决定性能的主要因素。单向总线通过重复插入驱动器而得到好处，但这对双向总线就非常困难。

8.3　ARM 参考外围规范

到目前为止，本章描述的对系统开发的支持基本集中于测试以及提供对处理器和系统状态的低层访问。AMBA 提供了在同一芯片上连接硬件元件的系统方法。但对每一个新的芯片，软件开发还必须从最基本的开始。

如果系统开发想从一个高的起点开始，例如根据特定的实时操作系统来开发软件，则需要许多部件以支持基本的操作系统功能。**ARM 参考外围规范**(reference peripheral specification)定义了一个基本部件集，提供了一个操作系统能够运行的基本框架，但又给专用系统留下了足够的扩展空间。

ARM 参考外围规范的目标是便于在符合规范的实现之间移植软件，从而提高新系统软件开发的起步层次。

基本部件

参考外围规范定义了下列部件：

- 存储器映射：它允许中断控制器、计数定时器和复位控制器的基地址浮动，但定义了各种寄存器从这些基地址的偏移。
- 中断控制器：带有已定义的功能集，包括一个已定义的发送/接收通信通道中断机制(但通道本身的机制没有定义)。
- 计数定时器：带有多种已定义的功能。
- 复位控制器：带有已定义的启动行为、加电复位检测、中断等待暂停模式和一个标识寄存器。

至于这些部件与哪种 ARM 核结合则没有规定，因为这不影响系统程序开发的模型。

存储器映射

系统必须定义中断控制器(**ICBase**)、计时定时器(**CTBase**)、复位和暂停控制器(**RPCBase**)的基址。

参考外围规范没有定义这些地址，但定义了所有寄存器相对于这些基址的地址。

中断控制器

中断控制器提供了最多 32 个电平敏感的 IRQ 源和一个 FIQ 源的状态的使能、禁止和测试的统一方法。每一个中断源都有一个中断使能的屏蔽位，它能使中断有效。还定义了从 ICBase 加固定偏移的存储器地址，用来检测未屏蔽、屏蔽的中断状态以及设置与清除中断源。

参考外围规范定义了 5 个 IRQ 中断源，分别是通信接收和发送功能、两个计数定时器，以及一个能直接由软件产生的 IRQ 中断源(主要是使能一个 FIQ 处理程序以产生一个 IRQ)。

计数定时器

需要两个 16 位计数定时器，但可能还会增加。这些定时器由相对于 CTBase 有固定偏移的寄存器控制。由系统时钟控制计数操作，并有 0、4、8 位可选的预计数(也就是说其输入频率是系统时钟频率的 1、16 或 256 分频)。

每个计数定时器有一个控制寄存器用于选择预计数，使能或禁止计数器，以及设定自由的或周期的工作模式；还有一个 Load 寄存器用于设定计数器计数的初始值。对 Load 寄存器的写操作将计数值初始化。当定时器减到 0 时产生一个中断。对“清除”寄存器的写操作将清除中断。在自由模式下，计数器值为 0 时继续减操作；而在周期模式下，重新读入 Load 寄存器的值并减计数。

计数器当前值可以随时从“数值”寄存器中读出。

复位及暂停控制器

复位及暂停控制器中包括相对于 RPCBase 有固定偏移的寄存器。可读寄存器给出标识及复位状态信息，包括是否发生加电复位。可写寄存器可以设置或清除复位状态(不包括加电复位状态位，这只能通过加电复位硬件设置)，清除复位映射(例如将

ROM 从 0 地址，即加电后需要使用的 ARM 复位向量地址，切换到正常存储器映射），以及将系统置为最小功耗的暂停模式直至中断将它唤醒。

系统设计

任何包含这个基本部件集的 ARM 系统都可以支持一个适当配置的操作系统内核。其后的系统设计包括进一步加入专用外围接口及软件，在这个功能基础往上构建。

由于多数应用需要这些部件，使用参考外围规范作为系统开发的起始点的开销很小，并且从能工作的系统开始是非常有益的。

8.4　建立硬件系统原型的工具

今天 SoC 设计工程师面对的任务是可怕的。芯片上门的数目已经是百万门级，且持续以指数增长。尽管市场上有最好的软件设计工具，在上市时间的限制下，设计者不能保证这种复杂度的系统是完全测试的。

就像以前指出的那样，解决这个问题的第一步是使设计的重要部分是已设计好的部件。设计复用可以将新设计的工作量减小为芯片上门总数的一小部分。片上互连使用如 AMBA 总线的系统方法进一步减小了设计工作量。但是仍旧有很难解决的问题，例如：

- 如何保证从不同的地方选用的复用块集成在一起能够正确工作？
- 如何保证专用系统达到通常有复杂的实时要求的性能指标？
- 如何能在芯片完成前进行软件设计？

使用软件工具仿真时，系统性能往往比最终系统低几个数量级，使软件开发及整个系统的验证不符合实际。

一个解决所有这些问题的可行方案是使用硬件原型：建造一个包括所需要部件的硬件系统。这个硬件系统不考虑对最终系统的功耗及尺寸限制，只提供一个系统验证及软件开发的平台。ARM 综合器(integrator)就是这样一个系统，VLSI 公司的快速硅原型(rapid silicon prototyping)也是一个这样的系统。

快速硅原型

VLSI 公司生产了一种开发系统称为“快速硅原型”。这个系统的基础是使用专门开发的参考芯片，每一个参考芯片都提供非常丰富的片上元件，并支持片外扩展。这个系统可以用于开发 SoC 设计的原型。目标系统建模分为两个步骤：

1) 对选择的参考芯片进行重构，使那些目标系统不需要的片上模块无效。
2) 目标系统需要但参考芯片又没有的模块由片外扩展来实现。这可以由已有的、具有必要功能的集成电路来实现，也可由 FPGA 实现(通常由高级语言如 VHDL 综合实现)。

使用已有的块，再用标准总线(如 AMBA)完成内部连接，减小了生产最终芯片时的技术风险。参考芯片中所有的模块都是可综合的部件。将所需功能的高级语言描述

加上用于配置 FGPA 的高级语言源代码，删除没有配置的功能，重新综合产生目标芯片。这个过程如图 8.13 所示。

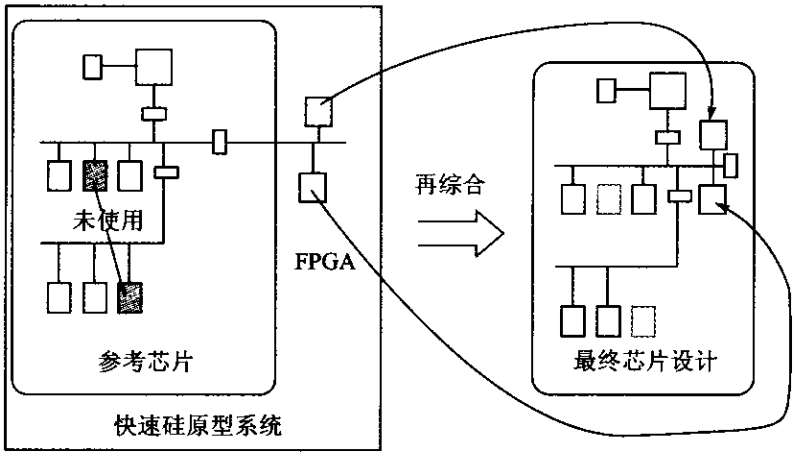


图 8.13 快速硅原型理论

显然，这种方法依赖于参考芯片包含适当的关键元件，如 CPU 核，还可能包含信号处理系统(在目标系统需要它的场合)。尽管理论上能够设计单个参考芯片使其包含每种不同的 ARM 处理器核(包括所有不同的 Cache 和 MMU 配置)，但实际上这样一种芯片即使作为原型机也是不经济的。因此，较好的方法是仔细选择参考芯片上的元件，以保证它能够覆盖大范围的系统。另外，正在构建包含不同的 CPU 和其他关键核的不同参考芯片，用于不同的应用领域。

8.5 ARM 仿真器 ARMulator

ARMulator 是 2.4 节介绍的交叉开发工具套件的一部分。它是一个 ARM 处理器的软仿真器，不需要 ARM 处理器芯片就能调试和评价 ARM 代码。

ARMulator 用于嵌入式系统开发。它支持系统的各种部件的高级原型，以支持软件开发和不同体系结构的评价。它由 4 个部分构成，即

- 处理器核模型：能够仿真现有的各种 ARM 核，包括 Thumb 指令集。
- 存储器接口：它能够模拟目标存储器系统的各种特征。提供各种原型以支持快速建模，但接口需要全定制以实现需要的细节。
- 处理器接口：它支持定制的协处理器模型。
- 操作系统接口：使个别系统调用可以由主机处理，或在 ARM 模型上模拟。

处理器核的模型包含远程调试接口，因此，处理器和系统状态对于 ARMsd(ARM 符号调试器)是可见的。通过这个接口，程序可以被装载、运行及调试。

系统建模

使用 ARMulator 可以建立一个完整的、时钟周期精确的系统软件模型，包括

Cache、MMU、物理存储器、外围器件、操作系统和软件。因为这可能是系统最高级别的模型，因此，最适合于完成设计方案的初始评价。

在设计适当稳定时，硬件开发就可以进入时序精确的 CAD 环境，但软件开发可以继续使用基于 ARMulator 的模型(可能从周期精度时序提升到指令精度时序以提高性能)。

在具体硬件设计时，原先软件模型中的一些时序假设有可能被证明是不能满足的。在设计进程中，重要的是保持软件模型的同步，使软件开发基于可得到的最精确的时序来估算。

目前，使用多个不同抽象级的目标系统计算机模型来支持复杂系统的开发是比较通用的做法。除非低抽象级模型是由更抽象的模型自动综合出来的，否则维护模型之间的一致性会花费很多精力。

8.6　JTAG 边界扫描测试结构

在专用嵌入式系统芯片产品的开发中有两个比较困难的地方，一个是 VLSI 元件的生产测试，另一个是组装后印制板的生产测试。

印制板测试可由 IEEE 标准 1149，即“标准测试访问接口和边界扫描结构”来解决。这个标准描述了一个用于数字电路引脚信号电平访问和控制的 5 引脚串行协议，并扩展到测试芯片上的电路。这个标准由“测试联合行动组”(Joint Test Action Group，简称 JTAG)开发。它描述的结构又称为 JTAG 边界扫描或 IEEE 1149。

JTAG 边界扫描测试接口的一般结构如图 8.14 所示。核逻辑与引脚之间的所有信号都被串行的扫描路径截取。在正常工作模式下，扫描路径能将逻辑核连接到引脚上；在测试模式下，扫描路径能够读取原始数据并以新的数据代替。

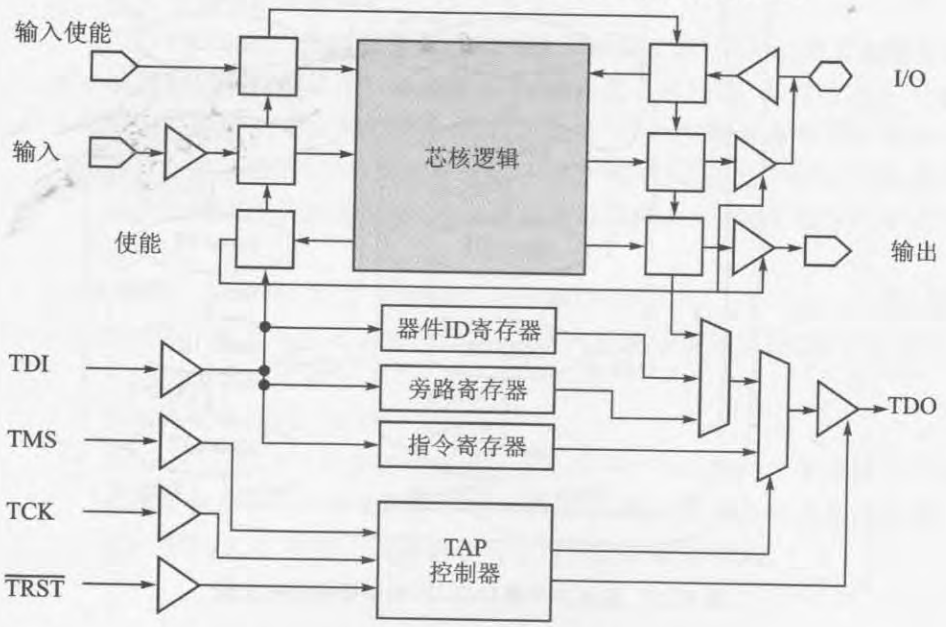


图 8.14　JTAG 边界扫描结构

测试信号

支持这个测试标准的芯片必须提供 5 个专用信号接口：

- $\overline{\text{TRST}}$：测试复位输入，用于测试接口的初始化。
- TCK：测试时钟，独立于任何系统时钟，用于控制测试接口的时序。
- TMS：测试模式选择信号，控制测试接口状态机的操作。
- TDI：测试数据输入，给边界扫描链或指令寄存器提供数据。
- TDO：测试数据输出。输出边界扫描链的采样值。在芯片串行测试时，将数据传送给下一个芯片。

若印制板上有多个支持 JTAG 的芯片，则测试电路的通常组织方法是将 $\overline{\text{TRST}}$、TCK 和 TMS 并行连接到每个芯片，将一个芯片的 TDO 连接到另外一个芯片的 TDI。这样使印制板测试接口同样具有如上所述的 5 个信号。

TAP 控制器

测试访问端口（**TAP**，Test Access Port）控制器控制测试接口的操作。这是个由 TMS 控制状态转换的状态机。其状态转换如图8.15所示。所有状态都有两个出口，

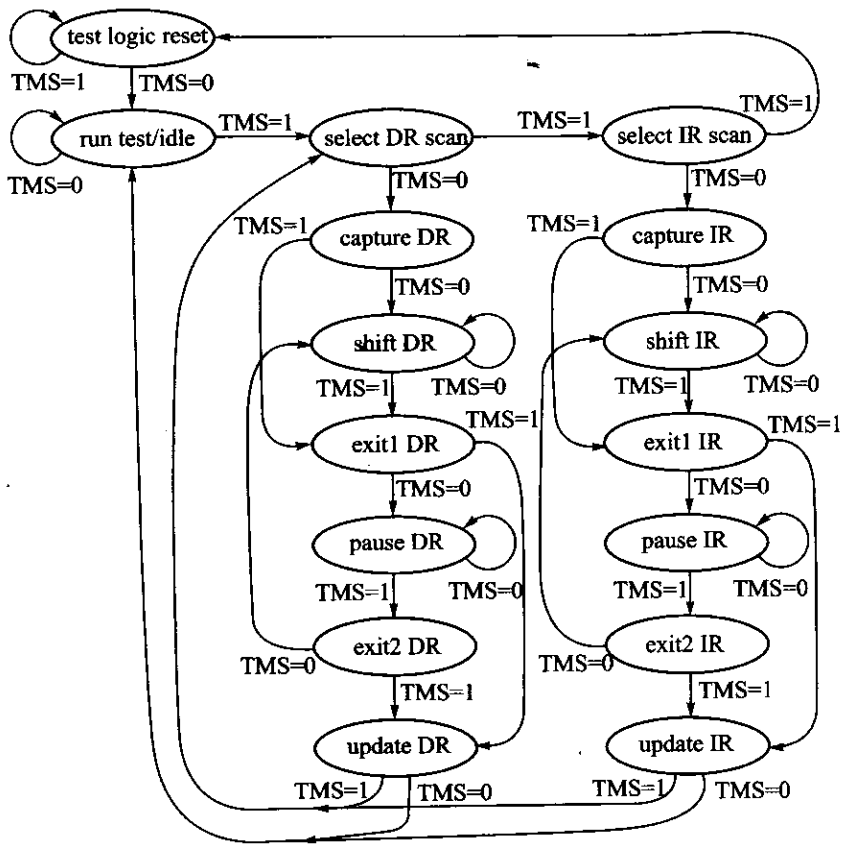


图 8.15 测试访问接口(TAP)控制器状态转换图

因此，转换可以由 1 个信号 TMS 控制。状态转换图中两个主要路径分别控制数据寄存器(DR，Data Register)和指令寄存器(IR，Instruction Register)的操作。

数据寄存器(DR)

特定芯片的行为由测试指令寄存器的内容决定。测试指令寄存器可用来选择各种不同的数据寄存器：

- *器件标识寄存器*：读出固定在芯片内的标识码。
- *旁路寄存器*：将 TDI 经过 1 个时钟周期的延迟连接到 TDO，使测试台可以快速访问同一电路板上测试环路中的另一个器件。
- *边界扫描寄存器*：截取核逻辑与引脚之间的所有信号。它由一个个寄存器位组成，如图 8.14 中连接到核逻辑的方块所示。
- 另外芯片上还可以有一些其他寄存器用于其他功能的测试。

指　令

JTAG 测试系统的正常操作过程是向指令寄存器送入指令，然后使用数据寄存器进行测试。测试指令说明下一步要进行的测试种类及测试要使用的数据寄存器。

指令可以是公开的，也可以是私用的。公开指令已经定义且用于通用测试。标准说明了一个兼容器件必须支持的最小公开指令集。私用指令用于片上专用测试，标准没有规定这些指令如何使用。

公开指令

兼容器件必须支持的最小集的公开指令如下：

- BYPASS：器件将 TDI 经 1 个时钟延时连接到 TDO。这个指令用于同一个测试环中其他器件的测试。
- EXTEST：将边界扫描寄存器连接到 TDI 和 TDO 之间。边界扫描寄存器能够捕获和控制引脚状态。参考图 8.15 中的状态转换图，引脚状态在 Capture DR 状态时被捕获并在 Shift DR 状态下通过 TDO 引脚从寄存器中移出。在捕获的数据被移出的同时，新的数据从 TDI 引脚移入，并在 Update DR 状态时加到边界扫描寄存器的输出端(从而加到输出引脚上)。这条指令用于支持板级连接测试。
- IDCODE：将 ID 寄存器连接到 TDI 和 TDO 之间。在 Capture DR 状态时，器件 ID(厂家赋与的固定标识数字，包括产品编号及版本码)拷贝到寄存器，并在 Shift DR 状态时移出。

  其他可以支持的公开指令还包括：
- INTEST：将边界扫描寄存器连接到 TDI 和 TDO 之间。寄存器可以捕获和控制核逻辑的输入及输出状态。应注意的是，输入完全由提供的值驱动。其他操作同 EXTEST 相似。这条指令用于内部逻辑核的测试。

PCB 测试

JTAG 测试电路的主要目的是测试印制板上走线与焊盘之间的连接。表面贴装封装不需要印制板上的通孔，这使得印制板的测试变得很困难。以前，探针测试仪能够从印制板背面接触到 IC 封装的全部引脚来检测其连接。表面封装技术使引脚间距缩小了，而且只能从印制板的元件面才能接触到走线，这使得探针技术变得不适用了。

若表面封装元件有 JTAG 测试接口，这个接口可用于控制芯片的输出并观测其输入(使用 EXTEST 指令)，而不依赖于芯片的正常功能。这样，可以检查芯片之间的板级连接。如果板上同时包含不支持 JTAG 测试接口的器件，则还需要使用探针技术。但是探针与 JTAG 接口相结合可以降低制造产品测试仪的成本及难度。

VLSI 测试

高复杂度集成电路在用于产品前须进行广泛的产品测试以鉴别失效器件。IC 测试仪是非常昂贵的仪器，每个器件在测试仪花费的时间是生产成本的重要部分。由于 JTAG 测试电路以串行方式工作，因此，不能高速地将测试向量加到核逻辑上，并且不能以器件正常工作速度通过 JTAG 接口加入测试向量以测试器件的性能。

因此，JTAG 结构不是解决 VLSI 产品测试所有问题的通用解决方案。但是它仍然能解决下列问题：

- JTAG 端口能用于 IC 内部电路功能测试(如果支持 INTEST 指令)。
- 能较好地控制 IC 引脚用于参数测试(测试输出缓冲的驱动能力、漏电、输入阈值等等)。这种测试只需要 EXTEST 指令，而这是所有 JTAG 兼容器件必须具有的指令。
- 可以用于访问内部扫描路径，以提高从引脚难以访问的内部节点的可控制性和可观察性。
- 可以用于访问片上调试功能，并且不需要额外引脚，也不会干扰系统的功能。这使用在 ARM EmbeddedICE 调试结构中。ARM EmbeddedICE 将在后面简单描述并在 8.7 节中详细介绍。
- 提供了下面将要介绍的基于宏单元设计的功能测试方法。

以上用途都是印制板产品测试这一基本功能的之外的附加功能。

EmbeddedICE

ARM 调试结构 EmbeddedICE 是基于 JTAG 测试端口的扩展，其详细结构将在 8.7节中介绍。EmbeddedICE 模块引入了附加的断点和观测点寄存器。这些数据寄存器可以通过专用 JTAG 指令来访问。一个跟踪缓冲器也可用相似的方法访问。ARM 核宏单元周围的扫描路径可以将指令加入 ARM 流水线，并且不会干扰系统的其他部分。这些指令可以访问及修改 ARM 和系统的状态。

调试结构提供了传统的在线仿真系统的大部分功能，可以调试一个复杂系统中的 ARM 宏单元。由于使用了 JTAG 测试访问端口控制调试硬件，使芯片不需要额外的引脚。

宏单元测试

使用大量复杂的、已设计好的宏单元，再加上一些专用用户逻辑来设计复杂的系统芯片的趋势越来越猛。ARM 处理器核本身就是一个宏单元。其他宏单元可以从 ARM 公司及其半导体合作伙伴或其他第三方供应商得到。在这种情况下，系统芯片的设计师对宏单元的了解有限，而每一个宏单元的产品测试向量主要依赖于宏单元提供商。

由于宏单元埋藏在系统芯片内部，设计时会遇到采取什么样的方法将得到的测试向量依次加到宏单元上的问题。设计中用户逻辑部分也需要产生测试图形，但我们认为设计者了解这部分逻辑。

有多种方法将测试图形加到宏单元上，即

- 可以提供这样一种测试模式，通过多路器使每个宏单元的信号依次连接到系统芯片的引脚上。
- 片上总线可以支持每个连接到总线上的宏单元的直接测试访问(参见 8.2 节)。
- 每个宏单元可以有一个边界扫描路径。使用扩展的 JTAG 结构，测试图形可以通过扫描路径加到宏单元上。

最后一种方法如图 8.16 所示。芯片外围的边界扫描路径支持公开的 EXTEST 操作，而其他的与宏单元一起设计的、环绕每个宏单元的路径用于加入功能测试图形。芯片中的用户专用逻辑可以有自己的扫描路径，或者如图所示那样，其所有接口信号必须截取一个已有的扫描路径。

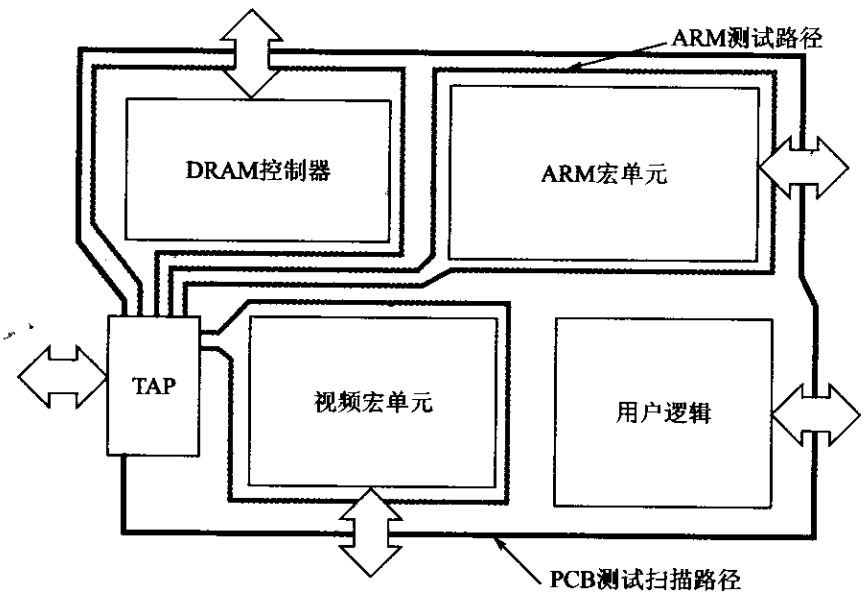


图 8.16　用于宏单元测试的可能的 JTAG 扩展

应该认识到，尽管功能测试是完全可行的，用于宏单元测试的扫描路径方法与使用 JTAG 边界扫描链测试芯片上逻辑核有同样的缺点。串行访问速度大大低于通过引脚

的并行访问速度，并且不可能进行全速性能测试。

基于宏单元的系统芯片的、最有希望的产品测试方法是采用片上总线对宏单元进行并行访问(特别是那些专门设计有这种总线访问接口的宏单元)。对于那些需要进行性能测试而又不便于通过片上总线访问的外围宏单元信号，则采用多路器实现外部访问。其他信号和内部状态在需要时使用 JTAG 端口通过扫描链访问。在 8.2 节介绍的 ARM 的 AMBA 总线支持这种测试方法，其测试方法也已在 8.2 节中介绍过。

JTAG 系统仍然是一种非常重要的板级测试方法。这种方法还可以用于逻辑核的内部测试以及访问片上调试工具。大多数 ARM 设计采用了 JTAG，并将它作为其测试和调试方法的重要组成。

8.7 ARM 调试结构

任何计算机系统的调试都是一件复杂的任务。调试有两种基本的方法，最简单的方法是使用如逻辑分析仪一类的测试仪器从外部监视系统；更强有力的方法是使用支持单步执行、设置断点等功能的工具从内部观察系统。

桌面调试

当要调试的系统是一个运行于台式机上的程序时，所有用户接口模块都已准备好，并且调试器本身是运行于这台机器上的另一个软件。当设置断点时用调用调试器来代替目标程序的指令。要记住原始指令，以便当程序的执行越过断点时恢复这条指令。

通常编译器有编译选项以产生扩展调试信息，如符号表。使用符号表，用户就可以在源代码级调试程序，用源代码中的名字而不是用存储器地址对变量寻址。源代码级调试非常有用，与目标代码级调试相比，它对机器环境需要较少的了解。

软件调试工具的一个普遍弱点是缺少观察点工具。一个观察点是一个存储器地址，当这个地址作为数据传送地址访问时会停止执行。由于大多数处理器不支持一个特定地址的捕获(要与存储器管理页失效故障相区别，失效页处理要粗糙得多)而缺少此功能。这是很遗憾的，因为在 C 程序中，一个非常普遍的代码编写错误是程序中一些不相关部分的错误指针造成数据破坏。若没有观察点工具，这是很难发现的。

嵌入式调试

如果系统是嵌入式的，则调试变得更为困难。由于系统中可能没有用户接口，因此，调试工具必须在远程主机上运行，并通过某种通信方式与目标机器连接。如果代码存放在 ROM 中，那么由于不能进行写操作，指令不能简单地由调试工具调用来代替。

一个标准的解决方案是采用在线仿真器 **ICE**(In-Circuit Emulator)。目标系统中的处理器被取掉，代之以与仿真器的连接。仿真器上的处理器可以是一个相同的芯片，也可以是一个有更多引脚的变型芯片(对内部状态有更高的可观察性)。但是仿真器上还有缓冲器，以便将总线上的活动复制到跟踪缓冲器(保存若干周期之内所有引脚上每个时钟周期的信号)，以及各种硬件资源，可以用来观察像执行通过一个断点这类的特

殊事件。跟踪缓冲器和硬件资源由运行在主桌面系统上的软件来管理。

当一个触发事件发生时，跟踪缓冲器冻结。这样用户可以观察感兴趣点附近的活动状态。主软件会显示跟踪缓冲器的数据，用户可以观察处理器和系统状态并进行修改，使其看起来与一个桌面系统调试器尽可能相似。

调试处理器核

ICE 方法依赖于系统中确实有能够去除并由 ICE 代替的处理器芯片。很明显，如果处理器是一个复杂的系统芯片上许多宏单元中的一个，那么这一点就是不可能的。

尽管使用软件模型如 ARMulator 仿真在物理实现前可以去除许多设计错误，但通常在仿真时运行整个软件系统是不可能的，并且精确描述所有实时约束也是困难的。由此看来，对整个硬件和软件系统进行调试是很有必要的。但怎样才能做到这一点呢？确定一个硬件开销可接受的、最好的、全面的策略仍然是一个研究热点，但是，最近几年在开发实用方法方面取得了相当的进展。本节余下的部分介绍 ARM 公司提出的方法。

ARM 调试硬件

为了提供与典型的 ICE 相似的调试工具，用户必须能够设计断点和观察点（对于运行在 ROM 中和 RAM 中的代码），检查并修改处理器和系统的状态，观察处理器在感兴趣点活动的轨迹，而且所有这些都可在有着良好用户界面的桌面系统上方便地做到。ARM 系统使用的跟踪机制与其他调试系统不同，这将在 8.8 节中讨论。本节将注重断点、观察点及状态监视的资源。

目标系统与主机之间通过扩展 JTAG 测试端口的功能来实现通信。为了便于板级测试，大多数设计中都有 JTAG 测试引脚。通过这些引脚访问测试硬件不需要额外的专用引脚，节省了芯片的宝贵资源以备将来使用。JTAG 扫描链用于访问断点及观察点寄存器，并向处理器施加指令来访问处理器及系统的状态。

实现断点及观察点寄存器的硬件代价非常小，一般是产品所能够接受的。主机系统运行标准的 ARM 开发工具，并通过一个串行口和/或并行口与目标系统通信。在主机串行口与目标的 JTAG 端口之间有专用的协议转换硬件。

除了断点和观察点事件外，当系统级事件发生时也可能希望使处理器停止。调试硬件有一些外部输入来实现这一点。

包含这些工具的片上单元称为 EmbeddedICE 模块。

EmbeddedICE

EmbeddedICE 模块包括两个观察点寄存器和控制与状态寄存器。当地址、数据和控制信号与观察点寄存器编程数据相匹配时，观察点寄存器可以中止处理器。由于比较是在屏蔽控制下进行的，因此，当 ROM 或 RAM 中的一条指令执行时，任何一个观察点寄存器可配置为能够中止处理器的断点寄存器。比较及屏蔽逻辑如图 8.17 所示。

链　接

每个观察点可以观察 ARM 地址总线、数据总线、$\overline{\text{trans}}$、$\overline{\text{opc}}$、$\overline{\text{mas}}[1:0]$和 $\overline{\text{r}}/\text{w}$ 控制信

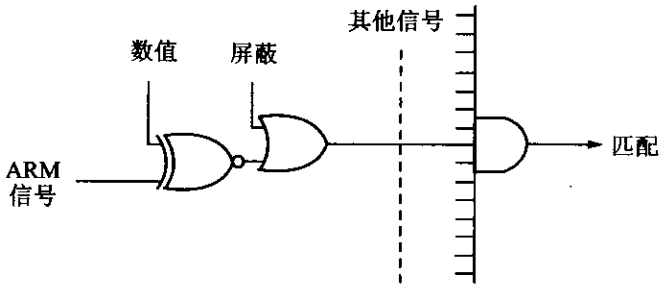


图 8.17 EmbeddedICE 信号比较逻辑

号的特定组合值。如何任何一个组合值匹配，则中止处理器。另外一种方式是把两个观察点链接起来，只有第一个观察点先匹配了，当第二个观察点再匹配时才使处理器中止。

寄存器

EmbeddedICE 寄存器通过 JTAG 测试端口使用专用扫描链编程。扫描链为 38 位长，包括 32 个数据位、5 个地址位和 1 个控制寄存器是读还是写的 $\overline{r}$/w 位。地址位指定特定的寄存器，具体的映射关系如表 8.1 所列。

JTAG 扫描链的用法如图 8.18 所示。当 TAP 控制器进入 updateDR 状态时进行读或写。

表 8.1 EmbeddedICE 寄存器映射

地 址	宽 度	功 能
00000	3	调试控制
00001	5	调试状态
00100	6	调试 comms 控制寄存器
00101	32	调试 comms 数据寄存器
01000	32	观察点 0 地址值
01001	32	观察点 0 地址屏蔽
01010	32	观察点 0 数据值
01011	32	观察点 0 数据屏蔽
01100	9	观察点 0 控制值
01101	8	观察点 0 控制屏蔽
10000	32	观察点 1 地址值
10001	32	观察点 1 地址屏蔽
10010	32	观察点 1 数据值
10011	32	观察点 1 数据屏蔽
10100	9	观察点 1 控制值
10101	8	观察点 1 控制屏蔽

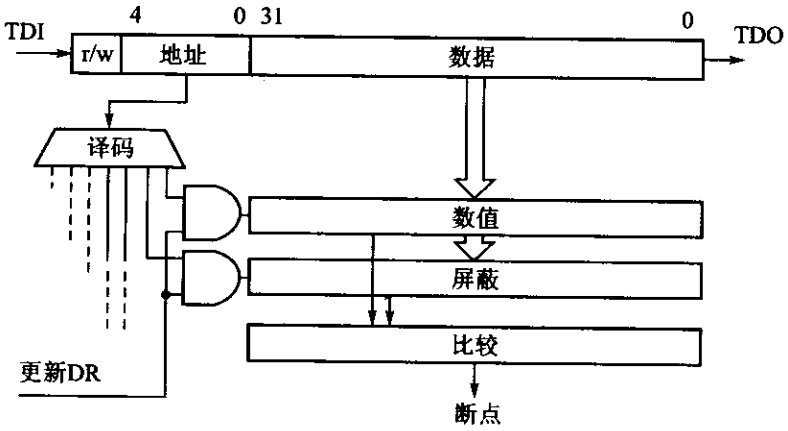


图 8.18　EmbeddedICE 寄存器读和写结构

访问状态

EmbeddedICE 模块允许程序在指定点中止，但不允许直接观测或修改处理器或系统状态。这通过另外的也是通过 JTAG 端口访问的扫描路径实现。

访问处理器状态的方法是中止处理器，再在处理器指令序列中强制插入一条指令，如多寄存器 Store 指令。然后通过扫描链向处理器加入时钟，使得处理器将寄存器送到数据端口。每个寄存器的值都被扫描链采集并移出。

由于扫描路径能提供的速度很低，使系统中有些位置不能读取，造成系统状态的收集比较困难。这时可在处理器中预装一些合适的指令，然后以系统速度访问这些位置。这样将所需的系统状态传送到处理器的寄存器，因而可以通过 JTAG 端口以上面所述方法把它传送到外部调试器中。

调试 comms 端口

除了断点和观察点寄存器外，EmbeddedICE 模块还有一个**调试 comms 端口**。通过这个端口，运行在目标系统上的软件可以与主机通信。运行在目标系统上的软件将comms端口视为一个 6 位控制寄存器和 32 位可读写寄存器，可以使用对协处理器 14 的 MRC 和 MCR 指令进行访问。主机将这些寄存器视为 EmbeddedICE 寄存器，其映射如表 8.1 所列。

调　试

基于 ARM 的、包括 EmbeddedICE 模块的系统芯片通过 JTAG 端口和协议转换器与主计算机连接。这种配置支持正常的断点、观察点以及处理器和系统状态访问，(除上面介绍的 comms 端口以外)这是程序员在本地或基于 ICE 的调试中习惯采用的方式。采用适当的主机软件，以较少的硬件代价得到完全的源代码级调试能力。

惟一的缺点是不能对代码进行实时跟踪。这是将在下一节介绍的嵌入式跟踪宏单元 ETM(Embedded Trace Macrocell)的功能。

8.8 嵌入式跟踪

在调试实时系统时，若不能观察其实时操作，则对应用程序的调试将非常困难。EmbeddedICE 宏单元提供的断点及观察点工具不足以完成这个功能，因为使用它们时处理器将偏离正常执行序列，破坏了软件的实时行为。

这里所需要的是：在程序执行时通过产生对处理器地址、数据及控制总线活动的跟踪来获得观察处理器全速操作情况的能力。问题是这需要巨大的数据带宽。一个以 100 MHz 运行的 ARM 处理器产生的接口信息超过 1 GB/s。将这些信息从芯片取出需要大量的引脚。芯片产品要具有这种能力是不经济的。因此需要专用的开发设备，这对开发新的 SoC 应用的成本产生不利的影响。

跟踪压缩

ARM 公司采用的方法是使用智能跟踪压缩技术减小接口带宽。例如：

- 大多数 ARM 地址是顺序的，因此，将每个地址送出芯片是没有必要的。相反，在大多数周期只送出顺序指示信息，而只在发生转移时才送出完整地址。
- 如果有片外逻辑能够访问处理器上执行的代码，那么处理器什么时候执行了一条转移指令及转移目标将都是可知的。必须送出芯片的惟一信息是是否执行了转移。
- 只有当转移目标不可预知时才需要较完整的地址信息，如子程序返回指令和表转移指令。即使如此，也只需要发送那些有变化的低端地址位。
- 这样处理后的地址有很高的突发性。可以使用一个 FIFO 缓冲器来减缓数据速率，以便这些必需的地址信息可以用稳定的速率以 4、8 或者 16 位包(packet)来传送。

通过使用一系列相似的技术，ARM 嵌入式跟踪宏单元可以将跟踪信息压缩到必要的长度，使这些信息依配置的不同通过 9、13 或 21 个引脚传送到片外。在不需要输出跟踪时，这些引脚可以用于其他目的。

实时调试

一个实时调试的完整解决方案如图 8.19 所示。EmbeddedICE 单元支持断点和观察点功能以及主机和目标软件的通信通道。嵌入式跟踪宏单元压缩处理器接口信息并通过跟踪端口送到片外。这两个单元都用 JTAG 端口控制。外部的 EmbeddedICE 控制器用于将主机系统连接到 JTAG 端口，外部跟踪端口分析器使主机系统与跟踪端口对接。主机通过一个网络可以与跟踪端口分析器和 EmbeddedICE 二者连接。

用户控制断点和观察点的设置及各种跟踪功能。可以跟踪所有应用软件，也可以跟踪某一特定程序。触发条件可以指定，跟踪采集可以在触发之前、之后，或以触发为中心。可以选择跟踪是否包括数据访问。跟踪采集可以只是数据访问的地址、只是数据本身，也可以是两者都有。

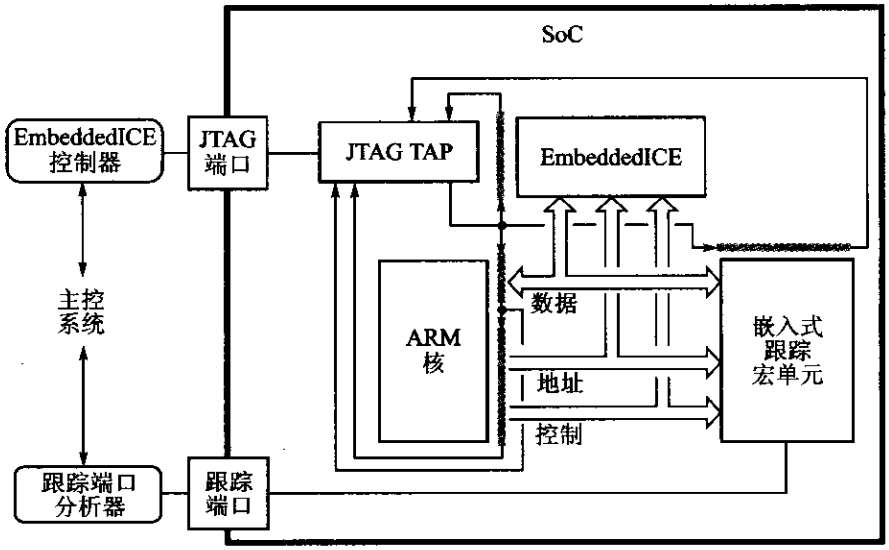


图 8.19　实时调试系统的组织

嵌入式跟踪选项

如前所述，嵌入式跟踪宏单元可以按照几种不同的配置进行综合，使这个单元的功能根据成本（以门和引脚的数目估算）进行折衷。

- 最小系统需要 5 个引脚送出流水线信息，4 个引脚送出数据（再加上 JTAG 接口的 5 个引脚）。这对执行跟踪是足够的，但只支持有限的数据跟踪，触发和过滤功能也有限制。一个 9 字节的 FIFO 用于平滑数据传输速率。该实现的硬件成本大约为 15 K 门。
- 最大系统使用 5 个引脚送出流水线信息，16 个引脚送出数据（也要再加上 JTAG 端口的 5 个引脚）。除最坏情况外，能够跟踪执行流及所有数据活动。一个 40 字节的 FIFO 用于平滑数据流，硬件成本大约为 50 K 门。

在这两种极端情况之间有多种中间配置。所有配置都允许使用外部引脚（也就是从芯片上其他逻辑输入）来控制跟踪触发，并从 EmbeddedICE 断点逻辑触发。

跟踪溢出

即使实现了全部的嵌入式跟踪宏单元，在有些情况下，跟踪 FIFO 缓冲器仍可能溢出。当发生这种情况时，单元可以设置为，或者停止处理器，或者放弃跟踪。无论怎样处理，实时跟踪是做不到了，尽管这只是暂时发生在 FIFO 耗尽时。

所发生事情的实质是信息带宽超过了压缩算法将信息压缩到跟踪端口带宽容量的能力。如果发生了这种情况，则必须修改过滤设置以减少要跟踪的数据。

N-Trace

实时跟踪技术是由 ARM 公司、VLSI Technology 公司和 Agilent Technology 公

司(以前是 HP 公司的一部分)合作开发的。VLSI Technology 公司提供了一个产品名为 N-Trace 的嵌入式跟踪宏单元的可综合版本。

跟踪端口分析器

跟踪端口分析器可以是一个传统的逻辑分析仪。但 Agilent Technology 公司开发了一种 ARM 专用低成本跟踪端口分析仪。其他开发商也会陆续提供类似的系统。

跟踪软件工具

嵌入式跟踪宏单元是使用软件通过 JTAG 端口进行配置的,所使用的软件是 ARM 软件开发工具的一个扩展。跟踪数据从跟踪端口分析仪下载并使用源代码信息解压。然后由带有分散数据访问的汇编列表表示,并反链接到源代码。

有了 EmbeddedICE 和嵌入式跟踪工具,ARM SoC 开发者得获得了传统的在线仿真器(ICE)工具能够提供的所有功能。通过这些技术能够全面观察应用代码的实时行为,并且能够设置断点,检查并修改处理器寄存器和存储器单元,还总是能够严格地反链接到高级语言源代码。

8.9 对信号处理的支持

许多使用 ARM 处理器作为控制器的应用系统还需要有良好的数字信号处理性能。一个典型的 GSM 移动手持电话就是这样的一个例子。第一代基于 ARM 的设计经常在同一个芯片上同时集成 ARM 核和一个 DSP 核。系统设计师需要仔细考虑一个系统功能是在 DSP 核上实现还是在 ARM 核上实现更有利。

DSP 核的编程模式与 ARM 核有很大的不同。DSP 核使用几个分开的数据存储器,程序员需要仔细地调度其内部流水线以得到最大的吞吐量。并行执行的 ARM 代码和 DSP 代码的同步是一个非常复杂的任务。

ARM 公司推出了 ARM 体系结构的两种不同的扩展:Piccolo 协处理器和 ARM v5TE 体系结构的信号处理指令集扩展,试图简化那些同时需要控制器和信号处理功能的系统设计任务。

Piccolo

Piccolo 协处理器是一个成熟的 16 位信号处理机,它使用 ARM 协处理器接口与 ARM 核协同操作,以便从存储器读取和向存储器传递操作数和结果,同时它还可以执行自己的指令集。

Piccolo 的组织如图 8.20 所示。通过 ARM 协处理器接口加载操作数并保存结果,因此,ARM 核必须产生合适的地址。输入/输出缓冲器使多个 16 位数据能在 1 条指令中传送,并且数据按对传送,以充分利用 ARM 的 32 位总线带宽。输入缓冲器保存数据直到信号处理代码调用它们为止,并且对这些值可以不按缓冲器的次序访问。

Piccolo 寄存器集保存的操作数可以是 16 位、32 位或 48 位宽。4 个 48 位寄存器

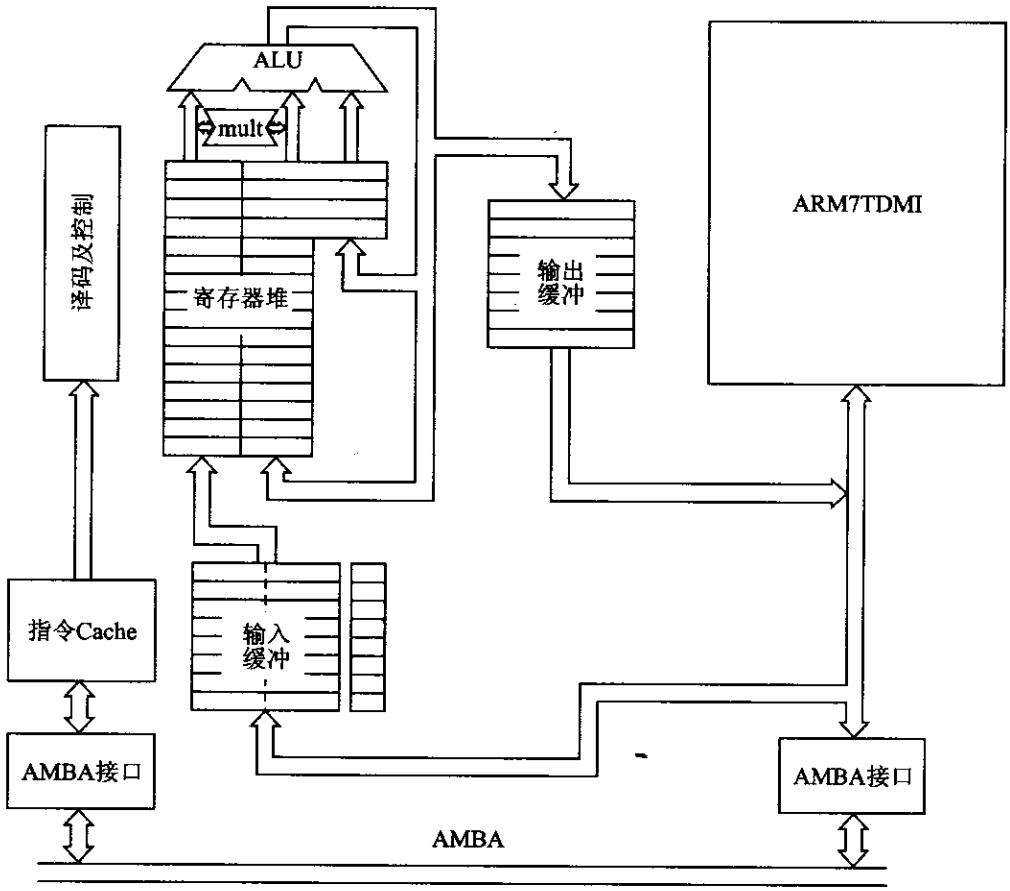


图 8.20　Piccolo 的组织

用于保存累加结果(如内部积)以防止溢出。处理逻辑能够在单周期内计算 16×16 的乘法,并将结果累加到一个 48 位累加寄存器中。它还对定点操作提供很好的支持,并支持饱合算法。

对寄存器文件(file)中保存的值进行的信号处理操作由单独的指令集指定。这些指令由 Piccolo 通过 AMBA 总线,从存储器中调入到局部的指令 Cache 中。

Piccolo 体系结构的一个目标是以寄存器、指令 Cache 以及输入/输出缓冲器的形式提供充足的局部存储。这样一条 AMBA 总线就能很好地支持其吞吐量。相反,传统的信号处理器使用两个独立的数据存储器和一个分开的指令存储器。这样,与 ARM 处理器核结合传统的信号处理核系统相比,Piccolo 具有更直接的编程模型。但是,有些情况下仍需要注意 ARM 代码与 Piccolo 代码同步。在一个强化的信号处理应用中,ARM 核忙于操作数和结果的传送,从而不可能同时完成大量的控制功能。ARM 体系结构 v5TE 指令集扩展提供了一个更加直接的编程模型。

v5TE 信号处理指令

ARM 体系结构 v5TE 定义的信号处理扩展指令集首先在 ARM9E-S 可综合核中实现。与 Piccolo 使用的指令集相比,它使用了不同的解决问题的方法。这里使用的所

有指令都是仔细挑选的，加到 ARM 本来的指令集中，以对信号处理应用中使用的数据类型提供更好的内在支持。

ARM 编程模型在 v5TE 体系结构中的扩展如图 8.21 所示。在 CPSR 的第 27 位增加了一个标记 Q，相应的所有 SPSR 也增加了一个标记 Q。Q 是粘接(sticky)溢出标记，可以由 v5TE 的特定指令设置，并由相应的 MSR 指令(参见 5.14 节)复位。“粘接”的意思是，标记一旦设置就一直保持，直到由 MSR 指令明确复位为止。这样，可以执行一系列指令，而只在最后检查一次标记 Q(使用 MRS 指令)，测试是否在指令序列中的某一点发生了溢出。

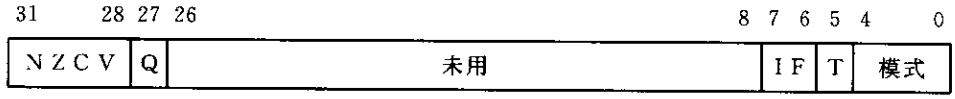


图 8.21 ARM v5TE PSR 格式

信号处理指令分为两组：乘法和加法/减法。加法/减法指令使用饱合算法，也就是当结果超过数据类型所能表示的范围时，返回一个能够表示的、最近似的值(同时标记 Q 置位)。这与传统的处理器算法(以及由 C 定义的模 2^{32} 算法数据类型)相反。在传统算法中，当结果稍大于最大值时，其值接近最小值。而这里当结果稍大于最大值时，则简单地返回最大值。在典型的信号处理算法中，这减小了误差，产生一个合理的结果。

v5TE 乘法指令

乘法指令提高了处理器处理 16 位数据类型的能力，并且使一个 32 位 ARM 寄存器可以保存两个 16 位值。因此，它们能够有效地访问寄存器的低 16 位和高 16 位中的数据。这些乘法指令的二进制编码如图 8.22 所示。

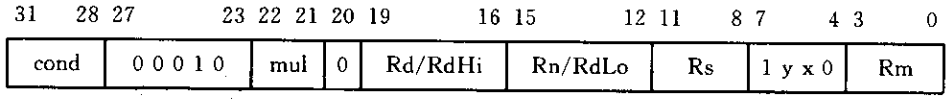


图 8.22 v5TE 体系结构乘法指令的二进制编码

这个格式支持的指令如下：

```
SMLAxy{cond}    Rd, Rm, Rs, Rn          ; mul = 00
```

这条指令计算两个有符号 16 位数的 16×16 乘积。这两个 16 位数是 Rm 的低 16 位(x=0)或高 16 位(x=1)与 Rs 的低 16 位(y=0)或高 16 位(y=1)。32 位乘积与 Rn 中的 32 位值相加，结果保存到 Rd。在汇编格式中，用 B 代替 x 和 y 表示低半字，用 T 表示高半字。

```
SMLAWy{cond}    Rd, Rm, Rs, Rn          ; mul = 01, x=0
```

这条指令计算 32×16 的乘法，32 位数保存到 Rm 中。16 位数可以是 Rs 的低 16 位(y=0)或高 16 位(y=1)。48 位乘积的高 32 位与 Rn 中的 32 位值相加，结果保存到

Rd 中。

```
SMULWy{cond}    Rd, Rm, Rs                ; mul=01, x=1, Rn=0
```

这条指令计算 Rm 中的 32 位数与 Rs 的低 16 位值(y=0)或高 16 位值(y=1)的 32×16 乘积。48 位结果的高 32 位保存到 Rd 中。

```
SMLALxy{cond}    RdLo, RdHi, Rm, Rs        ; mul = 10
```

这条指令计算 Rm 的低 16 位(x=0)或高 16 位(x=1)值与 Rs 的低 16 位(y=0)或高 16 位(y=1)值的两个有符号数的 16×16 乘积。32 位结果与 RdHi：RdLo 中的 64 位数相加，结果保存到 RdHi：RdLo 中。

```
SMULxy{cond}    Rd, Rm, Rs                ; mul=11, Rn= 0
```

这条指令计算两个 16 位有符号数的 16×16 乘法。这两个 16 位数是 Rm 的低 16 位(x=0)或高 16 位(x=1)与 Rs 的低 16 位(y=0)或高 16 位(y=1)。32 位结果保存到 Rd 中。

上面的所有指令不影响 CPSR 的标志位 N、Z、C 和 V，并且 PC(r15)不能作为操作数或目的寄存器。如果累加时(SMLA 和 SMLAW)溢出，则 CPSR 中的位 Q 置位，但加法使用传统的模 2^{32} 算法而不是饱合算法。

v5TE 加法/减法指令

v5TE 扩展体系结构的另一组指令是使用饱合算法的 32 位加法和减法指令。每种情况下都有一条附加指令，在进行加法和减法之前将一个操作数加倍。这提高了特定信号处理算法的有效性。这些指令的二进制编码如图 8.23 所示。

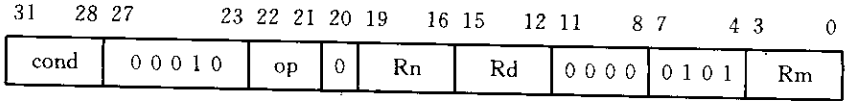


图 8.23　v5TE 体系结构加法/减法指令的二进制编码

对这种格式支持的指令如下：

```
QADD{cond}      Rd, Rm, Rn                ; op = 00
```

这条指令完成 Rm 和 Rn 的 32 位饱合加法，结果保存到 Rd。

```
QSUB{cond}      Rd, Rm, Rn                ; op = 01
```

这条指令完成从 Rm 减 Rn 的 32 位饱合减法，结果保存到 Rd。

```
QDADD{cond}     Rd, Rm, Rn                ; op = 10
```

这条指令先将 Rn 加倍(使用饱合算法)，然后完成其结果与 Rm 的 32 位饱合加法，结果保存到 Rd。

```
QDSUB{cond}     Rd, Rm, Rn                ; op = 11
```

这条指令先将 Rn 加倍(使用饱合算法)，然后完成从 Rm 减其结果的 32 位饱合减

法，结果保存到 Rd。

上面的所有指令不影响 CPSR 的标志位 N、Z、C 和 V，并且 PC(r15)不能作为操作数或目的寄存器。如果饱合加法或减法溢出，或者 Rn 加倍时产生溢出，则 CPSR 中的位 Q 置位。

v5TE 代码的例子

为了表明这些指令的用法，考虑保存在存储器中的两个 16 位有符号数向量的点积问题，分别在支持 v5TE 扩展的 ARM9E-S 核和不支持 v5TE 扩展的 ARM9TDMI 核上的计算。点积计算是信号处理应用中常用的过程。为了减小误差，应使用饱合算法。用于中央循环的 v5TE 代码为

```
loop    SMULBB    r3, r1, #2        ; 16×16 乘法
        SUBS      r4, r4, #2        ; 循环计数器减 1
        QADD      r5, r5, r3        ; 饱合的×2 及累加
        SMULTT    r3, r1, r2        ; 16×16 乘法
        LDR       r1, [r6], #4      ; 取两个乘数
        QDADD     r5, r5, r3        ; 饱合的×2 及累加
        LDR       r2, [r7], #4      ; 取两个被乘数
        BNE       loop
```

从代码举例中可以看出以下几个要点：

- 指令进行了调度以避免流水线阻塞。对于 ARM9E-S，这意味着 Load 或 16 位乘法的结果不应在随后的周期中使用。
- 尽管操作数是 16 位半字，但它们成对地以 32 位字读入。与半字读入相比，这更有效地使用了 ARM 的 32 位存储器接口，并且 v5TE 乘法指令可以直接对寄存器的半字进行独立访问。
- 饱合的加倍与累加指令用来在累加前对乘积调整。这是非常有用的，因为在信号处理中使用的定点算法通常假定操作数在－1～1 的范围内，但某些算法需要大于 1 的系数。加倍操作产生从－2～2 的有效范围，这对大多数算法是足够的。

性能比较

ARM9E-S 中的单周期 32×16 乘法使它能在 10 个周期内完成上述循环，其中 4 个周期是循环的开销(循环计数器减 1 和在循环末转移回去)。每个循环计算两个乘积，因此，每个乘积需要 5 个周期。若解开循环(复制代码在 1 个循环中计算更多的乘积)，则每个乘积可以减少到 3 个周期。在 ARM9TDMI 中完成 1 个乘积最少需要 10 个周期，差不多慢 3 倍。造成这个差别的一部分原因是 ARM9TDMI 的乘法慢，另一部分原因是处理 16 位操作数的效率低。其他原因是对饱合测试及校正需要额外的指令。

8.10　例题与练习

例题　8.1

估算通过 JTAG 和 AMBA 接口测试 ARM 核所需要的测试向量数目的比例。

ARM 核大约有 100 个连接接口(32 个数据、32 个地址、控制、时钟、总线和模式等等)。JTAG 接口是串行的。如果测试仪允许用一个向量说明一个 TCK 脉冲,则需要 100 个向量向 ARM 核加入一个并行测试图形。AMBA 测试接口分 5 个部分访问 ARM 外围接口(见 8.2 节"测试接口"一段),在标准测试仪上只需要 5 个测试向量。

因此,JTAG 接口看起来需要 20 倍的测试向量(应记住的是 JTAG 主要应用于印制板测试,而不是 VLSI 产品测试)。实际上基于 JTAG 的 EmbeddedICE 和 AMBA 接口都包括优化以提高向 ARM 核送指令的效率,这是测试的主要需求。考虑到这些因素,在估算时需要更为详细的分析。

练习　8.1.1

总结基于复杂宏单元的系统芯片的 VLSI 产品测试的问题所在,并讨论各种方案的相对优缺点。

练习　8.1.2

说明 VLSI 产品测试、印制板测试和系统调试的差别,以及在这些测试中如何使用 JTAG 测试端口。JTAG 方法在什么方面有效性高?在什么方面有效性差?

练习　8.1.3

AMBA 解决什么问题?ARM 参考外围规范解决什么问题?它们有什么相关性?

练习　8.1.4

勾画一个嵌入式系统芯片的开发计划,并给出在哪一阶段,ARMulator、AMBA、参考外围规范、EmbeddedICE 和 JTAG(i)被设计到芯片中,和/或(ii)被用于辅助开发过程。

第 9 章　ARM 处理器核

本章内容综述

ARM 处理器核是系统中的引擎，它从存储器读取 ARM(可能还有 Thumb)指令并执行这些指令。ARM 核非常小，典型的芯片面积只有几个平方毫米。现代的 VLSI 技术使得许多附加的系统部件可以集成在同一芯片中。它们可能与处理器核密切相关，如 Cache 和存储器管理硬件；也可能与系统部件无关，如信号处理硬件。它们甚至可能包含更多的 ARM 处理器核。在这些所有的元件中，处理器核最为突出，因为它是最为密集和复杂的部件，对软件开发和调试工具提出最高的要求。在确定一个新系统时，正确选择处理器核是最关键的决定之一。

本章将讲述目前主要的 ARM 处理器核产品，它们有着不同的价格、复杂度和性能，从中可以选择出最有效的解决方案。

在许多应用中，处理器核需要得到 Cache 和存储器管理子系统的支持。一些组合了这些元件的标准配置在第 12 章中介绍。

此外，与 ARM 兼容的处理器核将在第 14 章中讲述。这些处理器核是研究样品，还不是商业产品。

9.1　ARM7TDMI

*ARM7TDMI 是目前低端的 ARM 核，具有广泛的应用，其最显著的应用为数字移*动电话。

ARM7TDMI 是从最早实现了 32 位地址空间编程模式的 ARM6 核发展而来的。这种 ARM 核现在已被取代。ARM6 所使用的电路技术使它很难稳定地在低于 5 V 的电源电压下工作。ARM7 弥补了这一不足，而且在一个很短的时间内增加了 64 位乘法指令，支持片上调试、Thumb 指令集和 EmbededICE 观察点硬件，开发出 ARM7TDMI。

这项命名的由来如下：

- **ARM7**，32 位整数核 ARM6 的 3V 兼容版本，且具有：
- Thumb 16 位压缩指令集；
- 支持片上调试(Debug)，使处理器能够停止以响应调试请求；

- 增强型乘法器(Multiplier),与前代相比具有较高的性能且产生 64 位的结果;
- Embedded ICE 硬件以支持片上断点和观察点。

ARM7TDMI 组织

ARM7TDMI 组织如图 9.1 所示。ARM7TDMI 核是使用了 3 级流水线(参看 4.1 节)的基本的 ARM 整数核,它具有许多重要的特性及扩展。

- 它实现 ARM 体系结构版本 4T,支持 64 位结果的乘法,半字、有符号字节的 Load 和 Store 以及 Thumb 指令集。
- 它包含了 EmbeddedICE 模块以支持嵌入式系统调试(这部分内容已在 8.7 节讲解)。

因为调试硬件由 JTAG 测试访问端口访问,故 JTAG 控制逻辑(已在 8.6 节讲解)被认为是处理器宏单元的一部分。

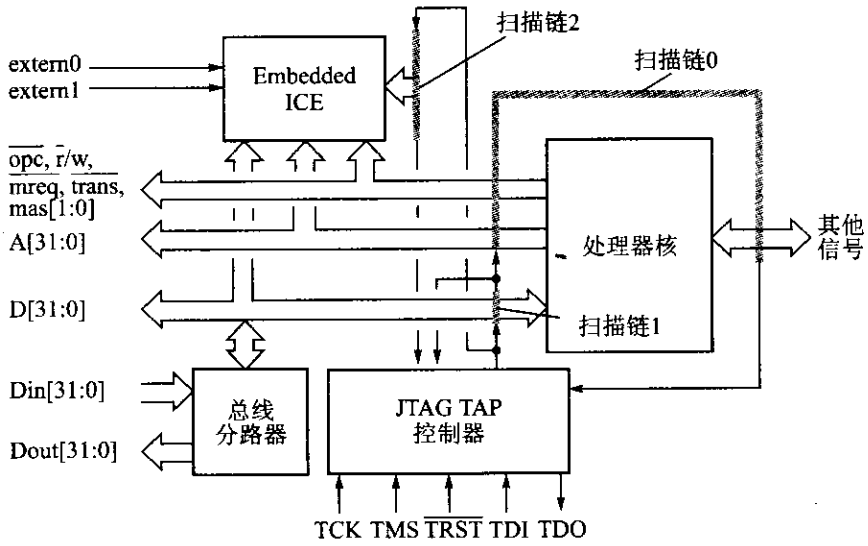


图 9.1　ARM7TDMI 的组织

硬件接口

ARM7TDMI 的硬件接口信号如图 9.2 所示,众多的信号数量让人看起来眼花缭乱。它暗示行为的复杂性而掩盖了基本 ARM 接口固有的简单性。从数量上说,接口信号主要是 32 位地址和数据总线。下面将会讲到,存储器接口将使用这些接口和一些控制信号。其他信号则专用于诸如片上调试、JTAG 边界扫描扩展等更深奥的功能。

在图 9.2 中,接口信号按功能分组。下面说明每组的作用,在适当时还要提到单个信号和接口时序的信息。

时钟控制

处理器所有的状态变化由存储器时钟 mclk 控制。尽管这个时钟可以由外部操纵

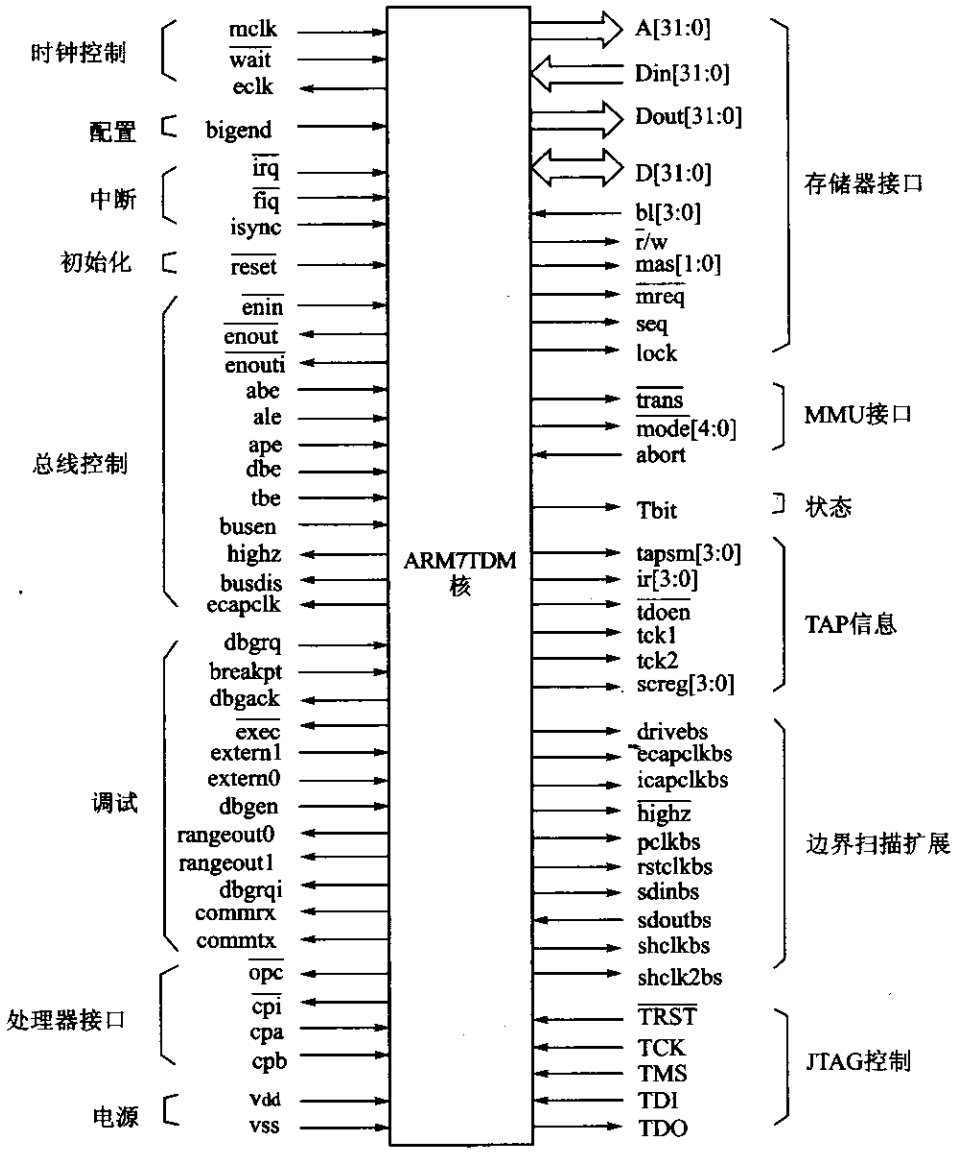


图 9.2 ARM7TDMI 核的接口信号

以便使处理器等待低速访问，但是通常只是提供一个自由的时钟，使用$\overline{\text{wait}}$跳过时钟周期。内部时钟实际上正好是 mclk 和$\overline{\text{wait}}$的逻辑“与”，因此，只有当 mclk 为低时$\overline{\text{wait}}$才能变化。

eclk 时钟输出反映了处理器核使用的时钟，因此，它一般反映了 mclk 在$\overline{\text{wait}}$门控后的行为。但在调试模式下它也反映了调试时钟的行为。

存储器接口

存储器接口包括 32 位地址(A[31：0])、双向数据总线 D[31：0]、分离的数据输出 Dout[31：0]和数据输入 Din[31：0]总线以及 10 个控制信号。

- $\overline{\text{mreg}}$　指示一个需要存储器访问的处理器周期。
- seq　指示存储器地址与前一周期使用的地址连续(也可能相同)。
- lock　指示处理器应保持总线,以确保 SWAP 指令读相和写相的不可分割性。
- $\bar{\text{r}}$/w　指示处理器执行读周期还是写周期。
- mas[1：0]　是对存储器访问大小的编码,指出访问的是字节、半字或字。
- bl[3：0]　由外部控制的使能信号,作用于数据输入总线上 4 个字节中每个字节的锁存,这使得少于 32 位宽的存储器易于实现接口。

指示存储器时钟周期类型的信号$\overline{\text{mreg}}$和 seq 要尽可能早地发给存储器控制逻辑,以便确定如何处理存储器访问。表 9.1 给出了对这两个信号的 4 种可能组合的解释。当顺序周期跟在非顺序周期后面时,地址将是非顺序周期的地址加上 1 个字(4 字节);在顺序周期跟在内部或协处理器寄存器传送周期后面时,地址和前一个周期没有变化。在一个典型的存储器组织中,增量的情况可以连同前一地址的信息一起,用来准备存储器进行快速的顺序访问。在地址保持不变的情况下,可以利用这一点在前一周期开始一个全存储器访问(因为既不是内部也不是协处理器寄存器传送周期使用存储器)。

表 9.1　ARM7TDMI 周期类型

$\overline{\text{mreq}}$	seq	周　期	应　用
0	0	N	非顺序存储器访问
0	1	S	顺序存储器访问
1	0	I	内部周期——总线与存储器非活动
1	1	C	协处理器寄存器传输——存储器非活动

关键接口信号的时序如图 9.3 所示。这些信号的用途和存储器接口逻辑的设计在 8.1 节已作了进一步讨论,在那里给出了特定的例子。

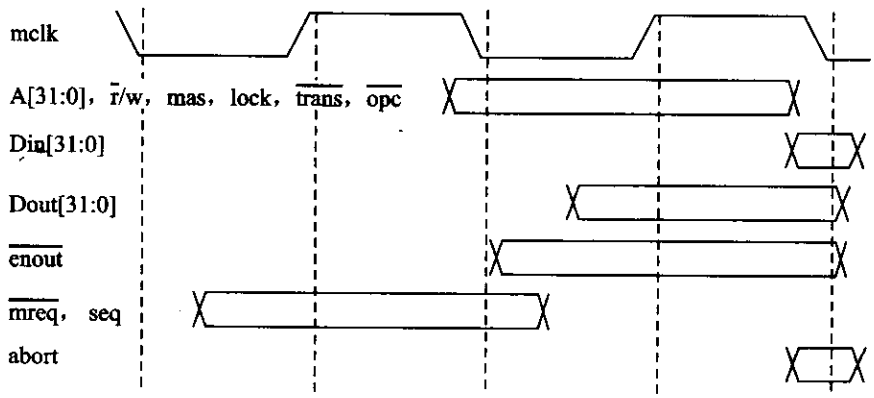


图 9.3　ARM7TDMI 核存储器和 MMU 接口时序

MMU 接口

MMU 的接口信号提供的信息用来控制对存储器区域的访问。$\overline{\text{trans}}$(转换控制)

信号指明处理器是用在用户($\overline{trans}$=0)模式还是特权($\overline{trans}$=1)模式,使得存储器的一些区域能被限制为仅用于监控访问,在适当的情况下用户和监控代码可以使用不同的转换表(尽管很少这样)。当需要关于操作模式更多的详细信息时,$\overline{mode}$[4:0]反映CPSR的低5位(反相),尽管存储器管理在这一级很少使用;当调试时,详细的模式信息可能最为有用。

当一个访问不被允许时,向abort输入端发出信号。abort及数据必须在时钟周期结束前有效。如图9.3所示。

一个中止的存储器访问使处理器执行预取或数据终止,这与在访问期间$\overline{opc}$的值有关。

如果希望支持只能执行的存储器区域,MMU也可以使用$\overline{opc}$信号。但是应该注意,这将阻止对代码区的文字库进行相对于PC的访问。因此,在ARM系统中并不广泛使用对只执行区域的保护(特别是将在11.6节讲述的ARM MMU体系结构中不支持)。

状 态

Tbit输出信号告诉环境当前处理器执行的是ARM指令还是Thumb指令。

配 置

bigend在小端和大端间转换字节的顺序(参看5.1节"存储器组织"一段中对小端和大端的解释)。这个输入信号对处理器的操作方式进行配置。尽管在需要时它可在时钟的第2相变化,但一般不会动态变化。

中 断

两个中断输入对处理器时钟而言可以是异步的,因为在进入处理器的控制逻辑前它们经过同步锁存。快速中断请求$\overline{fiq}$比一般的中断请求$\overline{irq}$有较高的优先级。

初始化

$\overline{reset}$从未知状态启动处理器,自地址00000000_{16}开始执行。

总线控制

通常ARM7TDMI核一经得到新地址就立即发出,以便MMU或存储器控制器有最长的时间来处理它。但是在简单的系统中,地址总线直接连接到ROM或SRAM,需要把原来的地址保持到周期末。处理器核有一个由ape控制的透明锁存器,当外部逻辑需要时它可以给地址重新定时。

ARM7TDMI核执行写周期时用信号$\overline{enout}$来指示。如果外部数据总线是双向的,就用$\overline{enout}$来将dout[31:0]加到总线上。有时希望推迟写操作以使其他部件可以驱动总线。可以使用数据总线的使能信号dbe来确保$\overline{enout}$在这个情况下保持无效。处理器核必须停止(用$\overline{wait}$或时钟展宽),直到总线可以使用为止。dbe按照外部逻辑的要求由外部定时。

其他总线控制信号$\overline{enin}$、$\overline{enouti}$、abe、ale、tbe、busen、highz、busdis和ecapclk执行

各种其他功能，读者应参考相应的 ARM7TDMI 数据手册来了解细节。

Debug 支持

ARM7TDMI 实现了在 8.7 节中讲述的 ARM 调试结构。EmbeddedICE 模块包含断点和观察点寄存器，使运行的代码能够停下来以便调试。这些寄存器通过 JTAG 测试端口使用扫描链 2(见图 9.1)进行控制。当遇到断点或观察点时，处理器停下来并进入调试状态。一旦进入调试状态，就可以使用扫描链 1 强制指令进入指令流水线，检查处理器的寄存器。对所有寄存器的存储将把寄存器的值送到数据总线，它们在数据总线上再用扫描链 1 采样并移出。访问特权模式寄存器需要强制加入指令来改变模式(注意，在调试状态，阻止从用户状态转换到特权模式的障碍已不存在)。

若需检查系统状态，可以让 ARM 以系统速度访问存储器，然后立即切换回调试状态。

Debug 接口

调试接口可扩展集成的 EmbeddedICE 宏单元所提供的功能。它使外部硬件能够支持调试(通过 dbgen)，并发出异步的调试请求(在 dbgrq 端口)或与指令同步的请求(在 breakpt 端口)。外部硬件通过 dbgack 得知处理器核什么时候处于调试模式。内部的调试请求信号在 dbgrqi 输出。

外部事件可以通过 extern0 和 extern1 来触发观察点，而 EmbeddedICE 观察点的匹配则由 rangeout0 和 rangeout1 端口的信号表示。

如果通信发送缓冲器是空的，则在 commtx 端口发出信号；如果接收缓冲器是空的，则在 commrx 端口发出信号。

处理器在$\overline{\text{exec}}$端口指示当前在执行级的指令是否被执行。如果指令没有被执行，就是它的条件码测试失败了。

协处理器接口

协处理器接口信号$\overline{\text{cpi}}$、cpa 和 cpb 在 4.5 节中已经讲过。另外提供给协处理器的信号是 opc，它指示存储器访问是取指令还是取数据。协处理器流水线跟随器使用它来跟踪 ARM 指令的执行。在不需要连接协处理器时，cpa 和 cpb 应该连接高电平。这将使所有协处理器指令产生未定义指令陷阱。

电　源

ARM7TDMI 核应在正常 5 V 或 3 V 电源电压下操作，尽管这依赖于工艺技术的水平和在核中使用的电路设计形式。

JTAG 接口

JTAG 控制信号符合标准的规定。该标准已在 8.6 节中讲过。这些控制信号通过专用引脚连到片外测试控制器。

TAP 信息

这些信号用来支持对 JTAG 系统增加更多的扫描链。关于边境扫描扩展信号在下面详述。

tapsm[3：0]指示 TAP 控制器所处的状态；ir[3：0]给出 TAP 指令寄存器的内容；screg[3：0]是 TAP 控制器当前所选择的扫描寄存器的地址；tck1 和 tck2 形成一对非重迭时钟来控制扩展扫描链；$\overline{\text{tdoen}}$指示何时在 tdo 有串行数据输出。

边界扫描扩展

ARM7TDMI 单元包含完整的 JTAG TAP 控制器，以支持 EmbeddedICE 功能。这个 TAP 控制器能够支持任何通过 JTAG 端口访问的片上扫描电路。因此，提供了接口信号 drivebs、ecapclkbs、icapclkbs、$\overline{\text{highz}}$、pclkbs、rstclkbs、sdinbs、sdoutbs、shclkbs 和 shclk2bs，使任意的扫描路径都可加入到系统中。读者应该参考相关的 ARM7TDMI 数据手册，以详细了解这些信号各自的功能。

ARM7TDMI 核

ARM7TDMI 处理器核的版图如图 9.4 所示。该 0.35 μm 的核在执行 32 位 ARM 代码时的技术特征总结在表 9.2 中。

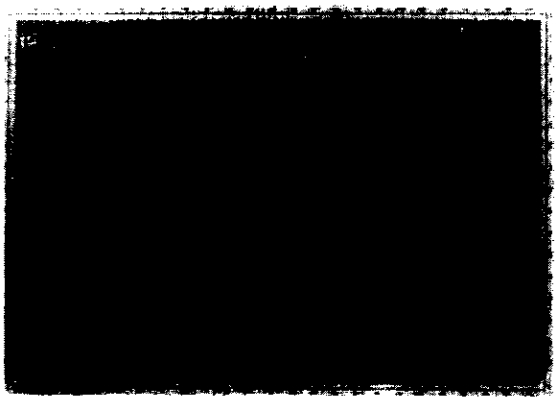


图 9.4 ARM7TDMI 处理器核

采用合适的工艺技术，ARM7TDMI 核得到了非常高的功率效率。另一个使用 0.25 μm工艺技术的芯片在 0.9 V 电源下达到 12 000 MIPS/W 的性能。

表 9.2 ARM7TDMI 性能

工 艺	金属层	Vdd	晶体管	核面积	时 钟	MIPS	功 耗	MIPS/W
0.35 μm	3	3.3 V	74 209	2.1 mm²	0～66 MHz	60	87 mW	690

可综合的 ARM7TDMI

标准的 ARM7TDMI 处理器核是“硬”的宏单元。也就是说，它以物理版图提供，定制为适当的工艺技术。ARM7TDMI-S 是 ARM7TDMI 的一个可综合的版本。它以高级语言模块的形式提供，可以使用任何目标工艺的适当的单元库来综合。因此，它比硬的宏单元更易于转移到新的工艺技术。

综合过程支持关于处理器核功能的若干选项，包括：

- 可省略的 EmbeddedICE 单元。
- 用仅支持产生 32 位结果的 ARM 乘法指令的、较小和较简单的乘法器来替代完全 64 位结果的乘法器。

每个选项都会导致综合出较小的、功能下降的宏单元。综合出的整个核比硬核大 50%，电源效率降低 50%。

ARM7TDMI 应用

ARM7TDMI 处理器核在存储器配置较简单的系统中已有许多应用，通常这些系统包含几千字节的片上 RAM。一个典型的例子是移动电话手机（同一芯片常常融合了复杂的数字信号处理硬件和相关的存储器）。在此应用中，ARM7TDMI 事实上已成为用于控制和用户接口功能的标准处理器。

如果需要非常高的性能，那么具有简单存储器系统的单纯的 ARM7TDMI 已不能满足要求，系统的复杂程度必然要增加。第一步是在 ARM7TDMI 上增加 Cache 存储器，可能以 ARM CPU 宏单元的形式。这将提高软件从片外存储器运行的性能。如果这还不能满足应用对性能的要求，则必须使用更复杂的、能够在高性能水平上运行的 ARM 核。ARM9TDMI 和 ARM10TDMI 就是这样的核，将在本章稍后讲解。

9.2　ARM8

ARM8 核是 1993—1996 年在 ARM 公司开发的，用以满足比 ARM7 的 3 级流水线性能更高的 ARM 核的需求。它后来被 ARM9TDMI 和 ARM10TDMI 替代。但它的设计提出了一些有趣的观点。

正如已经在 4.2 节中讨论的，可以通过以下途径提高处理器核的性能，即

- 增加时钟速率。这需要简化每级流水线的逻辑，因而流水线的级数将增加。
- 降低 CPI（每条指令的时钟周期）。这需要将 ARM7 中占据 1 个以上流水线槽的指令重新实现，以占据较少的流水线槽；或者减少由指令间的相关关系引起流水线的停止。也可以把两者结合起来。

减少 CPI

再重复一下以前提到过的论点：降低 ARM7 核 CPI 的基本问题与 von Neumann 瓶颈有关——任何带有单一指令与数据存储器的存储程序计算机，其性能都要受到可

用的存储器带宽的限制。ARM7 核几乎每个时钟周期都要访问存储器，或者取指令，或者传数据。为了得到比 ARM7 好很多的 CPI，存储器系统必须每个时钟周期读取 1 个以上的数据。为此，或者每个周期从单一存储器读取多于 32 位的数据，或者为指令和数据设置分开的存储器。

双倍带宽存储器

ARM8 保留了统一的存储器(无论以 Cache 的形式还是以片上 RAM 的形式)，但是利用多存储器访问的顺序性，从单一的存储器获得双倍的带宽。假设所连接的存储器在 1 个时钟周期可以读写 1 个字，而在以后顺序访问中与下一次访问一起每半个周期读写 1 个字。典型的存储器组织只花费很少的额外开销就可以提供额外的数据。由于增加的带宽仅限于顺序访问，看来它的作用有限，但是取指令是顺序性是很高的，而且 ARM 的多寄存器 Load 指令也产生顺序的地址(多寄存器 Store 指令也是如此，尽管在 ARM8 中这些指令并不采用双倍带宽存储器)。因此，对典型的 ARM 代码，顺序访问出现的概率是相当高的。

64 位宽度的存储器具有所需的特性，但是由于第 2 个字到达的时间延迟半个时钟周期，因而可以使用 32 位总线。因为 32 位总线与 64 位总线相比，布线需要的面积小，这可以节省芯片面积。

核的组织

ARM8 处理器包含预取指单元和整数数据通路。预取指单元负责从存储器取指令并将其缓冲(以便利用双倍带宽存储器)。它每个时钟周期向整数单元提供一条指令以及它的 PC 值。预取指单元负责转移的预测，使用基于转移方向的静态预测方法(将向后转移预测为“发生转移”，将向前转移预测为“不发生转移”)来猜测指令流将向什么方向发展；整数单元将计算准确的流向，并在需要时向预取指单元发送修改信息。

核的总体组织如图 9.5 所示。双倍带宽存储器通常在片上，在通用器件例如 ARM810(见 12.2 节)中为 Cache 存储器，在嵌入式应用中为可寻址的 RAM。也可以保留传统的单带宽存储器。但是如果在系统中没有某些快速存储器，与简单的 ARM 核相比，ARM8 核将显示不出什么优势。

流水线组织

处理器使用 5 级流水线，预取指单元为第 1 级，整数单元使用余下的 4 级：

1) 指令预取。
2) 指令译码和寄存器读。
3) 执行(移位和 ALU)。
4) 访问数据存储器。
5) 回写结果。

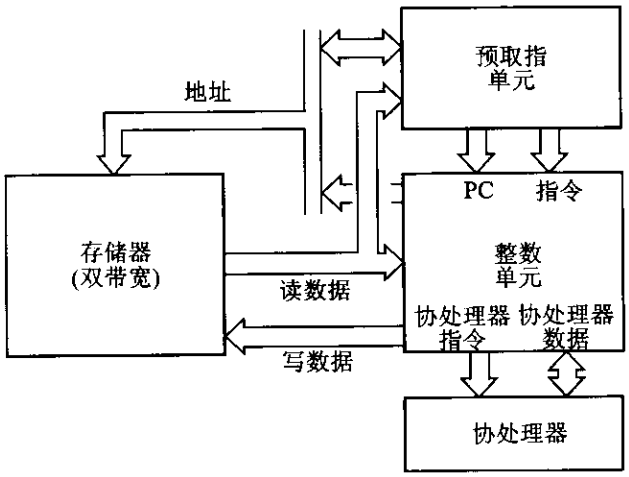


图 9.5　ARM8 处理器核的组织

整数单元的组织

ARM8 整数单元的组织如图 9.6 所示。指令流和相应的 PC 值由预取指单元通过示于图上部的接口提供；系统控制协处理器通过图左侧的专用协处理器指令与数据总线连接；连接数据存储器的接口在图的右侧，包括地址总线、写数据总线和读数据总线。

注意，读存储器数据总线对多寄存器 Load 指令支持双带宽传输，两个寄存器写端口用于多寄存器 Load 指令来存储双带宽数据流。由于多寄存器 Load 指令只传输字，所以，即使一半的数据流绕过字节对准和符号扩展逻辑也没有关系。

ARM8 的应用

ARM8 是一个通用的处理器核，可以容易地由 ARM 公司的许多特许商生产，所以，它没有针对特定的工艺技术进行过多的优化。它在相似的芯片面积上提供了比简单的 ARM7 高得多(2～3 倍)的性能。如果想实现它的全部潜能，则需要支持双倍带宽片上存储器。

ARM8 核的一项应用是构建高性能的 CPU，例如将在 12.2 节介绍的 ARM810。这里双倍带宽存储器为 Cache，芯片还包含存储器管理单元和系统控制协处理器 CP15。

ARM8 管芯

ARM8 核使用 124 554 个晶体管，工作速度高达 72 MHz，采用 3 层金属的 0.5 μm CMOS 工艺。

核的版图可参见图 12.6 的 ARM810 管芯照片，左上部为 ARM8 核。

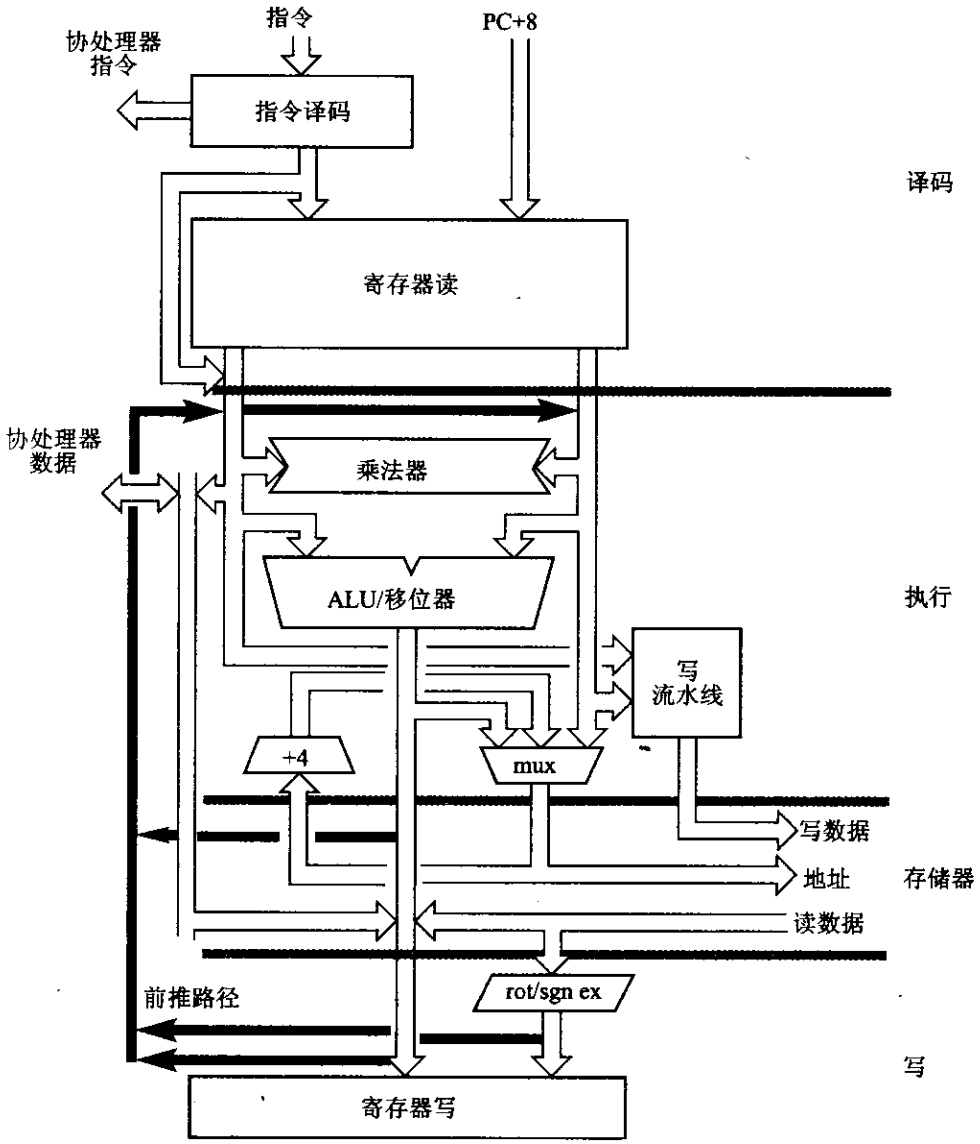


图 9.6 ARM8 整数单元的组织

9.3 ARM9TDMI

ARM9TDMI 核将 ARM7TDMI 的功能显著地提高到更高性能的水平。正如 ARM7TDMI 一样(不像 ARM8),它支持 Thumb 指令集,并含有 EmbeddedICE 模块以支持片上调试。通过采用 5 级流水线以增加最高时钟速率,使用分开的指令与数据存储器端口以改善 CPI(每条指令的时钟数——处理器在 1 个时钟周期所工作的量度),因而改善了性能。

改善的性能

由高性能的需求导致需要将流水线的级数从 3 级(如 ARM7TDMI 使用的)增加到 5 级,并改变存储器接口来使用分开的指令与数据存储器,其基本原理已经在 4.2 节讨论过。

ARM9TDMI 组织

ARM9TDMI 的 5 级流水线主要归功于将在 12.3 节中讲述的 StrongARM 的流水线(本章不讲解作为核的 StrongARM,因为它作为独立核的应用范围有限)。

ARM9TDMI 核的组织如图 4.4 所示。ARM9TDMI 与 StrongARM 核(如图12.8 所示)的主要区别在于:StrongARM 有一个与寄存器读出级并行操作的、专用的转移加法器,而 ARM9TDMI 使用主 ALU 来计算转移目标。这使 ARM9TDMI 为实现转移多损失了一些周期时间,但是得到的核较小和较简单,而且避免了在 StrongARM 中存在的一条非常关键的时序路径(如图 12.9 所示)。StrongARM 是为特定的工艺技术设计的,对它的时序路径可以仔细地管理;而 ARM9TDMI 需要方便地转移到新的工艺,这类关键路径很容易损害可用的最高时钟速率。

流水线操作

ARM9TDMI 5 级流水线的操作如图 9.7 所示。图中与 ARM7TDMI 的 3 级流水线做了比较。该图显示出处理器的主要处理功能如何在增加的流水线级之间重新分配,以便使时钟频率在相同工艺技术的条件下能够加倍(近似)。

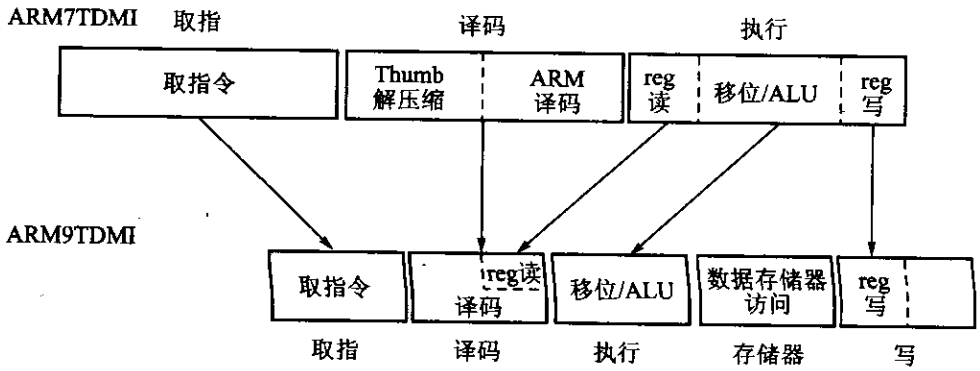


图 9.7　ARM7TDMI 与 ARM9TDMI

重新分配执行功能(寄存器读、移位、ALU 和寄存器写)并不是达到高时钟速率所需的全部。处理器还必须能够在 ARM7TDMI 所用的一半时间内访问指令存储器,还必须重新构造指令译码逻辑,使寄存器读与实际的译码部分同时进行。

Thumb 译码

ARM7TDMI 实现 Thumb 指令集的方法是使用 ARM7 流水线中的松弛时间将 Thumb 指令"解压缩"为 ARM 指令。ARM9TDMI 的流水线非常紧密,没有足够的松

弛时间能先将 Thumb 指令翻译成 ARM 指令再译码；相反，它有硬件直接对 ARM 指令和 Thumb 指令进行译码。

在 ARM9TDMI 的流水线中多出的“存储器”级在 ARM7TDMI 中没有直接的对应级。它的功能由中断流水线的额外的“执行”周期来执行。由于 ARM7TDMI 使用单一的存储器端口进行指令和数据的读写，这种中断是不可避免的。在读写数据时不能取指令。ARM9TDMI 通过设置分开的指令与数据存储器避免了这种流水线中断。

协处理器支持

ARM9TDMI 有一个协处理器接口，允许支持片上浮点协处理器、数字信号处理或其他专用的硬件加速要求。（在它支持的时钟速率下，几乎不可能使用片外协处理器。）

片上调试

如同在 ARM7TDMI 核中那样，ARM9TDMI 核中的 EmbeddedICE 功能也给出了系统级的调试特性（见 6.7 节），同时还有下列附加的特性：

- 支持硬件单步调试。
- 除了 ARM7TDMI 支持的地址/数据/控制条件之外，还可以在异常时设置断点。

低电压操作

虽然第一批 ARM9TDMI 核用 0.35 μm 3.3 V 技术实现，但它的设计已经转移到使用低至 1.2 V 电源的 0.25 μm 和 0.18 μm 的工艺。

ARM9TDMI 核

0.25 μm 的 ARM9TDMI 核在执行 32 位 ARM 代码时的特性综述于表 9.3。核的版图如图 9.8 所示。

表 9.3 ARM9TDMI 的特性

工 艺	金属层	Vdd	晶体管	核面积	时 钟	MIPS	功 耗	MIPS/W
0.25 μm	3	2.5 V	110 000	2.1 mm²	0～200 MHz	220	150 mW	1 500

ARM9TDMI 应用

ARM9TDMI 核有分开的指令与数据端口。原理上它可以把这两个端口连接到单一的统一存储器，但是实际上这样做就没有理由首先选择 ARM9TDMI，而是较小和较便宜的 ARM7TDMI 了。类似，ARM9TDMI 的 5 级流水线所支持的时钟速率高于 ARM7TDMI 的 3 级流水线，虽然不需要利用这样高的时钟速率，但是不这样做就没有理由使用 ARM9TDMI 了。因此，有理由使用 ARM9TDMI 核的任何应用都必须处理复杂的高速存储器子系统。

处理这种存储器要求的最常见方式，正像基于 ARM9TDMI 的各种标准 CPU 核

图 9.8　ARM9TDMI 处理器核

所示范的那样，将是使用分开的指令与数据 Cache 存储器。在 12.4 节将介绍 ARM920T 和 ARM940T CPU 核。在这些 CPU 核中的 Cache 能满足 ARM9TDMI 大部分存储器带宽的要求，并减少外部带宽要求，使经单一 AMBA 总线连接的、传统的统一存储器就能满足其带宽要求。

还有一种解决方案，特别适用于代码对性能有关键影响的嵌入式系统，这就是使用适当数量直接寻址的、分开的指令与数据本地存储器，而不使用 Cache。

ARM9E-S

ARM9E-S 是 ARM9TDMI 核的可综合版本。与“硬”核相比，它实现的是扩展的 ARM 指令集。除了 ARM9TDMI 支持的 ARM 体系结构 v4T 的指令外，ARM9E-S 还支持完整的 ARM 体系结构 v5TE(见 5.23 节)，包括在 8.9 节介绍的信号处理指令集扩展。

ARM9E-S 在相同工艺条件下比 ARM9TDMI 大 30%，使用 0.25 μm CMOS 工艺时面积为 2.7 mm^2。

9.4　ARM10TDMI

ARM10TDMI 是目前 ARM 处理器核的高端产品，在写本书时它仍在开发之中。正如 ARM9TDMI 的性能在相同工艺条件下近似达到 ARM7TDMI 两倍一样，ARM10TDMI 也以 ARM9TDMI 两倍的性能工作。若采用 0.25 μm CMOS 工艺，则在 300 MHz 时钟下预期它的性能将达到 400 dhrystone 2.1 MIPS。

为了达到这样的性能，从 ARM9TDMI 开始，将如下两个途径结合起来(还可见 4.2节的讨论)，即

1）增加最高时钟速率。

2）降低 CPI(每条指令的平均时钟数)。

由于 ARM9TDMI 的流水线已经相当优化了，若不采用诸如超标量执行这类非常复杂的组织结构，怎么能实现这些改善呢？而超标量执行又将损害低功耗和小面积这些 ARM 核的特点。

增加时钟速率

ARM 核可以支持的最高时钟速率是由任意流水线级的、最慢的逻辑路径决定的。

ARM9TDMI 的 5 级流水线已经平衡得很好了(见图 9.7)。5 级流水线中 4 级的负荷都很重。可以将流水线扩展到更多的级。但是这种超级流水线组织的好处往往被因增加流水线相关性而恶化了的 CPI 所抵消，除非采用非常复杂的机制来缩小它的抵消作用。

ARM10TDMI 没有这样做，它保留了与 ARM9TDMI 非常相似的流水线，但是采用特别的方式优化每一级(见图 9.9)，从而支持更高的时钟速率，即

- 通过提前提供下一周期所需的地址，取指和存储器级有效地由一个时钟周期增加到一个半时钟周期。为了在存储器级实现这一点，由一个单独的加法器计算存储器的地址，它可以比主 ALU 更快地产生结果(因为它只实现了 ALU 功能的一个子集)。
- 执行级使用改善了的电路技术和结构以缩短它的关键路径。例如，乘法器并不用 ALU 来解决部分和与乘积项，相反，它在存储器级有它自己的加法器(乘法从不访问存储器，所以这一级是空闲的)。
- 指令译码级是处理器逻辑中惟一不能充分流水化以支持高速时钟的部分，所以在这里增加了发射这一级。

结果产生了比 ARM9TDMI 的 5 级流水线运行更快的 6 级流水线，但是要求所支持的存储器比 ARM9TDMI 的存储器快不了多少。这是很重要的，因为非常快的存储器往往很费功耗。为了有更多译码时间，所增加的流水线级只是在执行未预料的转移时才增加流水线的相关性。由于增加的流水线级出现在读存储器之前，它不会带来新的操作数相关，也不需要新的前推路径。在有转移预测机制的情况下，这个流水线的 CPI 与 ARM9TDMI 流水线非常相似，但是支持更高的时钟速率。

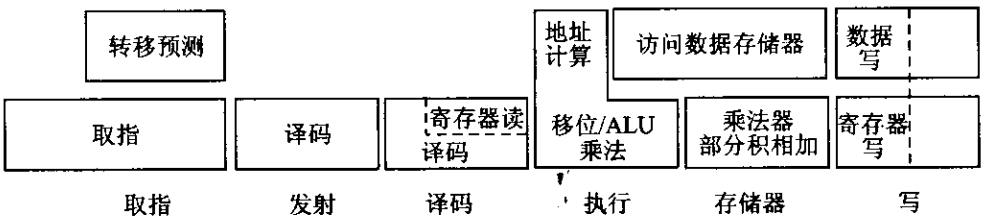


图 9.9 ARM10TDMI 的流水线

降低 CPI

以上所叙述的流水线改进方案支持提高 50%的时钟速率，而不损失 CPI。这是个好的起步，但是其性能还没有达到所需的 100%的提高。因此，在增加时钟速率之前必须改善 CPI。

任何改善 CPI 的计划必须从考虑存储器带宽开始。ARM7TDMI(几乎)在每个时钟周期使用其单一的 32 位存储器，所以，ARM9TDMI 改为哈佛存储器组织以释放更多的带宽。ARM9TDMI(几乎)在每个时钟周期使用其指令存储器。虽然它的数据存储器只有大约 50%的利用率，但很难利用这一点来改善 CPI。指令带宽必须以某种方式来增加。

ARM10TDMI 采用的途径是使用 64 位存储器。这有效地消除了指令带宽瓶颈，使得能够在处理器的组织中加入若干改善 CPI 的特性：

- 转移预测(branch prediction)：以上讨论的转移预测逻辑只是简单地为了维持流水线效率的需要，ARM10TDMI 的转移预测逻辑功能更强。因为按照每个周期两条指令的速率来取指，所以，转移预测单元(它在流水线中取指一级)可以经常在指令发射之前识别它们，并把它们从指令流中去除，将转移的周期代价降低到零。
- ARM10TDMI 采用**静态**转移预测机制：向后的条件转移预测为发生转移，先前的条件转移预测为不发生转移。
- 非阻塞的 Load 和 Store 执行：不能在单周期内完成的 Load 和 Store 指令，不管是因为它参考低速存储器，还是因为传输多寄存器，只要不发生操作数相关，都不会停止流水线的执行。
- 64 位的数据存储器使多寄存器 Load 和 Store 指令能够在每个时钟周期传送两个寄存器。

非阻塞 Load 和 Store 逻辑需要寄存器堆有独立的读与写端口，64 位多寄存器 Load 和 Store 指令需要读写端口各有两个。因此，ARM10TDMI 寄存器堆有 4 个读端口和 3 个写端口。

将这些特性合起来，使 ARM10TDMI 每 MHz 的 dhrystone 2.1 MIPS 数达到 1.25，ARM7TDMI 的同一参数为 0.9，ARM9TDMI 的同一参数为 1.1。这些数字直接反映出它们运行 dhrystone 基准程序时各自的 CPI 性能。其他程序可能给出相当不同的 CPI 结果，在执行复杂任务例如引导操作系统时，64 位数据总线使 ARM10TDMI 能够给出比 ARM9TDMI 好得多的有效 CPI。

ARM10TDMI 应用

在 9.3 节讨论 ARM9TDMI 应用时曾指出，至少需要某些本地的高速存储器来释放核的潜在性能。对于 ARM10TDMI 这也是正确的：没有分开的本地 64 位指令和数据存储器，处理器核就不能发挥它的全部性能，它也就不能比一个小而廉价的 ARM 核运行得更快。

还有，解决这个问题的通常(虽然不是惟一的)方法是置备本地 Cache 存储器，正如

在 12.6 节介绍的 ARM1020E 所显示的那样。由于 ARM10TDMI 核的性能严重依赖于快速 64 位本地存储器的实用性，对于其性能特性的讨论将在讲到 ARM1020E 时给出。

9.5 讨 论

所有早期的 ARM 处理器核，直到 ARM7TDMI，都是基于简单的取指—译码—执行流水线。从 Acorn Computers 公司在 20 世纪 80 年代早期开发的最初的 ARM1，直到今天在多数移动电话中使用的 ARM7TDMI 核，基本的操作原理几乎没变。ARM 公司在第 1 个 10 年的开发工作都集中于以下几个方面：

- 通过关键路径的优化和工艺尺寸的缩小来改善性能。
- 通过采用静态 CMOS 逻辑、降低电源电压和压缩代码（Thumb 指令集）来实现低功耗应用。
- 通过增加片上调试特性、片上总线和软件工具来支持系统开发。

ARM7TDMI 代表这个开发过程的顶点。在一个由 PC 和日益复杂的超标量、超流水线、高性能（功耗也很高）微处理器占优势的世界里，它的商业成功显示了原先非常简单的 3 级流水线的生命力。

在 ARM 开发的第 2 个 10 年里，在寻求高性能中出现了 ARM 组织的审慎多样化：

- 向 5 级流水线迈出的第一步获得了双倍的性能（所有其他因素都相等），代价是在核中加入了某些前推逻辑，以及双倍带宽存储器（例如在 ARM8 中）或分开的指令与数据存储器（例如在 ARM9TDMI 和 StrongARM 中）。
- 在 ARM10TDMI 中取得下一步性能加倍是相当艰难的。6 级流水线与以前使用的 5 级流水线颇为相似，但是存储器访问的时间槽分配被扩展了。这使存储器能够不用花费过多的功耗即可支持更高的时钟速率。处理器核还采用了更多的分隔：在取指单元，使转移能被预测并从指令流中去除；在数据存储器接口，使得当需要一些时间解决数据访问时（例如由于 Cache 的失效），处理器能够继续执行。

性能的改善是通过增加时钟速率和降低 CPI（每条指令的平均时钟数）来实现的。增加时钟速率通常需要增多流水线的级数，而这往往会损失 CPI。因此，需要采取补救措施来挽回 CPI 的损失，并进一步改善它。

到现在为止，所有 ARM 处理器都是基于每个时钟周期至多发射 1 条指令的组织结构，而且总是按程序的顺序发射。ARM10TDMI 和 AMULET3 处理器（在第 14.5 节介绍）处理无序的实现，以便在数据读写速度较慢时保持指令流。这两种处理器也包括转移预测逻辑，以减少在执行转移指令时对流水线的再填充。AMULET3 消除了预测的转移指令的取指，但仍然执行它；ARM10TDMI 读取转移指令，然而抑制了指令的执行。但是按照今天高端 PC 和工作站处理器的标准，这些仍然是很简单的机器。这种简单性在系统芯片应用中有直接的好处：这种简单的处理器比复杂的处理器需要较少的晶体管，因而使用较小的芯片面积，消费较少的功耗。

9.6　例题与练习

例题　9.1

为了与静态 RAM 或 ROM 接口，ARM7TDMI 地址总线应如何重定时？

通常 ARM7TDMI 在新地址一产生时就将其输出，时间接近前一个时钟周期的末尾。为了与静态 RAM 接口，必须在周期结束后还保持地址的稳定，所以必须消除该流水线。最简单的解决方法是使用地址流水线使能信号 ape。如果系统中某些存储器件从较早的地址中获益，而某些存储器件是静态的，那么应当使用外部锁存器给静态器件的地址重定时，或者应当控制 ape 来配合当前寻址的器件。

练习　9.1.1

回顾在本章中讲述的处理器核，讨论是哪些基本技术把从 ARM7TDMI 核到 ARM10TDMI 核的性能提高到 8 倍。

练习　9.1.2

如果设计者可以自由地改变处理器的电源电压，那么就有可能在性能（它与 Vdd 成比例）和功耗效率（它与 $1/V_{dd}^2$ 成比例）之间进行折衷。因此，$MIPS^3/W$ 就是扣除电源电压影响后对体系结构功耗效率的度量。

基于这种度量来比较本章介绍的各种处理器核。

练习　9.1.3

由前一个练习的结果继续考虑：低功耗系统的设计者为什么不能简单地选择在体系结构上效率最高的处理器核，然后按比例调整电源电压来达到所需的系统性能？

第 10 章　存储器层次

本章内容综述

现代微处理器能以很高的速率执行指令。为了开发出潜在的性能，处理器必须连接一个容量大、速度高的存储器系统。如果存储器容量太小，就不能装载足够的程序以保持处理器全力处理。如果太慢，就不能像处理器执行指令那样快地提供指令。

不幸的是，存储器容量越大速度越慢。因此，不可能设计一个足够大又足够快的单一存储器使高性能处理器充分发挥其能力。

但是有可能构建一个复合存储器系统，它包括一个小但速度快的存储器和一个大但速度慢的主存储器。根据典型程序的统计，这个存储器系统的外部行为在大部分时间像一个既大又快的存储器。这个小但速度快的元件是 Cache，它自动保存处理器经常用到的指令和数据的拷贝。Cache 的有效性依赖于程序具有空间局部性和时间局部性的特性。

两级存储器原理可以扩展为多级存储器层次。计算机后援(disk)存储可以看作是存储器层次的一部分。有了适当的存储器管理的支持，程序的大小不由主存储器限制，而是取决于容量远高于主存储器的硬盘空间。

10.1　存储器容量及速度

典型的计算机存储层次由多级构成，每级都有特性容量及速度。

- 微处理器寄存器组可看作是存储器层次的顶层。典型的 RISC 微处理器大约有 32 个 32 位寄存器，总共 128 字节，其访问时间为几个 ns。
- 片上 Cache 存储器的容量在 8～32 KB 之间，访问时间大约为 10 ns。
- 高性能桌上系统可能有第 2 级片外 Cache，容量为几百 KB，访问时间为几十 ns。
- 主存储器可能是几 MB 到几十 MB 的动态存储器，访问时间大约为 100 ns。
- 后援存储器，通常是硬盘，可能从几百 MB 到几个 GB，访问时间为几十 ms。

注意，主存储器与后援存储器之间的性能差别远大于其他相邻级之间的差别，即使系统中没有第 2 级 Cache。

保存在寄存器组中的数据可以由编译器或汇编语言直接控制，但其他层次中的内

容通常为自动管理。Cache 对于应用程序往往是不可见的，在硬件控制下，指令和数据以块或“页”的形式向上层级和下层级移动。主存储器与后援存储器之间的页映射由操作系统控制，对于应用程序是透明的。由于主存储器与后援存储器之间性能差异太大，决定何时在这两级间移动数据的算法更为复杂。

嵌入式系统通常没有后援存储器，因此也不采用页方式。但是许多嵌入式系统采用 Cache，ARM CPU 芯片采用了多种 Cache 组织结构。因此，这里对 Cache 的组织问题作较详细的讨论。

存储器成本

高速存储器的每位价格远高于低速存储器，因此，采用层次存储器的目的还在于以接近低速存储器的平均每位价格得到接近高速存储器的性能。

10.2　片上存储器

如果微处理器要达到最佳性能，那么采用片上存储器是必需的。在当前的时钟速率下，只有片上存储器能支持零等待状态访问速度。同时，与片外存储器相比，片上存储器有较好的功耗效率，并减少了电磁干扰。

片上存储器的优点

在许多嵌入式系统中采用简单的片上 RAM 而不是 Cache，这有许多原因：

- 它简单、便宜且功耗低。在下面几节中我们可以看出，为了使 Cache 有效工作需要巨大的逻辑开销。同时，如果没有合适、现成的 Cache，则设计费用也很高。
- 它有更不确定的行为。Cache 存储器有复杂的行为，这使得在某种情况下难于预测它将如何工作。特别是可能很难保证中断响应时间。

与 Cache 相比，片上 RAM 的缺点是需要程序员直接管理。而 Cache 对于程序员来说通常是透明的。

如果程序的混合(program mix)已定义好并在程序员控制之下，那么片上 RAM 就能有效地作为软件控制的 Cache 来使用。若应用程序的混合不能预知，那么控制任务将变得非常困难。因此，在应用程序的混合不可预知的通用系统中通常采用 Cache。

片上 RAM 的一个重要优点是，它使程序员能够根据对将来处理工作量的了解来划分 RAM 的空间。而 Cache 只知道前面程序的行为，因此，不会为将来的关键任务预先作准备。当关键任务必须满足严格的实时约束时，这个差别尤为重要。

系统设计师必须考虑所有的因素，在特定系统中采用正确的方法。无论选择什么样的片上存储器都要特别小心。它必须足够快以使微处理器满负荷，又要足够大以便能容纳关键程序。但又不能太快(功耗太大)太大(占用太多芯片面积)。

10.3 Cache

当第一代 RISC 微处理器刚出现时，标准存储器元件的速度比当时微处理器的速度快。但这种状况没有持续多久。后来半导体工艺技术的进展被用来提高微处理器的速度，但在改善存储器芯片方面的作用却截然不同。标准 DRAM 部件虽然也快了一点，但其开发的主要精力则放在提高其容量上。

处理器和存储器速度

1980 年，典型的 DRAM 部件的容量为 4K 位。1981 年和 1982 年开发出了 16K 位的芯片。这些部件的随机访问速率为 3 或 4 MHz，局部访问（页模式）时速率大约快 1 倍。当时的微处理器需要访问存储器 2M 次/s 左右。

在 2000 年，DRAM 部件每片的容量达到 256M 位，随机访问速率在 30 MHz 左右。微处理器每秒需要访问存储器几百兆次。如果处理器速率如此远高于存储器的，那么只有借助 Cache 才能满足其全部性能。

Cache 存储器是一个容量小但非常快的存储器，它保存最近用到的存储器数据的拷贝。对于程序员来说，Cache 是透明的。它自动地决定保存哪些数据，覆盖哪些数据。现在 Cache 通常与处理器在同一芯片上实现。Cache 能够发挥作用是因为程序具有**局部性**的特性。也就是说，在任何特定时间，微处理器趋于对相同区域的数据（如堆栈）多次执行相同的指令（如循环）。

统一的 Cache 和哈佛 Cache

Cache 有多种构造方法。在最高层次，微处理器可以采用下列两种组织中的一种，即

- 统一的 Cache。指令和数据用同一个 Cache，如图 10.1 所示。
- 指令和数据 Cache 分开。有时这种组织方式又称为**改进的哈佛结构**，如图 10.2 所示。

这两种组织方式各有优点。统一 Cache 能够根据当前程序的需要自动调整指令在 Cache 存储器中的比例，比固定划分有更好的性能。另一方面，分开的 Cache 使 Load/Store 指令能够单周期执行。

Cache 性能的度量

只有当所需要的存储器内容已经在 Cache 时，微处理器才能以高时钟速率工作。因此，系统的总体性能强烈依赖于存储器访问中不能由访问 Cache 来完成的比例。当要访问的内容在 Cache 时称为**命中**（hit），而要访问的内容不在 Cache 时称为**未命中**（miss）。所有能够由访问 Cache 来完成的存储器访问的比例称为**命中率**，通常以百分比来表示。不能由访问 Cache 来完成的存储器访问的比例称为**未命中率**。

如果一个现代微处理器要达到它的潜能，那么设计良好的 Cache 的未命中率应该只有百分之几。未命中率依赖于多个 Cache 参数，包括大小（Cache 中存储器字节数）和组织。

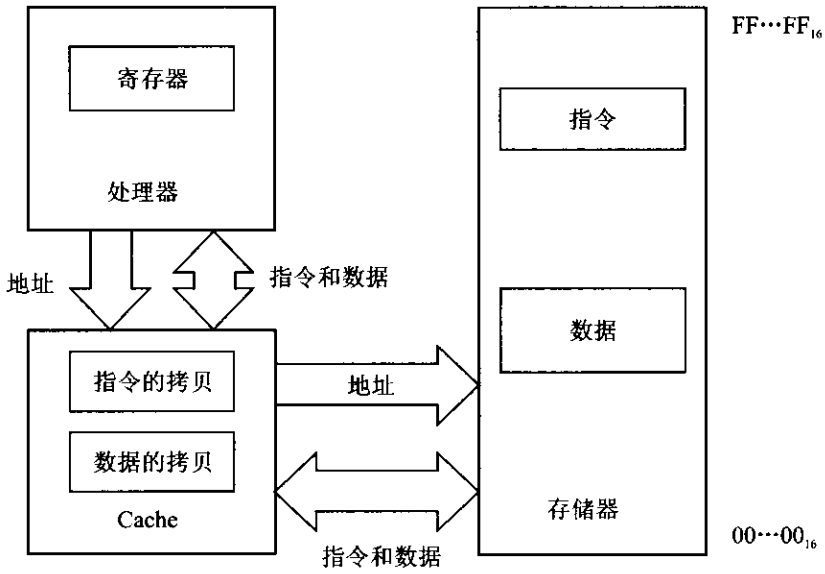


图 10.1　统一的指令和数据 Cache

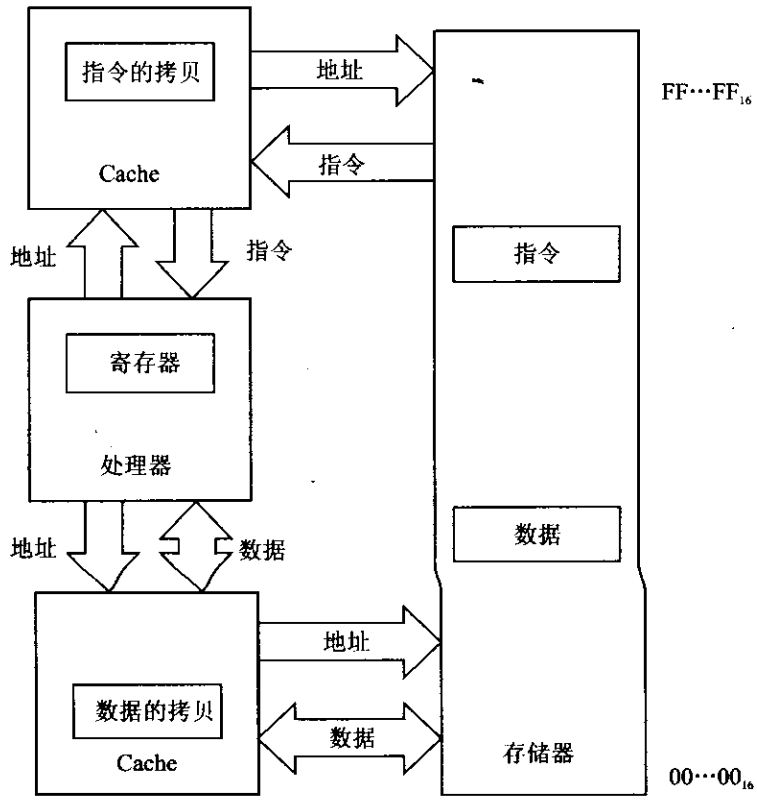


图 10.2　数据和指令分开的 Cache

Cache 组织

由于 Cache 所保存的主存储器内容是动态变化的，因此，Cache 必须同时保存数据及其在主存储器中的地址。

直接映射 Cache

最简单的 Cache 组织是直接映射方式，如图 10.3 所示。在直接映射 Cache 存储器中，一行数据连同在存储器中的地址 **Tag** 一起保存。Tag 由存储器地址的一部分(**index**)来寻址。

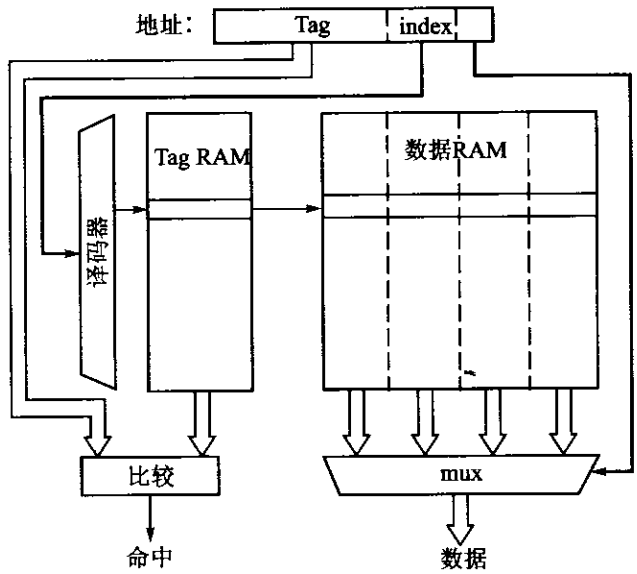


图 10.3 直接映射 Cache 组织

为了检查特定的存储器内容是否在 Cache 中，用地址的 index 位访问 Cache 的标志项。将高地址位与存储的 Tag 进行比较，如果相等，则说明访问内容在 Cache 中。低地址位用于访问行中相应内容。

与更为复杂的组织方式相比，直接映射组织具有一系列特性：

- 特定存储器项只能保存到 Cache 的惟一位置。有相同 Cache 地址的两个存储器项争夺使用该位置。
- Tag 存储器保存除行内寻址和 Cache RAM 寻址所需要的位之外的其他位。
- Tag 和数据访问可以同时进行，是所有组织方式中 Cache 访问速度最快的。
- 由于 Tag RAM 通常远小于数据 RAM，因此其访问时间短，使 Tag 的比较能够在数据访问时间内完成。

一个典型的直接映射 Cache 能保存 8 KB 数据，每行 16 字节，因此共有 512 行。一个 32 位地址中的 4 位用于行内寻址，9 位用于行寻址，其他 19 位为 Tag。因此，Tag RAM 存储器的大小超过 1 KB。

当向 Cache 装入数据时，从存储器中读出一块(block)数据。若存储器块的尺寸大

于 Cache 行的尺寸，问题还不是很大。但若存储块的尺寸小于 Cache 行的尺寸，那么 Tag 存储器必须扩展一些有效位用于指示行内的块。Cache 行和存储器块的大小一致是最简单的组织方式。

组相联 Cache

组相联 Cache 使存储器块可以保存到 Cache 多个位置以减少竞争问题，其复杂性较高。一个两路组相联 Cache 如图 10.4 所示。图中的 Cache 形式由两个可以有效地并行工作的直接映射 Cache 构成。给出一个地址后可能在两个 Cache 中任意一个找到数据，因此，存储器地址可以保存到两者中任何一个。在直接映射 Cache 中，竞争同一个位置的两个存储器项现在可以使用这些位置中的一个，使 Cache 中两者都可以命中。

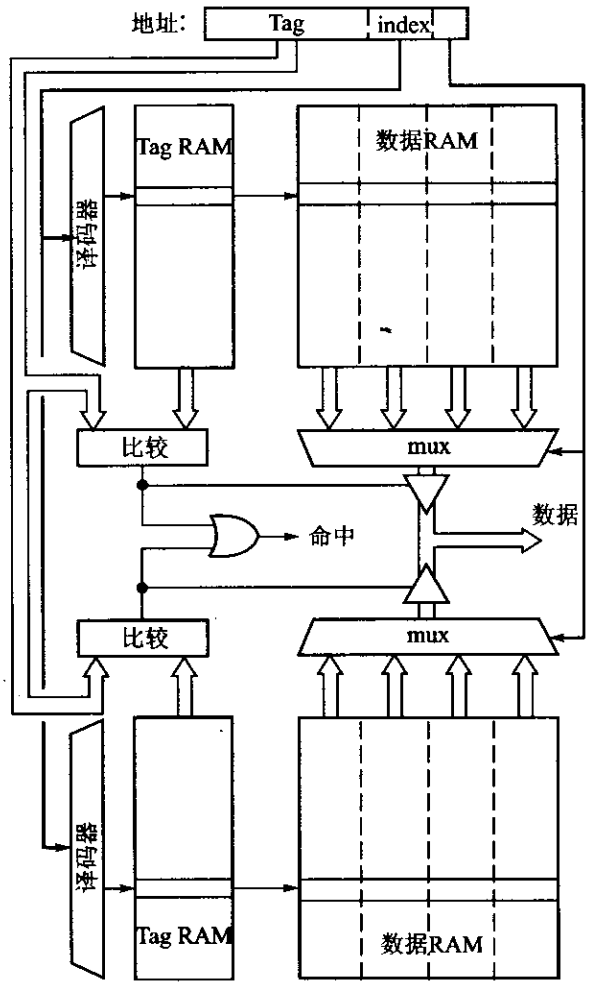


图 10.4　两路组相联 Cache 组织

每行 16 个字节的 8 KB Cache，在其每一半中有 256 行。因此，32 位地址中的其中 4 位用于从行中选择字节，8 位用于从每一半 Cache 中选择 1 行。因此 Tag 地址必须多 1 位，为 20 位。访问时间稍长于直接映射 Cache，时间增加是由于需要在两半之间

使用多路器选择数据。

当一个新数据项要放到 Cache 时，必须决定放到哪一半。有许多选择方式，最常用的方式如下：

- 随机存放。存放取决于一个随机或伪随机数。
- 最近未使用(LRU)。Cache 记录两个位置中哪一个是最后访问的，并将新数据放到另外一个。
- 循环使用(又称为“周期”)。Cache 记录哪一个位置是最近分配的，并将新数据放到另外一个。

组相联方法可以从两路扩展到任意路相联。但实际上超过 4 路相联后作用并不大，只会导致额外的复杂度。

全相联 Cache

另一个相联的极端是以 VLSI 技术设计一个全相联 Cache。不是将直接映射 Cache 继续分为更小的元件，而是使用内容寻址存储器(**CAM**, Content Addressed Memory)来设计 Tag RAM。一个 CAM 单元是具有内建比较器的 RAM 单元，因此，基于 CAM 的 Tag 存储器能够并行地查寻并定位任何位置的地址。全相联 Cache 的组织如图 10.5 所示。

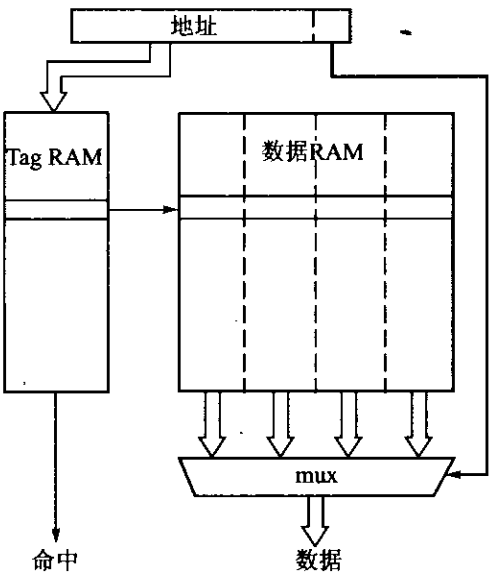


图 10.5 全相联 Cache 组织

由于没有任何地址位隐含 Cache 中数据的位置，因此，Tag 存储器必须保存除用于行内字节寻址的地址位以外的其他所有地址位。

写策略

上述方案的读访问工作方式是很明显的：当给出一个新的读地址时，Cache 检查是否保存了寻址的数据。如果是，则 Cache 给出这个数据；如果不是，则从主存储器读

取一个数据块，然后将其保存到 Cache 中合适的位置，并向微处理器提供所需的数据。

当微处理器执行写操作时有多种选择。按复杂度由低到高排序，常用的写策略(write strategies)如下：

1. 写直达(write through)

所有的写操作直接写入主存储器。如果所寻址的数据正好保存在 Cache 中，则 Cache 进行更新以保存新数据。在写操作时微处理器必须降到主存储器的速度。

2. 带缓冲的写直达

所有写操作仍然直接对主存操作，Cache 也在适当时机更新。但不是将微处理器降到主存速度，而是将要写的地址及数据保存到可以高速接收写信息的**写缓冲器**中。然后写缓冲器以主存速度将数据传送到主存中，而微处理器可以继续处理下一个任务。

3. 写回法(write back)

写回式 Cache 并不与主存保持一致。写操作只更新 Cache，因此，Cache 行必须记录是否被修改过(在每一行或块中设置一个脏位(dirty bit))。如果新的数据要调入到一个已脏的 Cache 行中，那么这个行必须先写回到主存中。

写直达 Cache 的实现最简单，其优点是主存随时更新，其缺点是在每个写操作过程中微处理器都降低到主存速度。加上写缓冲将使微处理器继续高速工作直至写速度超过外部写带宽。对于写回式 Cache，在最终值写回主存之前，一个位置可以多次写入，由此降低了对外部写带宽的需求。但是实现上更为复杂，并且由于缺乏一致性而难于管理。

Cache 特点总结

表 10.1 总结了定义 Cache 组织的各种参数。第一个参数是 Cache 与存储器管理部件(**MMU**，Memory Management Unit)的关系，这将在 10.5 节的“虚拟与物理 Cache”一段中进一步讨论。其他参数在本节中都已涉及。

表 10.1　Cache 组织选项的总结

组织特点	选　项		
Cache-MMU 关系	物理 Cache	虚拟 Cache	
Cache 内容	指令与数据统一的 Cache	分开的指令与数据 Cache	
相联	直接映射 RAM-RAM	组相联 RAM-RAM	全相联 CAM-RAM
替换策略	循环	随机	LRU
写策略	写直达	带缓冲的写直达	回写

10.4　Cache 设计示例

当选择 Cache 的组织方式时，需要考虑在 10.3 节中讨论过的几个因素，包括 Cache 的大小、相联度、行及块的大小、替换算法以及写策略。需要进行详细的体系结

构仿真来分析这些选择对 Cache 性能的影响。

ARM3 的 Cache

1989 年设计的 ARM3 是第一个带有片上 Cache 的 ARM 芯片。这些参数对性能及总线使用的影响已被细致地研究。在这些研究中，使用专门设计的硬件在 ARM2 运行几个基准程序时跟踪其地址，然后用软件分析这些地址流以便对多种组织形式的行为建模(现在已经不需要专门硬件了，台式机已经具有足够的性能，不需要硬件的支持就能仿真足够大的程序)。

在研究开始时设置一个期望通过 Cache 达到的性能上限。一个总是包含所需要数据的理想 Cache 模型用于设定这个上限。实际的 Cache 总要损失一些时间，因此，不可能比一个总是命中的理想 Cache 性能更好。

在比较实际的 Cache 和外部存储器速度(分别是 20 MHz 和 8 MHz)假设的基础上，建立了 3 种形式的理想模型：指令 Cache、指令和数据混合 Cache 以及数据 Cache。用没有 Cache 的系统性能进行规格化，其结果如表 10.2 所列。由该表可知，指令是 Cache 保存的最重要的内容。但若包括数据，则会使性能进一步提高 25%。

尽管早就做出 Cache 写策略将是写直达(理论上是最简单的)的决定，但是 Cache 检测到一个写未命中并从写地址处调入一行数据仍然是可能的。对写未命中调入策略作了简单研究，但可以证明其收效甚微且大大增加了复杂度，故这个策略很快就被放弃。因此，问题简化为对采用读未命中调入策略的统一的指令和数据 Cache，找出其最佳组织，与芯片面积和功耗相符。

表 10.2 理想 Cache 性能

Cache 形式	性 能
无 Cache	1
指令 Cache	1.95
指令和数据 Cache	2.5
数据 Cache	1.13

研究了多种不同的 Cache 组织及大小，其结果如图 10.6 所示。最简单的 Cache 组织是直接映射 Cache。但即使其大小为 16 KB，它仍比理想情况差很多。复杂性稍高的是 2 路组相联 Cache，当大小为 16 KB 时其性能与理想 Cache 差 1%左右。但在当时(1989 年设计 ARM3 时)，16 KB Cache 需要很大的芯片面积，而 4 KB Cache 的性能又不怎么好(结果强烈依赖于用于产生地址跟踪流的程序，但这些程序是典型程序)。

走向另一个极端，全相联 Cache 在小容量时的性能要好得多，使用基准程序测试时可产生近似理想的性能。使用的替换算法是随机算法。LRU(最近未使用)算法给出了非常相似的结果。

Cache 模型修改为一行 4 个字，这对减小 Tag 存储器面积成本是必要的。这个变化对性能影响很小。

全相联 Cache 需要一个巨大的 CAM(内容寻址存储器)Tag 存储器。即使一行 4

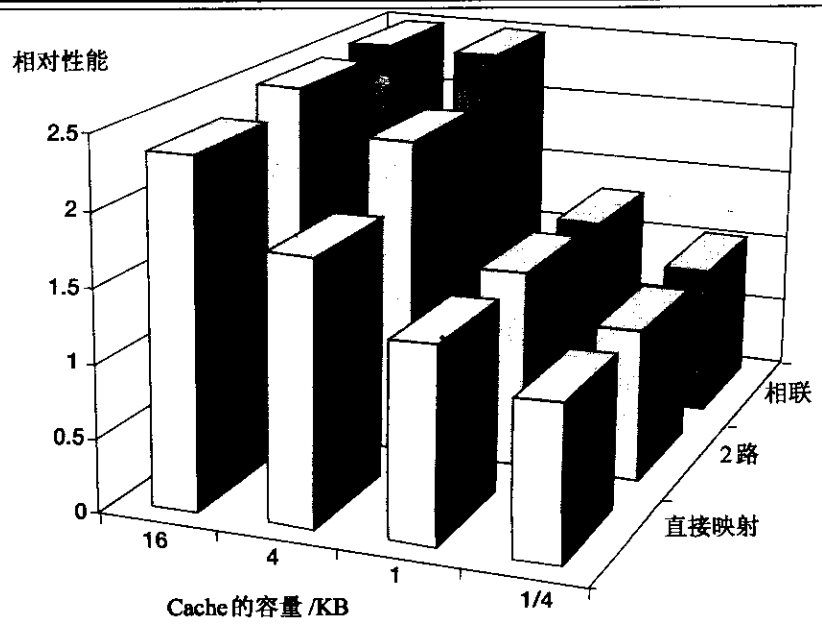


图 10.6　以容量和组织为函数的统一 Cache 性能

个字，其功耗也相当大。通过分段将 CAM 拆分为小的元件可大大减少功耗，但也减小了相联度。使用 4 KB Cache，分析系统性能对相联度的敏感性，如图 10.7 所示。图中给出了从全相联(256 路)到直接映射(1 路)的所有相联的系统性能。尽管从直接映射到 2 路组相联的性能提高最大，但是一直到 64 路相联，性能提高还是很明显的。

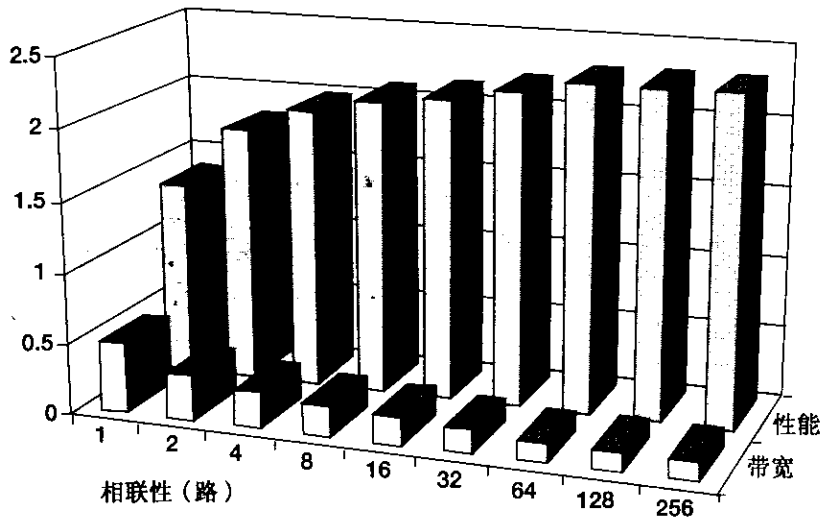


图 10.7　相联对性能及带宽需求的影响

由图 10.7 可以看出，64 路相联 CAM-RAM Cache 与全相联 Cache 性能相同，但 256 个 CAM 项被分为 4 个部分节以省功率。图 10.7 同时给出了各级相联所需的外部存储器的带宽(相对于没有 Cache 的微处理器)。应注意到图中最高的性能对应于最低的外部带宽需求。与内部访问相比，每一次外部访问都要消耗大量的能量，因此，Cache

在提高性能的同时减少了系统功耗需求。

由此采用的 Cache 组织如图 10.8 所示。虚拟地址的最低 2 位选择一个字中的字节；2～3 位从一个 Cache 行中选择字；4～5 位从 4 个 64 个表项的 CAM Tag 存储器中选择一个。虚拟地址的其他位送往所选择的 Tag 存储器(其他 Tag 存储器被禁止以节省功耗)，用以检查 Cache 数据 RAM 中的数据对应的地址以确定数据是否在 Cache 中。结果可能是命中，也可能是未命中。

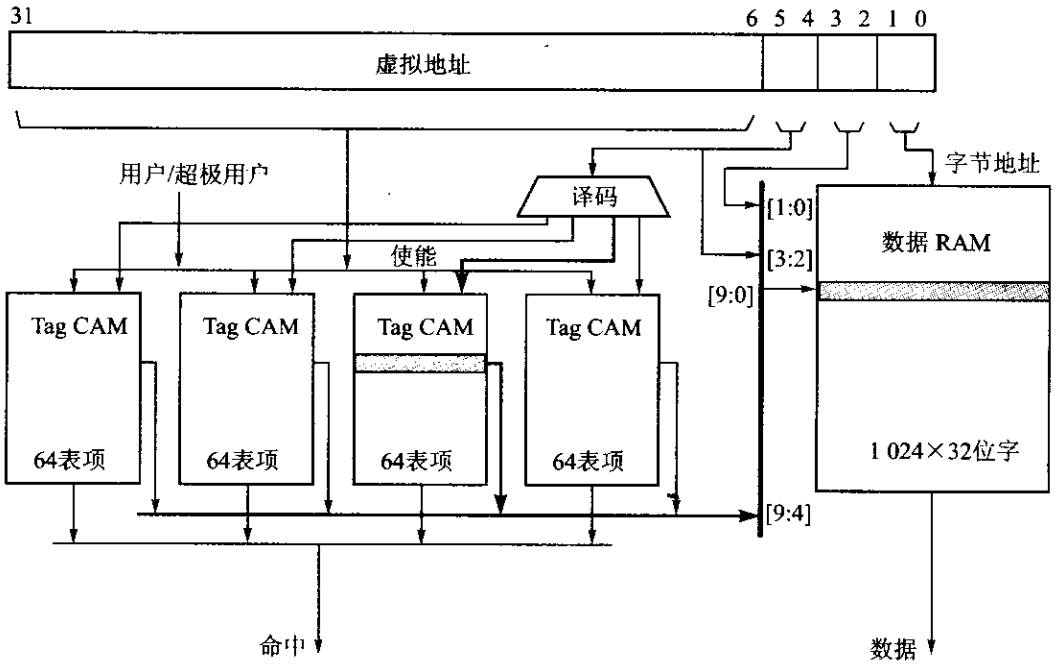


图 10.8 ARM3 的 Cache 组织

ARM600 Cache 控制 FSM

下面描述 ARM600 Cache 控制有限状态机，以说明管理 Cache 所需要的控制逻辑。ARM600 Cache 采用了 10.4 节中描述的 ARM3 的设计，同时还包括一个与 10.5 节中描述的转换系统相似的方案。

ARM600 使用两个时钟。从 Cache 读取或向写缓冲器写入时处理器使用**快速**时钟，而访问外部存储器时使用**存储器**时钟。处理器核使用的时钟在这两个时钟源之间动态切换。这两个时钟源之间可能是相互异步的。存储器时钟并不需要简单地由快速时钟分频得到，尽管如果这样处理器就可以通过配置来避免同步开销。

通常处理器基于 Cache 运行时使用快速时钟。在发生 Cache 未命中(或引用一个不能调入 Cache 的存储器)时，处理器与存储器时钟同步后进行单个外部访问或读一行数据到 Cache。由于在时钟之间切换时的同步会有一定的开销(以减小亚稳定性冒险到可接受的程度)，处理器检查下一个地址以决定是否切换回快速时钟。

控制这个动作的有限状态机如图 10.9 所示。初始化后，处理器在快速时钟下进入 check tag 状态。根据寻址数据是否在 Cache 中，处理器可能进入下列路径：

- 只要地址不是顺序的，没有 MMU 故障且是一个读命中或缓冲写，则状态机保持在 check tag 状态，并在每个时钟周期返回一个数据或进行一个写操作。
- 当下一个地址是在同一个 Cache 行的顺序读或顺序缓冲写时，状态机进入 sequential fast状态。此时可以不检查 Tag 及激活 MMU 就进行数据访问。这样既节省了功耗，处理器核又可以产生 seq 信号。在这个状态，每个时钟周期读回一个数据或进行一次缓冲写。
- 如果地址不在 Cache 中或写操作没有缓冲，则需要进行外部访问。状态机进入 start external 状态。在 external 状态，不可 Cache 存储器读或不可缓冲写被作为单个存储器事务来完成。如果存储器能读入 Cache，若 MMU 还没有准备好，那么在取了必要的转换信息后，读取一行 4 个字到 Cache。
- 处理器不使用存储器的周期在 idle 状态执行。

在转换过程中的几个点，可以清楚地显示出访问不能完成，并进入 abort 状态。外部硬件也可能使不可 Cache 读或不可缓冲写放弃。

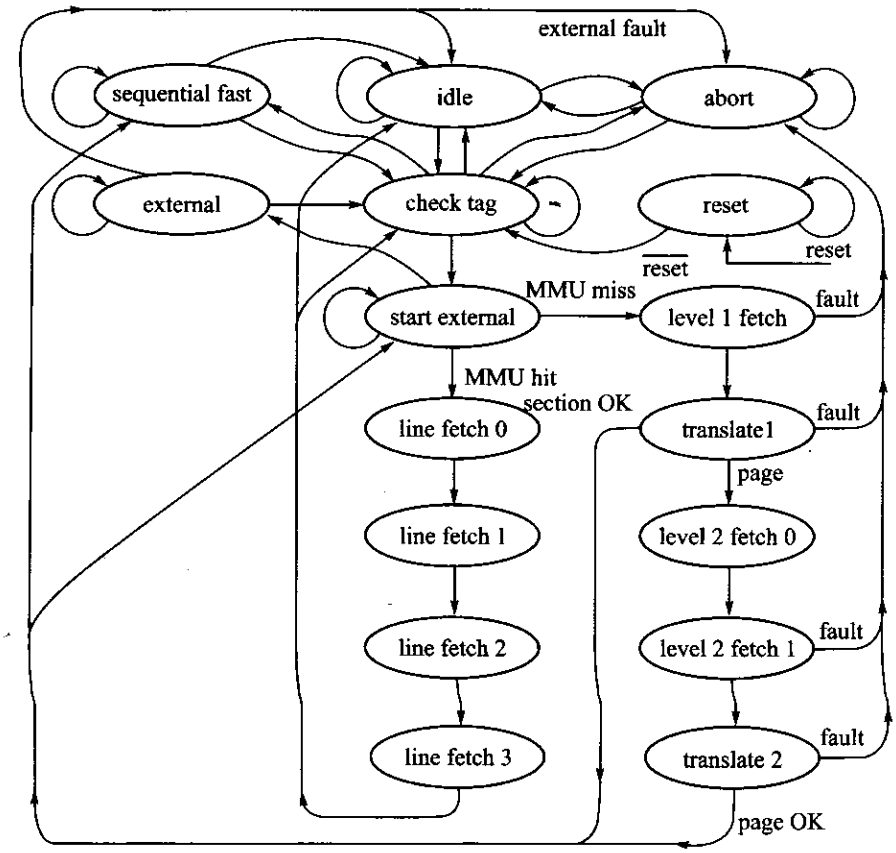


图 10.9　ARM600 Cache 控制状态机

10.5 存储器管理

现在典型的计算机系统在同一时间有多个程序在活动。单个处理器在同一时刻只能执行 1 个程序的指令，但可以在活动程序之间快速地切换，使它们看起来像是同时执行，至少从人的时间观念来看是这样。

快速切换是由操作系统管理的，因此，应用程序的程序员编写程序时可以不考虑这些，好像他可以占用整个机器一样。程序切换是由**存储器管理单元**(MMU)支持的。存储器管理有两种基本方法，称为**段式**管理(segmentation)和**页式**管理(paging)。

段式管理

最简单的一种存储器管理形式允许应用程序将其存储器视为一系列的段(segement)，每段包含一些特定类别的信息。例如，一个程序可以有一个包含所有指令的代码段、一个数据段和一个堆栈段。每次存储器访问都向 MMU 提供一个段选择和一个逻辑地址。每个段有一个基地址及一个相关的限制。逻辑地址是相对于段基地址的偏移。偏移不能超过限制，否则将发生一个访问违例，通常会产生一个异常。段也可以有一些其他的访问控制，例如代码段可以是只读的，试图对它进行写也将产生一个异常。

段式 MMU 访问机制如图 10.10 所示。

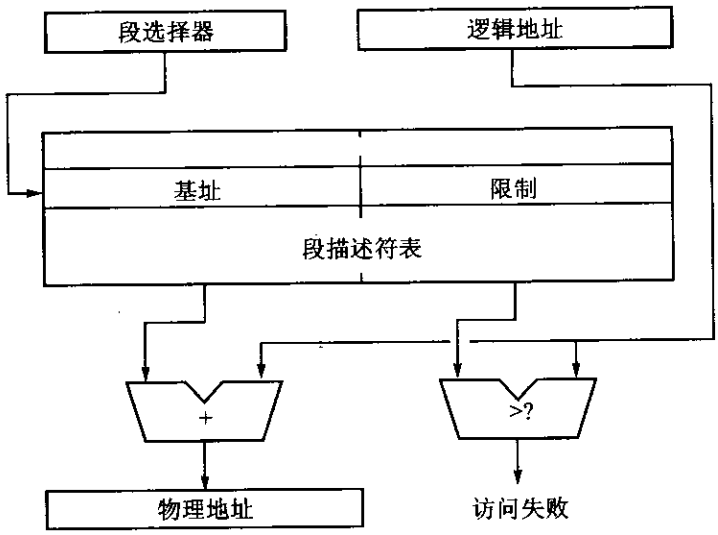


图 10.10 段式存储器管理方法

段式管理使程序拥有它自己的存储器视角，透明地与其他程序共存于同一存储空间。但是当共存程序变化，可用的存储器受到限制时，便会遭遇困难。由于段的大小是变化的，随着时间的推进，可用存储器变成碎片，使一个新的程序不能调入。其原因并不是因为存储空间不够，而是因为所有可用的存储器都是小的碎片，没有一片足够大的能够装入新程序的段。

操作系统可以通过移动存储器中的段将可用存储器拼接成一个大的片来解决这个问题，但其效率很低。现在大多数处理器使用基于固定大小的存储器块，又称为页(page)的存储器映射方法。有一些体系结构使用段式和页式，但许多处理器，包括ARM 仅支持页式管理。

页式管理

在页式管理中，逻辑和物理地址空间都划分为固定大小的页。页的大小通常为几KB，但对于不同的体系结构页的大小不同。逻辑和物理页之间的关系保存在页表中，而页表保存在主存储器中。

简单的计算表明，用单个表保存页表转换关系需要一个很大的表：如果 1 页为 4 KB，那么 32 位地址中的 20 位必须被转换。这就需要表中有 $2^{20} \times 20$ 位数据，也就是表的大小至少为 2.5 MB。对于一个小系统这个代价是不可接受的。

相反，大多数页式系统使用两级以上的页表。例如，地址的高 10 位用于确定一级页表目录中相应的二级页表；地址的次高 10 位确定页表项，即物理页数。这种转换方法如图 10.11 所示。

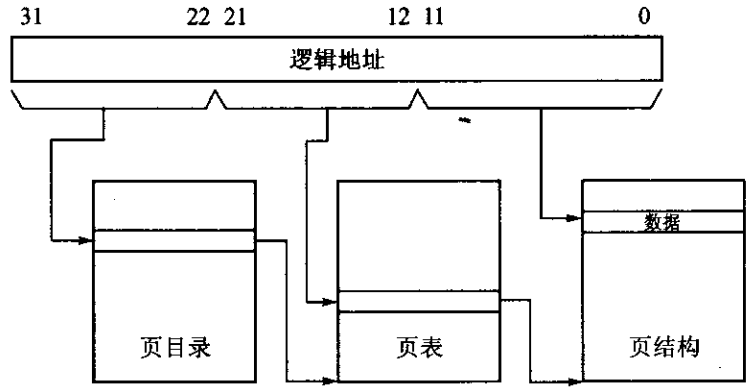


图 10.11　页式存储器管理方案

注意这里使用的特定数字。如果分配每个目录及页表项的大小为 32 位，则目录及每一个页表大小刚好为 4 KB，即刚好是一个存储器页。小规模系统的最小费用是 4 KB的页目录再加上 4 KB 的页表，这足以管理一个大小为 4 MB 的物理存储器。全配置至 4 GB 存储器需要 4 MB 的页表，但这个代价对于这么大的存储器是完全可以接受的。

将在 11.6 节中描述的 ARM MMU 使用的位分配方式(同时支持更大存储器块的单级转换)与这里描述的稍有不同，但原理是一样的。

虚拟存储器

另一种可能的存储器管理方案是允许将缺段或缺页标记出来，当对其访问时产生一个异常。当发生存储器分配溢出时，操作系统透明地将主存储器中的段或页移到后备存储器(通常是硬盘)中，并将其标记为无效。然后可以将物理存储器分配用做其他用途。如果程序试图访问无效的页或段，则会产生一个异常，操作系统将这个页或段调

回主存储器，然后允许程序重新访问。

当以页式存储器管理方案实现时，这个过程称为**请求分页式虚拟存储器**(demand-paged virtual memory)。编写的程序可以占用一个虚拟的存储空间，远大于所运行计算机的实际物理存储空间。操作系统轮番调入所需要的程序或数据。典型的程序重复执行部分代码，而其他部分则很少执行。将不常用的程序放在磁盘上不会明显地影响其性能。当然也可能发生操作系统频繁地切换存储器中页的情况。这称为系统失效(thrashing)，会严重影响系统性能。

可重起指令

在虚拟存储器系统中，一个重要需求是，任何可能造成存储器访问故障的指令必须能使处理器进入一个状态，以允许操作系统装入请求的存储器页，并使原始程序重新运行，就像故障从来没有发生过一样。这通常通过使所有存储器访问指令能够**重起**来实现。处理器必须保存足够的状态以使操作系统能够恢复足够的寄存器值，当请求页调入主存储器时，故障指令重新执行并得到与没有发生页故障时相同的结果。

地址转换后备缓冲器 TLB

上面描述的页方式使程序员完全自由透明地使用存储器。但可以看出，这是以付出相当的性能代价来得到的。原因是每次存储器访问都带来两次额外的存储器访问，即在访问所需数据之前要进行 1 次页目录访问和 1 次页表访问。

这些花销通常可以通过 **TLB**(Translation Look-aside Buffer)来避免。TLB 是最近用过的页转换的 Cache。同 10.3 节中描述的指令 Cache 和数据 Cache 一样，TLB 也有与相联度和替换策略有关的组织选项。行及块大小通常等于单个页表项。典型的 TLB 保存大约 64 个表项，容量远小于数据 Cache。典型程序的局部性特点使这个容量的 TLB 的未命中率仅为 1%左右。未命中时就要花费两个额外的存储器访问。

TLB 的操作如图 10.12 所示。

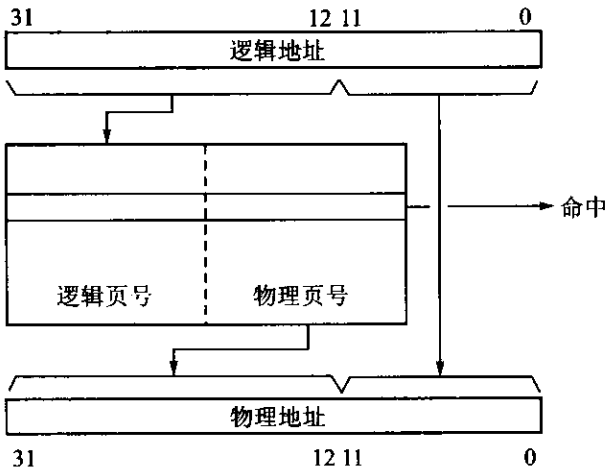


图 10.12 一个 TLB 操作

虚拟与物理 Cache

当系统同时实现了 MMU 和 Cache 时，Cache 可以工作于虚拟（MMU 之前）地址或物理（MMU 之后）地址。

虚拟 Cache 的优点在于处理器产生一个地址后可以立即开始一个 Cache 访问。如果发现数据在 Cache 中就不需要激活 MMU。其缺点是 Cache 可能包含**同义项**（synonyms），也就是同一主存数据项在 Cache 中有重复拷贝。出现同义项是因为地址转换机制通常允许重叠转换。如果处理器通过一条地址通道修改了数据，则 Cache 不可能更新第二个拷贝，从而导致 Cache 不一致。

由于物理存储器地址与数据项惟一相关，因而物理 Cache 避免了同义项问题。但是 MMU 必须在每个 Cache 访问时激活。但对于有些 MMU 和 Cache 组织，在 Cache 开始访问之前，MMU 必须完成地址转换，导致 Cache 延迟增加。

页式 MMU 只影响高地址位，而 Cache 访问只使用低地址位。利用这个事实安排物理 Cache 可以巧妙地避免顺序访问的开销。假定两种地址位不重叠，Cache 和 MMU 访问可以并行进行。MMU 产生的物理地址到达时刚好可以与 Cache 的物理地址 Tag 进行比较，可以将地址转换时间隐藏在 Cache Tag 访问中。这种优化方法不适用于全相联 Cache，只适用于 MMU 页尺寸大于 Cache 的每一个直接寻址部分。例如，4 KB 的页将直接映射 Cache 的最大尺寸限制为 4 KB，2 路组相联 Cache 的最大尺寸限制为 8 KB，以此类推。

实际上，虚拟和物理 Cache 在商业上都有应用。前者依赖于包括同义项问题的软件协定；而后者或者使用上述的优化，或者接受性能上的开销。

10.6　例题与练习

例题　10.1

当系统中页大小为 1 KB 时，4 路物理 Cache 的容量有多大？

假定我们想同 10.5 节“虚拟与物理 Cache”一段中描述的那样并行完成 TLB 和 Cache 访问，Cache 的每一部分最大为 1 KB，因此，全部 Cache 最大为 4 KB。

练习　10.1.1

如果 1 行的大小为 16 字节，那么在这个 Cache 中保存 Tag 需要多大的存储器？

练习　10.1.2

估计上例中 TLB、数据 Cache 的 Tag 与数据存储器的面积比例。

例题　10.2

如果要包含所有物理页的转换，那么 TLB 必须有多大？

当 1 页为 4 KB 时，1 MB 存储器有 256 页，因此 TLB 需要 256 个表项。TLB 不再需要是一个自动 Cache。由于 TLB 未命中意味着物理存储器缺页，因此，需要一个磁

盘传送。与磁盘传送相比，由软件维护 TLB 的开销可以忽略。

覆盖所有物理存储器的 TLB 是**反相页表**形式。早期在 Acorn Archimedes 机器中使用的 ARM 存储器控制器使用的就是这样的转换方法。参考图 10.12，转换硬件可以是一个 CAM。物理页数存储是一个简单的硬连线编码器。对每一个物理页，CAM 都有一个表项。

Acorn 存储器控制器芯片的 CAM 有 128 个表项，其页尺寸随系统中物理存储器的数量而变化。1 MB 系统的页为 8 KB；4 MB 系统的页为 32 KB。若超过 4 MB，则再加上一个存储器控制器，页尺寸固定为 32 KB。CAM 由软件维护，因此不需要复杂的查表硬件。整个转换表全部由软件定义。

练习 10.2.1

估算 128 个表项的反相页表芯片的尺寸与 64 个表项的 TLB 的比例。假定 1 位 CAM 的面积是 1 位 RAM 面积的两倍。

第 11 章　体系结构对操作系统的支持

本章内容综述

操作系统的任务是提供一个环境，使得当多道程序并行执行时，程序间不良冲突的危险最小，但又支持安全数据共享。操作系统还应提供一个与机器硬件设施的完全接口。

过程间冲突的减小是靠每个过程只能访问自己的存储区域这样一种存储器管理与保护方案来实现的。每个程序在系统存储器中都有自己的可见区域。程序切换时存储器可见区域也动态地转换到新程序，前一个程序使用的所有存储器从视野中移出。这一切若要高效操作，则需要复杂的硬件支持。

对存储器保护下的数据共享窗口必须非常仔细地控制。大多数程序不正常的深层原因是对共享结构的偶然访问造成的，因此需要采用强制的方法。

对硬件设备的访问涉及大量的底层位处理。这些细节通常是由操作系统集中处理，而不是由每个程序独自处理。这样程序可以通过系统调用方式在更高的层次上访问输入/输出函数。

ARM 体系结构中有专用结构来支持操作系统中所有类似的问题。

11.1　操作系统简介

操作系统为机器的底层硬件资源和运行于其中的应用程序之间提供一个统一和完全的接口。最高级的操作系统能为几个不同的用户在相同时间执行的多个通用程序提供支持。

多用户系统

在一个多用户操作系统中，并行运行的程序的数量及类型是未知的，而且每次都不同。如果一个多用户系统要求每一个程序都顾及同一机器上其他程序的存在，这会很不方便。因此，多用户操作系统为每个程序提供一个完整的**虚拟机器**。编写程序时认为它是同一时刻在机器上运行的惟一程序。存在其他程序仅有的明显影响是运行时间加长了。

尽管在同一时间机器运行多个程序，但处理器只有一组寄存器(这里我们不考虑多

处理器系统),因此,在一个特定时间只有 1 个程序在执行。外在的并行性由**时间分片**(time-slicing)实现,也就是程序在处理器中轮流执行。由于从人的标准来看处理器以极高的速度运行,在一段时间如 1 s 内每个程序在处理器运行了几次,因此,每个程序都有一定程度的前进。操作系统负责**调度**(决定一个程序何时执行),使每个程序平均分享 CPU 时间,或使用**优先级**信息使某些程序优先执行。

处理器切换一个程序,或者是因为由定时器中断调用的操作系统认为这个程序现在已执行了足够的时间,或者是因为这个程序请求了一个低速外围访问(如磁盘访问),且在得到响应之前不会再做任何有用工作。这时操作系统不会让程序在处理器中空转,而是进行切换并调度其他能够进行有效工作的程序。

存储器管理

为了构造一个运行程序的虚拟机器,操作系统必须建立一个环境,使程序能够在其期望的存储器位置访问其代码及数据。由于程序将使用的期望存储器位置可能与其他程序冲突,因此操作系统进行存储器转换,使向程序调入代码及数据的**物理**存储器位置出现在相应的**逻辑地址**处。程序通过由操作系统管理的逻辑-物理地址转换机制查看存储器。

保　护

对于在同一个机器上运行程序的多个用户,非常希望能够保证一个用户程序的错误不会影响到任何其他程序的操作。同时,保护其他程序不受恶意攻击也是必要的。

存储器映射硬件不但给每个程序一个虚拟机器,而且能保证一个程序看不到属于其他程序的任何存储器,从而提供了一种存储器保护方法。但太过于强调保护会影响效率,这是因为共享那些包含有常用函数库的存储器域可以节省存储器使用。这里采用的解决方法是让这些区域成为只读的或只执行的。这样一个程序就不会破坏其他程序也会使用的代码。

常用的恶意破坏其他程序的方法是通过假装操作系统状态来克服由存储器管理系统设置的保护,然后修改转换表。大多数系统采用的解决方法是提供特权系统模式。该模式具有可控的访问。只有在这个模式下才能访问转换表。

设计一个能够避免来自聪明个人恶意攻击的计算机系统是一个复杂的问题,需要在体系结构上提供一些支持。ARM 的结构支持是提供具有可控制性访问的超级处理器模式,并在存储器管理单元中提供多种存储器保护形式。但 ARM 很少用于那种需要保护以避免恶意用户的系统。大多数情况下,这些设施用于捕捉因疏忽产生的程序错误,协助软件调试。

资源分配

两个并行执行的程序发出的系统资源请求可能发生冲突。例如,一个程序可能请求从一部分磁盘读取数据。这个程序在磁盘驱动器寻找数据时切换出去。而切换进来的程序也可能立即请求从另一部分磁盘读取数据。如果磁盘驱动器直接响应这些请求,则很容易产生两个程序交替控制磁盘控制器,使其在两个请求之间振荡的情况,而

每个程序都没有足够的时间找到所需数据。这样将产生系统活锁，直到磁盘驱动器损坏。

为了避免这类情况，所有激活输入/输出的请求都由操作系统设置一个通道。通道接收第一个程序的请求，把第二个程序的请求放在队列中。第一个请求满足后再响应第二个请求。

单用户系统

对于单用户系统，在同一时间执行多道程序的情形仍然可能发生，上述大多数情况仍然存在。尽管消除了恶意用户共享同一机器的威胁，但每道程序都运行于自己的空间，以使一个程序的错误不会影响其他程序，这仍然是非常有用的。在消除恶意用户的顾虑后系统可以简化，不再需要禁止一个程序假装成系统特权。这种情形即使偶尔发生也是极其不可能的。

然而，越来越多的单用户桌面机器连接到计算机网络，而网络允许其他用户在上面远程执行程序。这类机器应被明确地看做多用户系统，并对其进行适当级别的保护。

嵌入式系统

嵌入式系统明显区别于上面讨论的单用户和多用户通用系统。它通常执行一系列固定程序而不会引入新的程序，因此不存在恶意用户问题。

操作系统继续扮演相似的角色，就是给每道程序提供一个干净的虚拟机器，保护程序以防止其他程序中错误的干扰，并调度使用 CPU 时间。

许多嵌入式系统工作时有实时约束。这些约束决定调度的优先级。成本问题使嵌入式系统不会采用通用机器中常用的操作系统，因为这种操作系统需要大量的存储器资源。这导致了实时操作系统(**RTOS**, Real-Time Operating Syetem)的开发。实时操作系统提供嵌入式系统所需的调度及硬件接口工具，并且只占用几 KB 的存储器。

小型嵌入式系统甚至可能连这种开销也不能承受，或者可能只有非常简单的调度需求(例如在所有时间里只执行 1 个固定程序)，根本不需要一个“操作系统”。这时有一个简单的“监视”程序，提供几个系统功能(如输入/输出接口功能)就足够了。这种系统基本上不需要存储器管理硬件，直接使用微处理器的逻辑地址访问存储器。如果一个嵌入式系统包括一个 Cache 存储器，就需要一些机制以定义哪些范围的存储器是可 Cache 的(cacheable)(因为 I/O 域是不能 Cache 的)。但这比整个存储器管理系统要简单得多。

本章结构

存储器管理的一般原理已经在第 10 章中介绍过。本章后面的各节介绍 ARM 系统控制协处理器及其控制的存储器管理系统，包括一个完整的、有地址转换的 MMU，以及一个用于不需要地址转换的嵌入式系统的、简单的保护单元。

随后几节介绍与操作系统相关的重要问题：同步、上下文切换以及输入/输出器件的处理，包括中断的使用。

11.2 ARM 系统控制协处理器

ARM 系统控制协处理器是一个片上协处理器，使用逻辑协处理器号 15，控制片上 Cache 或 Cache、存储器管理或保护单元、写缓冲器、预取缓冲器、转移目标 Cache 以及系统配置信号的操作。

CP15 指令

通过读写 CP15 寄存器进行控制是有效的。

所有寄存器都是 32 位长，并且只能在超级模式下由 MRC 和 MCR 指令访问(参见 5.19 节)。使用其他协处理器指令，或试图在用户模式下访问，都将产生无定义指令陷阱。这些指令的格式如图 11.1 所示。在大多数情况下，不使用 CRm 和 Cop2 域(field)，且应将其置为 0。它们只在特定操作中使用。

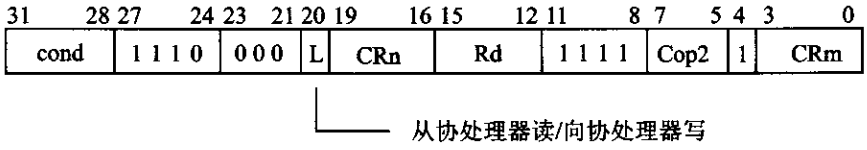


图 11.1 CP15 寄存器传输指令

保护单元

如果应用于嵌入式系统的 ARM CPU 执行固定的或可控制的应用程序，则不需要完整的、具有地址转换能力的存储器管理单元。对于这样的系统，一个简单的保护单元就足够了。

ARM 保护单元中 CP15 寄存器的组织将在 11.3 节中描述，保护单元的操作将在 11.4 节中描述。

采用了保护单元的 ARM740T 的 CPU 将在 12.1 节中描述，ARM940T 将在 12.4 节中介绍。

MMU

用于一般应用的 ARM CPU，其应用程序的范围及数量在设计时是未知的，因此，通常采用具有地址转换的、完整的存储器管理单元。

ARM MMU 中 CP15 寄存器的组织将在 11.5 节中介绍，MMU 的操作将在 11.6 中介绍。

采用了完整的 MMU 的 CPU 以及其他所有 CPU 将在 12 章中介绍。

11.3　保护单元寄存器 CP15

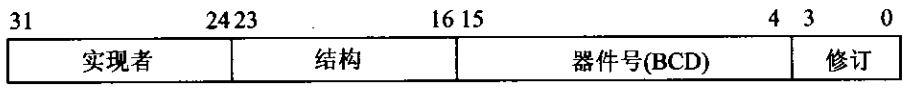
保护单元寄存器的结构如表 11.1 所列。对寄存器进行读/写的 CP15 指令如图 11.1所示，其中 CRn 指定要访问的寄存器。

表 11.1　保护单元中寄存器 CP15 的结构

寄存器	目　的
0	ID 寄存器
1	配置
2	Cache 控制
3	写缓冲控制
5	访问允许
6	界基及大小
7	Cache 操作
9	Cache 锁定
15	测试
4，8，10～14	未用

寄存器的详细功能如下：

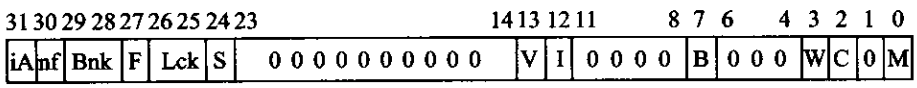
- 寄存器 0：只读。返回器件标识信息。



位[3：0]为修订编号；位[15：4]为 3 位 BCD 码表示的器件编号；位[23：16]为结构版本(0：版本 3；1：版本 4；2：版本 4T；3：版本 5T)；位[31：24]为生产商标记的 ASCII 码(ASCII 码“A”＝41_{16}表示 ARM 公司；ASCII 码“D”＝44_{16}表示 Digital 公司等等)。

有些 CPU 并不完全遵循上述寄存器 0 的格式。最近的 CPU 有第二个寄存器 0(通过改变 MRC 指令的 Cop2 域来访问)，给出了 Cache 的组织细节。

- 寄存器 1：可读/写。包含一些控制系统功能使能以及控制系统参数的信息位。

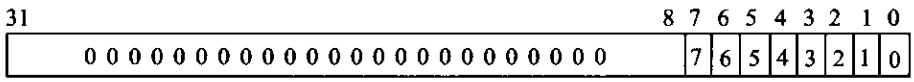


所有位在复位时清 0。若置位，则位 M 使能保护单元；位 C 使能数据或统一 Cache；位 W 使能写缓冲器；位 B 将小端格式转换为大端格式；位 I 在数据和指

令 Cache 分开时使能指令 Cache；位 V 将异常向量(exception vector)移到地址高端附近；S、Lck、F 和 Bnk 用于 Cache 控制(在 ARM740T 中)；nf 和 iA 控制各种时钟机制(在 ARM940T 中)。

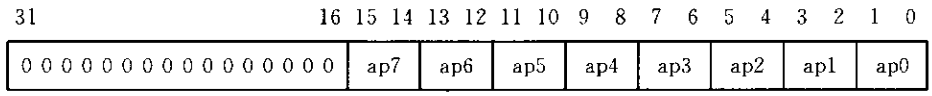
应注意，所有这些位并不是在所有实现中都提供。

- 寄存器 2：可读写。控制 8 个各自的保护区域(region)的 Cache 能力。



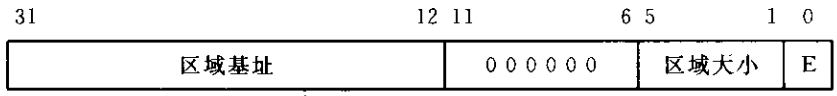
位[0]使能区域 0 的读 Cache；同样位[1]使能区域 1，以此类推。ARM940T 的指令及数据端口有分开的保护单元。Cop2(参见图 11.1)确定访问哪个单元。Cop2＝0 访问数据端口的保护单元；Cop2＝1 访问指令端口的保护单元。

- 寄存器 3：可读写。确定一个保护区域是否使用写缓冲器。其格式与寄存器 2 的相同。但由于 ARM940T 的指令端口是只读的，写缓冲器只能用于数据端口，因此，Cop2 应设为 0。
- 寄存器 5：可读写。定义每个保护区域的访问许可。



访问许可包括不能访问(00)、仅超级模式(01)、超级模式下全访问和用户只读(10)、全访问(11)。同样，ARM940T 使用 Cop2 域来区分指令(1)和数据(0)保护单元。

- 寄存器 6：可读写。定义 8 个区域的起始地址和大小。



区域基址必须是区域大小的整数倍。区域大小域(field)的编码如表 11.2 所列。E 使能该区域。

CRm 域指定特定的区域(参见图 11.1)，其值应设置为 0～7。对于一个硬核，如 ARM940T，指令及数据存储器端口有分开的区域寄存器。与寄存器 2 相同，Cop2 指定寻址的存储器端口。

- 寄存器 7：控制各种 Cache 操作。ARM740T 和 ARM940T 有不同的操作。
- 寄存器 9：在 ARM940T 中用于锁定 Cache 范围。(ARM740T 中用寄存器 1 中的特定位来实现该功能。)
- 寄存器 15：在 ARM940T 中用于将 Cache 分配算法从随机算法改变为循环算法，仅在产品测试时使用。

11.4　ARM 保护单元

用于嵌入式应用的 ARM CPU 中实现了存储器保护单元，定义了不同存储器区域的各种保护及 Cache 功能。例如，I/O 区域可以仅限于超级访问，并且是不可 Cache (uncacheable)的。保护单元不对地址进行转换。需要地址转换的系统应使用完整的存储器管理单元，如 11.6 节所述。实现了保护单元的 CPU 包括 ARM740T 和 ARM940T。

保护单元的结构

保护单元允许将 ARM 的 4 GB 的寻址空间映射为 8 个区域。每个区域都有可编程的起始地址及大小、可编程的保护及 Cache 性质。这些区域可以重叠。对重叠区域的寻址有固定的优先级。

区域定义

通过写 CP15 寄存器 6 可以定义 8 个区域中每一个的起始地址及大小。这个寄存器的格式在前面已定义。这些区域的大小最小为 4 KB，最大为 4 GB。当写 CP15 寄存器 6 时，Rd[5：1]定义区域大小，在最大和最小值之间以 2 的倍数设置，如表 11.2 所列。起始地址由 Rd[31：12]定义，必须是所选大小的倍数。在 Rd[0]置位后区域才能起作用。

表 11.2　保护单元区域大小编码

Rd[5：1]	区域大小
01011	4 KB
01100	8 KB
01101	16 KB
01110	32 KB
⋮	⋮
11100	512 MB
11101	1 GB
11110	2 GB
11111	4 GB

区域优先级

区域之间设置为重叠是合法的。当对重叠部分寻址时，保护单元以固定的优先级确定使用哪个区域来定义重叠部分的属性。区域 7 的优先级最高，而区域 0 的优先级最低。其他区域为中间优先级，按其数字排序。

将一个区域的起始地址限制为其大小的倍数，就可以通过比较寻址地址的某些最高有效位(20 位以上)与该区域的起始地址的对应位，来检查一个地址是否在该区域内。如果匹配，则这个地址就落在该区域内；否则落在区域之外。由于不需要加法或减法，这个比较过程非常快。由此，保护单元的组织如图 11.2 所示。图中区域 0 覆盖了整个 4 GB 寻址空间，因此，所有寻址都将落在该区域。区域 1、2 和 4 为 4 KB，区域 3、6、7 和 5 为 8 KB。假如寻址地址落在区域 0、3 和 6 内(以黑箭头线标注)，则优先编码器选择使用最高优先级区域，也就是区域 6 的属性。

如果地址没有落在任何使能区域内，则保护单元将产生一个异常中断。

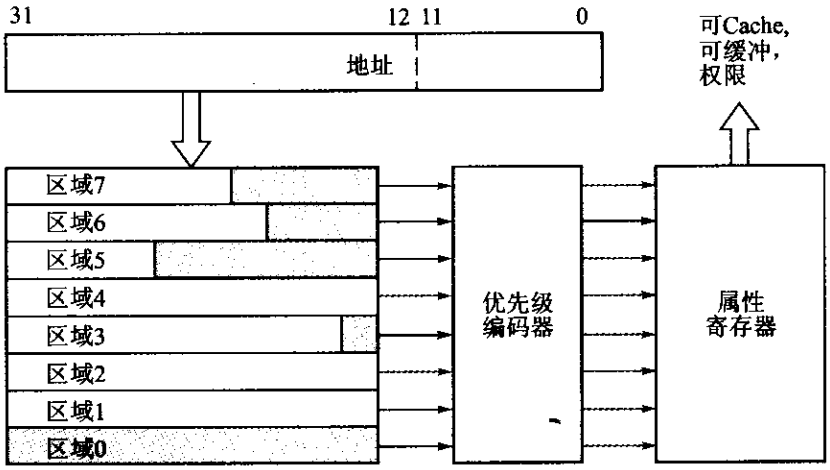


图 11.2 保护单元的组织

Harvard 核

使用 Harvard 结构的 ARM 核，如 ARM940T，在其指令及数据端口上有分开的保护单元，因此总共有 16 个区域。

11.5 CP15 MMU 寄存器

MMU 寄存器结构如表 11.3 所列。这些寄存器使用如图 11.1 所示的 CP15 指令进行读和写，其中 CRn 指定要访问的寄存器。

表 11.3 CP15 MMU 寄存器结构

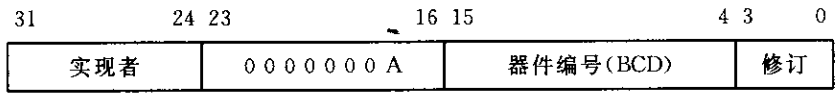
寄存器	目 的
0	ID 寄存器
1	控制
2	转换表基址
3	页域(domain)访问控制

续表　11.3

寄存器	目　的
5	故障状态
6	故障地址
7	Cache 操作
8	TLB 操作
9	读缓冲操作
10	TLB 锁定
13	程序 ID 映射
14	调试支持
15	测试及时钟控制
4,11～12	未用

这些寄存器功能的详细描述如下：

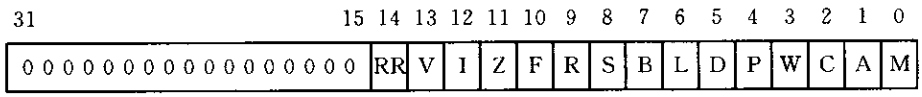
- 寄存器 0：只读。返回标识信息。



位[3：0]为修订号；位[15：4]为 3 位 BCD 码的器件编号；位[23：16]为体系结构版本（“A”=0 为版本 3；“A”=1 为版本 4）；位[31：24]为 ASCII 码的开发商标记（ASCII “A”=41_{16}表示 ARM 公司；“D”=44_{16}为 Digital 公司等等）。

有些 CPU 并不完全遵循上述寄存器 0 格式。最近的 CPU 有第二个寄存器 0（通过改变 MRC 指令的 Cop2 域访问），它给出了 Cache 的组织细节。

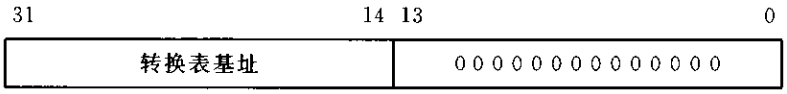
- 寄存器 1：在体系结构版本 3 中只写；在体系结构版本 4 中可读/写。包含一些控制系统功能使能以及控制系统参数的信息位。



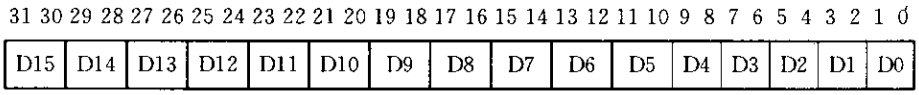
所有位在复位时清 0。若置位，则位 M 使能 MMU 单元；位 A 使能地址对准故障检查；位 C 使能数据或统一 Cache；位 W 使能写缓冲器；位 P 将 26 位异常入口切换为 32 位；位 D 将 26 位地址范围切换为 32 位；位 L 切换为后中止（late abort）时序；位 B 切换小端格式为大端格式；位 S 和 R 修改 MMU 系统和 ROM 保护状态；位 F 控制外部协处理器通信速度；位 Z 使能预转移；位 I 在数据和指令 Cache 分开时使能指令 Cache；位 V 将异常向量基地址由 0x00000000 移到 0xffff0000；位 RR 控制 Cache 替换算法（伪随机或循环）。应注意，所有这些位并不是在所有实现中都提供。

位[31:15]在读时给出不确定值，在读—修改—写访问时应当保留。例如，位[31：30]在 ARM920 和 ARM940 中用于时钟控制功能。

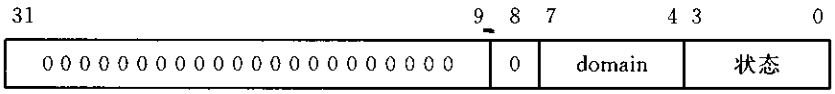
- 寄存器 2：在体系结构版本 3 中只写，在版本 4 中可读写。包含当前活动的第一级转换表的起始地址。



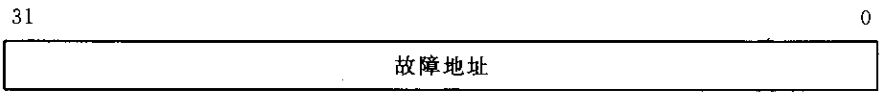
- 寄存器 3：在体系结构版本 3 中只写，在版本 4 中可读写。包含 16 个域(field)，每个 2 位，指定 16 个页域(domain)的访问权限。页域将在 11.6 节作详细介绍。



- 寄存器 5：在体系结构版本 4 中可读写。在版本 3 中为只读，对其进行写操作将刷新整个 TLB。指示故障类型以及中止的最后的数据访问域。位 D 在一个数据断点被置位。

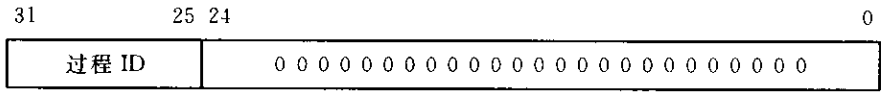


- 寄存器 6：在体系结构版本 4 中可读写。在版本 3 中为只读，对其写将刷新特定的 TLB 表项。包含中止的最后的数据访问地址。



- 寄存器 7：在体系结构版本 4 中可读写。在版本 3 中为只写并简单地刷新 Cache。用于完成一系列 Cache、写缓冲、预取缓冲和转移目标 Cache 的清除以及/或刷新操作。提供的数据应当为 0 或一个相对虚拟地址。

  访问寄存器 7 时用 Cop2 和 CRm 域指定特定的操作。其可用功能随实现而变化。
- 寄存器 8：在体系结构版本 4 中可读写，在版本 3 中没有这个寄存器。用于完成一系列 TLB 操作，刷新 TLB 单个或整个表项，支持统一或分开的指令和数据 TLB。
- 寄存器 9：如果有该寄存器，则用于控制读缓冲器。在一些 CPU 中，这个寄存器用于控制 Cache 锁存功能。
- 寄存器 10：用于控制 TLB 锁存功能(若支持这一功能的话)。
- 寄存器 13：用于通过过程 ID 寄存器重新映射虚拟地址。这个机制用于支持 Window CE，并只在特定的 CPU(如 ARM720T 和 SA-1100)中出现。如果虚拟地址的位[31：25]为 0，则用这个寄存器的位[31：25]代替。



- 寄存器 14：用于调试支持。
- 寄存器 15：用于测试。在某些 CPU 中还用于时钟控制。

11.6　ARM MMU 结构

MMU 完成两个基本功能：

- 将虚拟地址转换为物理地址。
- 控制存储器访问权限，中止非法访问。

ARM MMU 使用具有步行表(table-walking)硬件的两级页表和一个保存最近使用的页转换的 TLB。对于指令和数据 Cache 分开的微处理器，也可能有分开的指令和数据 TLB。

存储器粒度

存储器映射通过相同的基本机制以几种不同的粒度(granularity)来实现。可以使用的单位如下：

- 段(section)：为 1 MB 存储器块。
- 大页：为 64 KB 存储器块。在大页中，将访问控制施加到单独的 16 KB 子页上。
- 小页：为 4 KB 存储器块。在小页中，将访问控制施加到单独的 1 KB 子页上。
- 微页(ting page)：某些最新的 CPU 还支持 1 KB 的微页。

通常的粒度为 4 KB 的小页。大页及项可以用单个 TLB 表项映射到大块数据存储器。若强制将大块数据映射到小页，则在特定情况下，会造成 TLB 效率低下。

页域

页域(domain)是 ARM MMU 体系结构所特有的。一个页域是一组具有特定访问权限的段和/或页。这允许多个不同过程用同一个转换表执行，同时不同程序之间又有一些保护。它是一种更为轻便的过程切换机制，而不必使每个过程都有自己的转换表。

访问控制基于两类程序：

- 客户程序：是页域的使用者，必须保留构成页域的每个段和页的访问权限。
- 管理程序：是页域的管理者，可以越过个别段或页的访问权限。

在任何一个时间，一个程序可能是某些页域的客户程序，是另一些页域的管理程序，而对剩下的页域则完全没有访问权限。这由 CP15 寄存器 3 控制。对 16 个页域的每一个，寄存器 3 中有两位用于描述当前程序在每一个页域的状态。表 11.4 给出了对这两位的解释。向 CP15 寄存器 3 写一个新值可以修改一个程序对所有页域的关系。

表 11.4 页域访问控制位

值	状 态	描 述
00	禁止访问	任何访问将产生一个页域故障
01	客户程序	检查页和段的权限位
10	保留	未用
11	管理程序	不检查页和段的权限位

转换过程

对一个新的虚拟地址的转换总是从取第一级开始(在这里先忽略 TLB。TLB 只是一个 Cache,用来加速下面描述的过程)。这要用到在 CP15 寄存器 2 中保存的转换基址。转换基址寄存器的位[31：14]结合虚拟地址的位[31：20]形成一个存储器地址,用于访问第一级描述符,如图 11.3 所示。

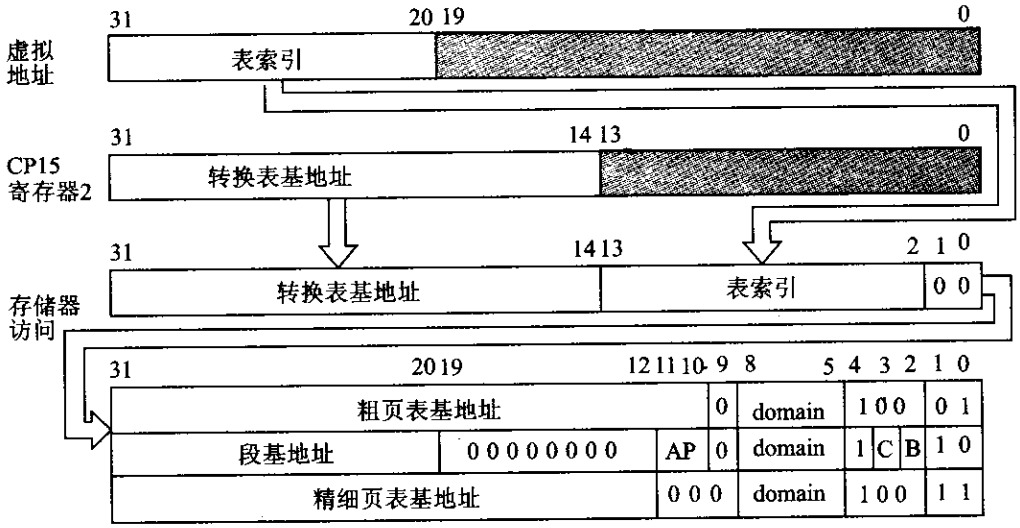


图 11.3 第一级转换的读取

第一级描述符可能是一个段的描述符,也可以是第二级页表的指针,这取决于其最低两位。"01"表示是第二级粗略页表的指针;"10"表示段描述符;"11"表示是第二级精细页表的指针(只由特定 CPU 支持);"00"用于指示一个产生转换故障的描述符。

段转换

当第一级描述符指示将虚拟地址转换为段时,对页域(段描述符中的"domain")进行检查,并且若当前过程是页域的客户程序,则对访问权限(段描述符中的"AP")也进行检查。如果允许访问,则存储器地址由段描述符的位[31：20]与虚拟地址的位[19：0]级联构成。这个地址用于访问存储器中的数据。段转换的完整次序如图 11.4 所示。

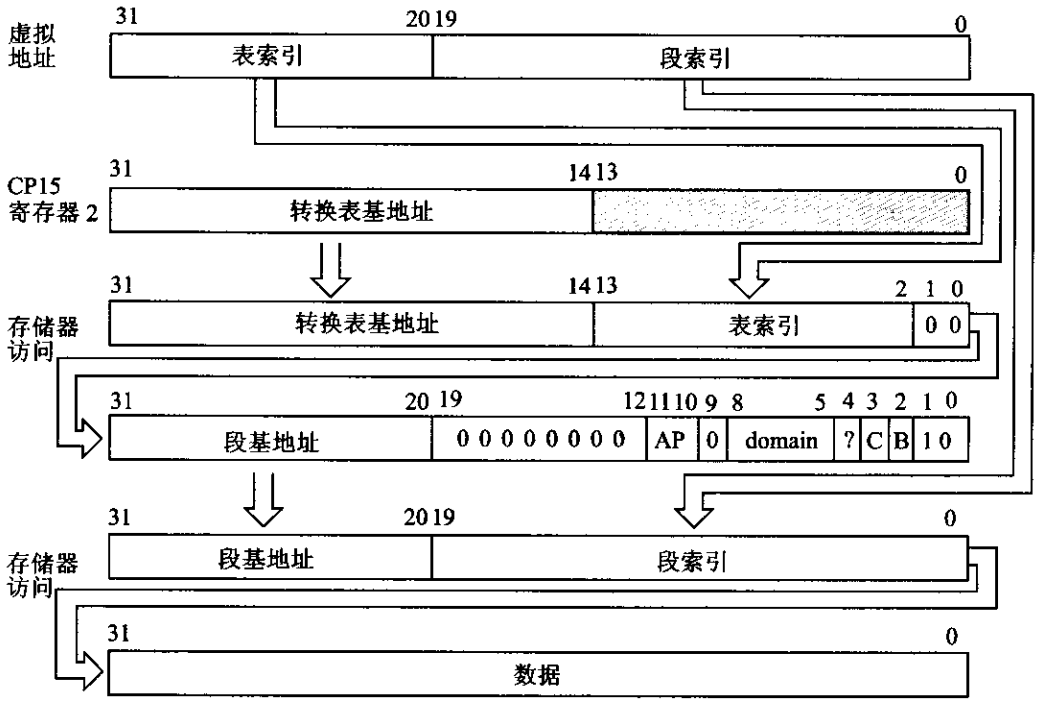


图 11.4　段转换次序

访问权限位(AP)将在本节后面的“访问权限”一段中描述。可缓冲(B)和可 Cache(C)位的操作将在本节后面的“Cache 和写缓冲控制”一段中描述。

页转换

当第一级描述符指示将虚拟地址转换为页时，需要进一步访问第二级页表。第二级粗页描述符的地址由第一级描述符的位[31：10]和虚拟地址的位[31：12]连接构成。第二级精细页描述符的地址由第一级描述符的位[31：12]和虚拟地址的位[19：10]连接构成。

第二级粗页描述符可以是一个大页(64 KB)描述符，也可以是小页(4 KB)描述符，这取决于其最低两位。“01”表示是大页；“10”表示是小页。其他值将产生陷阱，“00”用于产生一个转换故障；“11”值不应使用。第二级精细页描述符可以是一个微页(1 KB)描述符，由其最低两位为“11”指示；也可以像上面一样，是大页或小页描述符。

小页基地址保存在页描述符的位[31：12]。位[11：4]包含 4 个子页的访问权限，分别由两位构成(AP0～3)。其中子页是页大小的 1/4。位[3：2]为缓冲使能和 Cache 使能位。(标记为“?”的位用于专用实现。)

小页的整个转换次序如图 11.5 所示。除了虚拟地址的位[15：12]同时用于页表索引和页表偏移外，大页的转换次序相似。因此，大页的每个页表项必须在页表中将用于页表索引的那些位的值拷贝 16 次。

微页转换方法也是相似的，但必须从一个精细的第一级描述符开始。微页不支持子页，因此，在第二级描述符中只有一组访问权限。

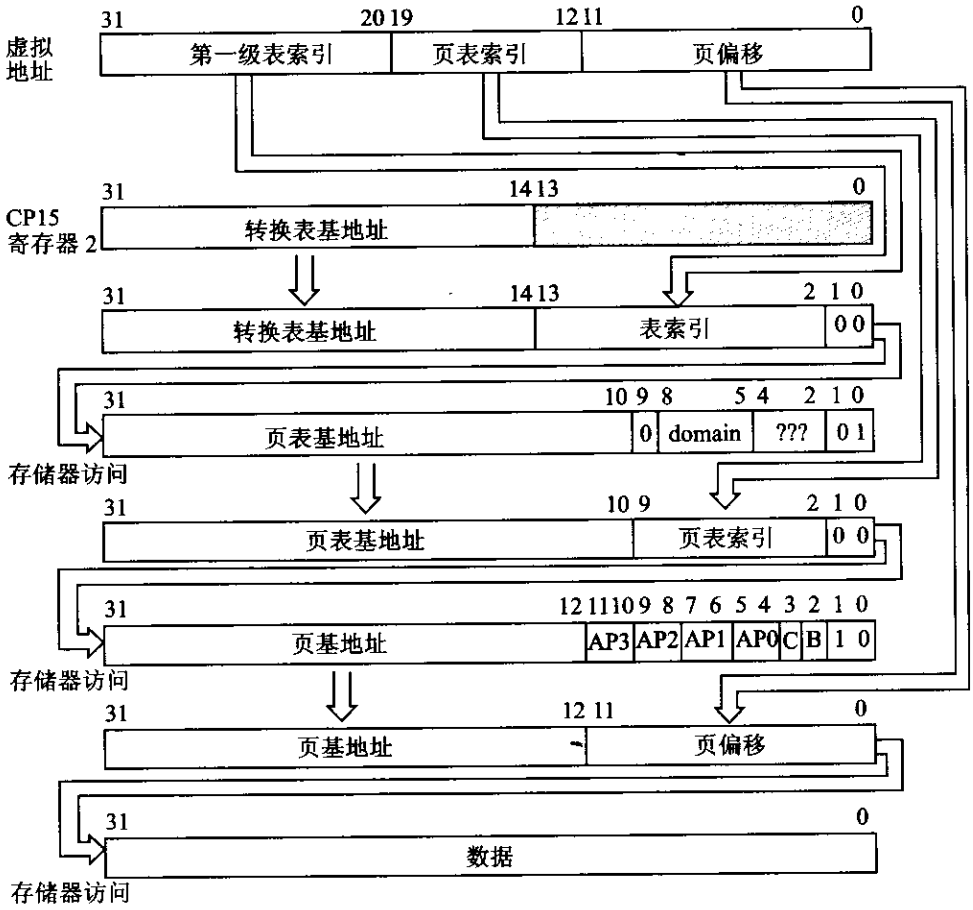


图 11.5 小页转换次序

访问权限

每一个段及子页的 AP 位与第一级描述符的页域信息、CP15 寄存器 3 的页域控制信息、CP15 寄存器 1 的 S 和 R 控制位以及微处理器的用户/超级用户状态一起，用来确定对寻址位置的读或写访问是否允许。权限检查操作过程如下：

1) 如果允许对准检查(CP15 寄存器 1 的位[1]置位)，则检查地址对准。若未对准(也就是说，如果字没有与 4 字节边界对准或半字没有与 2 字节边界对准)，则产生故障。

2) 以第一级描述符的位[8：5]确定寻址位置的页域(若第一级描述符不存在，则取描述符时产生故障)。

3) 检查页域访问控制寄存器，即 CP15 寄存器 3，确定当前过程对这个页域是用户还是管理程序。如果都不是，则产生故障。

4) 如果是这个页域的管理程序，则不管访问权限而继续进行。如果是用户程序，则使用 CP15 寄存器 1 的位 S 和 R 按表 11.5 来检查访问权限。如果不允许访问，则产生故障。若允许则继续访问数据。

表 11.5　访问权限

AP	S	R	超级用户	用　户
00	0	0	禁止访问	禁止访问
00	1	0	只读	禁止访问
00	0	1	只读	只读
00	1	1	不使用	不使用
01	-	-	读/写	禁止访问
10	-	-	读/写	只读
11	-	-	读/写	读/写

权限检查方法如图 11.6 所示。图中显示出，在地址转换过程中可能产生各种故障。MMU 可能产生对准、转换、页域以及权限故障。另外，外部存储器系统可能产生 Cache 行取(不是所有 CPU 都支持)、不可 Cache 或不可缓冲访问(中止缓冲写是不支持的)以及转换表访问故障。所有这些故障都称为中止(abort)，由处理器作为预取或数据中止异常来处理，这依赖于访问的是指令还是数据。

数据访问故障将使故障状态寄存器(CP15 寄存器 5)和故障地址寄存器(CP15 寄存器 6)更新，以提供故障产生原因和位置的信息。指令访问故障只会在指令执行时产生异常(由于指令可能是在转换后被取的，因此可能不被执行)，并且不会更新故障状态和地址寄存器。故障地址可以由返回的在链接寄存器中的地址推断出来。

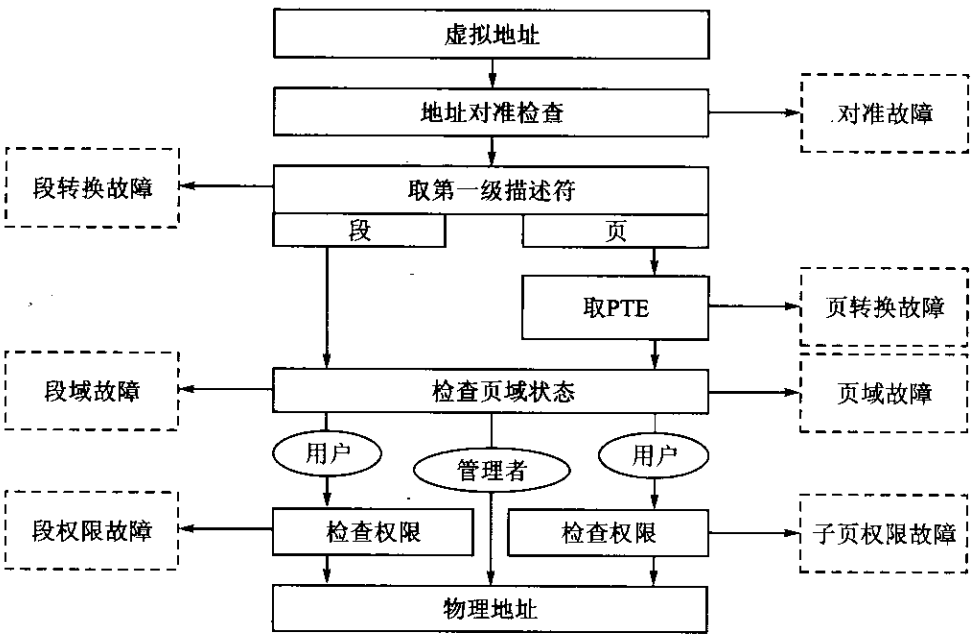


图 11.6　访问权限检查方法

Cache 和写缓冲控制

段及第二级页描述符中的位 C 和 B 用来控制是否将段或页中的数据复制到 Cache 中和/或通过写缓冲器回写到存储器中。

对于采用写直达(write-through)方法的 Cache,位 C 控制数据是否是可 Cache 的,位 B 控制是否可以写缓冲。对于采用回写方案的 Cache,可以用"可 Cache、不可缓冲"组合来描述"写直达、可缓冲"的行为。(这些 Cache 术语在 10.3 节的"写策略"中描述过。)

外部故障

应注意,当采用缓冲写时,若发生外部故障,则微处理器是不能恢复的。这是因为到指示发生故障时,微处理器可能已经执行了几条指令,因此,不可能恢复原先状态并重新执行发生故障的存储指令。若需要恢复(例如,允许微处理器在发生总线故障后重新执行存储指令),则必须采用不可缓冲写。

在典型的 ARM 应用中,没有外部故障的潜在的可恢复资源,因此这不是一个问题。

11.7 同 步

在多个进程(process)共享数据结构的系统中,常遇到的问题是如何控制共享数据的访问以保证正确操作。

例如,在一个系统中,一个进程对一系列传感器数据进行采样并保存到存储器中,以供另一个进程在任意时间使用。如果第二个进程总是能看到这些数据的一个窗口,在数据没有完全更新前保证不将第一个进程切换出去并切入第二个进程是非常重要的。这个过程采用的机制称为**进程同步**,这需要对数据结构进行互斥(mulually exclusive)访问。

互 斥

若一个进程要对共享数据结构进行操作,且操作要求不能有其他进程访问这些数据,那么必须等待其他进程完成访问后,设置一些锁以防止在完成操作之前,其他进程访问这些数据。

实现互斥的一个方法是使用特殊的存储器位置来控制对数据结构的访问。例如,这个位置可以包含一个布尔值来指示数据结构是否正在使用。想要使用数据结构的进程必须等待,在空闲后使用这些数据并将其标记为忙。在完成操作后再将其标记为空闲。可能存在的问题是数据在变为空闲和正要标记为忙之间可能产生一个中断。中断造成进程切换,而新的进程看到的是结构处于空闲状态,将其标记为忙并修改其中一位。此时,另外一个中断将控制返回给第一个进程,而这个进程处于认为结构是空闲的状态,但这是错误的。

一个常用的解决这类问题的方法是布尔值测试及置位时禁止中断。但如果处理器处于被保护的监控模式(例如 ARM),那么用户级代码不能禁止中断,因此需要系统调用。而完成系统调用并将控制返回给用户进程需要几个时钟周期。

SWAP

一个更为有效的方法是使用**原子**(atomic,也就是不可中断的)"测试及设置"指令。ARM 的 SWAP 指令(参见 5.13 节)正是为了这种用途而加入到指令集中的一条此类指令。一个寄存器先置为忙值,然后该寄存器与存储器中包含布尔值的位置进行交换。如果调入的值为空闲,则进程可以继续进行;如果是忙,则该进程必须等待,重复测试直到得到空闲值为止。

应注意的是,这是 ARM 指令集中加入 SWAP 指令的惟一原因。这条指令不会提高微处理器性能,其动态使用频率也可以忽略。它只是提供这一功能。

11.8　上下文切换

上下文(context)是保证进程正确执行所必须建立的所有系统状态。这些状态包括:

- 所有微处理器寄存器的值,包括程序计数器、堆栈指针等等;
- 浮点寄存器的值(如果进程使用的话);
- 存储器转换表(但不是 TLB 的内容。TLB 的内容只是存储器值的 Cache,在需要时会自动调入);
- 进程使用的存储器中的数据(但不是 Cache 值,因为在需要时会自动调入);
- 当发生进程切换时,必须保存上一个进程的上下文,并调入新进程的上下文(这里是假定进程不是第一次执行)。

何时切换

在产生外部中断时可能进行上下文切换,例如:

- 定时器中断使操作系统根据时间片(time-slicing)算法唤醒一个新的进程。
- 正在等待一个特定事件的高优先级进程因响应这个事件而被激活。

完成了工作的进程可以调用操作系统将其置为睡眠状态,等待外部事件将其激活。

在所有情况下,都是操作系统进行控制并负责保存老的上下文并调入新的上下文。在基于 ARM 的系统中,这些工作通常由微处理器在监控模式下完成。

寄存器状态

如果所有的上下文切换都在响应 IRQ 或内部故障或超级调用时发生,而监控代码又不重新开放中断,则进程寄存器状态可以只限于用户模式寄存器。如果上下文切换可以在响应 FIQ 时发生,或超级代码可重新开放中断,那么可能也需要保存和恢复某些特权模式寄存器。

由于认识到在特权模式下保存和恢复用户寄存器的困难性，ARM 在体系结构上对保存和恢复寄存器给予支持，提供专用指令协助实现。这些指令是多寄存器 Load/Store 指令的特殊形式(参见 5.12 节)，允许代码在非用户模式下从非用户模式寄存器寻址的存储器区域来保存和恢复用户寄存器。

若没有这些指令，则操作系统需要切换到用户模式来保存和恢复用户寄存器组，然后通过保护屏障返回超级模式。虽然这种方法可行，但它的效率很低。

浮点状态

硬协处理器中的浮点寄存器或用软仿真器保留在存储器中的浮点寄存器是进程状态的一部分。为了减小每次进程交换时保存和恢复寄存器的开销，当使用浮点的进程被切换出去时，操作系统只是简单地禁止用户级使用浮点系统。如果新的进程也需要使用浮点系统，那么第一次使用会产生陷阱。此时，操作系统保存上一个进程的状态并调入新的状态，然后重新使能浮点系统，使新的进程可以自由地使用浮点系统。

因此，只有在确实需要时才会发生浮点上下文切换的开销。

转换状态

当新老进程使用独立的转换表时，上下文切换的工作量很大。切换整个转换表结构的一个简单方法是改变 CP15 寄存器 2 的第一级页表基地址。但由于这将导致已有的 TLB 和(虚拟地址)Cache 表项无效，因而必须刷新它们。可以简单地将所有表项标记为无效而将 TLB 和指令或写直达数据 Cache 刷新。在 ARM 处理器芯片中，每个 TLB 或 Cache 的刷新只需要 1 条 CP15 指令；但对于写回式 Cache，对所有“脏”行都必须清理，因此需要花费很多指令。

(应注意，物理寻址的 Cache 不存在这种问题。但现在所有的 ARM CPU 都使用虚拟寻址的 Cache。)

当新老进程共享同一个转换表时，进程切换的工作量较小。由于 ARM MMU 结构中“页域”的机制，只需要更新 CP15 寄存器 3 就可以重新配置虚拟寻址空间中 16 个不同子集的保护状态。

为了保证 Cache 不会成为保护系统的漏洞，进行 Cache 访问的同时，必须进行权限检查。在调入一行 Cache 数据的同时，通过存储页域及访问权限的信息就可以实现这一点。但现在的 ARM 微处理器在 Cache 访问的同时，使用 MMU 中的信息进行权限检查。

11.9 输入/输出

在 ARM 系统中，输入/输出(I/O)功能是通过存储器映射的可寻址外围寄存器和中断输入的组合来实现的。某些 ARM 系统也可能有直接存储器访问(**DMA**，Direct Memory Access)硬件。

存储器映射的外围设备

外围设备(如串行线控制器)中包含一些寄存器。在存储器映射系统中,这些寄存器就像特定地址的存储器区域一样。(在其他的系统组织中,I/O 功能可能与存储器件有不同的寻址空间。)串行线控制器可能有如下寄存器:

- 发送数据寄存器(只写):写入这个位置的数据被送往串行线。
- 接收数据寄存器(只读):保存从串行线送来的数据。
- 控制寄存器(读/写):设置数据速率,管理 RTS(请求发送)和其他类似的信号。
- 中断使能寄存器(读/写):控制产生中断的硬件事件。
- 状态寄存器(只读):指示读数据是否有效,写缓冲是否满等等。

要接收数据,必须用软件适当地设置器件。通常在接收到有效数据或检测到错误时产生一个中断。中断程序必须将数据复制到缓冲器中,并进行错误检查。

存储器映射问题

应注意的是,存储器映射外围寄存器的行为与存储器不同。连续两次读数据寄存器,即使对该寄存器没有写操作,其结果也很可能不同。而对真正存储器的读是**幂等的**(idempotent)(可多次重复读,结果一致)。对外围寄存器的读操作可能清除当前值,致使下一次读结果不同。这种寄存器称为**读敏感**的。

当涉及读敏感的寄存器时,编程必须非常小心。特别是不能将这种寄存器的数据复制到 Cache 存储器。

在许多 ARM 系统中,对 I/O 寄存器不能在用户模式下访问。要访问这些器件,只能通过监控调用(SWI)或通过使用这种调用的 C 库函数。

直接存储器访问

如果 I/O 功能需要很高的数据带宽,处理来自 I/O 系统的中断可能会花费相当部分的处理器性能。许多系统采用 DMA 硬件来处理不需要处理器干预的最低级 I/O 数据传输。通常,DMA 硬件从外围端口传送数据块到存储器的缓冲区,只有发生错误或缓冲区满了时才中断处理器。这样微处理器中断将出现在每一缓冲区而不是出现在每个字节。

但是要注意,DMA 数据传送占用了部分存储器带宽,I/O 活动仍将使微处理器性能下降(尽管比利用中断来处理数据传送要小得多)。

快速中断请求

ARM 的快速中断(FIQ)结构比其他异常模式(参见图 2.1)有更多的寄存器,以减小处理这类中断时保存和恢复寄存器的开销。寄存器的数量是根据实现一个 DMA 通道的软仿真器所需要的数量来选择的。

如果一个不支持 DMA 的 ARM 系统有一个 I/O 数据传送源,并且其带宽要求远高于其他需求,那么就值得考虑将 FIQ 分配给该数据源,而用 IRQ 支持其他源。FIQ 同时支持几个不同的数据源的效率很低,尽管在不同源之间进行粗粒度切换可能比较

合适。

中断延迟

中断延迟(interrupt latency)是微处理器的一个重要参数。中断延迟是在最坏情况下响应中断的最长时间。对于 ARM6,在最坏情况下 FIQ 的延时由以下要素决定,即

1) 信号通过 FIQ 同步锁存器所需要的时间,最坏为 3 个时钟周期。
2) 最长指令(也就是 16 个寄存器的多寄存器 Load 指令)完成时间,为 20 个时钟周期。
3) 数据中止进入序列所需的时间,为 3 个时钟周期(数据中止的优先级高于 FIQ,但不屏蔽 FIQ 输出,参见 5.2 节)。
4) FIQ 进入序列所需的时间,为 2 个时钟周期。

由此,总的最坏延迟为 28 个时钟周期。在这个时间后,ARM6 执行 0x1C 处,也就是 FIQ 入口处的指令。这些周期可以是顺序的,也可以是非顺序的。若访问的存储器速度低,则延迟会更长。最好延迟为 4 个时钟周期。

IRQ 延迟的计算方法相似,但必须包括 FIQ 响应程序的最长绝对延迟(这是由于 FIQ 的优先级高于 IRQ)。

Cache 和 I/O 交互作用

通常假定 Cache 会使处理器更快。一般情况下,若性能指的是在一段合理时间内的平均值,那么这种假定是正确的。但在很多情况下会使用中断,这时最坏情况的实时响应是非常关键的。在这种情况下,Cache 会使性能显著下降。MMU 也会使情况变坏。

在第 12 章中将会介绍的 ARM710 的最坏中断延迟包括:

1) 请求信号通过 FIQ 同步锁存器所需要的时间,像前面一样为 3 个时钟周期(最坏)。
2) 完成最长指令(16 个寄存器多寄存器 Load 指令)所需时间,同前面一样为 20 个时钟周期。但这可能造成写缓冲器刷新,这需要 12 个时钟周期;然后会发生 3 个 MMU TLB 未命中,需要 18 个时钟周期;6 个 Cache 未命中,需要 24 个时钟周期。原先的 20 个时钟周期与 Cache 行取重叠,但这一步总的花销为 66 个时钟周期。
3) 数据中止进入序列所需的时间,同前面一样,为 3 个时钟周期。但从向量空间读取可能增加 1 个 MMU 未命中和 Cache 未命中,这会增加 12 个时钟周期。
4) FIQ 进入序列所需的时间,同前面一样为 2 个时钟周期。但可能造成另一次 Cache 未命中,需花费 6 个时钟周期。

现在的总延迟为 87 个时钟周期,其中大多数为非顺序存储器访问。由此可见,支持存储器架构的自动机制在平均情况下会加速通用程序,但对关键代码段的最坏情况计算则有相反作用。

减小延迟

当必须满足实时约束时，如何才能减小延迟呢？

- 固定区域的快速片上 RAM(如将向量空间保存在存储器的低端)将加速异常的进入。
- **锁定** TLB 的项和 Cache 以保证关键代码段永不发生未命中问题。

应注意到，在 Cache 和 MMU 通常有效的通用系统中也经常存在实时约束问题，例如磁盘数据交换或局域网管理，特别是在只有很小的 DMA 硬件支持的低成本系统中。

其他 Cache 问题

对于 Cache 还有一些其他事情需要当心，例如：

- Cache 假定对同一地址，在没有新数据写入之前，每次读都会返回相同数据。但 I/O 器件不是这样，每次读操作返回的是下一片数据。
- Cache 每次取数据块(通常为 4 个字)的地址是顺序的。I/O 器件中通常相邻地址的寄存器有不同的功能。同时对它们读会给出不可预测的结果。

因此，将存储器的 I/O 区域通常标记为非 Cache 区，并绕过 Cache 访问。通常 Cache 与读敏感的器件互相排斥。显示帧缓冲器也需要仔细考虑，通常也设为不可 Cache。

操作系统问题

通常，所有 I/O 器件寄存器的底层细节以及中断处理都由操作系统负责。发送数据到串行口的通常作法是将要发送的下一字节数据装入 r0，然后进行相应的超级调用。操作系统调用一个称为**器件驱动程序**的子程序来检查发送缓冲器是否空闲，传送线是否活动(active)，以及是否发生传送错误等等。甚至可以调用一个进程向操作系统传送一个指针，将整个缓冲器中的数据发送出去。

由于将装满数据的缓冲器送到串行线需要一些时间，操作系统可以返回进程控制直至发送缓冲器有空闲空间为止。传输线硬件设备发来的中断将控制返回给操作系统，操作系统向缓冲器填充数据后将控制返回给中断进程。下一次中断引起下一次发送，直至整个缓冲器中的数据发送出去为止。

有可能发生这样的情况：请求传输线的进程完成了工作，从定时器发来的中断或其他源激活了其他进程。操作系统在修改转换表时必须小心，保证不会造成数据缓冲器不能访问。对其他请求向传输线发送数据的进程的处理也要谨慎，避免影响正在传送数据的第一个进程。使用**资源分配**以保证在共享资源的使用上不会发生冲突。

一个进程可能在请求一个输出功能后进入停止状态(inactive)，直到完成输出为止，也可能进入停止状态等待一个特定的输入。也可以将一个请求存入操作系统，在发生输入/输出事件时激活它。

11.10 例题与练习

例题 11.1

在 ARM 中为什么用户级代码不能禁止中断?

若允许用户禁止中断,则不可能实现安全的操作系统。下列代码显示了一个恶意用户如何破坏所有正在活动的程序,即

```
        MSR     CPSR_f, #&c0        ; 禁止 IRQ 和 FIQ
HERE    B       HERE                ; 死循环
```

一旦禁止中断,操作系统没有办法再获得控制权,程序陷入死循环。惟一的方法是硬复位,这将破坏所有当前正在活动的程序。

如果用户不能禁止中断,则操作系统可由定时器建立一个周期性的规则中断,可打断无限循环并调度其他程序。若操作系统给这个程序设置了一个 CPU 时间上限,则这个程序将产生时间溢出;或者在切入时继续循环,在有计费的系统中得到一个大账单。

练习 11.1.1

在一个安全的操作系统中,对低层存储器(这里保存着中断向量)最小级别的保护是什么?

练习 11.1.2

如果 ARM 没有 SWAP 指令,则设计一个可支持同步的硬件外围设备。(提示:不能由标准存储器实现,该存储部件必须是读敏感的。)

第 12 章　ARM CPU 核

本章内容综述

虽然某些 ARM 应用只采用简单的整数微处理器核作为基本处理元件，但还有一些应用需要其他一些紧密相关的功能，如 Cache 存储器和存储器管理硬件。ARM 公司提供了一系列这类基于整数微处理器核的 CPU 配置。

这里介绍的 ARM CPU 核包括 ARM710T/720T/740T、ARM810（现在已由 ARM9 系列代替）、StrongARM、ARM920T/940T 以及 ARM1020E。这些 CPU 包括各种流水线和 Cache 组织，很好地说明了设计低功耗、高性能处理器时会遇到的各种问题。

Cache 存储器的基本作用是满足处理器核的指令和数据带宽需求，因此，Cache 组织与特定微处理器核需要紧密结合。在 SoC 设计中，Cache 的目的是降低 CPU 核对外部存储器带宽的需求，使得片上总线能够承担。如果直接与 AMBA 总线连接，则高性能 ARM 处理器核的执行速度要稍快于 ARM7TDMI 的。因此，通常采用高速局部存储器或 Cache。

存储器管理也是一个复杂的系统功能，需要与微处理器核紧密结合。它可以是一个完全基于转换的系统，也可以是一个简单的保护单元。ARM CPU 核在单个宏单元（macrocell）中集成了处理器核、Cache、MMU，通常还有 AMBA 接口。

12.1　ARM710T/720T/740T

ARM710T/720T/740T 在 ARM7TDMI 微处理器核（参见 9.1 节）的基础上增加了一个 8 KB 的指令和数据混合 Cache。外部存储器和外围器件通过 AMBA 总线主控单元访问，同时还集成了写缓冲器以及存储器管理单元（ARM710T/720T）或存储器保护单元（ARM740T）。

ARM710T 和 ARM720T 的 CPU 有相似的组织结构，如图 12.1 所示。

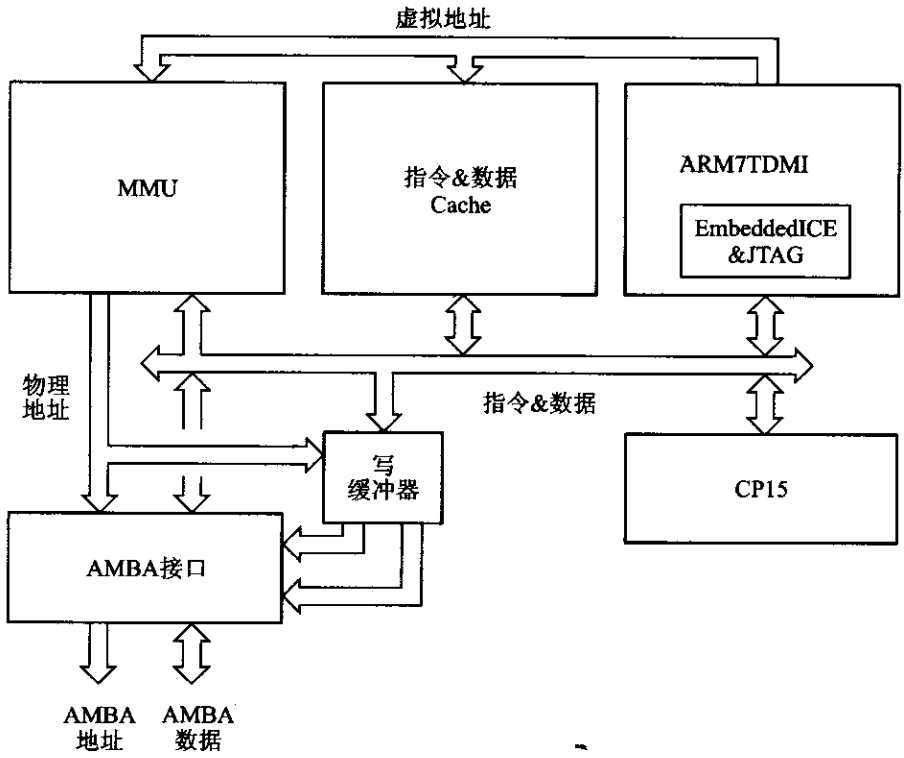


图 12.1 ARM710T 和 ARM720T 的组织结构

ARM710T 的 Cache

由于 ARM7TDMI 微处理器核只有一个存储器接口，因此，在逻辑上适合采用统一的指令和数据 Cache。ARM710T 集成了一个 8 KB 的 Cache。Cache 采用 4 路组相联结构，每个 Cache 行为 16 字节。采用随机替换算法，在 Cache 未命中时，有 4 个位置可以装入新数据。由于目标时钟速率只比标准片外存储器件的高几倍，因此 Cache 采用写直达策略。

ARM710T 的 Cache 组织如图 12.2 所示。虚拟地址的位[10：4]用于 4 个 Tag 存储器索引。Tag 包含了虚拟地址的位[31：11]，因此，Tag 与当前虚拟地址的位[31：11]进行比较。如果有一个 Tag 匹配，则 Cache 命中。用匹配 Tag 存储器的 2 位编号加上虚拟地址的位[10：4]访问数据 RAM 的对应行。虚拟地址的位[3：2]选择行中的字。如果需要访问字节或半字，则用位[1：0]对字进行选择。

将 ARM710T 的 Cache 组织与 ARM3 和 ARM610(详细介绍见 10.4 节)进行比较很有意义，因为它们的 Cache 设计都有类似的高性能和低功耗操作问题。虽然在这类应用中如何设计 Cache 还没有最终定论，但是，设计者可以通过对这些 ARM 芯片的探讨来获得设计的指导。

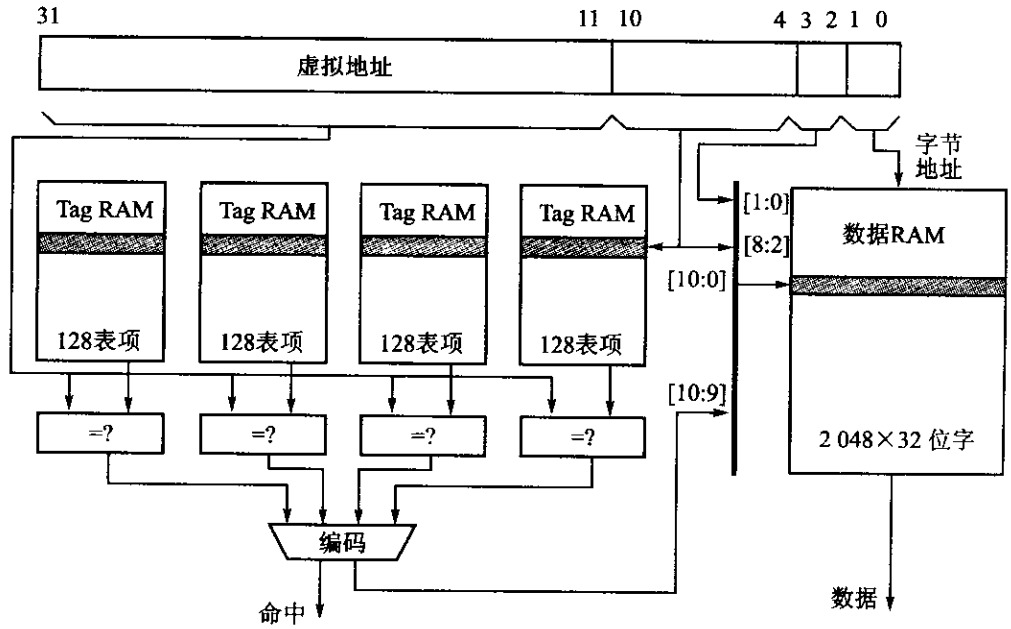


图 12.2　ARM710T Cache 的组织

Cache 速度

高相联度 Cache 的命中率最好，但需要顺序访问 CAM 和 RAM，而这限制了一个周期的时间。低相联度 Cache 可并行访问 Tag 和数据，使其周期时间最短。虽然直接映射 Cache 的命中率远低于全相联 Cache 的，但从直接映射变为 2 路或 4 路映射后，已经获得由相联带来的大部分好处，超过 4 路后效果提高就很不明显了。但是，一个全相联 CAM-RAM Cache 比 4 路组相联 RAM-RAM Cache 要简单得多。

Cache 功率

CAM 需要在每个周期对每个数据项进行并行比较，因此耗电稍微大些。通过稍微减小相联度来将 Cache 分段会增加一些复杂性，但可以使 CAM 只有部分活动，从而显著降低功耗。

在静态 RAM 中，主要的功耗来源是模拟灵敏放大器。在 4 路相联 Cache 中，Tag 存储器中的灵敏放大器活动次数 4 倍于直接映射 Cache。如果并行访问 Tag 和数据以提高速度，那么数据存储器中灵敏放大器的活动次数也为 4 倍。（相反，RAM-RAM Cache 可以串行访问 Tag 和数据。若 Tag 存储器命中，则访问对应的数据 RAM，从而降低功耗。）数据一旦有效，采用自定时关电电路来关闭灵敏放大器即可降低功耗，但灵敏放大器的功耗仍然非常可观。

顺序访问

微处理器访问的存储器数据位于 Cache 的同一行时，应该可以在第一次访问之后跳过其他的 Tag 查找。ARM 产生一个信号来指示下一个存储器访问是否是顺序访

问。用这个信号和当前地址可以推断出访问是否在同一 Cache 行。(这不会检测到所有的同一 Cache 行访问,但可以检测到大多数,并且实现非常简单。)

对于同一 Cache 行访问,跳过 Tag 查找提高了访问速度并降低了功耗。顺序访问可以使用低灵敏放大器(可以使用标准逻辑而不是模拟电路),并大大降低功耗。但必须小心以保证位线上增加的电压摆幅产生的功耗不高于在灵敏放大器上节省的功耗。

功率优化

设计 Cache 时必须注意要降低的是整个系统的功耗,而不只是 Cache 的功耗。片外存储器访问的功耗远高于片上存储器访问的,因此,首要任务是使存储器组织有好的命中率。确定使用高相联 CAM-RAM 组织还是组相联的 RAM-RAM 组织,则需要对所有设计问题进行详尽调查。低层电路,如灵敏放大器或 CAM 命中检测器的低功耗新设计方法会对其产生很大的影响。

采用顺序访问降低功耗并提高性能是一个很好的想法。对 ARM 典型程序动态执行的统计表明,75%的访问是顺序的。由于对顺序访问的处理基本上比较简单,因此,这个统计结果不应忽视。对于功耗要求非常高的情况,牺牲性能而采用两个时钟周期进行非顺序访问是值得的。这会使性能只下降 25%,但可以将 Cache 的功耗需求降低 2～3 倍。

低功耗研究方面一个有意思的问题是,一个在功耗上最好的 Cache 组织是否必须同时在性能上也是最好的。

ARM710T 的 MMU

ARM710T 存储器管理单元实现了 11.6 节中描述的 ARM 存储器管理结构。它使用 11.5 节中介绍的系统控制协处理器。

TLB 是一个 64 数据项的相联 Cache,并采用最近使用的算法。算法去除了大部分访问的两级查表,加速了转换速度。

ARM710T 的写缓冲器

写缓冲器中有 4 个地址和 8 个数据字。存储器管理单元定义哪个地址是可缓冲的。每个地址可以与任何数量的数据字相关,因此,写缓冲器可以保存写往 1 个地址的 1 个数据字(或字节)和写往其他地址的 7 个数据字,或写往不同地址的两块(每块 4 个字)等等。与特定地址相关的数据字被顺序写入以这个地址开始的存储器中。

(很明显,多个数据字与一个地址相关基本上是由多寄存器 Load 指令产生。其他可能的情况只能是外部协处理器的传送数据。)

映射如图 12.3 所示。图中第 1 个地址映射 4 个数据字,第 2 个和第 3 个地址各映射 1 个数据字。第 4 个地址在当前还没有用上。当所有 4 个地址都被用或所有 8 个数据字为满时,写缓冲器满。

微处理器以快速(Cache)时钟速度写入缓冲器,并继续执行 Cache 中的指令,同时写缓冲器以存储器时钟速度向存储器写入数据。当微处理器所需的指令和数据在 Cache 中,并且在写操作时写缓冲器还未填满时,微处理器速度完全不受存储器速度的

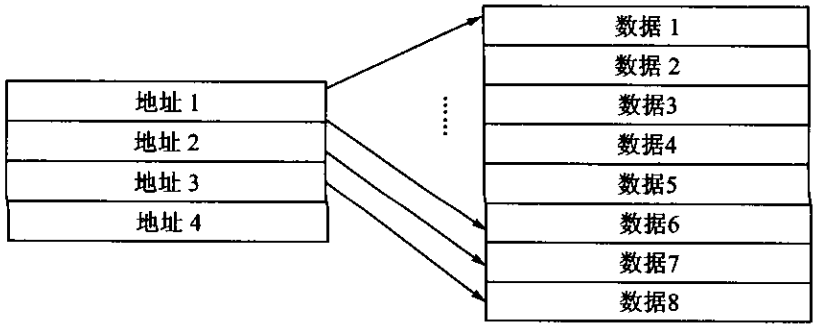


图 12.3　写缓冲器映射的例子

影响。写缓冲器以不大的硬件代价换来约 15%的性能提高。

写缓冲器的主要缺点是当缓冲写产生外部存储器故障时，由于微处理器状态不可恢复而使指令不可能重新执行。微处理器仍然可以支持虚拟存储器，这是因为转换故障可由片上 MMU 检测，使异常在数据写入写缓冲器之前产生。但如果使能写缓冲器，则不能支持基于软件错误恢复的存储器纠错。

ARM710T 芯片

以 0.35 μm CMOS 工艺实现的 ARM710T 的特点如表 12.1 所列。

表 12.1　ARM710T 的特点

工　艺	金属层	Vdd	晶体管	核面积	时　钟	MIPS	功　耗	MIPS/W
0.35 μm	3	3.3 V	N/A	11.7 mm²	0～59 MHz	53	240 mW	220

ARM720T

ARM720T 与 ARM710T 很相似，但有以下扩展：

- 位于低 32 MB 地址空间的虚拟地址可以重定位到进程 ID 寄存器(CP15 寄存器 3)指定的 32 MB 存储器范围。
- 异常向量可以从存储器低端移到 0xffff0000，以避免上面的机制对其产生影响。这个功能由 CP15 寄存器 1 的位 V 控制。

这些扩展用于提高 CPU 核的能力以支持 Window CE 操作系统。这由 11.5 节介绍的 CP15 MMU 控制寄存器实现。

ARM740T

ARM740T 与 ARM710T 的惟一不同之处是，用一个简单的存储器保护单元代替 ARM710T 的存储器管理单元。存储器保护单元采用 11.4 节描述的结构，并由 11.3 节中介绍的系统控制协处理器来实现。

保护单元不支持虚拟到物理存储器地址的转换，但以较小的代价提供了基本的保护及 Cache 控制功能。这适合于运行固定软件系统的嵌入式应用。对于这类应用，完

整的地址转换的代价过于高昂。同时由于一次 TLB 未命中会导致几次外部存储器访问，因此，省掉地址转换硬件提高了性能和功耗效率。

ARM740T 的组织如图 12.4 所示。

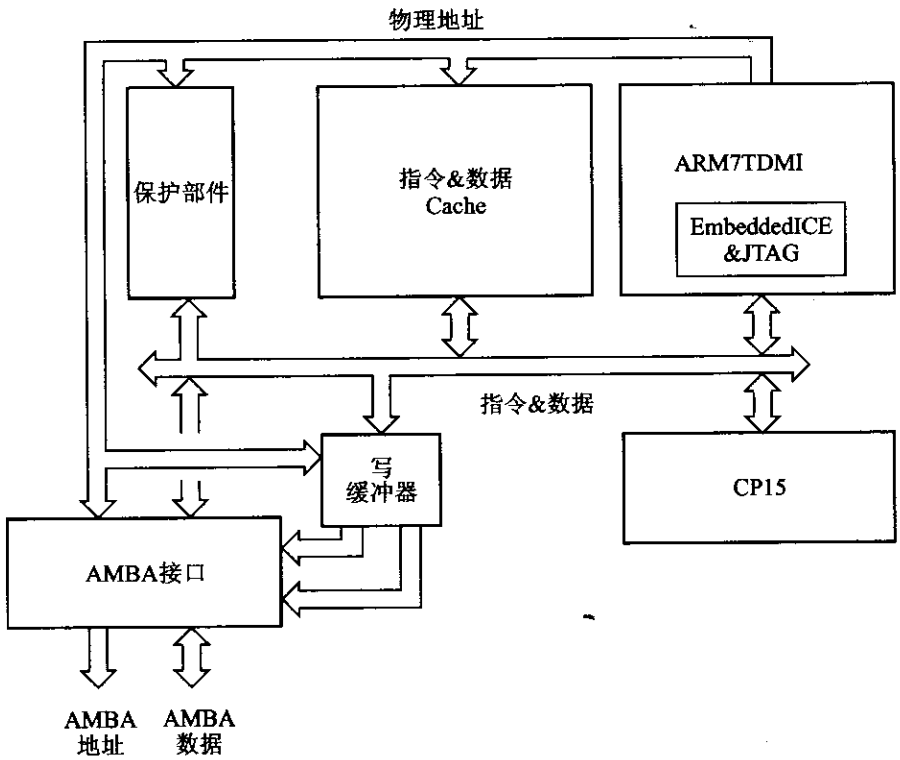


图 12.4 ARM740T 组织

ARM740T 芯片

以 0.35 μm CMOS 工艺实现的 ARM740T 的特点如表 12.2 所列。

表 12.2 ARM740T 的特点

工 艺	金属层	Vdd	晶体管	核面积	时 钟	MIPS	功 耗	MIPS/W
0.35 μm	3	3.3 V	N/A	9.8 mm^2	0～59 MHz	53	175 mW	300

12.2 ARM810

ARM810 是高性能 ARM CPU 芯片，具有片上 Cache 及存储器管理单元。ARM810 的流水线与用于原 Acorn Computer 公司设计的 ARM 芯片并继续用于 ARM6 和 ARM7 的流水线根本不同，第一次实现了 ARM 公司开发的 ARM 指令集。现在 ARM810 已由 ARM9 系列取代。

ARM8 核是 ARM810 的整数处理单元，在第 9.2 节已介绍过。ARM810 在基本 CPU 的基础上增加了如下片上部件：

- 8 KB 虚拟地址的指令和数据统一 Cache：Cache 采用写回(或写直达，由页表项控制)策略，并按 ARM8 核的要求提供了双倍带宽的能力。Cache 为 64 路相联，并由 1 KB 的部件构造，以简化将来嵌入式系统芯片中较小 Cache 或采用更为先进的工艺技术的较大 Cache 的开发。它被设计为 Cache 的各区域可以锁定，以保证在许多嵌入式系统中常遇到的对速度要求比较严格的那部分代码不会被冲刷掉。
- 存储器管理部件：采用 11.6 节中描述的 ARM MMU 结构，使用 11.5 节中介绍的系统控制协处理器来实现。
- 采用写缓冲器：当微处理器写外部存储器时可以继续执行其他指令。

ARM810 的组织如图 12.5 所示。

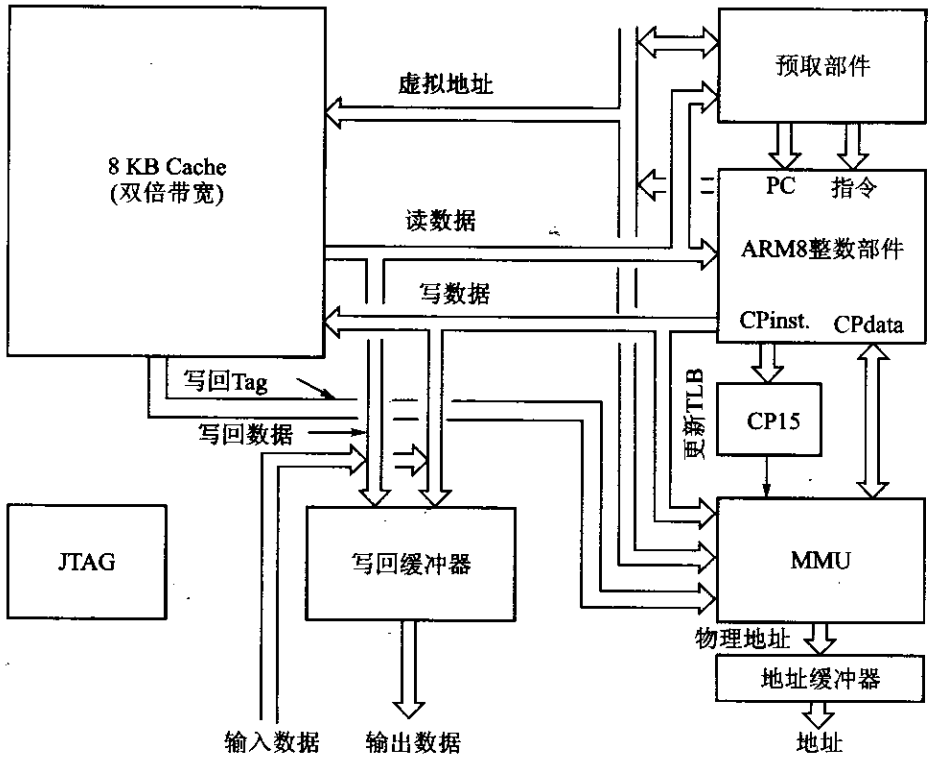


图 12.5　ARM810 的组织

双倍带宽 Cache

ARM8 核的双倍带宽需求由 Cache 实现。外部存储器访问还是采用传统的行再填充(line refill)和单独的数据传送协议。只有在顺序存储器访问时，也就是在指令预取和多寄存器 Load 时，Cache 才能提供双倍带宽。

由于 ARM7TDMI 的流水线结构使存储器接口几乎在每个周期都使用，因此，如

果要减小微处理器的 CPI(每条指令的时钟数),则必须采用一些方法以提高存储器的有效带宽。StrongARM(在 12.3 节介绍)采用分开的指令和数据 Cache 来增加带宽(由于 ARM 产生的指令流差不多是数据流的两倍,致使这些带宽不能充分利用,因此,增加的有效带宽大约为 50%)。ARM810 通过每个时钟周期读取两个顺序字来增加带宽。但一般只有约 75% 的 ARM 存储器访问是顺序的,因此,增加的可用带宽约为 60%。虽然 ARM810 的方法增加了带宽,但指令和数据访问之间产生冲突的机会也增加了。上述两种方法之间的优缺点比较也不是一件简单的事情。

由于 Cache 采用写回策略并进行虚拟寻址,因此清除"脏"行需要地址转换。ARM810 采用的机制是将虚拟 Tag 送到 MMU 进行转换。

ARM810 芯片

图 12.6 是 ARM810 芯片的照片。图的上方是 ARM8 核的数据通路,正下方是控制逻辑。MMU TLB 在图的右上角。芯片的下半部分是 8 个 1 KB Cache 块。

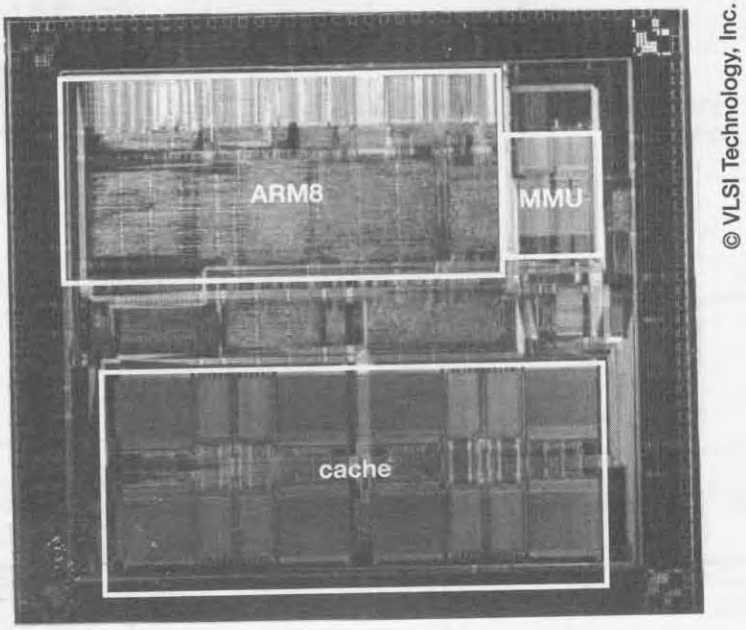


图 12.6 ARM810 管芯版图照片

ARM810 的特点总结如表 12.3 所列。

表 12.3 ARM810 的特点

工 艺	金属层	Vdd	晶体管	管芯面积	时 钟	MIPS	功 耗	MIPS/W
0.5 μm	3	3.3 V	836 022	76 mm^2	0~72 MHz	86	500 mW	172

12.3　StrongARM SA-110

StrongARM CPU 是由 DEC 公司和 ARM 公司合作开发的。它首次采用修改的 Harvard 结构(分开的指令和数据 Cache)的 ARM 微处理器。

自 1998 年 Intel 公司接管 Digital 半导体公司到现在,SA-110 由 Intel 公司生产。

Digital 公司的 Alpha 背景

在微处理器行业中,Digital 公司的 Alpha 微处理器闻名遐迩。Alpha 微处理器是一个工作频率非常高的 64 位 RISC 微处理器。获得这样高的时钟频率是基于先进的 CMOS 工艺技术、仔细平衡的流水线设计、精心设计的时钟方案以及用内部设计工具实现的对这些部件非常良好的控制。

在 StrongARM 设计中采用了同样的技术,并且进一步考虑了功耗效率。

StrongARM 组织

StrongARM 的组织如图 12.7 所示。其主要特点如下:

- 具有寄存器前推的 5 级流水线。
- 除 64 位乘法、多寄存器传送和存储器/寄存器交换指令外,其他所有普通指令均是单周期指令。
- 16 KB、32 路相联的指令 Cache,每行 32 字节。
- 16 KB、32 路相联的写回式数据 Cache,每行 32 字节。
- 分开的 32 数据项指令和数据地址转换后备缓冲器。
- 8 数据项的写缓冲器,每个数据项 16 个字节。
- 低功耗的伪静态操作。

微处理器使用系统控制协处理器 15 来管理片上 MMU 和 Cache 资源,并且集成了 JTAG 边界扫描测试电路以支持印制板连接测试。(没有实现器件内部电路测试的 JTAG in-test 指令。)

第一个 StrongARM 芯片采用 Digital 的 0.35 μm 3 层金属 CMOS 工艺来实现,约 2 500万个晶体管,面积为 50 mm^2(对于这种性能的微处理器来说是非常小的)。时钟频率在 160～200 MHz 时,达到 200～250 Dhrystone MIPS。供电电压为 1.65～2 V 时,功耗在 0.5～1 W 之间。

StrongARM 处理器核

处理器核采用典型的 5 级流水线,并实现全旁路(寄存器前推)和硬件互锁。ARM 指令集要求在寄存器堆读访问之前进行指令部分译码,并要求移位操作和 ALU 串行工作。所有这些额外的逻辑功能被嵌在相应的流水级中,并没有增加流水深度。这 5 级流水线是:

1) 取指(从指令 Cache)。

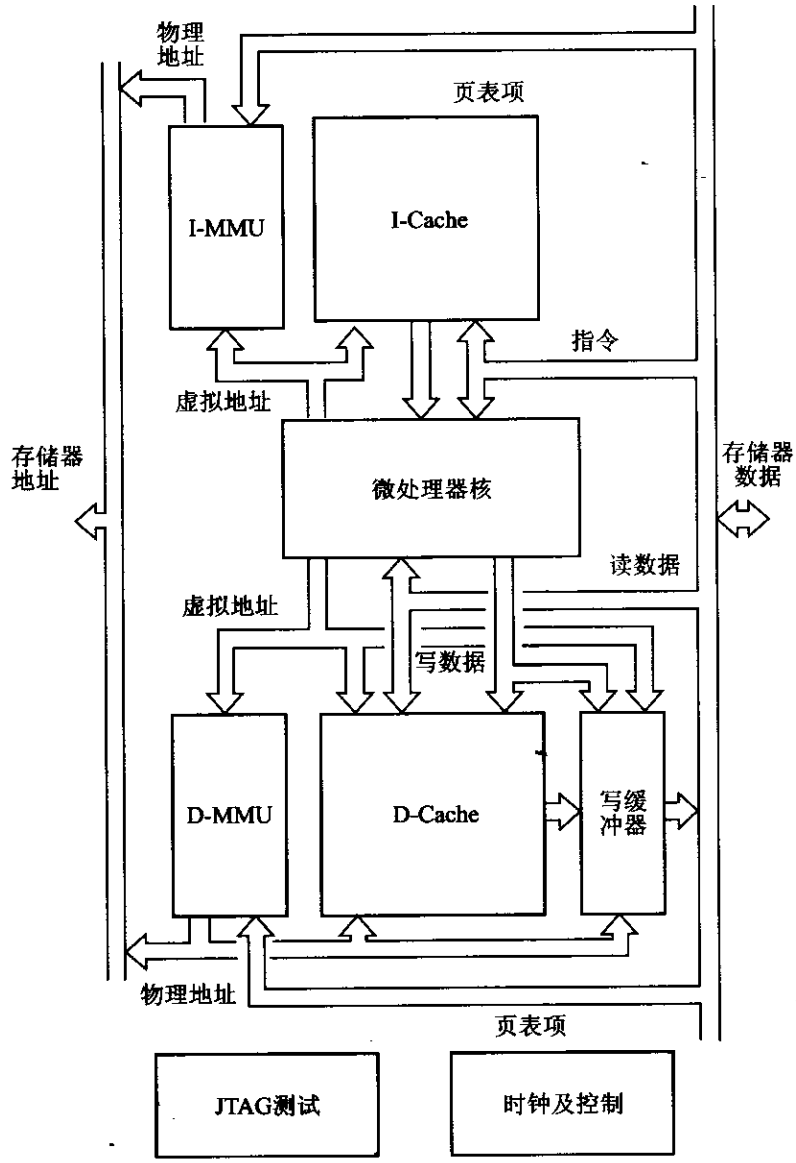


图 12.7 StrongARM 组织

2）指令译码及寄存器读，转移目标计算及执行。

3）移位及 ALU 操作，包括数据传送的存储器地址计算。

4）数据 Cache 访问。

5）结果写回到寄存器文件。

主要流水线部件的组织如图 12.8 所示。流水级由阴影条划分。穿过这些阴影条的数据会在交叉点锁存。从阴影条两端通过的数据则向前或向后跨过流水线级，例如：

- 寄存器前推通路将中间结果传给下一条指令，以避免读后写冒险引起的寄存器互锁停顿。
- 从下一条指令的取指级传送 PC＋4 的 PC 通路给出当前指令的 PC＋8，作为 r15 并用于转移目标计算。

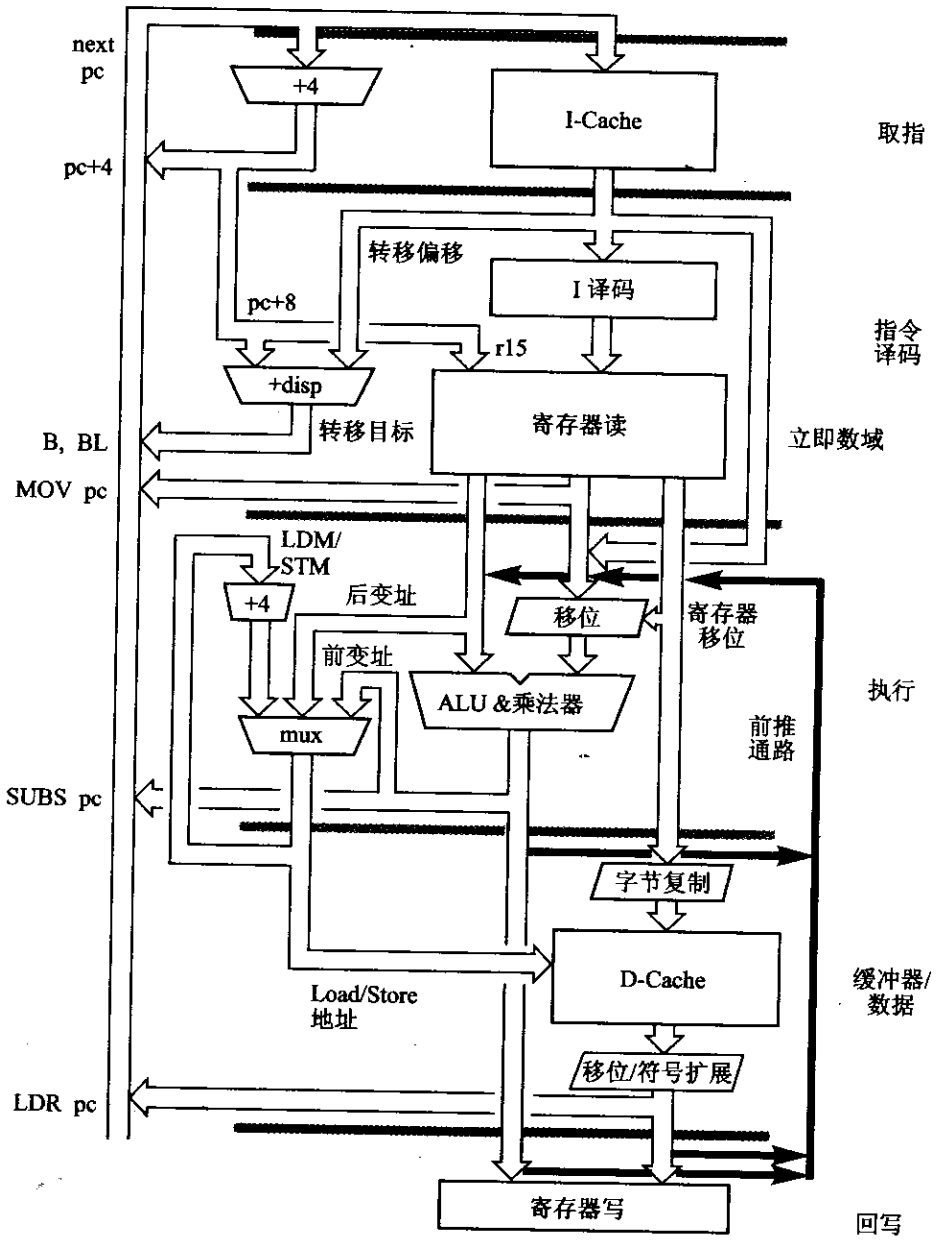


图 12.8　StrongARM 核流水线组织

流水线的特点

在这个流水线结构中需要特别指出的是：

- 要在 1 个周期内完成寄存器控制的移位和带基址变址寻址的存储操作，寄存器需要有 3 个读端口。
- 要在 1 个周期内完成自动变址的 Load 操作，寄存器需要两个写端口。
- 执行级的地址增值器支持多寄存器 Load/Store 指令。
- 有很多源可以产生下一个 PC 值。

最后一点表明，由于在 ARM 结构中 PC 作为寄存器堆的 r15 是可见的，因而对 PC 的修改有多种途径。

PC 的修改

下一 PC 值大多是由取指级的 PC 增值器产生的。这个值在下一个周期开始时有效，这样可以在每个周期取一条指令。

其次 PC 值还常产生于转移指令。目标地址是在指令译码周期由专用的转移位移加法器计算出来的。当进行一次转移时，除执行转移指令的周期外，在转移指令后还要有 1 个周期的损失。由于偏移域在指令中的位置是固定的，因此，位移加法与指令译码可以并行操作。如果转移条件不成立，则只是简单地丢弃计算出的目标地址。

例如，通常从循环中退出的代码序列为

```
CMP     r0, #0
BNE     label
...
```

在这个序列中流水线的操作如图 12.9 所示。应注意的是如何及时地产生条件码以降低转移损失。紧跟在转移指令后面的指令在取指后放弃。在下一个周期处理器取转移目标处的指令。转移目标地址与决定是否转移的条件是并行产生的。

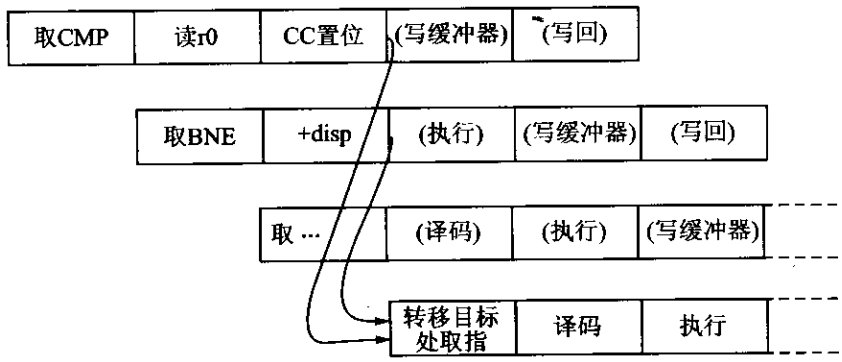


图 12.9 StrongARM 循环测试的流水线行为

同样，转移和链接指令也要损失 1 个周期。其转移过程与转移指令相似，但在执行级和写回级计算 PC+4 并写到链接寄存器 r14。正常的子程序返回指令“MOV PC, OR”也要损失 1 个周期。其转移目标地址来自寄存器文件而不是转移位移加法器。转移目标地址也是在译码周期末有效。

若必须计算出返回地址，例如从一个异常中返回，那么由于 ALU 结果在执行级末才能给出，因此会有 2 个时钟周期的损失；若新的 PC 值要从存储器中读入（从一个跳转表，从堆栈中返回的或子程序），则有 3 个时钟周期的损失。

前推通路

在流水线执行级，3 个寄存器操作数的每一个都有前推通路，以避免在读后写冒险

发生时流水线产生停顿。需要前推的值如下：

1) ALU 的结果；

2) 从数据 Cache 加载的数据；

3) 缓冲后的 ALU 结果。

除了加载的数据值被后面的指令使用的情况外(在这种情况下需要 1 个周期的停顿)，这些通路消除了几乎所有因数据相关而引起的停顿。

中止的恢复

从表面上看，如果在执行级后面一级将 ALU 结果写入寄存器文件，而不是将其缓冲并延迟到最后一级，则可以减少 1 条前推通路。这种延迟方案的优点是，在数据 Cache 访问期间可能产生数据中止，此时可能需要补救动作以恢复基址寄存器值(例如，在故障产生前多寄存器 Load 序列覆盖了基址寄存器)。这种方法不但能够恢复，还支持最清洁的恢复机制，使基址寄存器一直保持指令开始时的原值，在处理异常时就不必再撤消任何自动变址。

乘法器

StrongARM 的乘法部件很有特色。不论处理器的时钟速率有多高，乘法器每周期都计算 12 位，用 1～3 个时钟周期计算两个 32 位操作数的乘积。对于数字信号处理性能要求很高的应用来说，StrongARM 的高速乘法器有很大的潜力。

指令 Cache

16 KB 的 I-Cache 分 512 行，每行 8 条指令(32 字节)。指令 Cache 为 32 路相联(使用 CAM-RAM 组织)，采用循环替换算法并使用微处理器的虚拟地址。1 行即为 1 块，因此，整行同时从存储器调入。用存储器管理表将存储器个别区域标记为可 Cache 或不可 Cache。可以在软件控制下禁止和(全部)刷新 Cache。

数据 Cache

数据 Cache 的组织与指令 Cache 相同：16 KB 分 512 行，每行 32 字节；32 路相联；虚拟寻址；采用循环替换算法，但增加了数据存储功能(指令 Cache 为只读)。块大小也为 32 字节。Cache 可以由软件禁止。存储器的个别区域可以设置为不可 Cache。(I/O 区域最好设置为不可 Cache。)

数据 Cache 采用写回策略。每行设有两个脏位，这样当收回该行 Cache 时，写回存储器的数据可以是整行、半行，或不需写回。由于半脏的情况经常发生，使用 2 个而不是 1 个脏位可以减小存储器流量。Cache 保存了所用每一行的物理地址，当收回的 Cache 行被写回存储器时，可能被放到写缓冲器中。

由于采用写回策略，有时必须将所有“脏”行写回存储器。在 StrongARM 中，可以通过软件向所有行调入新数据来实现。

同义项

使用虚拟寻址 Cache 时，必须保证所有可 Cache 的物理存储器位置与虚拟地址一一对应。当两个虚拟地址映射到相同的物理位置时，这两个虚拟地址为**同义项**(synonyms)。当同义项存在时，两个虚拟地址都不应该是可 Cache 的。

Cache 的一致性

对于指令和数据 Cache 分开的情况，同一个存储器位置可能在两种 Cache 中有不同的拷贝。当一个存储器区域在一个时间作为数据(可写)，而在另一个时间作为可执行的指令时，就必须加倍小心以避免不一致的情况发生。

常遇到的一种情形是一个程序加载(或从一个存储器位置复制到另一个位置)然后执行。在加载过程中，程序作为数据并通过数据 Cache 传递。在执行时程序装入指令 Cache(指令 Cache 中可能有相同地址的以前指令的拷贝)。要保证操作正确，必须：

1) 加载过程必须完成。

2) 整个数据 Cache 必须是“干净”的(同前面描述的那样，向每一行加载新的数据)；或者，如果受影响的 Cache 行地址已知，则可以显式地清除并刷新这些行。

3) 指令 Cache 应当被刷新(清除旧指令)。

另一个解决办法是在加载过程中将特定存储器区域置为不可 Cache。

对于 ARM 代码中常用到的常数(literal)(指令流中的数据项)则不存在这个问题。若一个存储器块混装指令和常数，则这个存储器块可能在指令和数据 Cache 都加载。尽管如此，只要将某个字(或字节)始终如一地作为指令或数据对待就不会产生问题。甚至程序会修改(尽管这很少使用，这也是一个很不好的习惯)常数的值，但只要不影响可能在指令 Cache 中的指令数据也是可以接受的。尽管如此，应避免使用常数并隔离数据区和代码区。

编译器问题

对于指令和数据 Cache 分开的情况，编译器应将编译过程中遇到的所有常数汇集在一起，而不是放到各个子程序的末尾。这样减少了在数据 Cache 中加载指令和在指令 Cache 中加载数据的情况。

写缓冲器

写缓冲器平缓了写数据带宽上小的峰值，减少了因存储器总线饱合而产生的微处理器停顿。大的峰值会造成缓冲器填满而导致微处理器停顿。对同一个 16 字节区域互不相关的写操作会在写缓冲器中合并，虽然仅合并到最后写入缓冲器的地址。缓冲器最多保存 8 个地址(每个地址与 16 字节边界对准)，复制虚拟地址以供写合并时使用，复制物理地址用于对外部存储器寻址。每个地址最多可以有 16 字节的数据(因此，每个地址可处理半个“脏行”，或一条多寄存器 Store 指令最多可有 4 个寄存器)。

写缓冲可以由软件禁止。使用 MMU 页表可以将个别存储器区域标记为可缓冲或不可缓冲的。所有可 Cache 的区域都是可缓冲的(清除的 Cache 行通过写缓冲器写

回),但不可 Cache 的区域可以是可缓冲的或不可缓冲的。通常,I/O 区域是不可缓冲的。一个不可缓冲的写要等到写缓冲器空后才能写入存储器。

当读数据且数据 Cache 未命中时,要检查写缓冲器以确保一致性。但读指令时不检查写缓冲器。当一个曾作为数据使用的存储器位置作为指令使用时,必须使用一条专用指令来确保写缓冲器已排干。

MMU 组织

StrongARM 实现了标准的 ARM 存储器管理结构,对指令和数据使用了分开的转换后备缓冲器(TLB)。每个 TLB 有 32 个转换项,这些转换项以全相联 Cache 方式组织,并使用循环替换算法。当 TLB 未命中时启动 table-walking 硬件从主存储器中读取转换及访问权限信息。

StrongARM 芯片

StrongARM 管芯的照片如图 12.10 所示。图中文字标注了主要功能块。指令 Cache(ICACHE)和数据 Cache(DCACHE)占用大部分管芯面积。每个 Cache 有自己的 MMU(DMMU 和 IMMU)。微处理器核有指令发射部件(IBOX)和带有高速硬件乘法器(MUL)的执行部件(EBOX)。芯片还有写缓冲器和外部总线控制器。

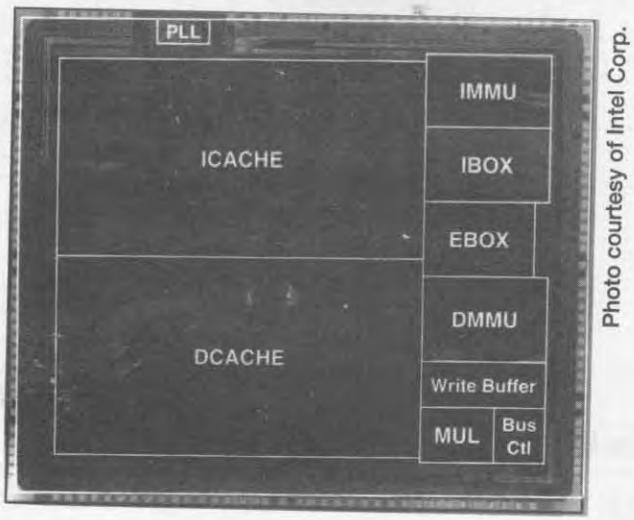


图 12.10　StrongARM 管芯照片

高速片上时钟由锁相环(PLL)产生。锁相环的外部时钟输入频率为 3.68 MHz。StrongARM 的特点总结如表 12.4 所列。

表 12.4　StrongARM 的特点

工　艺	金属层	Vdd	晶体管	管芯面积	时　钟	MIPS	功　耗	MIPS/W
0.35 μm	3	1.65/2 V	2 500 000	50 mm^2	100/233 MHz	115/268	300/1 000 mW	380/268

12.4 ARM920T 和 ARM940T

ARM920T 和 ARM940T 在 ARM9TDMI(参见 9.3 节)的基础上增加了指令和数据 Cache。指令和数据端口通过 AMBA 总线主控单元合并在一起。片上还集成了写缓冲器和存储器管理单元(ARM920T)或存储器保护单元(ARM940T)。

ARM920T

整个 ARM920T 的组织如图 12.11 所示。

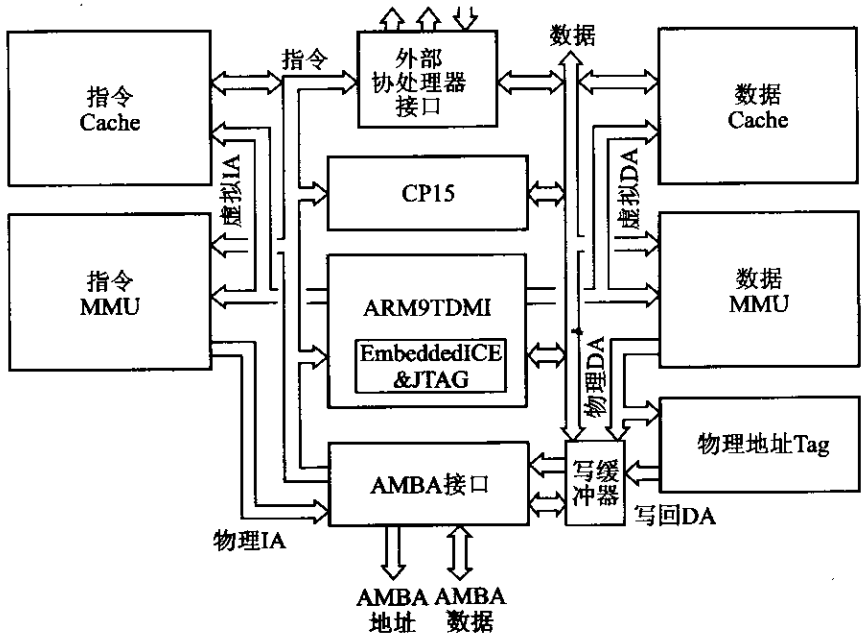


图 12.11 ARM920T 组织

ARM920T 的 Cache

指令和数据 Cache 的大小都是 16 KB,采用 64 路相联的分段式 CAM-RAM 组织。每个 Cache 分为 8 段,每段 64 行。段由 A[7:5]寻址。每行 8 个字(32 字节),支持以 256 字节为单位的锁定(对应每段一行)。替换策略为伪随机或循环,由 CP15 寄存器 1 的 RR 位(位 14)决定。在 Cache 未命中时一次将整行 8 个字全部重新读入。

指令 Cache 为只读。数据 Cache 采用写回策略,并且每行有 1 个有效位、2 个脏位和 1 个写回位。写回位复制了通常在转换系统中才有的信息,使 Cache 在写直达或写回时不必通过 MMU 而直接进行写操作。在 Cache 行被替换时,送到写缓冲器的脏数据的数量可以是 0、4 或 8 个字,这由两个脏位决定。数据 Cache 的空间分配只在读未命中时进行,而在写未命中不进行空间分配。

当采用虚拟地址访问数据 Cache 时，由于写操作需要物理地址，就存在一个脏数据什么时候写回到主存储器的问题。ARM810 解决这个问题的方法是将虚拟地址送回 MMU 进行转换，但肯定不能保证所需的转换项仍然在 TLB 中。因此，整个过程可能花费很多时间。为了避免这种问题，ARM920T 采用了第二个 Tag 存储器，用来保存每一个 Cache 行的物理地址。这样 Cache 行刷新不需要启动 MMU，并且处理过程没有将脏数据传送到写缓冲器的延时。

ARM920T 可以强制脏 Cache 行刷新，并使用 Cache 索引或存储器地址写回主存储器（这个过程称为清洁），因此，可以清洁与特定存储器区域相应的所有入口。

ARM920T 写缓冲器

写缓冲器可以保存 4 个地址和 16 个数据字。

ARM920T 的 MMU

MMU 采用 11.6 节中介绍的存储器管理结构，并由 11.5 节介绍的系统控制协处理器 CP15 控制。由于 ARM920T 有分开的指令和数据端口，所以它有两个 MMU，每个端口一个。

存储器管理硬件包括指令存储器端口 64 个数据项的 TLB 和数据端口 64 个数据项的 TLB。ARM920T 还有支持 Windows CE 所需要的进程 ID 逻辑。进程 ID 插入后启动 Cache 和 MMU，因此，上下文切换不会使 Cache 或 TLB 无效。除支持 64 KB 的大页和 4 KB 的小页外，ARM920T 的 MMU 还支持 1 KB 的微页转换。

ARM920T 的 MMU 中支持可选择的 TLB 项锁定，以保证关键进程（如一个实时进程）的转换项不会被清除。

ARM920T 芯片

ARM920T 的特点总结如表 12.5 所列。

表 12.5　ARM920T 的特点

工　艺	金属层	Vdd	晶体管	核面积	时　钟	MIPS	功　耗	MIPS/W
0.25 μm	4	2.5 V	2 500 000	23～25 mm^2	0～200 MHz	220	560 mW	390

ARM940T

ARM940T 是另一个基于 ARM9TDMI 整数核的 CPU 核。它比 ARM920T 简单，不支持虚拟到物理地址的转换。ARM940T 的组织如图 12.12 所示。

存储器保护单元

存储器保护单元采用 11.4 节中描述的结构。由于 ARM940T 采用分开的指令和数据存储器，因此，存储器保护单元也是分开的。

这个配置没有虚拟到物理地址的转换机构。许多嵌入式系统不需要地址转换，而且

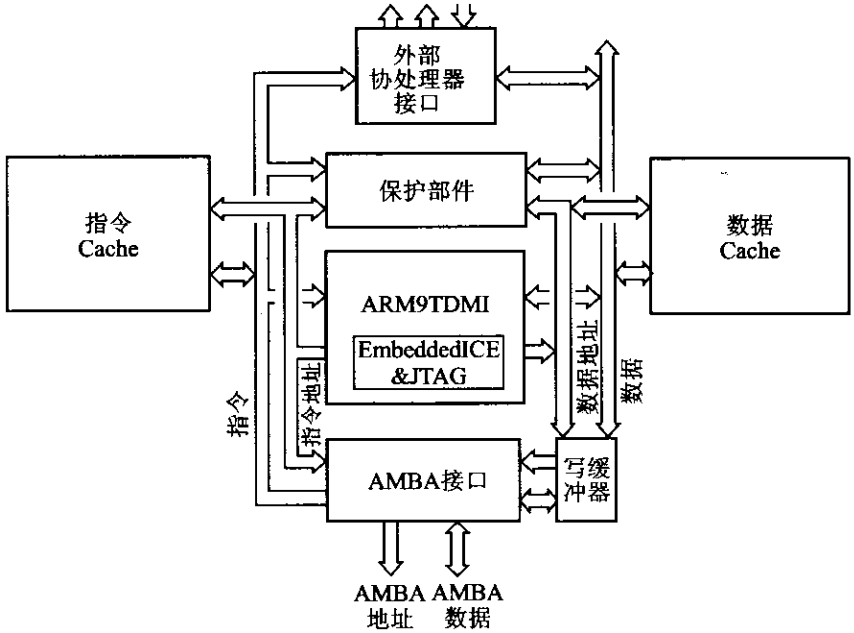


图 12.12 ARM940T 的组织

由于整个 MMU 占用很大的芯片面积，去掉 MMU 可以大大降低成本。去掉 AMBA 接口和存储器转换硬件后，嵌入式应用系统可以在同一芯片上集成其他的 AMBA 宏单元。

ARM940T 的 Cache

指令和数据 Cache 的大小均为 4 KB，由 4 个 1 KB 的段(segment)构成。两者都采用全相联的 CAM-RAM 结构(这些 Cache 术语的解释见 10.3 节)。每个 Cache 行的大小为 4 个字。如果地址可 Cache，则 Cache 未命中时总是整行调入。地址是否可 Cache 则由存储器保护单元指示。

Cache 支持锁定，也就是说部分 Cache 内容可被装入，且可以受到保护而不被冲掉。Cache 的锁定部分不能再作为通用 Cache，但保证特定的关键代码总在 Cache 中可能比提高 Cache 的命中率更为重要。Cache 锁定的单位可以是 16 字节。

ARM9TDMI 的指令端口用于加载指令，因而只进行读操作。惟一相关性问题是程序在 Cache 中保存有备份，而对应的主存储器却已修改。对代码的任何修改(例如向主存储器调入新的程序，而在此之前这部分存储器由一个不再需要的程序占用)都要小心处理。在新程序执行前指令 Cache 应有选择地或者全部刷新。

ARM9TDMI 的数据端口同时支持读和写操作，因此，数据 Cache 必须采用某种写策略(参见 10.3 节的“写策略”一段)。

由于 ARM9TDMI 的时钟工作频率很高，因此，设计的 Cache 支持写回操作，同时还支持简单的写直达。特定地址的写模式由存储器保护单元定义。只在读未命中时对数据 Cache 进行分配。

ARM940T 写缓冲器

写缓冲器最多可以保存 8 个数据字和 4 个地址。由存储器保护单元定义哪些地址是可缓冲的。数据 Cache 脏行在刷新时也被传送到写缓冲器。

ARM940T 芯片

以 0.25 μm CMOS 工艺实现的 ARM940T 的特点如表 12.6 所列，其版图如图 12.13 所示。可以看出，在一个低功耗的 SoC 设计中，整个 CPU 核所占面积比例相对较小。

表 12.6　ARM940T 的特点

工　艺	金属层	Vdd	晶体管	核面积	时　钟	MIPS	功　耗	MIPS/W
0.25 μm	3	2.5 V	802 000	8.1 mm²	0～200 MHz	220	385 mW	570

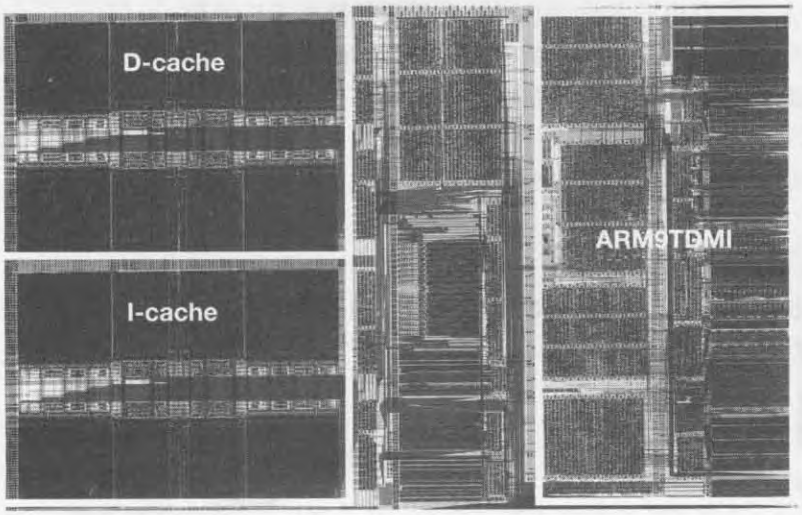


图 12.13　ARM940T CPU 版图

12.5　ARM946E-S 和 ARM966E-S

ARM946E-S 和 ARM966E-S 是基于 ARM9E-S 整数核的、可综合的 CPU 核（参见 9.3 节中“ARM9E-S”一段）。

这两种 CPU 核都有 AMBA AHB 接口，可与一个嵌入式跟踪宏单元（参见 8.8 节）一起综合。它们没有地址转换硬件，主要用于嵌入式应用。

ARM946E-S 的 Cache

ARM946E-S 使用 4 路组相联的 Cache。选择组相联而不是 ARM920T 和 ARM940T 的 CAM-RAM 组织主要是由于用标准 ASIC 库中可综合 RAM 结构构造一

个组相联 Cache 比较容易。对于大多数设计系统，综合 CAM-RAM Cache 结构仍然是比较困难。

指令和数据 Cache 的大小可以各为 4～64 KB，并且两个 Cache 的大小可以不同。两 Cache 的行都为 8 个字，支持锁定，并且替换算法由软件选择，可以是伪随机或循环算法。写策略也由软件选择，可以是写直达或写回。

ARM946E-S 集成了存储器保护单元，其整个组织与 ARM940T 的相似（如图 12.12所示）。

ARM966E-S 的存储器

ARM966E-S 中没有 Cache。这个可综合的宏单元集成了一个紧密耦合的 SRAM。这个 SRAM 映射到固定的存储器地址。存储器的大小可以变化。第一个存储器只连接到数据端口，而第二个存储器连接到指令和数据端口。通常第二个存储器由指令端口使用。但能够由数据端口访问也是非常重要的，原因如下：

- 嵌入在代码中的常数（如地址）必须通过数据端口读取。
- 必须有一种手段能够在使用前对指令存储器进行初始化。指令端口是只读的，因此不能用于指令存储器初始化。

ARM966E-S 中还包含写缓冲器，以便提高对 AMBA AHB 总线带宽的利用。同时，ARM966E-S 支持片上协处理器。

软 IP 核

这些可综合的 CPU 核针对的是对随时可由新工艺技术实现的高性能微处理器的强烈市场需求，是对 ARM 技术以硬宏单元发送方式的补充。

12.6 ARM1020E

ARM1020E 是基于 9.4 节介绍的 ARM10TDMI 核而设计的 CPU。它采用 v5TE 版本的 ARM 体系结构，包括 Thumb 指令集和信号处理指令集扩展（参见 8.9 节）。

ARM1020E 的组织与 ARM920T（如图 12.11 所示）非常相似，不同之处是 Cache 的大小和总线的宽度。（在图中所示的详细程度上）惟一的实质区别是 ARM1020E 在指令和数据上使用了两个 AMBA AHB 总线主模块请求-应答握手信号，而 ARM920T 在内部对它们进行判决以产生单个外部请求-应答接口。

ARM1020E 的 Cache

ARM1020E 集成了 32 KB 的指令 Cache 和 32 KB 的数据 Cache。两个 Cache 都是 64 路相联，并使用了段式 CAM-RAM 结构。行的大小都是 32 字节，都使用伪随机或循环替换算法并支持锁定。

为了满足 ARM10TDMI 的带宽需求，两个 Cache 的数据总线带宽都是 64 位。指令 Cache 每周期能够提供两条指令；数据 Cache 在执行多寄存器 Load/Store 指令时，

每周期能够提供两个字。

指令 Cache 是只读的。数据 Cache 采用写回策略，每个 Cache 行有 1 个有效位、1 个脏位和 1 个写回位。当一个 Cache 行被替换时，可能需要将 8 个字写回主存储器，这取决于脏位的状态。由于存储器数据带宽为 64 位，将 8 个字全部写入存储器需要 4 个存储器周期。

写回位是通常在转换系统中写回信息的复制，这使得 Cache 以写直达或写回方式进行写操作时不用启动 MMU。

hit-under-miss 缓冲器

ARM1020E 也支持 hit-under-miss 操作，也就是说如果一个数据引用引起 Cache 未命中，那么在读取包含未命中数据的行的同时，Cache 可以继续支持后续的数据引用（假定后续的数据引用没有再次引起未命中）。

ARM1020E 写缓冲器

写缓冲器有 8 片，每片可以保存 1 个地址和 64 位数据（两个字）。还有一个 4 片的写缓冲器，用来处理 Cache 更新，以保证在主写缓冲器中 Cache 更新与 hit-under-miss 写操作之间没有冲突。

ARM1020E MMU

存储器管理系统基于两个 64 表项的地址转换后备缓冲器（TLB），每个 Cache 1 个。TLB 还支持可选择的锁定，以保证关键的实时代码部分不会被冲刷。

ARM1020E 总线接口

外部存储器总线接口与 AMBA AHB 兼容（参见 8.2 节）。ARM1020E 的指令和数据存储器使用不同的 AMBA AHB 接口。尽管两个接口共享 32 位地址和 64 位单向读和写数据总线接口，但每个接口向总线判决器发出它自己的请求。

ARM1020E 芯片

以 0.18 μm CMOS 工艺实现、工作电压为 1.5 V 的 ARM1020E 的目标特性总结如表 12.7 所列。CPU 的工作电压可以降为 1.2 V 以提高功耗效率。

表 12.7　ARM1020E 的目标特性

工　艺	金属层	Vdd	晶体管	核面积	时　钟	MIPS	功　耗	MIPS/W
0.18 μm	5	1.5 V	7 000 000	12 mm^2	0～400 MHz	500	400 mW	1 250

ARM10200

ARM10200 是基于 ARM1020 CPU 核的参考芯片，它增加了下列部件：

- VFP10 向量浮点单元；
- 高性能同步 DRAM 接口；

· 锁相环电路，用于产生高速 CPU 时钟。

设计 ARM10200 主要用于评估 ARM1020E CPU 核，同时支持基准测试和系统原型设计。

VFP10

ARM10TDMI 通过浮点协处理器 VFP10 提供高性能的向量浮点运算。VFP 采用 5 级 Load/Store 流水线和 7 级执行流水线，同时支持单精度和双精度 IEEE754 浮点运算(参见第 6.3 节)。VFP10 能够在每个时钟周期发射一条浮点乘加操作，它使用 ARM10TDMI 的 64 位数据存储器接口，每个时钟周期可以 Load/Store 1 个双精度数据或 2 个单精度数据。算术和 Load/Store 指令中都有向量变量，能够对一组寄存器进行相同操作。由于向量运算和向量 Load/Store 能够并行执行，因此，吞吐量峰值可以达到 800 MFLOP(400 MHz 时)。

ARM10200 芯片

0.25 μm 的 ARM10200 芯片版图如图 12.14 所示。芯片第一个版本的 VFP10 核全部由综合实现，对 Cache 采用普通设计规则设计。在芯片的 0.18 μm 版本中，VFP10 核的数据通路版图由手工完成，而控制由综合实现。对 Cache 采用专门工艺的设计规则设计，使其相对面积大大减小。

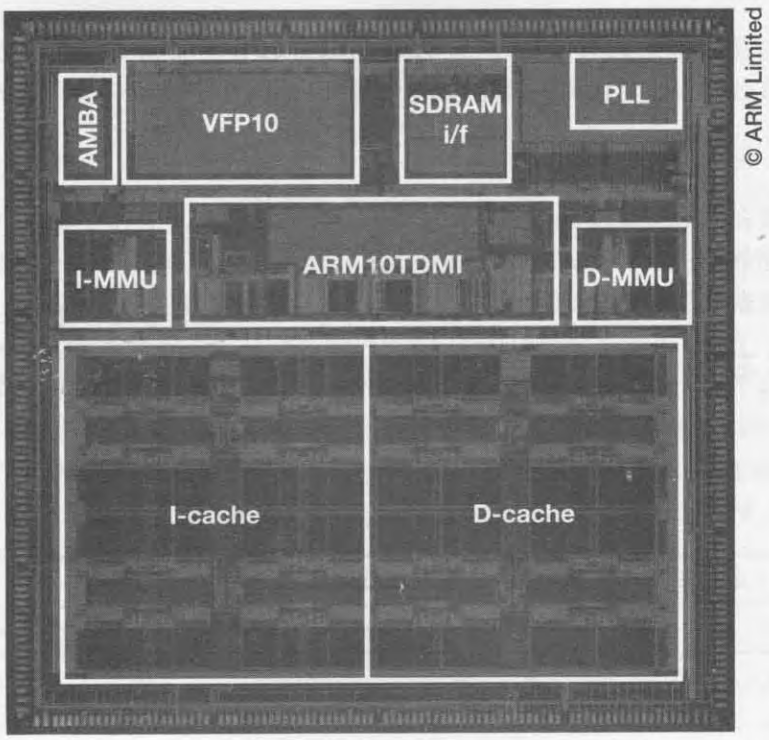


图 12.14 ARM10200 芯片版图

12.7 讨 论

本章介绍的 ARM CPU 核强调了开发高性能、低功耗微处理器子系统的一些重要方面。

与微处理器核设计相关的问题已在第 9 章讨论，并在 9.5 节进行了总结。这里关注的是其他一些部件，它们与微处理器核密切相关，并对实现其固有的潜在性能来说是非常关键的。

存储器带宽

相关存储器系统的带宽基本上限制了处理器的性能。ARM CPU 核系列显示出如何优化 Cache 存储器系统以满足不同的性能需求。

- ARM7TDMI 充分利用了每一个存储器周期。它要求存储器在每个时钟周期提供一个数据字。由于采用了 Von Neumann 结构(也就是用单个存储器端口传送指令和数据)，统一 Cache 可以支持其存储器带宽需求。ARM710T、720T 和 740T 都采用这样的 Cache。
- ARM9TDMI 要求每个时钟周期传送的数据字多于 1 个。为达到这个目的，采用了 Harvard 结构(分开的指令和数据端口)。分开的 Cache(以及存储器管理单元)可以满足这种要求。ARM920T 和 940T 都采用了分开的 32 位 Cache。
- 由于要在转移指令执行前对其进行预测，ARM10TDMI 要求每个时钟周期提供的指令多于 1 个。同时也要在执行多寄存器 Load/Store 指令时，每个周期传送两个数据字。因此，ARM10TDMI 要求 Cache 分开，并且要求每个 Cache 的带宽高于 32 位，例如 ARM1020E 的 Cache 带宽为 64 位。

ARM9 和 ARM10 系列 CPU 采用分开的 Cache 可能产生相关性问题，这留由软件解决。提供必需带宽的另一种方法是像 ARM810 那样使用双倍带宽 Cache，以及像 AMULET3H(将在 14.6 节介绍)那样使用双端口局部存储器。这两种方法使用统一的 Cache 模型，简化了软件设计，但增加了硬件复杂度。

Cache 的相联度

在本章和第 10 章已对组相联 RAM-RAM Cache 和全相联 CAM-RAM Cache 的相对优点进行了详细的讨论。为了得到较好的性能和功耗效率，早期的 ARM CPU 设计采用段式 CAM-RAM 结构。由于在理论上 RAM 比 CAM 小(每位的版图面积小大约一半)，并且在 ARM 必须定位的目标工艺的标准单元库中一般都有 RAM 单元，ARM7 系统 CPU 转而采用 4 路组相联 RAM-RAM 形式。后面的 ARM8、ARM9 和 ARM10 CPU 又回到段式 CAM-RAM 形式。这样至少可以支持 Cache 锁定(在例题 12.1 中将进一步讨论)。

Cache 写策略

写直达 Cache 最易于设计，并且当处理器时钟速率只是主存储器周期速率几倍时，处理效果很好。当处理器时钟速率是存储器周期速率 10 倍以上时，数据存储指令产生的数据写传送(write date traffic)会使外部存储器总线饱合，处理器将频繁地停顿以等待写周期完成。解决这个问题的惟一方法是使用一种 Cache 策略。它使数据不一定都写到主存储器中，如采用写回式 Cache。

当前一般的主存储器周期大约为 10 MHz，因此，工作于 60 MHz 的 ARM7 系列 CPU 采用写直达 Cache 能够工作得很好。

ARM810 首次实现的时钟速率处于边缘状态，但若没有被 ARM9 系列取代。它将会达到更高的时钟速率，并因此而采用写回式 Cache。如果不想损失其性能，那么 ARM9 和 ARM10 必须采用写回式 Cache。

Cache 行的长度

大的 Cache 行减小了 Tag 存储器的容量(对于给定大小的数据存储器)，减小了 Cache 未命中率，并增加了未命中的代价(调入一 Cache 行所花费的时间)。所有 ARM CPU 核的 Cache 行大小为 4 个字(16 字节)或 8 个字(32 字节)。在嵌入式应用并且 Cache 容量比较小时，最好使用较小的 Cache 行；而在通用系统并且 Cache 容量比较大时，使用较大 Cache 行。

在 ARM1020E 中使用的 64 位总线使得 32 字节 Cache 行调入及刷新所需的存储器周期数与 32 位总线 16 字节 Cache 行所需的相同。

存储器管理

ARM MMU 是非常复杂的单元，它占用的芯片面积与处理器核本身差不多。那些在设计时应用程序还不能确定的通用系统可能需要这个功能。运行确定应用程序的嵌入式系统可以不设计存储器管理。对于这种系统，对 MMU 成本进行评估是很困难的。

在 ARM CPU 中使用的 MMU 有两种不同的系列：一种是完整的 MMU，用于通用系统；另一种是非常简单的存储器保护单元，用于应用程序固定的嵌入式系统。ARM740T 和 ARM940T 属于后者。通用 CPU 包括支持 Window CE 的(ARM720T、ARM920T 和 ARM1020E)和不支持 Window CE 的(ARM710T 和 ARM810)。

最新开发的存储器管理结构支持在 ARM810、ARM920T 和 ARM1020E 中的 TLB 锁定。TLB 锁定与 Cache 锁定的目的相似，都是用于确保关键的实时代码能够尽可能有效地执行。

12.8　例题与练习

例题　12.1

为什么段相联 Cache 比组相联 Cache 更适于支持锁定？

早期的 CPU，如在 10.4 节研究的 ARM3，其 Cache 采用了高相联的 CAM-RAM 结构。ARM700 系列 CPU 放弃了使用 CAM-RAM 结构的 Cache，转而使用 RAM-RAM 组相联结构的 Cache。这是因为 RAM 类型的 Tag 存储器与等价的 CAM 结构的 Tag 存储器相比，占用的芯片面积小。从 ARM810 往后的 ARM CPU 又使用了 CAM-RAM Cache，至少其部分原因是需要支持部分 Cache 锁定。

为什么 CAM-RAM Cache 能对锁定提供更好的支持？其原因可能简单地归于相联度。特定存储器位置能够映射到的 Cache 位置由对地址中某些位的译码决定。在直接映射 Cache 中，只能映射到 1 个 Cache 位置；2 路组相联 Cache 可以提供对 2 个位置进行选择；4 路组相联可以对提供 4 个位置进行选择；以此类推。

在支持 Cache 锁定的 ARM CPU 中，锁定操作通过减少可选择的范围来实现。对于 4 路组相联 Cache，可以锁定 1/4 的 Cache(留下 3 路 Cache)、1/2 的 Cache(留下 2 路 Cache)、3/4 的 Cache(留下 1 路 Cache，变为直接映射 Cache)。4 路 Cache 的粒度比较粗，有效性比较差。例如，锁定的 Cache 只保存一个很小的中断处理程序。

典型的 CAM-RAM Cache 为 64 路相联，使 Cache 锁定的单位为总 Cache 的 1/64，这个粒度就更好一些。

练习　12.1.1

讨论下列 Cache 特性在性能和功耗方面的影响，即

1) 将 Tag 和数据 RAM 分成可执行并行查表的多块以增加相联度。

2) Tag 和数据 RAM 串行访问。

3) 采用顺序访问模型以越过 Tag 查找及优化数据 RAM 访问机制。

4) 指令和数据 Cache 分开(像 StrongARM 那样)。

练习　12.1.2

解释为什么片上写缓冲器不能与片外存储器管理单元一起使用。

练习　12.1.3

为什么当处理器速度相对存储器速度提高时，Cache 写策略采用写回式而不是写直达式变得更重要？

第 13 章　嵌入式 ARM 的应用

本章内容综述

嵌入式系统设计的趋势是将某些存储器件之外所有主要的系统功能集成到单一的芯片中。由此在器件成本、可靠性和功耗效率方面获得的利益是显著的。正是半导体工艺技术的进步使这些成为可能，而该技术使得人们能以低廉的价格制造出包含数百万晶体管的芯片，而且在几年之内将能使一个芯片包含数千万晶体管。

因此，在单一芯片上实现复杂系统的时代正向我们走来。ARM 公司为这个时代的到来发挥了领导作用，因为 ARM 核的面积非常小，使其他系统功能有更多可用的硅资源。在本章中，我们将介绍几个基于 ARM 的片上系统(SoC)实例，但是实际上我们只是在这一领域走马看花。在今后几年中将会看到嵌入式应用的洪流，其中许多都将基于 ARM 处理器核。

设计一个 32 位计算机系统是一项复杂的工程。把它设计到一个必须一次成功的单一芯片上依然是非常具有挑战性的。没有成功的公式，但是有许多非常强大的设计工具可以帮助设计者。正如在多数工程实践中一样，一个人通过研究问题和其他人的经验只能获得已有的知识。搞懂一个现存的设计比创造一个新的设计要简单得多。开发一个新的系统芯片是数字电子学领域今天最伟大的挑战。

13.1　VLSI Ruby II 先进通信处理器

VLSI Technology 公司是 ARM 公司的第一个半导体伙伴，它与 Acorn Computers 公司、Apple Computers 公司一起在建立独立的 ARM 公司过程中发挥过作用。它们与 ARM 公司的关系可追溯到 ARM 公司成立之前，它们在 1985 年制造了第一个 ARM 处理器，并于 1987 年从 Acorn Computers 公司获得了技术授权。

VLSI 公司曾为 Acorn Computers 公司制造过许多基于 ARM 的标准芯片，为 Apple Newtons公司生产过 ARM610 芯片。它们还生产过几个为特定用户设计的、基于 ARM 的产品，并使其中一些成为标准器件。Ruby II 就是一种用于便携式通信设备的标准器件。

Ruby II 的组织结构

Ruby II 的组织结构如图 13.1 所示。该芯片基于 ARM 核并包含 2 KB 的快速(零等待状态)片上 SRAM。关键的程序可以由应用控制加载到 RAM 中以获得最好的性能和最小的功耗。有一组外围模块共用若干引脚,包括 1 个 PCMCIA 接口、4 个单字节宽的并行接口和 2 个 UART。由一个模式选择模块来控制在什么时候可以使用哪些接口。而单字节宽 FIFO 缓冲器的作用是隔离处理器,使之不必对传输的每个字节立刻作出响应。

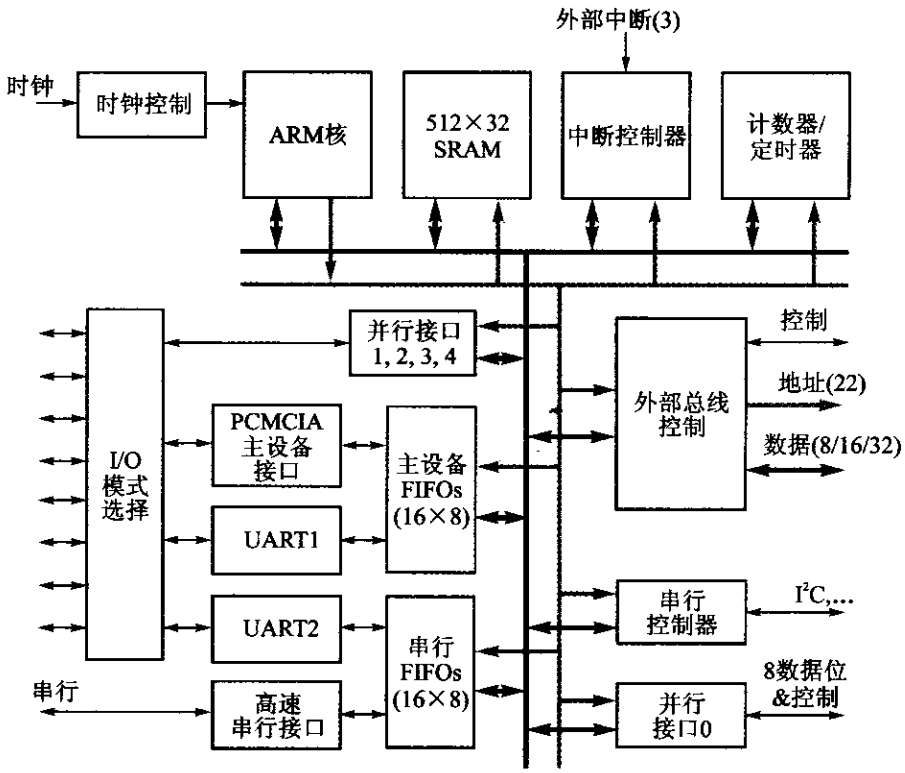


图 13.1　Ruby II 选取进通信控制器的组织结构

同步通信控制器模块支持一系列标准串行通信协议,还有串行控制器模块提供软件控制的数据端口。该端口可以用来实现不同的串行控制协议,例如由 Philips 公司定义的 I^2C 总线。该总线可以连接多种串行器件,如带电池的 RAM、实时时钟、E^2PROM 和音频编码译码器等。

外部总线接口支持具有 8 位、16 位和 32 位数据总线的器件,而且能灵活地产生等待状态。计数-定时器模块有 3 个 8 位计数器,连接成一个 24 位预定标器。另外,中断控制器提供所有片内和片外中断源的可编程控制。芯片具有 4 种功耗管理模式:

1) 在线模式:所有电路加全速时钟。

2) 命令模式:ARM 核的运行有 1～64 个等待状态,但所有其他电路以全速运行。可由中断将系统立即切换到在线模式。

3) 睡眠模式：除定时器和震荡器之外的所有电路都停止，可由特定的中断将系统转回为在线模式。

4) 停止模式：所有电路(包括振荡器)都停止，可由特定的中断将系统转回为在线模式。

封　装

Ruby II 有 144 引脚和 176 引脚两种封装，采用 TQFP 封装形式。使用 5 V 电源时可以工作到 32 MHz。如果工作于 20 MHz，则使用 32 位 1 个等待状态的存储器，芯片在在线模式下耗电为 30 mA，在命令模式下耗电为 7.9 mA，在睡眠模式下耗电为 1.5 mA，在停止模式下耗电为 150 μA。

13.2 VLSI ISDN 用户处理器

VLSI ISDN 用户(subscriber)处理器(VIP)是一个用于 **ISDN**(Integrated Services Digital Network——综合服务数字网，一种数字电话标准)用户通信的可编程引擎。它是由 Hagenuk GmbH 公司为了用于它们的 ISDN 产品而设计开发的，后来授权给了 VLSI Technology 公司作为 **ASSP**(Application Specific Standard Part——专用标准部件)销售。该芯片包含了实现全功能 ISDN 终端所需的大多数电路，支持在同一条线路上进行声音、数据和视频服务。它可以实现的应用种类包括：

- ISDN 终端设备，例如内部和数字的 PABX(专用自动交换分机)电话，H.320 视频电话及集成的 PC 通信。
- ISDN 对 **DECT**(Digital European Cordless Telepone——数字欧洲无绳电话)的控制器，允许若干无绳电话互连并连接到 ISDN，供家庭或商业应用。
- ISDN 到 PCMCIA 的通信卡。

VIP 芯片包含了连接 ISDN S0 接口所需的专用接口，支持电话接口，如数字键盘、数码显示、麦克风和耳机，提供与外部信号处理器或编码/译码器的数字连接，还包含功率管理部件，如可编程的时钟和用来监视电池状态的模数转换器。

ARM6 核执行总的控制和 ISDN 协议功能。一个 3 KB 的片上 RAM 在 36.864 MHz 的处理器全速时钟频率下，执行无等待状态的操作。关键代码子程序可以根据需要加载到 RAM 中，例如，支持免提操作所需要的信号处理子程序。

VIP 的组织结构

VIP 芯片的组织结构如图 13.2 所示，典型的系统配置如图 13.3 所示。

存储器接口

外部存储器接口支持 8 位、16 位和 32 位片外静态 RAM 和 ROM，以及 16 位和 32 位动态 RAM。可寻址的存储器被划分为 4 个区域，每个区域在操作时，等待状态的数目都是可编程的(最小为 1 个等待状态，等于 54 ns 的访问时间)。

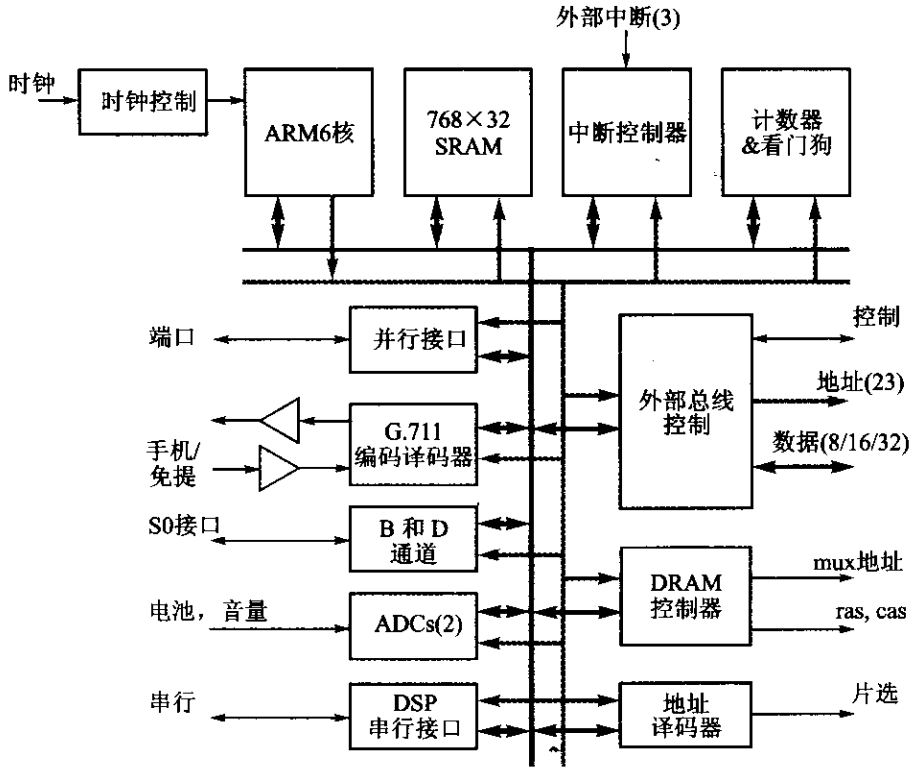


图 13.2　VIP 的组织结构

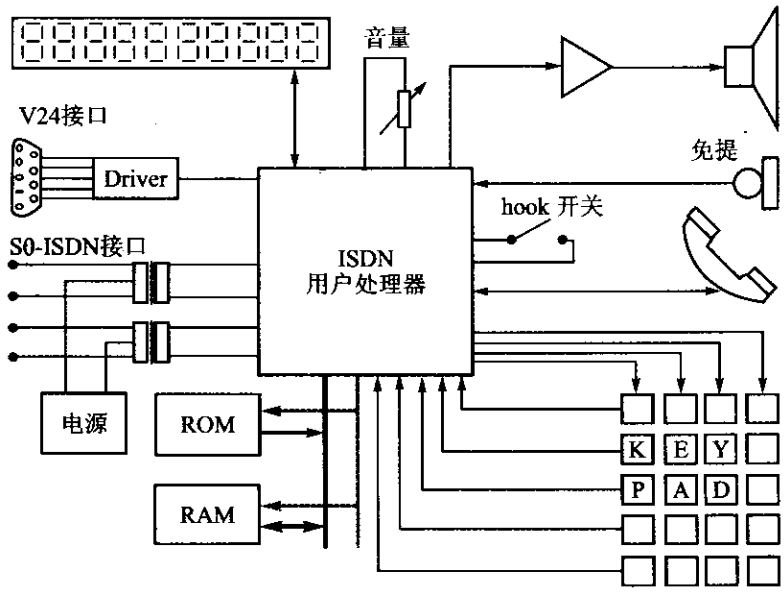


图 13.3　典型的 VIP 系统配置

S0 接口

片上 ISDN S0 接口可以通过隔离变压器连接到 S0 接口总线，并提供电涌保护。片上功能包括用于数据和时钟恢复的锁相环、帧同步以及低级别的协议。192 kbit/s 的数据速率包括两个 64 kbit/s 的 B 通道和一个 16 kbit/s 的 D 通道。当用于电话时，B 通道以 6 kHz 的采样频率传送 8 位语音采样，D 通道则用于控制。

编码译码器

G.711 编码译码器包括一个片上模拟前端，通过它编码译码器可以直接与电话听筒和免提麦克风及扬声器连接。输入和输出通道有可独立编程的增益。放大级有省电模式以便在其不活动期节省功耗。

模数转换器

片上模数转换器是根据一个电容器放电到输入电平需要多长时间来转换的。这是一种测量缓变电压的、非常简便的方法。除了一个片上比较器、一个在转换开始时给电容充电的输出，以及测量从转换开始到比较器跳变所用时间的方法之外，别的几乎不再需要。其典型应用为测量音量控制电位器的电压或检测便携式装置的电池电压。

键盘接口

键盘接口使用并行输出端口来选通键盘的列，使用并行输入端口及内部下拉电阻来检测键盘的行。在输入端口的一个“或”门可以产生一个中断。如果所有的列输出都是激活的，那么按压任何键都能产生中断，因而 ARM 可以激活单一的列并检测单一的行来确定是哪一个键被按压了。

时钟和定时器

芯片有两个时钟源。正常操作时使用 38.864 MHz，省电模式下使用 460.8 kHz。如果 1.28 s 没有动作，那么一个看门狗定时器使 CPU 复位，一个 2.5 ms 定时器可中断处理器的睡眠模式以便用于 DRAM 刷新和多任务用途。

13.3 OneC™ VWS22100 GSM 芯片

由 VLSI Technology 公司开发的 OneC VWS22100 是用于 GSM 移动电话手机的片上系统芯片(SoC)。由于增加了外部程序与数据存储器以及适当的无线模块，它可提供手机所需的全部功能。一个使用 OneC VWS22100 集成基带器件的 GSM 手机的实例是三星公司的 SGH2400，这是一款双频带(GSM900/1800)带有免提语音拨号功能的手机，如图 13.4 所示。

图 13.4　使用 OneC VWS22100 的三星公司 SGH2400 GSM 手机

VWS22100 的组织结构

OneC VWS22100 的系统结构是典型的用于当今移动电话手机的控制器。它嵌入了一个作为通用控制器的 ARM7TDMI 核和一个 DSP 核。ARM7TDMI 核用来处理用户接口及某些 GSM 协议层;DSP 核用来在一些专用信号处理硬件模块的协助下执行基带信号处理。它的结构如图 13.5 所示。

DSP 子系统

DSP 子系统(见图 13.5 中阴影部分)基于 16 位 Oak DSP 核。它执行全部实时信号处理功能,包括:

- 声音编码;
- 均衡;
- 通道编码;
- 回声消除;
- 噪声抑制;
- 声音识别;
- 数据压缩。

ARM7TDMI 子系统

ARM7TDMI 核负责系统控制功能,包括:

- 用户接口软件;

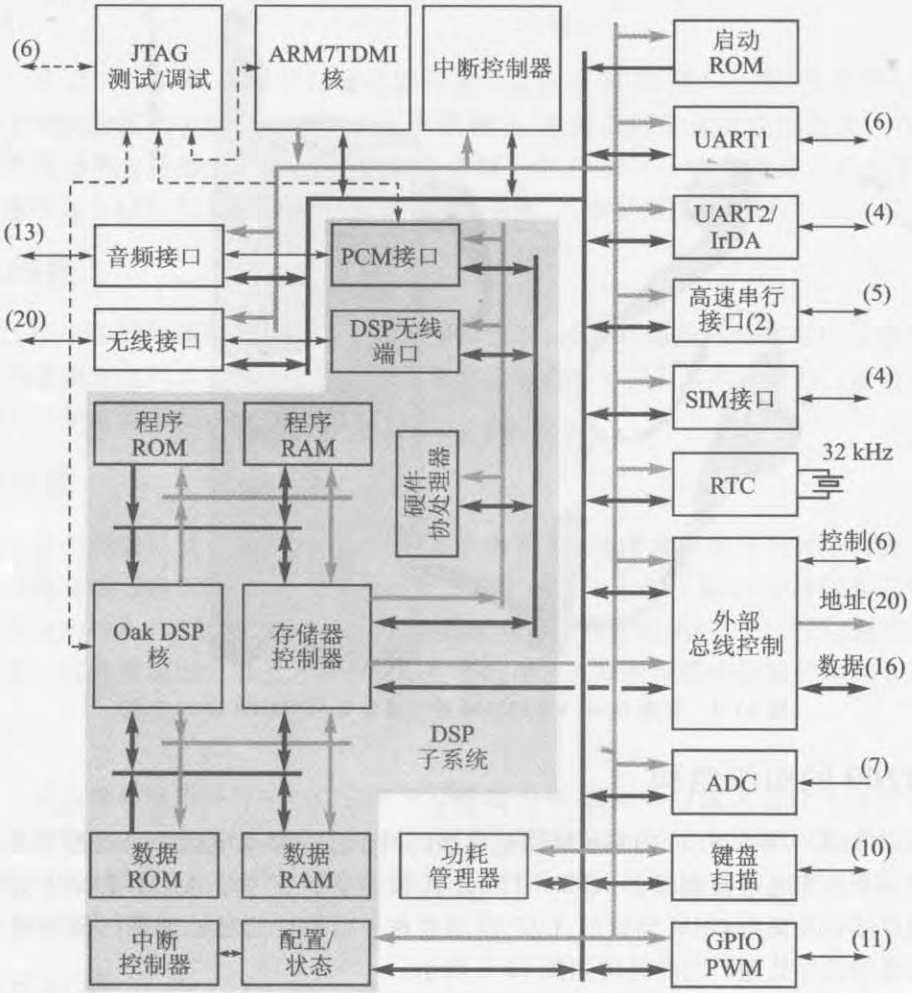


图 13.5 OneC VWS22100 GSM 芯片组织结构

- GSM 协议栈；
- 功耗管理；
- 驱动外围接口；
- 运行一些数据应用软件。

职责分配

通过分析图 13.5 的一些细节，可以说明 ARM 核与 Oak 核的任务分割。

音频和无线接口(在图的左侧)连接到两个处理系统。ARM 系统设置放大器的增益，控制无线电传送功率和频率综合器，有呼叫时以振铃通知用户，等等。

数据流，包括译码的声音数据及通过无线电链接发送和接收的码元(symbol)，直接在 Oak 系统与外围接口之间传输。

片上调试功能

芯片上有两个不同种类的处理系统。片上调试硬件使用单一 JTAG 接口访问若干调试部件，其中包括 ARM7TDMI EmbeddedICE 模块、Oak DSP 核的调试技术以及其他测试与调试装置。

功耗管理

移动电话手机的电池寿命在当前市场上是个重要问题。“通话时间”和“待机时间”在产品的广告中起着极重要作用。OneC VWS22100 内设若干功率管理部件以优化产品的性能。该产品控制：

- 全局的和选择性的节电模式；
- 在等待模式下降低系统时钟频率的能力；
- 模拟电路也能低功耗操作；
- 片上脉宽调制输出控制电池充电；
- 片上模/数转换器提供对温度和电池电压的监测以得到优化操作。

GSM 手机

基于 OneC VWS22100 的 GSM 蜂窝电话手机的典型硬件结构如图 13.6 所示。

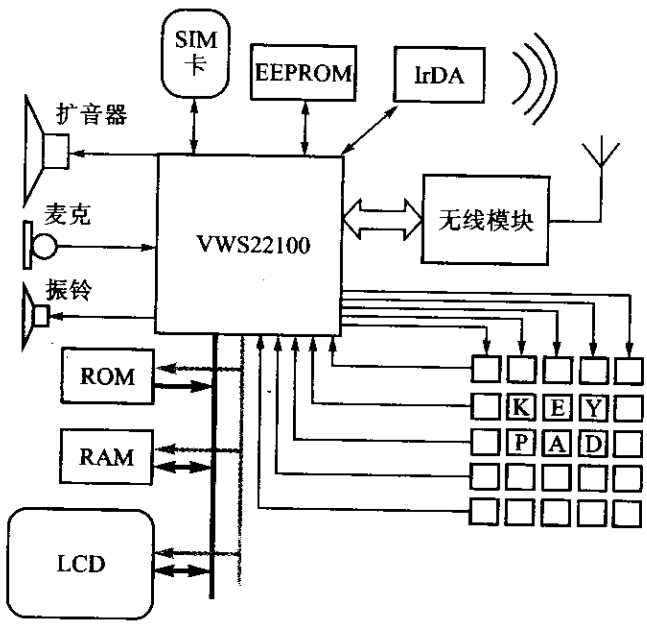


图 13.6　典型的 GSM 手机结构

13.4 爱立信-VLSI 蓝牙基带控制器

蓝牙是用于 2.4 GHz 频带无线数据通信的事实上的标准，它是由爱立信、IBM、Intel、诺吉亚和东芝公司组成的联盟开发的。该标准是为了支持短距离(10 cm～10 m)通信，就像现在采用 IrDA 标准实现的红外通信那样，但可避免 IrDA 的视线、校准和互相干涉等限制。蓝牙使用无线通信可支持膝上电脑到蜂窝电话、打印机、PDA、桌上电脑、传真机、键盘等等的连接，它还可提供一个连接现行数据网络的桥梁。这样，它可以用做个人网络的电缆替代技术。

这一标准支持 1 Mbit/s 的总数据率，使用跳频方案和向前纠错在多噪声和不协调的环境中实现稳固的通信。

爱立信- VLSI 蓝牙基带控制器芯片是一个联合开发的标准部件，用于基于蓝牙的便携式通信设备。

蓝牙“皮可网”

蓝牙设备动态地形成特别的“皮可网(piconet)”，这是由 2～8 个运行同一个跳频方案的设备组成的设备群。所有的设备都是同等的，尽管在皮可网建立时会有一个设备作为主控设备。由主控设备规定时钟和跳频次序来实现皮可网的同步。

由多个皮可网可以连接成离散网。

蓝牙控制器的组织

蓝牙基带控制器的组织结构如图 13.7 所示。芯片以同步的 ARM7TDMI 核为基础，包含 64 KB 快速(零等待状态)片上 SRAM 和 4 KB 指令 Cache。关键的子程序可以加载到 RAM 以得到最佳性能。Cache 改善驻留于片外存储器的代码的性能和功耗效率。

有一系列外围模块共用若干引脚，其中有 3 个 UART、1 个 USB 接口和 1 个 I^2C 总线接口。FIFO 缓冲器隔离处理器，使之不必对通过这些接口传输的每一字节作出响应。

外部总线接口支持带有 8 位和 16 位数据总线的设备，而且能灵活地产生等待状态。计数定时器模块有 3 个 8 位计数器，连接成一个 24 位预定标器。另外，中断控制器提供所有片内和片外中断源的控制。

爱立信的蓝牙核

蓝牙基带控制器包含一个功耗优化的硬件模块，即爱立信蓝牙核(EBC)。该模块处理蓝牙规范之内所有的连接控制器功能，并包括通向蓝牙无线电路的接口逻辑。EBC 执行点对点、多槽和点对多槽通信的所有信息包处理功能。

基带协议将电路和信息包切换结合起来使用。槽可以用做同步通道，例如支持声音传输。

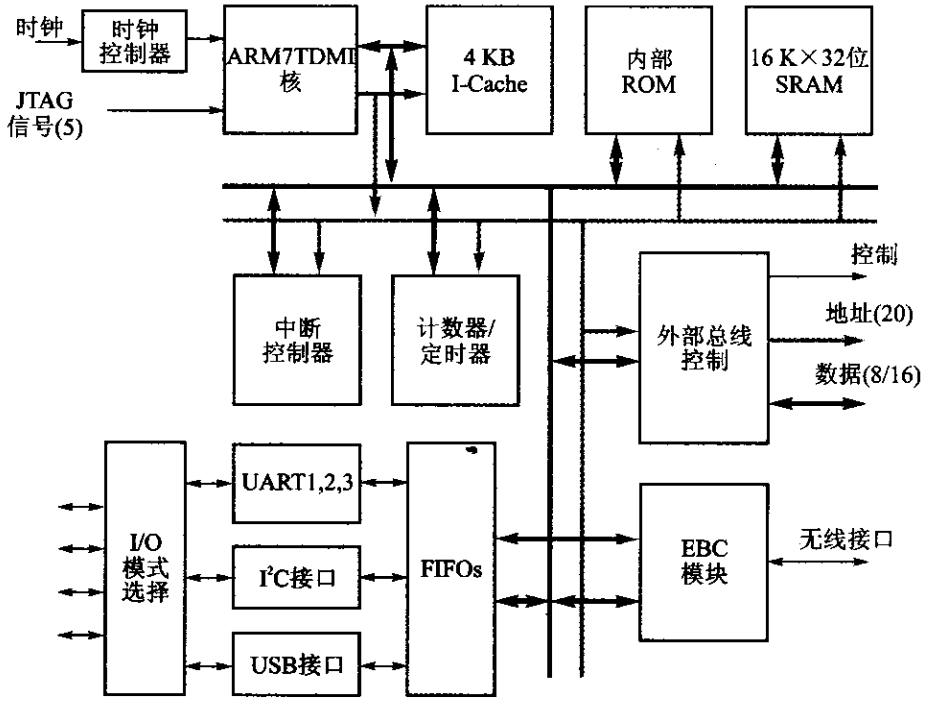


图 13.7　爱立信-VLSI 蓝牙基带控制器的组织结构

功耗管理

芯片具有 4 种功耗管理模式：

1）在线模式：所有模块的时钟采用其正常速度。ARM7TDMI 核的时钟根据应用不同位于 13～40 MHz 之间。在最大数据传输速率下，电流消耗约为 30 mA。

2）命令模式：ARM7TDMI 核通过插入等待状态而降低时钟频率。

3）睡眠模式：ARM7TDMI 核的时钟停止，正如其他模块的可编程子集的时钟一样。在这种模式下消耗的电流约为 0.3 mA。

4）停止模式：时钟振荡器关断。

蓝牙系统

典型的蓝牙系统如图 13.8 所示。基带控制器芯片需要一个外部无线模块和程序 ROM 来组成完整的系统。高度的集成使得一个复杂且功能强大的无线通信系统能够以非常紧凑和经济的形式实现。

蓝牙硅片

图 13.9 是一个蓝牙管芯的照片。管芯面积的主体是 64 KB 的 RAM，综合产生的 EBC（右下部）为第二大模块。

位于芯片右上角综合产生的 ARM7TDMI 核，其结构的清晰度远不如图 9.4 的硬宏单元。这是由于硬宏单元是手工布图的，手工设计者使用非常规则的数据通路结构，

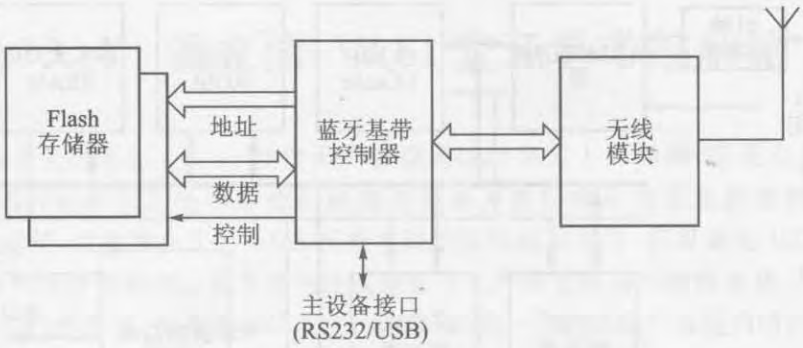


图 13.8 典型的蓝牙应用

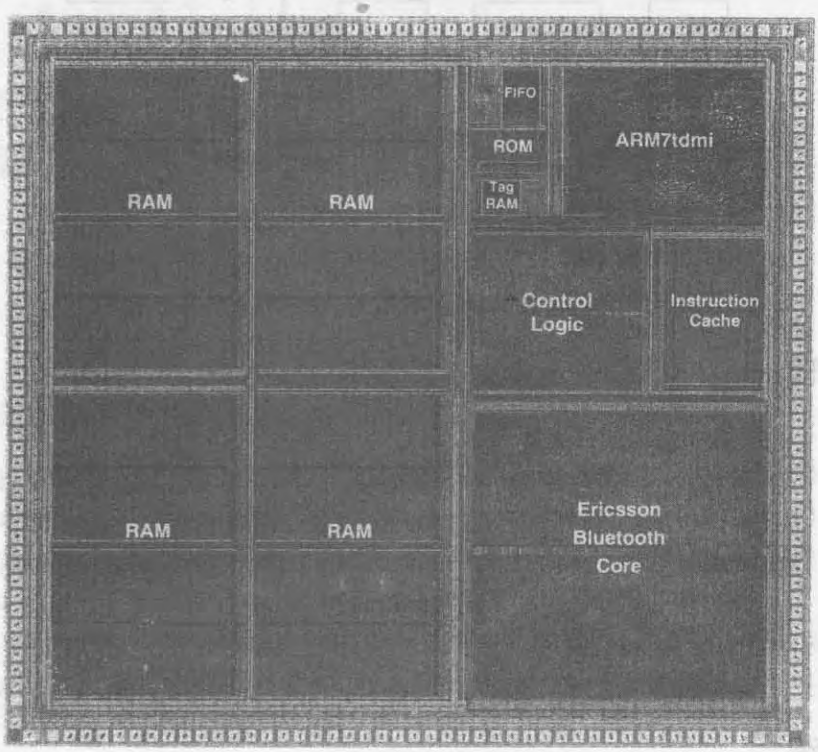


图 13.9 蓝牙基带控制器管芯照片

以便获得高密度的版图和减少必须新建的、不同单元的数量。综合的单元使用密度较低和较不规则的结构。综合方法的优点在于它能以快得多的速度转换到一种新的CMOS 工艺。

蓝牙核的特性总结如表 13.1 所列。处理器的工作频率可以高达 39 MHz,但是表中显示的数据代表着典型的 GSM 应用。芯片的 I/O 工作于 3.3 V,但是对于核的逻辑电路,其工作电压的典型值为 2.5 V。

表 13.1　蓝牙的特性

工　艺	金属层	Vdd	晶体管	管芯面积	时　钟	MIPS	功　耗	MIPS/W
0.25 μm	3	2.5 V	4 300 000	20 mm²	1～13 MHz	12	75 mW	160

数字无线通信

就像移动电话市场的快速增长所显示出来的那样，低价格、高性能数字系统的出现使无线通信能够找到新的应用领域。数字通信可以使数据在传送时由于干扰造成的错误能够被检测和纠正。数字技术还可以使复杂的无线电技术（例如跳频）受到控制，进一步降低干扰的影响。

蓝牙将这一优点扩展到了很小的通信网络，并且指出将来不仅个别的而且他们所有的个人数据设备都会无形地互连，并连接到全球网络。

蓝牙 SoC

蓝牙的应用领域已经成为行动的焦点，其目标是把数字和无线功能集成到单一 CMOS 芯片上。图 13.8 所示的典型应用就可以在单一芯片上实现。

EBC 模块也是可购买使用权的知识产权（IP），可用于基于 AMBA 的其他设计上。

13.5　ARM7500 和 ARM7500FE

ARM7500 是集成度很高的单片计算机，它将 Acorn RISC PC 的主要部件（除去存储器）组合到单一芯片上。这是第一块这样的系统芯片，它把整个 ARM CPU 用做它的处理器宏单元，包括 Cache 和存储器管理，而不是仅仅包括整数处理器核。ARM7500FE 增加了 FPA10 浮点协处理器作为片上宏单元，还进行了若干其他次要的改变。

这两种器件都很适合于消费类多媒体应用，例如机顶盒、因特网设备和游戏控制台。但是，它还有许多其他潜在的应用领域。

ARM7500 和 ARM7500FE 中主要的宏单元如下：

- ARM CPU 核；
- FPA10 浮点协处理器（只在 ARM7500FE 中）；
- 视频和声音宏单元；
- 存储器和 I/O 控制器。

ARM CPU 核

ARM CPU 核包含 ARM710 CPU 的大部分功能，惟一的让步是将 Cache 由 8 KB 减少到 4 KB，以便使其他宏单元有足够的空间。

CPU 的基础是 ARM7 的整数核（ARM7TDMI 的前身，不支持 Thumb 和嵌入式

调试),带有 4 KB 4 路组相联的指令数据混合 Cache 和一个存储器管理单元。该存储器管理单元基于 2 级页表,并带有 64 个数据项的备用缓冲器和写缓冲器。

FPA10 浮点单元

FPA10 曾在 6.4 节中比较详细地介绍过。它能有效地增强系统处理浮点数据的能力(在没有协处理器的情况下,浮点数据用 ARM 浮点库子程序来处理)。浮点性能在 40 MHz 下可达 6 MFLOPS(使用 Linpack 基准程序通过运行双精度浮点代码进行测试)。

视频和声音宏单元

视频控制器可以使用高达 120 MHz 的像素时钟(由简单的片外压控振荡器使用片上相位比较器产生)产生显示画面。它包括一个 256 色的调色板,带有用于红、绿、蓝各个输出的片上 8 位数/模转换器,以及用于外部混色和褪色的附加控制位。支持分离的硬件游标,其输出可以驱动高分辨率的彩色监视器,或者单、双画面的灰度或彩色液晶显示器。显示的时序是完全可编程的。

ARM7500 的声音系统可以产生 8 个独立声道的 8 位(对数)模拟立体声,通过片上指数的数/模转换器播放。也可以通过串行数字声道和外部 CD 品质的数/模转换器产生 16 位的声音样品。ARM7500FE 只支持后一种 16 位声音系统。

用于视频、游标和声音数据流的数据通道是由存储器中的 DMA 控制器和 I/O 控制器来产生的。

存储器和 I/O 控制器

存储器控制器支持高达 4 个 DRAM 存储体(bank)和两个 ROM 存储体的直接连接。每个存储体都可以编程为 16 位宽或 32 位宽。对 16 位宽存储体中的 32 位数据,存储器控制器将进行两次访问。

在传送多达 256 个数据的突发中,DRAM 控制器在若干个顺序周期中采用页访问模式,并支持一系列 DRAM 刷新模式。只要使用合适的 ROM 器件,ROM 控制器也支持突发模式。3 个 DMA 控制器用于处理视频、游标和声音的数据流。

I/O 控制器管理一个 16 位的片外 I/O 总线(使用扩展缓冲器可以扩展到 32 位)和若干片上接口。片外总线支持简单的外设、智能化外设模块、PC 型外设以及 PCMCIA 卡的接口。

片上接口包括 4 个可以用来支持 4 路模拟输入通道的模拟比较器、2 个键盘和/或鼠标的串行端口、计数-定时器、8 个通用开漏 I/O 线,以及 1 个可编程的中断控制器。同时还有功耗管理装置。

系统框图

有很多方法来使用这些芯片,但是一个典型的系统组织结构如图 13.10 所示。

应　用

ARM7500 已应用于 Acorn Risc PC 的低价格机型和在线媒体交互式视频机顶盒。

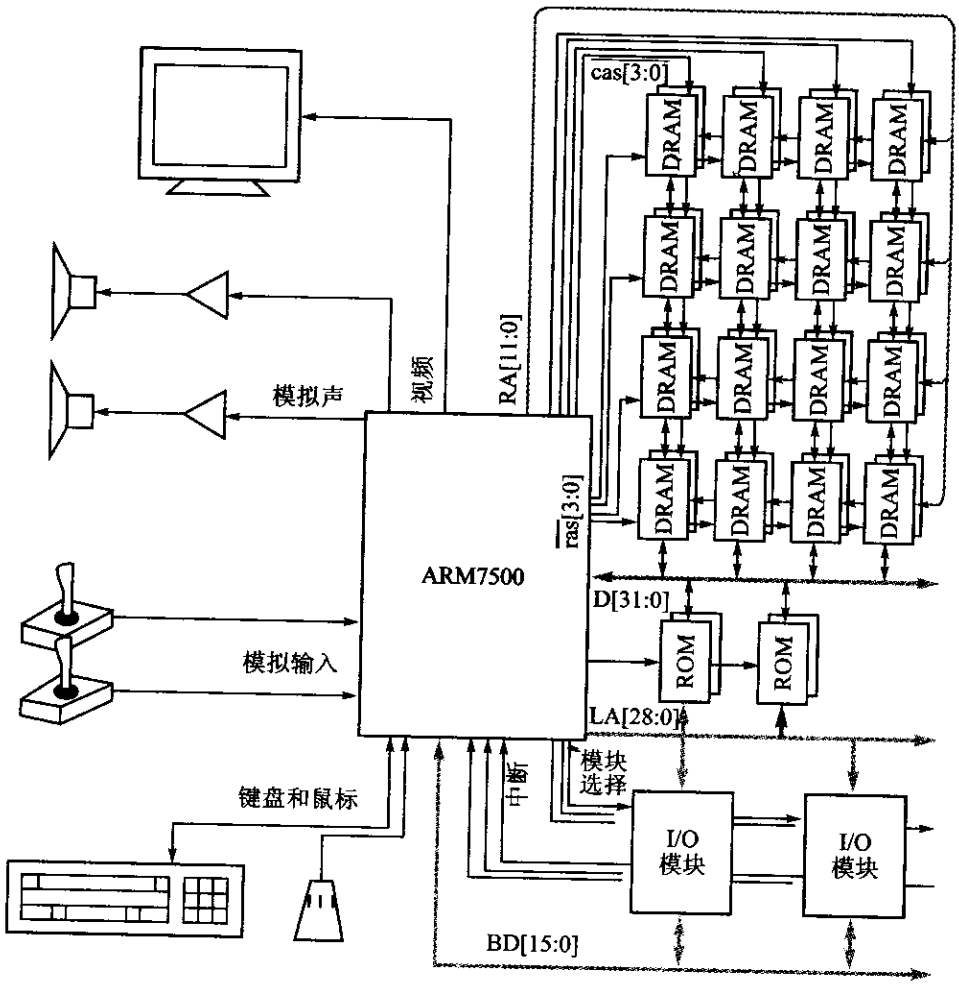


图 13.10　典型的 ARM7500 系统组织结构

与原来的 Risc PC 芯片组相比，它的主要局限在于将视频数据流限制于 DRAM，而标准 Risc PC 的高分辨率显示则使用 VRAM。这使得它不能用于在高分辨率显示器上显示很多种颜色。但是对于液晶显示器或电视监视器、VGA 或 Super-VDA 的分辨率，这种限制是不明显的。只有当分辨率达到 1 280×1 024 及以上时，颜色的数量才会受到限制，这是由于标准 DRAM 带宽的限制。

由于交互式视频和游戏机使用电视机品质的显示器，便携式计算机使用 VGA(640×480 像素)分辨率的液晶显示器，因此，ARM7500 最适用于这些应用。它的高集成度和节电特性使得它适用于手持式测试设备，它高品质的声音和图像对于多媒体应用是很好的性能。

ARM7500 芯片

ARM7500FE 的管芯照片如图 13.11 所示。在表 13.2 中总结了 ARM7500 的特性。注意，在管芯左上角的 ARM7 核只占用管芯面积的 5% 。

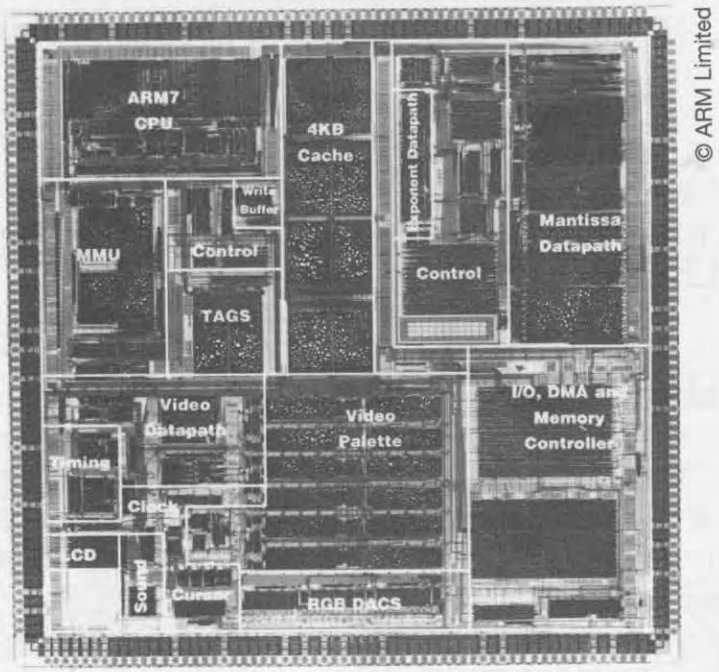


图 13.11 ARM7500FE 的管芯照片

表 13.2 ARM7500 的特性

工 艺	金属层	Vdd	晶体管	管芯面积	时 钟	MIPS	功 耗	MIPS/W
0.6 μm	2	5 V	550 000	70 mm²	0～33 MHz	30	690 mW	43

13.6 ARM7100

ARM7100 是高集成度的微控制器，它适用于某些移动应用，例如智能型移动电话和掌上电脑。它是 Psion 5 系列掌上电脑的基础。

图 13.12 是 Psion 5MX 系列掌上电脑的照片(它使用 ARM7100 较晚的版本，其体系结构经过修改，并采用更先进的工艺技术)。

ARM7100 的组织结构

ARM7100 的组织结构如图 13.13 所示。ARM710a CPU 包含一个 ARM7 处理器核(ARM7TDMI 的前身，不支持 Thumb)、ARM 存储器管理单元、8 KB 4 路相联 4 字线 Cache 和 4 地址 8 数据字写缓冲器。这类产品中 Cache 存储器的用途基本上是改善功耗效率，这是通过减少片外存储器的访问次数来实现的。加入存储器管理单元是为了支持那些为运行多种通用应用程序所需的操作系统。

片上系统的组织结构基于 AMBA 总线。外部电路包括 LCD 控制器、串行和并行的 I/O 端口、中断控制器，以及一个 32 位的外部总线接口，以便能有效地访问片外

图 13.12　Psion 5MX 系列

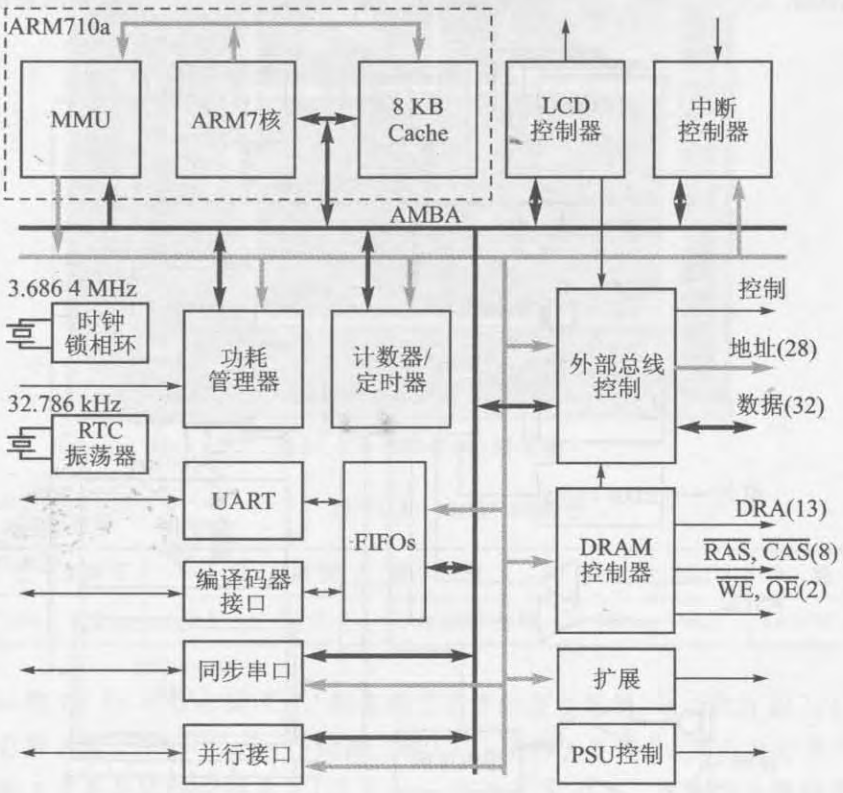


图 13.13　ARM7100 的组织结构

ROM、RAM 和 DRAM。DRAM 控制器提供这类存储器所必需的多路复用地址和控制信号。

电源管理

ARM7100 是供电池供电的便携式设备使用的。它要求在有用户输入时能提供高性能，在等待用户输入时工作在极低的功耗水平。为达到这个要求，芯片实行 3 级电源管理，即

- 在全操作模式下，ARM CPU 的性能达到大约 14 MIPS，在 3.0 V 电源下耗电为 24 mA。
- 在等待模式下，CPU 停止工作而其他系统运行，这时耗电为 33 mW。
- 在旁通模式下，只有 32 kHz 的时钟运行，耗电为 33 μW。

其他增强功耗效率的特点还包括支持自刷新 DRAM 存储器。这种存储器不需 ARM7100 干预就可以保持其内容。

Psion 5 系列

Psion 5 系列的组织结构如图 13.14 所示。这张图显示出各种用户接口如何连接到 ARM7100 上，从而在芯片级上以最小的复杂度得到一个结构完善的系统。

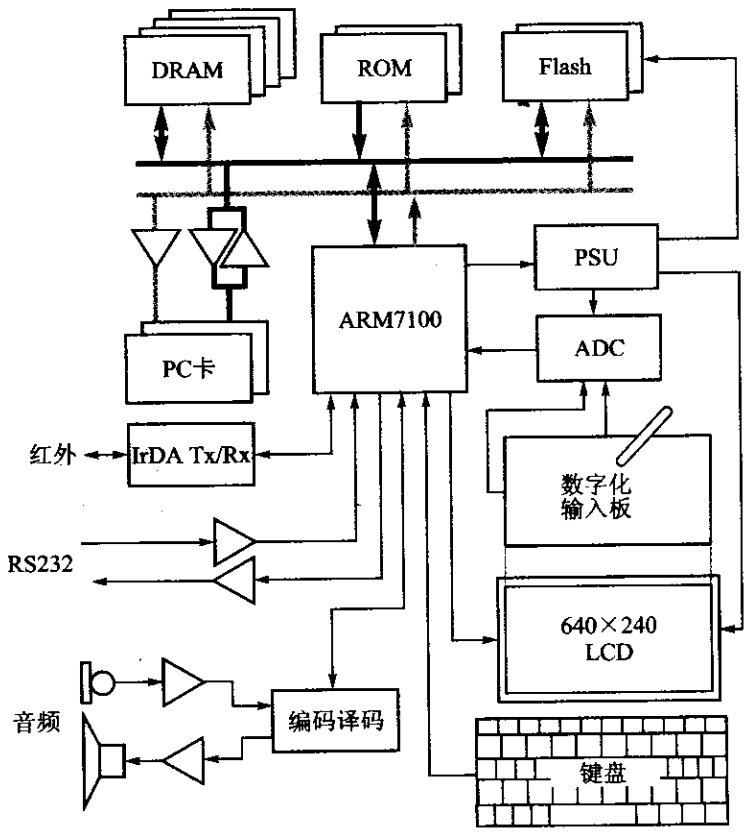


图 13.14 Psion 5 系列的硬件组织

主要的用户输入设备是键盘和输入笔。前者简单地连接到并行 I/O 引脚，后者使用覆盖在 LCD 显示器上并通过模/数转换器(ADC)连接的透明的数字化输入板。

通信设备包括用于有线连接的 RS232 串行接口和用于无线连接的 IrDA 红外接口。该红外接口连接打印机、调制解调器和主控设备 PC(用于软件加载和备份)。音频编码译码器可将来自内置麦克风的声音数字化,存入存储器,然后通过小型的内置扩音器回放。

通过 PC 卡槽可以扩展硬件。

ARM7100 芯片

图 13.15 是 ARM7100 管芯的版图。表 13.3 概述了它的主要特性。

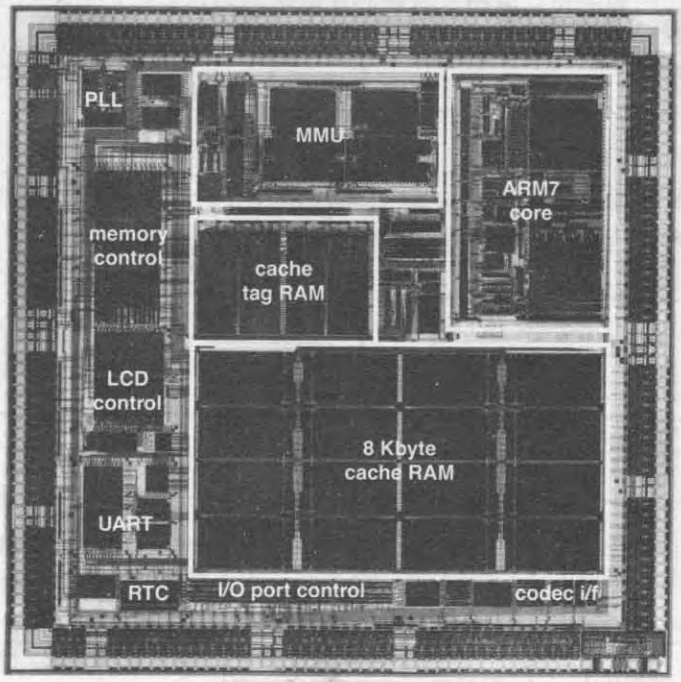


图 13.15　ARM7100 管芯图

表 13.3　ARM7100 特性

工　艺	金属层	Vdd	晶体管	管芯面积	时　钟	MIPS	功　耗	MIPS/W
0.6 μm	2	3.3 V	N/A	N/A mm^2	18.432 MHz	30	14 mW	212

从图 13.15 可以看到,CPU 核占据了芯片的主要面积,除了最左边 1/4 以外,芯片都被它所占据。8 KB Cache 存储器占据 CPU 核的下半部分,其左上方是 Cache Tag,在它的上方是存储器管理单元,而 ARM7 核位于右上方。所有的外围部件和 AMBA 总线占据左边 1/4 部分。

与 ARM7500FE 比较

在这里,平衡是与 ARM7500FE 的区别(见图 13.11)。在 ARM7500FE 中,为驱动 CRT 监视器所需的高速彩色查找表比 ARM7100 中的 LCD 控制器占据更大的面积。

ARM7500FE 中的浮点硬件也占据了相当大的面积。为了补偿，ARM7500FE 的 Cache 小了一半，但仍占据了很大面积。

13.7 SA-1100

在第 12.3 节中曾介绍过 SA-110 StrongARM CPU 核。SA-1100 是基于这种核的改良版本，是高性能、集成化系统芯片。它可用于移动电话手机、调制解调器和其他需要高性能、低功耗的手持式应用。这种芯片的组织结构如图 13.16 所示。

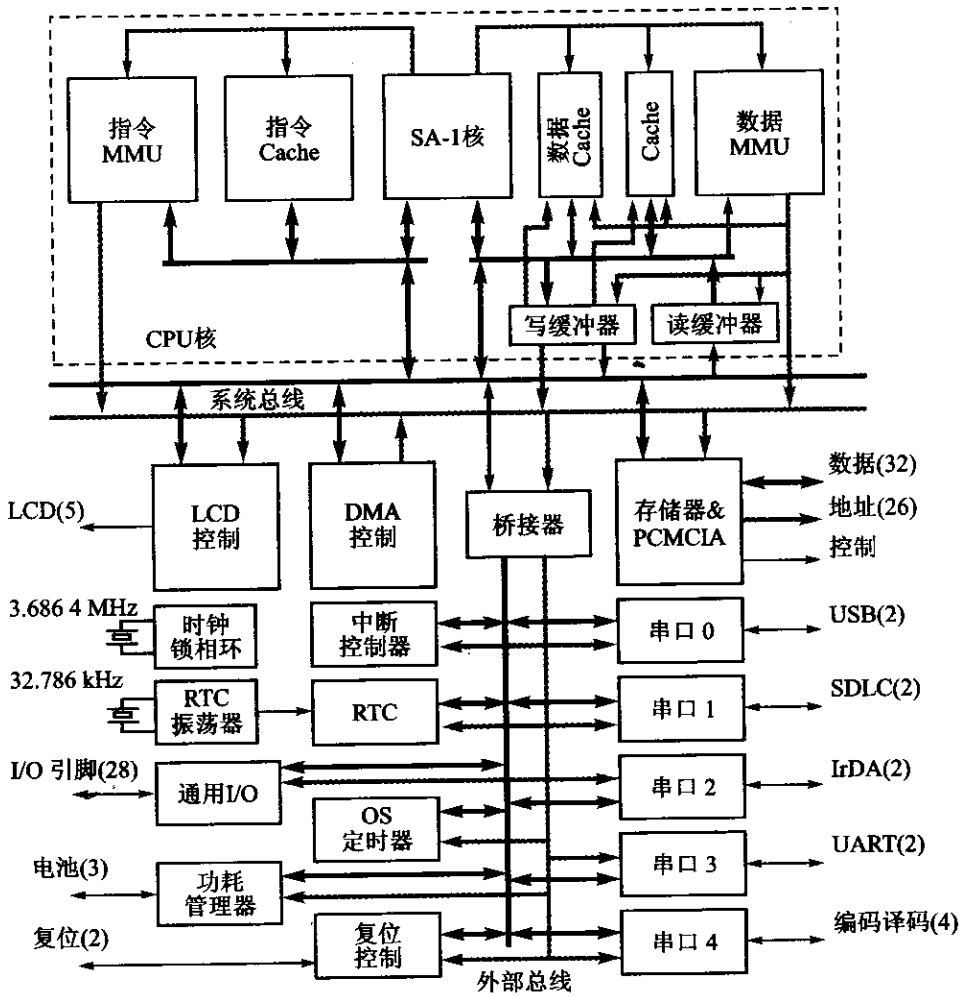


图 13.16 SA-1100 组织结构

CPU 核

SA-1100 的 CPU 核使用与 SA-110 同样的 SA-1 处理器核，但是进行了少量修改以支持 Windows CE 所需的中断向量再定位机制。指令 Cache 也是相似的，为使用 32

路相联 CAM-RAM 结构和 8 字线的 16 KB Cache。存储器管理系统未作其他改变，只是包含了 ProcessID 机制，这也是 Windows CE 所需要的。

与 SA-110 的主要区别在于数据 Cache，原来的 16 KB Cache 被换为一个 8 KB 32 路相联 Cache 和一个与之并列的 512 字节 2 路组相联 Cache。用存储器管理表来确定一个特定的存储器单元应该影射到(如果有影射)哪一个 Cache。第二个微型 Cache 的目的是使大数据结构存入 Cache 时不会造成主数据 Cache 有较大污染。

在数据 Cache 方面的其他区别是增加了读取缓冲器。它可以用来在处理器需要数据之前将它预加载。这样在需要该数据时，可以花费较短的等待时间得到。读取缓冲器时经由协处理器寄存器用软件控制。

对 CPU 核最后的扩展是增加了硬件断点和监视点寄存器。这些寄存器也是经由协处理器寄存器编程的。

存储器控制器

存储器控制器支持多达 4 个存储体的 32 位片外 DRAM。片外 DRAM 可以是传统的，也可以是 EDO(扩充的数据总线)类型的。也支持 ROM、Flash 存储器和 SRAM。

通过 PCMCIA 接口可以进一步扩展存储器(需要一些外部的胶合(glue)逻辑)。该接口支持两个卡槽。

系统控制

片上系统控制功能包括：

- 复位控制器；
- 电源管理控制器，处理电池过放电报警和系统在各种操作模式间的切换；
- 操作系统定时模块，支持一般时序和看门狗功能；
- 中断控制器；
- 实时时钟，源于 32 kHz 晶体振荡源；
- 28 个通用 I/O 引脚。

外围器件

外围子系统包括 LCD 控制器，以及用于 USB、SDLC、IrDA、编码译码器和标准 UART 功能的串行端口。一个 6 通道的 DMA 控制器使 CPU 不必直接地处理数据传输。

总线结构

从图 13.16 可以看出，SA-1100 中有两条总线，它们通过桥路相连，即

- 系统总线：连接所有总线主控器件和片外存储器。
- 外围总线：连接所有从动外围器件。

该双总线结构类似于 AMBA 总线中 ASB-APB(或 AHB-APB)的分割。它使具有高占空比的总线最小化，也减小了所有外围器件必须具备的总线接口的复杂度和成本。

应　用

SA-1100 的典型应用需要一定数量的片外存储器，其中也许包括一些 DRAM、一些 ROM 和/或 Flash 存储器。除此之外，只需要用于各种外围接口、显示器等必要的接口电路。组成的系统在 PCB 级非常简单，但在处理能力和系统结构方面功能强大而且完善。

SA-1100 芯片

表 13.4 总结了 SA-1100 芯片的特性，图 13.17 是管芯照片。该芯片可以在低达 1.5 V 的电源电压下工作以达到优化的功耗效率。它在稍高的电源电压下工作时，可以达到较高的性能，代价是功耗效率较低。

表 13.4　SA-1100 的特性

工　艺	金属层	Vdd	晶体管	管芯面积	时　钟	MIPS	功　耗	MIPS/W
0.35 μm	3	1.5/2 V	2 500 000	75 mm^2	190/220 MHz	220/250	330/550 mW	665/450

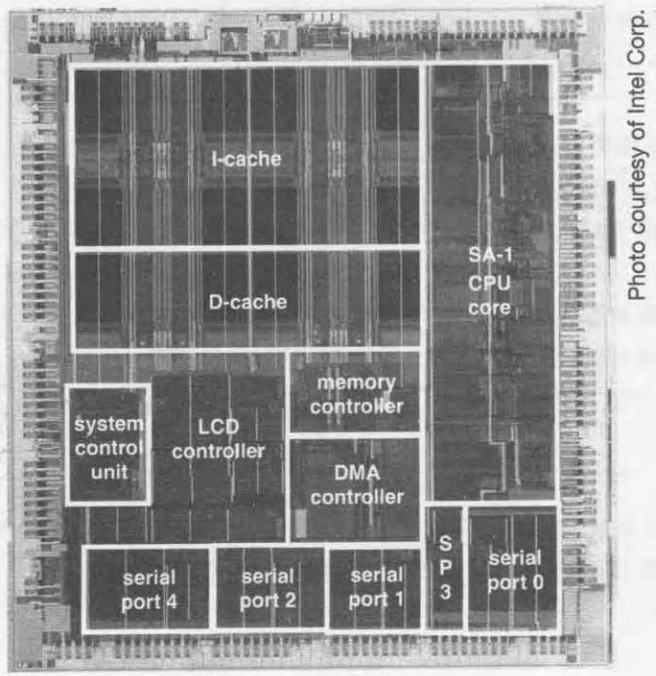


图 13.17　SA-1100 管芯图

13.8　例题与练习

例题　13.1

运行一个零等待状态片上 RAM 中的关键 DSP 程序，与 2 等待状态片外 RAM 相比，估算其性能的改进。

典型的 DSP 程序主要是进行乘加运算。其代码可能如下：

```
        ...                                 ; 初始化
LOOP    LDR     r0, [r3], #4                ; 读取下一个数据
        LDR     r1, [r4], #4                ; 读取下一个系数
        MLA     r5, r0, r1, r5              ; 累加下一项
        SUBS    r2, r2, #1                  ; 循环次数减 1
        BNE     LOOP                        ; 若没结束，则继续循环
```

这个循环需要读取 7 条指令（包括在转移盲区中的 2 条）和 2 个数据。在标准 ARM 核中乘法占用的运算周期数是与数据相关的，每个周期计算 2 位。如果我们假设系数为 16 位，而且在乘法中由系数确定周期数，则乘法需要 8 个内部周期。每次读取数据也需要 1 个内部周期。

如果数据和系数总是在片上存储器中，那么，如果由片上 RAM 读取指令执行，一次循环使用 19 个时钟周期；而如果由 2 等待状态片外 RAM 读取指令执行，则还需另外 14 个等待状态周期。

因此，使用片上 RAM 使循环加快大约 75%。

注意，若比较存储器的访问速度，可望达到 3 倍，上述循环速度的增快远小于这个数字。

练习　13.1.1

估算使用片上 RAM 带来的功耗节省（假设片上访问耗费为 2 nJ，而片外访问耗费为 10 nJ）。

练习　13.1.2

假设外部存储器宽度限制为 16 位，重新估算。

练习　13.1.3

假设处理器支持 Thumb 模式（而且程序改写为使用 Thumb 指令）和高速乘法，对外部存储器为 16 位和 32 位两种情况重新估算。

例题　13.2

研究本章中的设计，总结应用于这些系统芯片的节能技术。

首先，所有这些系统芯片都有很高的集成度，这使得在同一芯片的两个模块交换数据时节省能量。

所有芯片都基于 ARM 核，它本身就是非常省功耗的。

虽然这些芯片都没有足够的片上存储器以完全消除对片外存储器的需要，但是一些芯片具有数千字节的片上 RAM，它可以加载关键程序来节省能量（并改善性能）。

一些芯片包含时钟控制电路，当芯片不是全速工作时能降低时钟频率（或完全停止时钟），当个别模块处于非活动状态时，切断这些模块的时钟。

芯片主要基于静态或准静态 CMOS 技术，只要稍加小心，数字电路只在切换时耗费能量。一些芯片还包含模拟功能模块，它们需要连续的偏置电流。因此，当这些电路处于非活动状态时，通常可以转入省电模式。

练习 13.2.1

在嵌入式系统中通常没有复位按钮来重新启动崩溃了的系统。用什么技术来恢复软件崩溃？

练习 13.2.2

本章描述的一些系统包含计数/定时模块，它们是用来做什么的？

练习 13.2.3

ARM7500 有一个片上 Cache，AMULET2e（见 14.4 节）有一个片上存储器，它可以配置成 Cache 或直接寻址存储器。RudyII 和 VIP 只有片上直接寻址存储器。解释片上 RAM 和片上 Cache 的区别，再解释这些芯片的设计者为什么这样选择。

第 14 章　AMULET 异步 ARM 处理器

本章内容综述

AMULET 处理器核使用完全异步的方式实现了 ARM 体系结构，这就是说它是自定时的，在没有任何外部提供时钟的条件下工作。

自定时数字系统在功耗效率和电磁兼容性方面具有潜在的优势。但是，如果完全采用异步时序框架，则它需要从根本上对系统的结构进行再设计。而且，如果要它以这种方式工作，则设计者需要学习一套新的思路。提供设计工具的行业对自定时设计的支持也很缺乏。

AMULET 处理器核是在英国曼彻斯特大学开发的研究原型机。它的商业前景尚不明朗。本书没有足够的篇幅提供对它的完整描述，但是将提供对它的概述，因为它代表着实现 ARM 指令集的另一种非常不同的方法。请见本书的参考文献，可以从那里找到有关 AMULET1 和 AMULET2 进一步的细节。有关 AMULET3 更完全的细节将另行发表。

最新的自定时核 AMULET3 融合了时钟型 ARM 核中的大部分特性，包括支持 Thumb 指令集、调试硬件和支持 SoC 模块设计的片上宏单元总线等等。它将异步技术提高到准备进入商业应用的高度，在以后几年里将会看到这个技术是否会在未来的 ARM 结构中发挥作用。

14.1　自定时设计

现在以及过去的四分之一世纪中，实际上所有数字设计都是基于使用全局时钟信号，以时钟信号控制系统中所有部件的操作。复杂的系统可能使用多个时钟，但是，其中在每一个时钟控制的区域内都设计成一个同步的子系统，而且不同区域之间的接口都必须认真地设计以确保操作可靠。

时钟并非一直这样占据支配地位。很多早期计算机采用的控制电路都是使用局部信息来控制局部动作。然而，当人们开发出以使用中心时钟为基础的高效率设计方法来满足与提供了快速增长的晶体管资源的集成电路技术相关产业的需求时，这种异步设计方式就不受欢迎了。

时钟问题

但是，时钟型设计风格最近开始遇到如下实际问题：

- 不断提高的时钟频率意味着保持全局同步越来越困难了；**时钟偏斜**（芯片上不同点上时钟时序的差别）会损害电路性能，并可能最终导致电路功能错误。
- 高时钟频率还会导致过高的**功耗**。时钟分配网络可能会消耗全芯片功耗的重要部分。尽管门控时钟技术可以在一定程度上控制时钟网络的活动度，但是，这种控制通常是粗粒度的，而且对时钟偏斜有负面影响。
- 全局同步加大了电源电流的瞬变，导致**电磁干扰**。EMC（电磁兼容性）法规变得越来越严格，而加大时钟频率会使遵守这类法规变得更加富于挑战性。

自定时设计的动机

由于这些原因，最近异步设计技术重新引起了人们的兴趣。针对上述问题，异步设计：

- 因为没有时钟，所以没有**时钟偏斜**问题。
- 在异步设计中，只有当电路响应请求完成有用的工作时才会产生跳变。这就避免了由时钟信号造成的连续漏电，以及为复杂的功率管理系统而付出的开销。它可以在零功耗和最高性能之间实行即时切换。由于很多嵌入式应用的工作量都变化得很快，所以异步处理器在**节省功耗**方面具有很大潜力。
- 由于内部活动度很少相关，异步电路**电磁辐射**很少。

另外，异步电路有潜力达到典型的而不是最坏情况的性能，因为它的时序是按照实际情况调整的，而时钟型电路必须按照最坏情况留出容限。

AMULET 系列异步微处理器开发于英国曼彻斯特大学，是世界上发展中的异步设计研究的一部分。还有在其他地方开发的、支持其他指令集结构的其他异步微处理器，但是只有 AMULET 处理器实现 ARM 指令集。

自定时信号

异步设计是具有许多不同侧面和许多不同方法的复杂学科。全面介绍异步设计已超出本书的范围，但是，少量的基本概念应该能使读者掌握 AMULET 核的最重要的特性。这些概念中首要的就是异步通信的思想。在没有任何基准时钟的情况下，数据流是如何控制的呢？

AMULET 设计都使用请求-应答握手的形式来控制数据流。构成从发送端到接收端的数据通信的动作顺序如下：

1）发送端将有效数据送到总线。

2）发送端发出**请求**事件。

3）接收端在准备好后接收数据。

4）接收端向发送端发出**应答**事件。

5）发送端可以从总线上**删除**数据，在准备好后开始下一次通信。

数据通过总线时使用传统的二进制编码，但是有两种方式来发送请求和应答信号，

即跳变信号和电平信号。

跳变信号

AMULET1 使用跳变编码，即用电平的改变(从高到低或从低到高)来标志一个事件，见图 14.1。

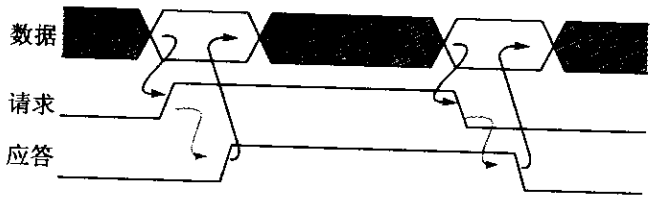


图 14.1　跳变信号通信协议

电平信号

AMULET2 和 AMULET3 使用电平编码，即以上升沿来标志一个事件，它必须返回到零才可以标志下一个事件，见图 14.2。

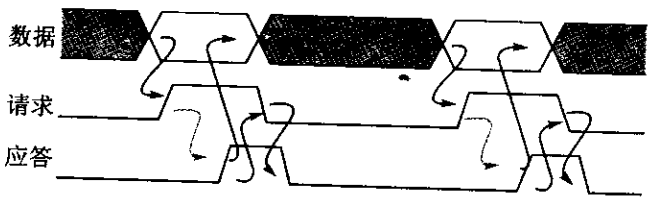


图 14.2　电平信号通信协议

AMULET1 使用跳变信号是由于它概念清晰。每次跳变都有作用，因而它的时序由电路的功能决定。它使用最少数量的跳变，因而应该是省功耗的。但是，用于实现跳变控制的 CMOS 电路相对较慢，而且效率较低。因此 AMULET 2 和 AMULET3 使用了电平信号，应用了较快和功率效率较高的电路，尽管使用了双倍的跳变次数。但是，协议中必须进行有关恢复(返回零)时序的某些判决。

自定时流水线

可以用自定时技术来构造一个异步流水线处理单元，在流水线中经过每一级的处理延迟和采用上面讲到的一种协议，将结果送到下一级。如果设计正确，则它可以适应可变的处理延迟和外部延迟。重要的仅仅是局部的事件时序(当然，长时间延迟会导致低性能)。

在时钟型流水线中，整个流水线必须永远受时钟控制，而时钟的频率是由在最坏环境(电压和温度)条件下最慢的流水线级以及假定最坏的数据情况而确定的。与此不同，异步流水线将工作于由当前条件决定的可变速率。极坏的条件有可能会使处理单元用的时间稍长一些。当出现这种情况时将会损失一些性能，但是，只要这种情况出现得足够少，它对整体性能的影响会是很小的。

14.2 AMULET1

AMULET1 处理器核具有如图 14.3 所示的高级组织结构。它的设计基于一系列相互作用的异步流水线，每一级流水线都是在自己特有的时间以自己特有的速度工作。这些流水线可能看起来会给处理器造成长得无法容忍的执行时间；但是，与同步流水线不同，异步流水线可以具有非常短的执行时间。

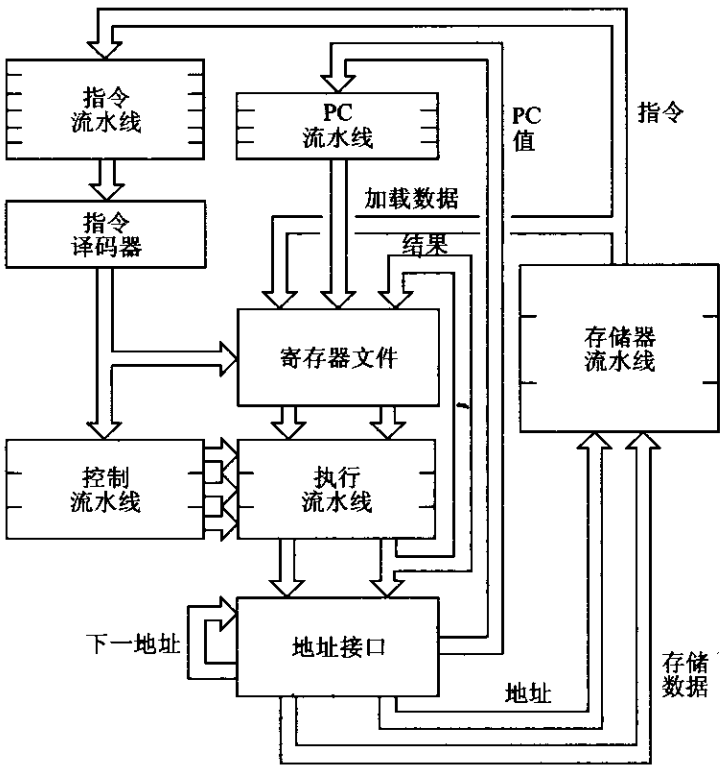


图 14.3 AMULET1 的内部组织

处理器的操作以向存储器发出取指请求的地址接口开始。地址接口有一个独立的地址递增器，它使得地址接口可以在各个流水线缓冲器所容许的范围内尽量早地进行预取指。

地址不确定论

需要产生存储器新地址的那些指令，例如数据传输指令和非预知的转移指令，在执行流水线(execution pipeline)中计算新地址，然后把新地址送到地址接口。由于新地址到达的时间相对于地址接口内部递增环路的时序是任意的，因而新地址在地址流中的插入点是不确定的。因此，处理器在转移指令之后的预取指深度从根本上是不确定的。

寄存器相关

如果要求执行流水线的工作效率高，则寄存器堆必须能够在前一条指令的结果返回之前发出指令所需的操作数。但是，在一些情况下，操作数可能依赖于前一个结果（这就是“先写后读”冒险）。在这种情况下，除非有前推的机制来提供流水线下游的正确数值，否则，寄存器堆在结果返回之前不能发出操作数。

应用于很多 RISC 处理器（包括 ARM8 和 StrongARM）的寄存器前推机制是基于同步流水线特性的，因为它涉及到把一个流水线级中的源操作数寄存器号与另一个流水线级中的目标寄存器号进行比较。在异步流水线中，流水线级都在不同的时间移动，这种比较只能在相关流水线级之间引入明确的同步后才能进行，这就损失了异步操作的大部分好处。

寄存器锁定

在 AMULET1 中，寄存器相关是通过一种基于锁定寄存器的 FIFO（先进先出队列）的新型的寄存器锁定形式实现的。目标寄存器号以译码的格式存放在 FIFO 中，直到相关的结果从执行流水线或存储器流水线返回到寄存器堆为止。

锁定 FIFO 的组织形式如图 14.4 所示。FIFO 的每一级在相当于目的寄存器的位置有一个“1”。在该图中，FIFO 控制 16 个寄存器（在 AMULET1 中，对于包括供系统使用的寄存器在内的每个物理寄存器，FIFO 都有相应的一列）。该图表示，到达的第 1 个结果写到 r0，第 2 个结果写到 r2，第 3 个结果写到 r12，FIFO 的第 4 级是空的。

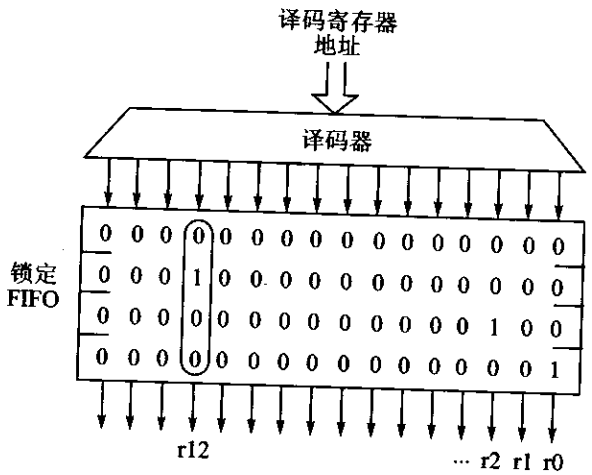


图 14.4　AMULET 中锁定寄存器的 FIFO 的组织

如果下一条指令需要 r12 作为源操作数，则检查 FIFO 中与 r12 相应的一列（图 14.4 中圈出的一列），以发现 r12 是否有效。只要在这一列中存在“1”，就说明 r12 处于等待写入的状态，因而它的当前值无效。这时需要等待，直到“1”被清除后才能执行读操作。

检查是由硬件对每个寄存器在 FIFO 中对应的列进行“或”运算来实现的。这看来有些冒险，因为在使用“或”的输出结果时，FIFO 中的数据可能会移出 FIFO。但是，在异步 FIFO 中，数据的移动是将一级的数据复制到下一级，数据只能从第一级删除，所以，传输中的“1”将交替地出现在一个或两个位置，决不会完全消失。因此，“或”的输出即使在数据移动过程中也会是稳定的。

AMULET1 完全依靠寄存器锁定机制来维持寄存器的相关，结果执行流水线在运行典型代码时会很频繁地停止。（因为标准的 ARM 处理器对连续指令之间的寄存器相关是不敏感的，所以，编译器并不尽力去避免这种情况，因而这种寄存器相关在典型代码中是常见的。）

AMULET1 的性能

开发 AMULET1 是为了显示设计全异步实现的商业微处理器结构的可行性。样品实现了功能，并且运行了用标准 ARM 开发工具产生的测试程序。样品的性能汇总于表 14.1。该表显示出分别用 European Silicon Systems(ES2)和 GEC Plessey Semiconductors(GPS)两种不同工艺技术制造的器件的特性。采用 1 μm 工艺设计的 AMULET1核的版图如图 14.5 所示。基于 Dhrydtone 基准的性能指标显示，AMULET1的性能与用相同工艺技术制造的 ARM6 的性能属于同一数量级，但确实并不优于 ARM6。然而，制造 AMULET1 的根本目的是显示自定时设计的可行性，这个目的显然达到了。

表 14.1 AMULET1 的特性

	AMULET1/ES2	AMULET1/GPS	ARM6
工艺/μm	1	0.7	1
面积/mm^2	5.5×4.1	3.9×2.9	4.1×2.7
晶体管	58 374	58 374	33 494
性能/kDhrydtones	20.5	~40[a]	31
乘法器/(ns/bit)	5.3	3	2.5
条件	5 V, 20℃	5 V, 20℃	5 V, 20 MHz
功率/mW	152	N/A[b]	148
MIPS/W	77	N/A	120

a 估算的最高性能；b GPS 不支持功率测量。

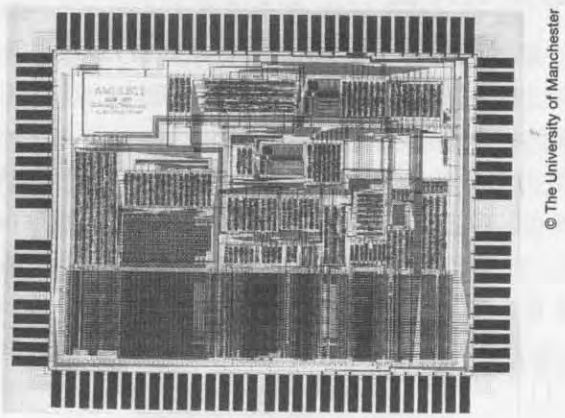


图 14.5　AMULET1 版图

14.3　AMULET2

AMULET2 是第二代异步 ARM 处理器。它采用的组织形式与图 14.3 所示的 AMULET1 的非常相似。如前所述，在 AMULET1 中使用的两相(跳变)信号被四相(电平)信号所取代。另外，增加了一些组织部件以增强性能。

AMULET2 寄存器前推

AMULET2 使用与 AMULET1 相同的寄存器锁定机制。但是，为了减少因寄存器相关停止而带来的性能损失，它还使用了前推机制来处理一般情况。在时钟型处理器流水线中使用的旁路机制不适用于异步流水线，所以，需要开发新的技术。在 AMULET2中使用的两项技术如下：

- 最新结果寄存器。指令译码器保持对执行流水线输出结果的目标寄存器进行记录。如果后面紧接的指令使用该寄存器作为源操作数，那么读寄存器的操作便被旁通，从最新结果寄存器中获取数值。
- 最新加载数据寄存器。指令译码器保持对从存储器最新加载的数据项的目标寄存器进行记录。只要该寄存器被用做源操作数，那么读寄存器操作便被旁通，其数值直接由最新加载数据寄存器取得。一个类似于锁定 FIFO 的机制作为寄存器的保护装置，确保它采集正确的数据。

这两种机制都依赖于所需的结果已经存在。当出现某些不确定性时(例如结果是由条件执行指令产生的)，指令译码器可以退回到锁定机制，发挥异步组织的能力来处理提供操作数时的可变延迟。

AMULET2 跳转跟踪缓冲器

AMULET1 顺序地从 PC 的当前值进行预取指，而每当需要偏离顺序执行时，都必须有地址作为矫正信息从执行流水线发往地址接口。每次当 PC 需要矫正时，都会因读取了将被废弃的指令而使性能受损和浪费能量。

AMULET2 力图降低这种低效率的情形。方法是记住以前在哪里发生过转移，并推测控制系统将在随后遵循相同的路径。跳转跟踪缓冲器(jump trace buffer)的组织如图 14.6 所示。它与 1969—1974 年间在曼彻斯特大学开发的 MU5 大型计算机(它的操作也采用异步控制)中使用的结构相似。

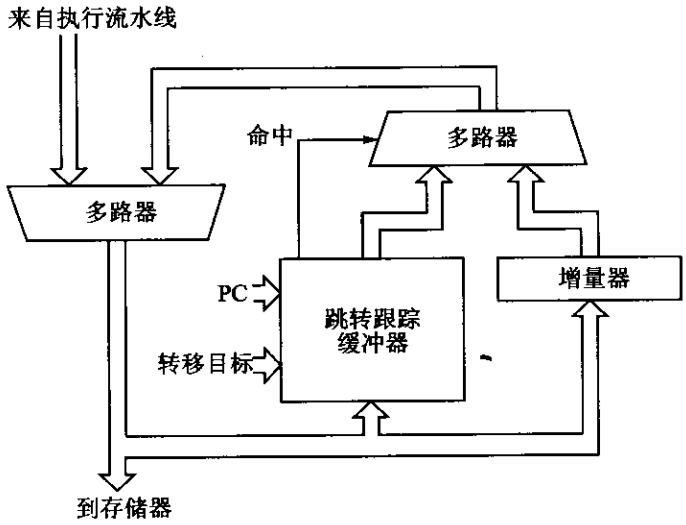


图 14.6 AMULET2 跳转跟踪缓冲器

缓冲器把最近执行转移指令时程序计数器的内容及转移目标存储起来，只要发现从它存储的地址中读取指令，就把原来预计的顺序控制流修改成以前的转移目标。如果这个预测被证实为正确的，那么就按照这个指令顺序进行取指；如果它被证明为错误的，即在此不应该执行转移，那么便将转移作为“非转移”指令来执行，返回到以前的顺序控制流。

转移统计

跳转跟踪缓冲器的效力依赖于典型的转移行为的统计。典型的数据如表 14.2 所列。

表 14.2 AMULET2 转移预测统计

预测算法	正确/%	错误/%	多余取指
顺序	33	67	每个转移 2 次(平均)
跟踪缓冲器	67	33	每个转移 1 次(平均)

在没有跳转跟踪缓冲器的情况下，缺省的顺序取指方式等效于预测所有转移均不发生。这对于所有转移中的 1/3 是正确的，对其余的 2/3 是不正确的。具有 20 个存储项的跳转跟踪缓冲器把这个比例翻转过来，预测正确的约占 2/3，预测失误的约占 1/3。

尽管转移之后的预取指深度是不确定的，但在 AMULET2 观察到的预取指深度大约为 3 条指令。当转移预测正确时，所取的指令被使用。但是，当转移预测失误或转移没有被预测时，所取的指令被丢弃。因此，跳转跟踪缓冲器将每个转移的多余取指数目由 2 减少到 1。由于在典型的代码中大约每 5 条指令发生 1 次转移，所以，可以预期，跳转跟踪缓冲器将取指带宽降低约 20%，将总存储器带宽降低 10%～15%。

在系统性能被现有存储器的带宽限制的情况下，这种节省将直接转变为性能的改善。在任何情况下，它都会由于消除冗余的活动度而节约功耗。

暂　停

与某些其他微处理器不同，ARM 没有明确的暂停指令。当程序发现不再有有用的工作要做时，它通常会进入一个空闲的循环，执行：

```
B　　.　　; 循环，直到中断
```

在此，“.”表示当前 PC，所以转移的目标就是转移指令本身。这时程序落在这个单指令的循环之中，直到有中断使它去做其他事情为止。

显然，当处理器进入空闲循环之中时，它便不再做有用的工作，所以，它用的全部功率都被浪费掉。AMULET2 检测与自身循环的转移相对应的操作码。检测到以后，在异步控制网络的某一点上将一个信号停止下来。这个停止的动作迅速传递到整个控制网络，使处理器进入不活跃的零功耗状态。由一个活跃的中断请求来释放这个停止状态，使处理器立即恢复正常的处理能力。

这样，AMULET2 以很快的速率在零功耗状态和最大处理能力之间切换，而且不需要软件开销。实际上，许多现有的 ARM 代码都能使用这个方案来实现最优化的功耗效率，即使它们编写的时候并没有想到这个方案。这使该处理器非常适用于带有突发性实时加载特性的低功耗应用。

14.4　AMULET2e

AMULET2e 是一种结合了 4 KB 存储器和灵活的存储器接口（漏斗）的 AMULET2。它的存储器可以配置成 Cache 或固定的 RAM 区。存储器接口可以直接连接 8、16 或 32 位外部器件，包括由 DRAM 构成的存储器。AMULET2e 的内部组织如图 14.7 所示。

AMULET2e 的 Cache

Cache 由 4 个 1 KB 数据块组成，每个数据块都是全相联的随机替换式存储器，带有 4 倍字线和数据块容量。在 CAM 区域和 RAM 区域之间的流水线寄存器，使得在

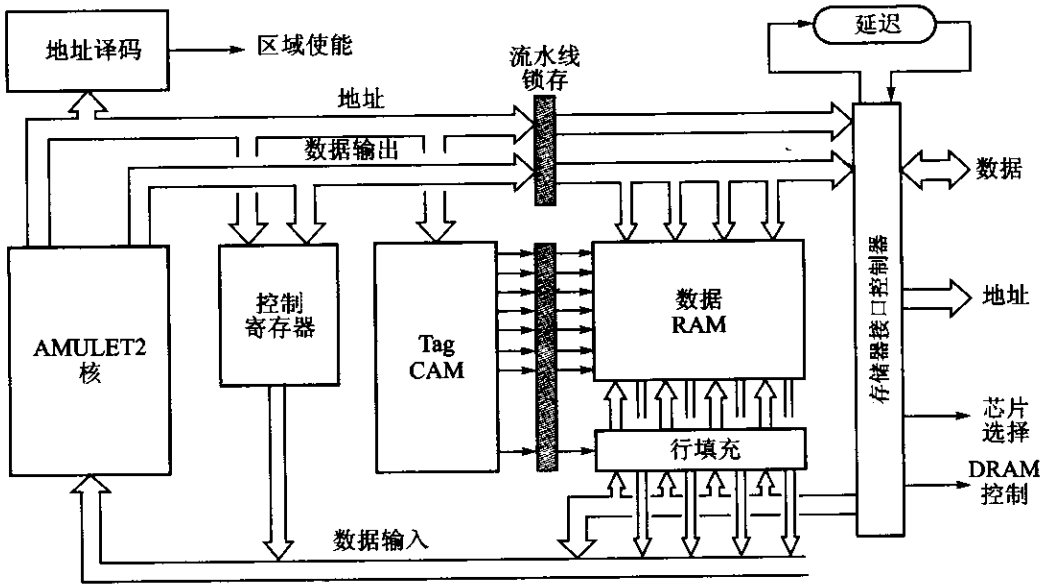


图 14.7 AMULET2e 的内部组织

RAM 完成前一次读写后，下一次读写可以开始其 CAM 查找。这利用了 AMULET2 核在第一个存储器请求的数据返回之前可以发出多个存储器请求的能力。顺序读写可以被检测出来，并使 CAM 查找被旁路，因而节约了功耗并改进了性能。

Cache 从外部存储器的行取数(line fetch)是非阻塞的，首先读取被寻址的数据，然后在取其他行数据时允许处理器继续操作。在处理器访问 Cache 的同时，行取数的自动装置继续加载行取数缓冲器。在 CAM 中还有一个附加项，它如同行取数缓冲器中所存数据的索引。实际上，这个数据保留在行取数缓冲器内，它可以同 Cache 中的数据一样被访问，直到 Cache 发生下一次未命中为止。那时，这个缓冲器被拷贝到 Cache 中，而新的数据从外部存储器加载到行取数缓冲器。

AMULET2e 的管芯

AMULET2e 管芯图如图 14.8 所示。器件特性综述如表 14.3 所列。AMULET2e核使用了 93 000 个晶体管，Cache 使用了 328 000 个晶体管，其他用于控制逻辑和压焊点。

表 14.3 AMULET2e 的特性

工 艺	金属层	Vdd	晶体管	管芯面积	时 钟	MIPS	功 率	MIPS/W
0.5 μm	3	3.3 V	454 000	41 mm^2	无	40	140 mW	285

时序基准

由于在异步系统中没有基准时钟，使按时序访问存储器成为需要认真考虑的问题。AMULET2e 的解决方案是使用一个直接连接芯片的外部基准延迟，以及启动时加载的配

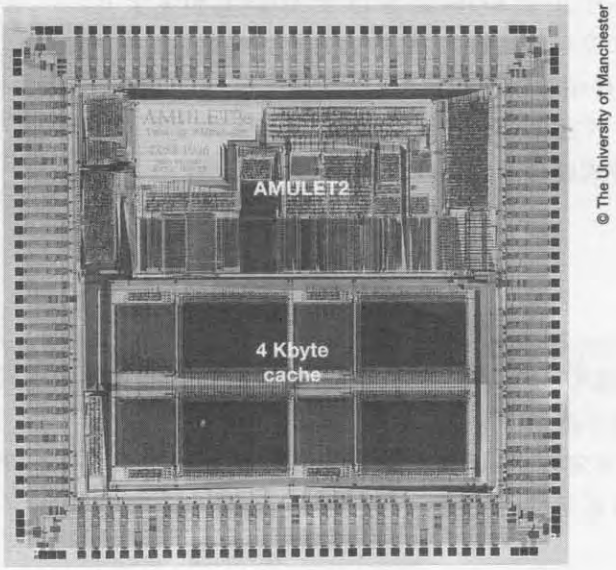


图 14.8　AMULET2e 的管芯图

置寄存器，它规定每个存储区的组织和时序特性。例如，基准延迟可以反映外部 SRAM 的读写时间，RAM 将被配置为占用 1 个基准延迟。较慢的 ROM 可能被配置为占用几个基准延迟。（注意，基准延迟只用于片外时序，所有片内延迟都是自定时的。）

AMULET2e 系统

AMULET2e 的配置使它能尽可能简单地构建小型系统。作为一个例子，图 14.9

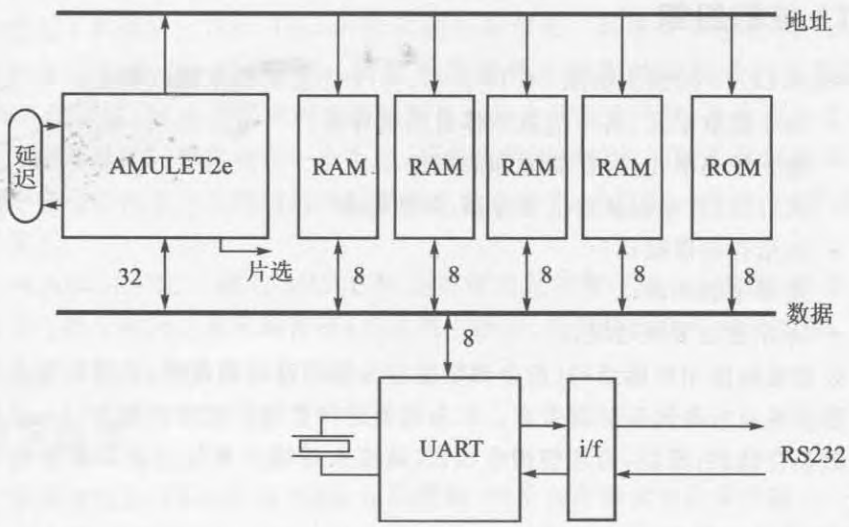


图 14.9　AMULET2e 测试卡的组织

画出了一个包含 AMULET2e 的测试卡。除了 AMULET2e 本身外，仅有的部件为 4 个SRAM 芯片、1 个 ROM 芯片、1 个 UART 和 1 个 RS232 有线接口。UART 使用晶体振荡器来控制其比特率，而系统的全部时序功能由 AMULET2e 控制。

ROM 中存储着标准的 ARM“天使”代码，而在 RS232 串行线的另一端，宿主计算机运行 ARM 开发工具。该系统显示，只要认真考虑了存储器接口，使用异步处理器并不比使用时钟式处理器更困难。

14.5 AMULET3

开发 AMULET3 是为了建立异步设计的商业生存能力。就像它的前几代设计一样，AMULET3 是支持中断和存储器故障且全功能的 ARM 兼容处理器。AMULET1 和 AMULET2 实现了 ARM 6 的体系结构(ARM 体系结构的 3G 版本)。AMULET3 支持 ARM 体系结构的 4T 版本，包括 16 位 Thumb 指令集。

性能目标

AMULET3 项目的目标是实现异步 ARM 的 v4T 体系结构。这种结构在功耗效率和性能方面与 ARM9TDMI 相匹敌。这意味着采用 0.35 μm 工艺达到 100 MIPS (用 Dhrystone2.1 测量)以上的性能指标。对比之下，AMULET2 采用 0.5 μm 工艺达到 40 MIPS 的性能。

为达到超过两倍的性能增长，需要从根本上改变核的组织结构。像时钟型处理器一样，基本的途径也是增加处理器流水线的周期频率和减少每条指令的平均周期数。但是，这里的“周期”不是由外部时钟决定，而且长度也不固定。

AMULET3 核的组织

AMULET3 的组织如图 14.10 所示。6 个主要的流水线级如下：

- 指令读取单元，其中包括转移目标缓冲器；
- 指令译码单元，寄存器读和前推级；
- 执行级，其中包括移位寄存器、乘法器和 ALU；
- 数据存储器接口；
- 重排序缓冲器；
- 寄存器结果回写级。

处理器核使用哈佛结构(指令和数据有分开的存储器接口)。这种结构可提供支持高性能所需且较高的存储器带宽。只有需要访问数据存储器的指令(Load 和 Store)才经过数据存储器，所以，对这些指令而言，其流水线深度要超过诸如简单的数据处理指令的。

下面给出每一级流水线进一步的细节。

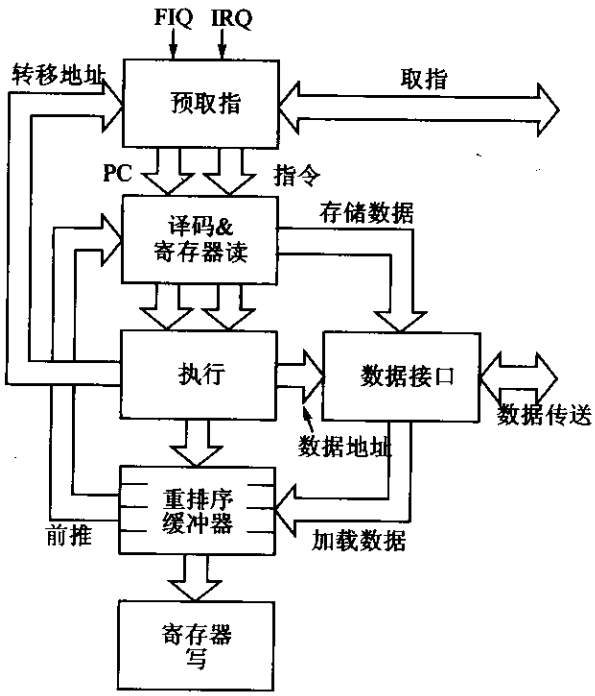


图 14.10　AMULET3 核的组织

指令预取单元

预取指单元自动地操作。只要它有空间存放，就从存储器读取 32 位(1 个 ARM 指令或 2 个 Thumb 指令)指令包。

它包含一个转移预测单元。该单元同 AMULET2 中使用的跳转跟踪缓冲器类似，但是进行了扩展以支持对 Thumb 代码的转移预测。两条指令组成的 Thumb 指令包可能包含 0、1 或 2 个转移指令，而转移预测器必须能处理所有这些情况。当执行 Thumb 代码时，16 个记录项的转移预测单元分为两半来存储，每半边 8 个记录项。将偶数半字地址的转移存储在一个半边，而将奇数半字地址的转移存储在另一半边。一个指令包在任何半字的转移都可能在两个半边命中，于是第一条指令(在偶数地址)的目标优先。

与 AMULET2 一样，AMULET3 也有零功耗的暂停指令。这里，暂停在预取指单元的作用胜于在执行单元的作用，当出现中断后，能更快地恢复到全功能。这里，中断本身也经过处理，使执行时间缩短。

译码和寄存器读

译码级包括 Thumb 和 ARM 译码逻辑，以及寄存器读与前推机制。

当 ARM7TDMI 实现 Thumb 译码功能时，先把 Thumb 指令转换成与它相应的 ARM 指令，再进行 ARM 译码。ARM9TDMI 对 Thumb 指令直接译码，缩短了译码的执行时间。AMULET3 采取了一种中间方式，对某些在时间上要求较高的控制信号直

接译码，对其他信号则采用 ARM 指令译码器。异步流水线在操作中容许有一些弹性，有一个单独的 Thumb 译码级，它在 ARM 译码工作时会有效地收缩。

寄存器读和前推级从寄存器堆中取得操作数，并在重排序缓冲器中搜索更新的数值，将正确的最新数值送到执行单元。如果正确的数值还没有准备好，则前推进程将使操作停止，直到它准备好为止。

遵循 StrongARM 设计者的见解，AMULET3 的寄存器堆有 3 个读出端口。这使得几乎所有指令都可以在单一周期内发出。

执　行

执行级包括加法器、移位寄存器和乘法器，PSR 在逻辑上也属于这一级。乘法器在 1 个周期内计算 8 位乘积。但是，该周期比典型的处理器周期快得多，所以，它在大约 20 ns 时间内就可计算一次 32×32 的乘积。加法器使用曾在 ARM9TDMI 中使用过的进位判决方案（见第 4.4 节的“进位判决加法器”）。

数据存储器接口

数据存储器接口在执行 Load 和 Store 指令时作为流水线中单独的一级，但是被其他指令旁路。这样，从慢存储器模块中取数据时将不阻止流水线中其他指令的执行，只要那些指令不使用所取的数据即可。

重排序缓冲器

重排序缓冲器按照数据形成的先后顺序从执行单元和数据存储器接口接收结果数据，并存储这些数据，直到它们可以按照程序的顺序回写到寄存器堆为止。一旦一个结果被存储到重排序缓冲器，它就可以被前推用于需要它的任何指令。

用来实现这一功能的结构如图 14.11 所示。缓冲器有 4 个槽，一个槽在指令发出时就分配给该指令产生的结果。（槽也被分配来存储指令，以便正确处理存储器的故障。）结果一产生，就被送到槽中。

重排序缓冲器的前推需要搜索缓冲器的内容。条件执行的 ARM 指令可能向缓冲器返回有效的结果，也可能不返回；多条指令可能使用同一个目的寄存器。为了鉴别特定寄存器的数据是否为最新数据，需要等待结果存入槽中，查看它是否有效，然后（最坏情况）顺序搜索其他所有的槽。这种情形非常少，而且很容易适用于异步时序框架。

寄存器回写

由重排序缓冲器输送的回写流（write-back stream）是依照程序的顺序，所以，流水线的这一级只是简单地把数据送到寄存器堆，检查每个数据的有效性，查找是否有存储器故障。如果检测到故障，就会引发一个异常，后续的结果将被丢弃，直到异常处理机制被激励为止。

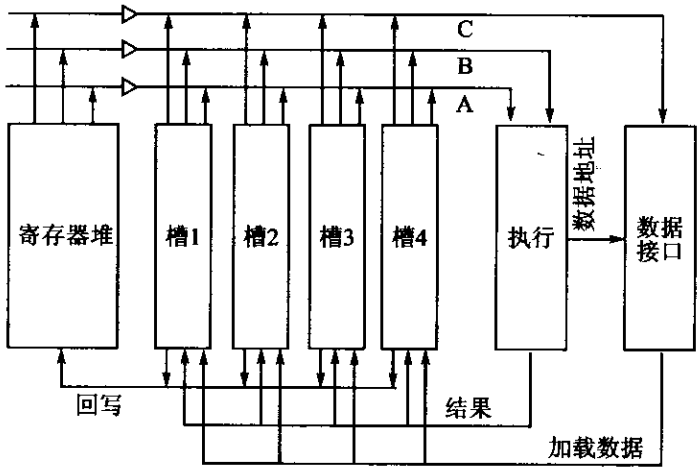


图 14.11　AMULET3 重排序缓冲器的组织

AMULET3 的性能

表 14.4 总结了 AMULET3 的性能。可以看出，实现在相同工艺条件下与 ARM9TDMI 可比性的目标已经达到了。

AMULET3 被用做 DRACO 电信控制器中的处理器核，这是下一节的题目。

表 14.4　AMULET3 的特性

工　艺	金属层	Vdd	晶体管	核面积	时　钟	MIPS	功　率	MIPS/W
0.35 μm	3	3.3 V	113 000	3 mm^2	无	120	154 mW	780

14.6　DRACO 电信控制器

DRACO(DECT Radio Communication Controller)是第一个基于 AMULET3H 的商业设计芯片。它使用在本节中将会介绍的 AMULET3H 自定时处理子系统作为计算和控制引擎。

DRACO 是由 Hagenuk GmbH 公司(设计时钟型电信外设)和曼彻斯特大学(负责 AMULET3H 子系统)在欧洲联盟的资助下协作开发的。

自定时基本原理

决定在 DRACO 芯片中采用自定时处理子系统主要是受到自定时逻辑电磁兼容性(EMC)优势的影响。高速时钟系统产生的无线电干扰可能会危及 DECT 无线电通信的性能，时钟的谐波或许会落到某个 DECT 通道的频带之中，致使该通道不能使用。

可能会作出合乎逻辑的结论，即整个芯片都以自定时逻辑工作。但是，许多电信接口在本质上是高度同步的，而且很容易用同步综合工具来设计。

因为外围的特征频率都要比处理速率低很多，所以，同步的外围子系统不应危及由自定时处理子系统带来的 EMC 优势。所有的高速处理操作和高负载外部存储器总线都异步工作，既能发挥全自定时系统最大优势，又不用花费开发自定时电信外围部件的设计费用。

DRACO 的功能

DRACO 芯片的同步外围子系统包含如下功能：

- 带有 16 kbit/s HDLC 控制器和无变压器模拟 ISDN 接口的 ISDN 控制器；
- DECT 无线电接口基带控制器和与 DECT 无线电子系统的接口；
- DECT 加密引擎；
- 四通道全双工 ADPCM/PCM 转换信号处理器；
- 用做 DECT 控制器缓冲空间的 8 KB 共用 RAM；
- 带有语音输入/输出模拟前端的电信编码译码器，它也可以用于模拟电信端口；
- 两个高速 UART，以及一个可以被任何一个 UART 使用的 IrDA 接口；
- 中断控制器；
- 计数定时器和看门狗定时器；
- 带有可编程切换功能的 2 Mbit/s IOM2 公用通道控制器；
- I^2C 接口；
- 模/数转换器(ADC)接口；
- 65 个灵活的 I/O 端口，包括具有握手能力的 8 位并行端口；
- 两个通用脉冲宽度调制器；
- 片内时钟振荡器产生 38.864 MHz 时钟，片内锁相环产生 ISDN 接口需要的 12.288 MHz 主时钟和 DECT 控制器需要的 13.824 MHz 主时钟；
- ISDN-DECT 同步锁相环避免在 ISDN 和 DECT 时钟域之间传输数据时的位损失；
- 时钟系统具有低功耗功能。

另外，自定时 AMULET3H 处理子系统包含：

- 100 MIPS AMULET3 32 位处理器，它实现了 ARM 体系结构 v4T(包括 Thumb 16 位压缩指令集)，带有调试硬件；
- 8 KB 双端口高速 RAM(处理器的本地存储)；
- 自定时片上总线，带有连接同步片上总线的桥路，通过它与同步的外围部件相连接；
- 32 通道的 DMA 控制器；
- 16 KB ROM，存放标准电信软件；
- 异步事件驱动加载模块，当完成模/数转换时，将处理器保持在零功耗等待模式，由此使软件与外部数据速率同步；
- 可编程的外部存储器接口，它支持对 SRAM、DRAM 和 Flash 存储器的直接连接；
- 由软件校准的片上基准延迟线，以控制片外存储器的访问时序。

AMULET3H

AMULET3H 是基于 AMULET3 核的子系统，它在 DRACO 电信控制器的心脏形成一个异步的“岛”。在其中，它是连接一系列同步外围控制器的接口。为了获取异步操作的好处，核必须能够访问一些异步操作的存储器。而且若有片外异步操作的存储器，则会对电磁兼容具有重要的益处。

MARBLE 片上总线

异步子系统的组织结构如图 14.12 所示。AMULET3 核直接连接到双端口 RAM（下面将进一步讨论），然后再连接到 MARBLE 片上总线。MARBLE 在概念上类似于 ARM 的 AMBA 总线，主要区别是它不使用时钟信号。它的传输机制基于分离的事项基元(transaction primitive)。

除了本地 RAM 之外，访问系统部件都要经过 MARBLE 总线。这些部件包括片上 ROM、一个 DMA 控制器、一个连接同步总线（其上接有专用外设控制器）的桥路，以及连接外部存储器的接口。

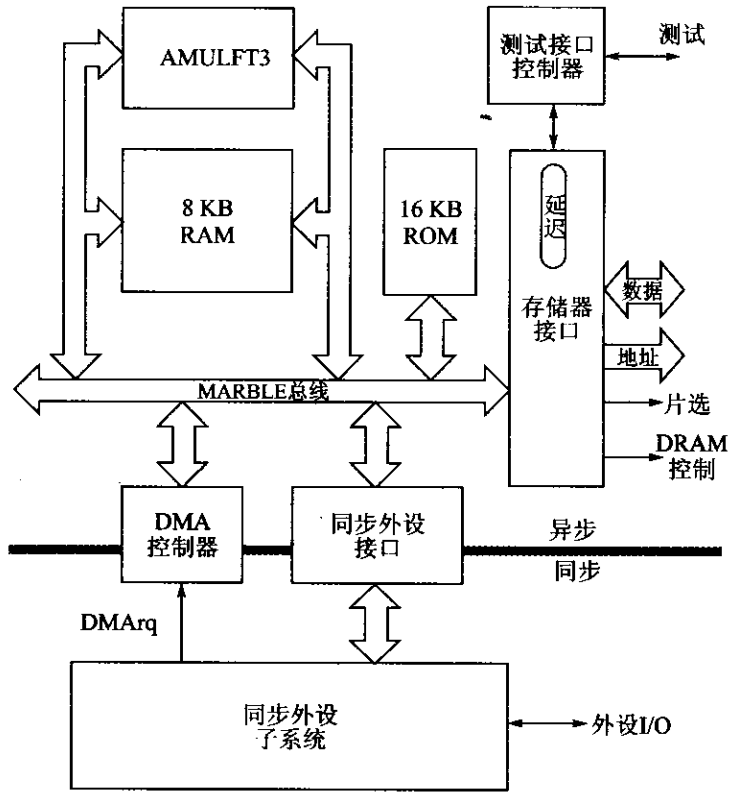


图 14.12　AMULET3H 的异步子系统

外部存储器接口

外部存储器接口为片外器件提供一组传统的信号。它类似于 AMULET2e 的接口

(见 14.4 节),高度可配置并使用基准延迟为外部访问定时。但是,AMULET3H 没有使用片外基准延迟,而是有一个可以由软件按时序基准进行校准的片上延迟。或许要在同步外围子系统中使用一个定时/计数器。

直接支持片外 DRAM,就像支持传统的 ROM、SRAM 以及存储器影射的外围器件一样。

AMULET3 的存储器组织

AMULET3 处理器核有分开的地址与数据总线,用来访问指令与数据存储器。一般这会要求分开的指令和数据存储器。RISC 系统经常使用一种改良的哈佛结构,其中有分开的指令和数据 Cache 以及统一的主存储器。

AMULET3H 控制器使用直接影射的 RAM 而不是 Cache,这是因为它更省成本,而且对实时应用具有更确定的行为。通过使用双端口存储器结构(见图 14.13),它还避免了使用分开的指令和数据存储器(以及连带的维持它们一致性的难题)。在每一位的级别上将存储器作成双端口的代价也是很高的,所以,把存储器划分为 8 个 1 KB 的模块,每个模块有两个经内部判决的端口。当向不同模块同时访问数据和指令时,可以互不妨碍地进行。当它们在同一个模块发生冲突时,一个访问将被推迟,等到另一个完成后再进行。

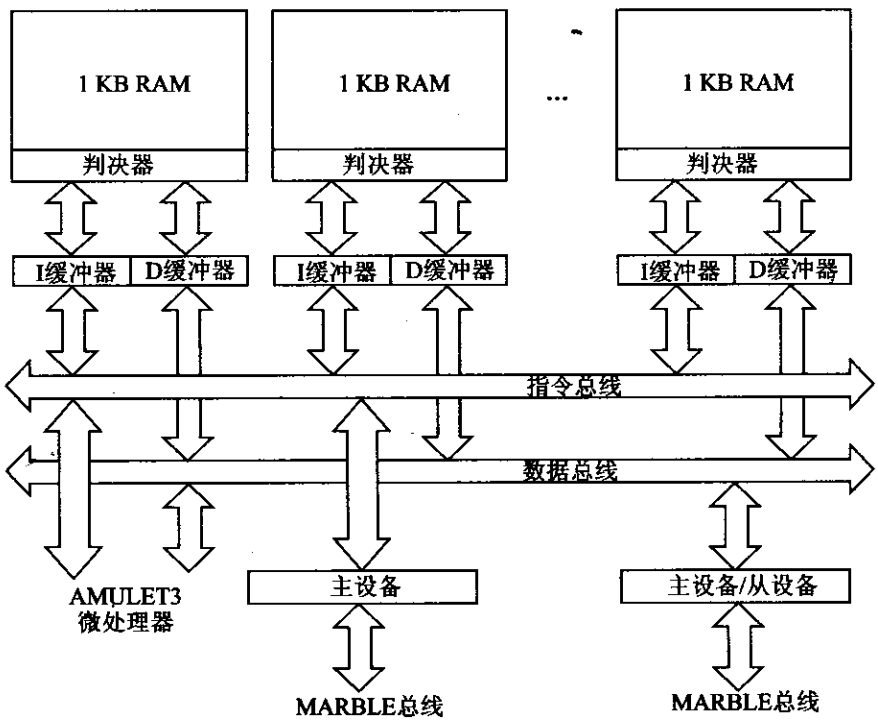


图 14.13 AMULET3H 的存储器组织

由于在每个 RAM 模块中包含有分开的 4 字长指令和数据缓冲器,冲突(及存储器的平均访问次数)会进一步减少。每次访问一个模块时,首先检查所需的数据是否在缓

冲器里，只有不在缓冲器里，才会冒着冲突的危险去询问 RAM。仿真表明，大约 60% 的取指能从这些缓冲器中得到，而且许多短的、时间关键的循环将完全从缓冲器中取指。

这些缓冲器实际上在 RAM 模块前面形成了一个简单的 128 字节的第一级 Cache。这是一个特别灵巧的类比，因为避免访问 RAM 阵列能使读出周期加快，并自动地被异步流水线所接受。

测试接口控制器

AMULET3H 是为商业应用而开发的，因而在生产中必须是可测试的。设计中有许多功能部件来保障这一点，其中最明显的是测试接口控制器（见图 14.12）。该控制器是外部存储器接口逻辑的扩展，它在控制外部存储器接口时遵循 AMBA 确定的范例（见 8.2 节的“测试接口”）。在正常操作时，它是 MARBLE 从动（slave）部件，测试时变为 MARBLE 主动（master）部件。在测试模式中，生产测试设备可以成为 MARBLE 主动部件，它能够读取片上 ROM，读写片上 RAM，还可以访问 AMULET3H 系统中许多其他测试装备。所有这些都由连接于 MARBLE 总线的测试寄存器来控制。

AMULET3H 的性能

表 14.5 总结了仿真得到的 AMULET3H 子系统的性能指标。（关于晶体管数目、尺寸和功耗的数字仅适用于异步子系统，DRACO 硅片的总面积为 7.8 mm × 7.3 mm。）由于片上 RAM 不能满足处理器峰值指令的带宽要求，所以，系统的最高性能稍低于 AMULET3 核（报告于表 14.4）。系统的功耗效率低于单独的处理器，这是因为将存储器系统的功耗也同处理器核本身的功耗一起计算进来。

表 14.5　AMULET3H 的特性

工　艺	金属层	Vdd	晶体管	核面积	时　钟	MIPS	功　率	MIPS/W
0.35 μm	3	3.3 V	825 000	21 mm^2	无	100	215 mW	465

可以看出，AMULET3H 子系统在性能和功耗效率方面与同等的时钟型 ARM 系统不分上下。它有更好的电磁干扰特性和系统功率管理特性这些优点。与这些优点相对照，必须考虑在异步设计的工具支持方面当前所存在的缺点。

DRACO 的应用

DRACO 芯片可以应用于若干电信应用系统。一个典型的应用是将 ISDN 终端与 DECT 基站结合起来。这可以使若干用户用无绳手机互相通信，并经过 ISDN 线缆或无绳 ISDN 网络端子连接到电信网络。该芯片主要的市场目标在于无绳 DECT 数据通信的应用系统。

DRACO 的管芯

图 14.14 是 DRACO 管芯的版图。AMULET3H 异步子系统占据核区域的下半

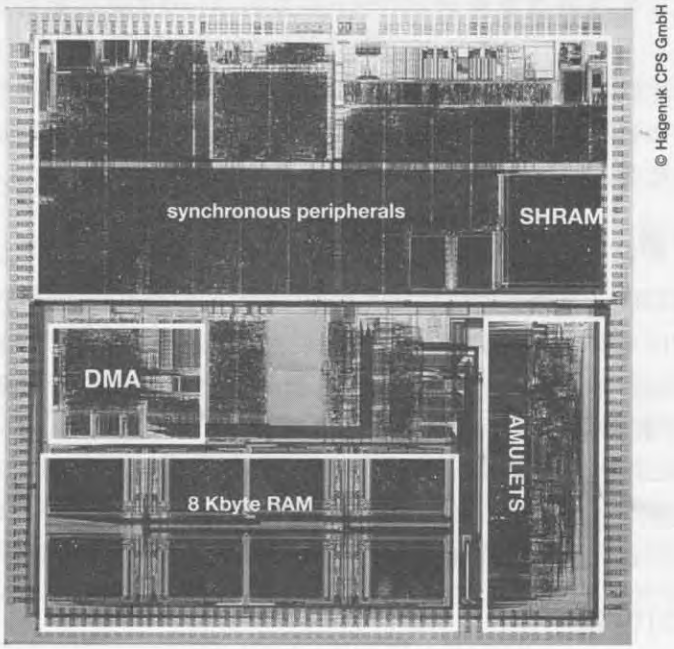


图 14.14 DRACO 的管芯图

部，同步的电信外围电路占据上半部。第一个管芯制造于 2000 年的早期。

14.7 自定时系统的未来

尽管 AMULET3 将获得商业应用，但是 AMULET 项目至今从根本上仍具有研究的性质，而且 AMULET 核也没有在广泛的应用领域中立即取代时钟型 ARM 核的前景。但是，在世界范围内对异步设计方式潜力的兴趣正在复苏。人们希望采用异步设计方式，来节省功耗，来改善电磁兼容，来获得设计更模块化的计算硬件的方法。

消除全局时钟，让每个子系统在任何需要的时候都执行自己的时序功能，这些可导致节省功耗，这在理论上是清楚的。但是，还没有多少实例说明这些好处在电路达到具有商业意义的足够复杂度时是否可以实际地实现。AMULET 研究的直接目的就是为增加对异步技术的价值有说服力的例证。

自定时设计方式被广泛接受的阻碍是现有设计团体的知识基础。由于早期一些异步计算机的设计者经受过的困难，多数 IC 设计者被培养成对异步电路有很强的厌恶。那些困难来自对自定时设计未经训练的设计方法。而现代的发展提供了异步设计框架，它能克服大多数问题，其中本质的是比在时钟框架内提供更加反常的逻辑设计方法。

以后几年将能看出，AMULET 和世界上类似的开发项目是否可以展示出异步设计足够多的优点，使设计者抛弃它们过去他们所受的大部分教育，学会履行他们职责的新方法。

AMULET 的支持

AMULET1 是使用欧共体基金在 OMI-MAP(Open Microprocessor system Initiative——Microprocessor Arcnitecture Project)中开发的。AMULET2 是在 OMI-DE/ARM 计划(Open Microprocessor system Initiative——Deeply Embedded ARM)中开发的。AMULET3 和 DRACO 的开发主要是从欧洲联盟资助的 OMI-DE2 和 OMI-ATOM 计划中得到支持。本工作的各方面都得益于来自英国政府的支持。这种支持是通过工程和物理科学研究理事会(EPSRC)以资助博士生和工具开发的形式在 ROPA 赠款 GR/K61913 中实施的。

本工作还得到 ARM Limited 公司、GEC Plessey Semiconductor 公司(现在属于 MITEL)和 VLSI Technology 公司各种形式的支持。Compass Design Automation 公司(现在属于 Avant!)的工具对这些项目的成功甚为重要。EPIC Design Technology 公司(现在属于 Synopsys)的 TimeMill 对于 AMULET2 和 AMULET3 的精确建模是至关重要的。

14.8　例题与练习

例题　14.1

总结自定时设计的优点和缺点。

在此讨论的优点如下：

- 改善了电磁兼容性(EMC)；
- 改善了功耗效率；
- 改善了设计模块化；
- 消除了时钟偏斜问题；
- 有潜力达到典型的性能而不是最坏情况性能。

缺点如下：

- 尚没有用于自定时设计的设计工具；
- 缺乏设计经验；
- 在设计教育中对自定时技术的厌恶受到鼓励；
- 缺乏对上述优点商业规模的示范。

一般来说，寻求以自定时的解决方案来解决设计问题，这仍然是一个高风险的选择。工具的缺乏可能会使它非常费力。

练习　14.1.1

如果处理器核具有分开的指令和数据端口，它们都连接到单一双端口存储器，那么试估算存储器冲突对时钟型处理器和自定时处理器性能的影响。假设存储器像 AMULET3的存储器那样由 8 个带有判决器的存储段构成，但是没有在线缓冲器(见 14.6 节中“AMULET3H 存储器组织”)。

可以设想，处理器连续地取指，而且大约每两条指令就需要访问一次数据存储器。从时钟型处理器读取指令和数据将是(被时钟)同步进行的，所以，每次发生竞争时，读指令和读数据当中将有其中一个访问停顿 1 个周期。异步处理器则不是这样同步操作，所以，读指令和读数据具有随机的相对时序。当它们发生竞争时，停顿只在第 2 个读被请求时第 1 个读剩余的时间部分出现。

练习 14.1.2

估算在 AMULET3H 存储器系统中，指令和数据在线缓冲器对性能的影响。假设条件与练习 14.1.1 的相同。

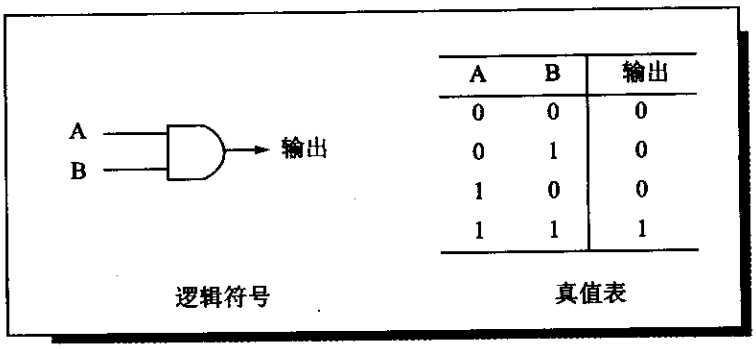
附录　计算机逻辑

计算机逻辑

计算机设计的基础是布尔逻辑，一个信号的值用“真”或“假”来表示。一般把接近地线的电压称为“假”，把接近电源的电压称为“真”。然而，可以使用任何一种能够可靠地表示两种不同状态的方法。有时也把“真”称为逻辑“1”，把“假”称为逻辑“0”。

逻辑门

逻辑门有一个或多个逻辑输入，其输出是输入的函数。例如 2 输入“与”门，当它第一个输入为“真”“与”第二个输入为“真”时，产生的输出值为“真”。由于每个输入非“真”即“假”，因此，只有 4 种可能的输入组合，门的全部功能可以用如图 A.1 所示的**真值表**来表示。图 A.1 还给出了“与”门的逻辑符号。“与”门的输入可以扩展到两个以上(尽管现行的 CMOS 技术将**扇入**(fan-in)限制为 4 个输入，对某些亚微米工艺减少到 3 个输入)。当所有输入都为“1”时，输出为“1”；当至少有一个输入为“0”时，输出为“0”。



A	B	输出
0	0	0
0	1	0
1	0	0
1	1	1

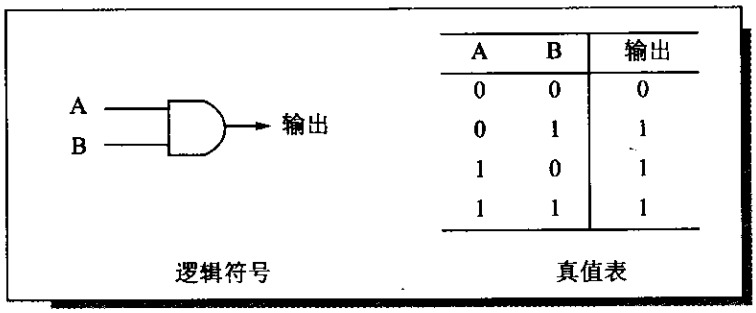
真值表

图 A.1　“与”门的逻辑符号和真值表

可以类似地定义“或”门。当所有输入都为“0”时，输出为“0”；当至少有一个输入为“1”时，输出为“1”。2 输入“或”门的逻辑符号和真值表如图 A.2 所示。

反相器是个很重要的逻辑部件。它有一个输入，产生反相的输出。它的逻辑功能为“非”。当输入为“真”(“1”)时输出为“假”(“0”)，反之亦然。输出端带有反相器的“与”门为“与非”门，输出端带有反相器的“或”门为“或非”门。通常在门的输入或输出端加一个小圆圈表示反相。

事实上，常规的简单 CMOS 门本身就是反相的，因此“与非”门比“与”门简单。后者是由前者加反相器构成的。与之类似，“或”门是由“或非”门加反相器构成的。



A	B	输出
0	0	0
0	1	1
1	0	1
1	1	1

真值表

图 A.2 “或”门的逻辑符号和真值表

布尔代数

只用 2 输入“与非”门就可以构成所有逻辑电路。这个结论是由布尔代数的规则得出的。首先，如果“与非”门的两个输入连接到同一个信号，则输出就是输入的反相。这样我们得到了反相器。其次，在“与非”门的每个输入端连接一个反相器，简单地考察真值表就会发现这个电路执行“或”的功能。通常使用传统的算术符号书写逻辑算式，用“·”表示“与”，用“+”表示“或”。这样，“A 与 (B 或 C)”就写成“A · (B + C)”。逻辑反相用上划线表示，“A 与非 B”写为“$\overline{A \cdot B}$”。这种表示法是非常方便的，只是要通过上下文表明它是一个 1 + 1 = 1 的布尔逻辑方程。

二进制数

在计算机中，通常以二进制计数法来表示数(在 6.2 节更完整地讨论了数据类型和数的表示法)。在我们熟悉的基于 10 的计数法中，每个数字都在 0～9 之间，而且每一位的取值比率都是 10 的幂(个、十、百…)。这里不采用这种计数法，而代之以二进制计数法。每个数字只能是 0 或 1，每一位的取值比率都是 2 的幂(1、2、4、8、16…)。因为每个二进制数位(bit)都只能在两个数中取值，因而可以用布尔值来表示。而完整的二进制数则可以用一组顺序排列的布尔值来表示。

二进制加法

可以使用上述的逻辑门来形成两个 1 位二进制数的和。如果两位都是 0，则和为 0。1 与 0 的和为 1。但是两个 1 的和为 2，在二进制计数法中要用两位“10”来表示。因此，两个输入都是 1 位的加法器必须有两个输出位：一位是与输入同权重的和，另一位是其权重为输入两倍的进位。

如果输入为 A 和 B，则和(sum)与进位(carry)由下式决定：

$$\text{sum} = A \cdot \overline{B} + \overline{A} \cdot B \tag{18}$$

$$\text{carry} = A \cdot B \tag{19}$$

和函数(sum)在数字逻辑中经常使用，称为“**异或**”或 XOR。“异或”的英文原意为“排他的或”，这是因为当 A 或者 B 为“真”时其结果为真，而 A 与 B 皆为“真”时其结果为“假”。“异或”有其特有的逻辑符号，如图 A.3 所示。图中还给出了用 4 个“与非”门

实现“异或”逻辑的方法。

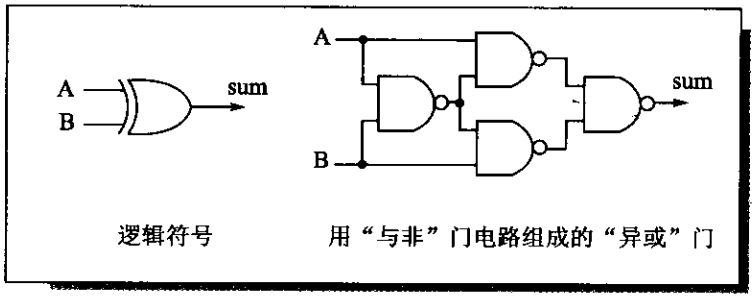


图 A.3　“异或”门的逻辑符号和“与非”门电路

可以用 1 位二进制数的加法器来构建 N 位加法器。但是除了第一位以外，其他位可能要接受来自低一位的进位输入。加法器的每一位都由输入及进位输入来产生和及进位输出。

$$sum_i = A_i \cdot \overline{B_i} \cdot \overline{C_{i-1}} + \overline{A_i} \cdot B_i \cdot \overline{C_{i-1}} + \overline{A_i} \cdot \overline{B_i} \cdot C_{i-1} + A_i \cdot B_i \cdot C_{i-1} \quad (20)$$

$$C_i = A_i \cdot B_i + A_i \cdot C_{i-1} + B_i \cdot C_{i-1} \quad (21)$$

式中：$i = 1 \sim N$；C_0 为 0。

多路器

在处理器的实现中经常需要根据不同的周期，从若干供选择的输入中选择出源操作数。执行这一功能的逻辑部件是**多路器**（或简称为 **mux**）。一个 2 输入多路器有一个布尔 select 输入和两个二进制输入 A_i 和 B_i，其中 $1 \leqslant i \leqslant N$，N 为每个二进制数的位数。当 S 为 0 时，输出 Z_i 等于 A_i；当 S 为 1 时，输出 Z_i 等于 B_i。这是一个简单易懂的功能。

$$Z_i = \overline{S} \cdot A_i + S \cdot B_i \quad (22)$$

时　钟

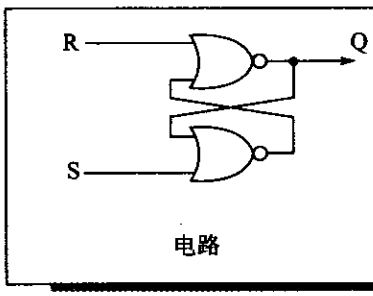
几乎所有处理器都由一个称为**时钟**的、自主运行的时序基准来控制（但有例外，见第 14 章的 AMULET 处理器核。该处理器核没有任何外部时序基准信号）。时钟信号控制处理器中状态的改变。

一般地说，所有状态都由**寄存器**来保持。在时钟周期之内，**组合逻辑**（其输出仅依赖于当前输入值）使用如前所述的布尔逻辑门计算出下一状态的值。在时钟周期结束时用有效时钟沿把所有寄存器同时切换到下一状态。为了获得最高的性能，设定时钟频率时，要在保证最坏条件下，所有组合逻辑在下一有效时钟沿到来之前能够完成运算的前提下，取最高的时钟速率。

时序电路

寄存器在两个有效时钟沿之间存储状态，在有效时钟沿到来时改变其内容。这是一种**时序**电路，它的输出不仅依赖于当前的输入值，还与以前输入值是如何变化的有关。

最简单的时序电路是 R-S(复位-置位)触发器。这是这样一种电路,只要 Set 输入有效,输出就设置为高电平;只要 Reset 有效,输出就复位为低电平。如果两个输入同时有效,则触发器的行为取决于它的实现方式。如果两个输入都无效,则触发器保持前一个状态。使用两个"或非"门实现的 R-S 触发器如图 A.4 所示。图 A.4 还给出了它的时序表。这不再是简单的真值表,因为输出不是当前输入值的组合函数。



电路

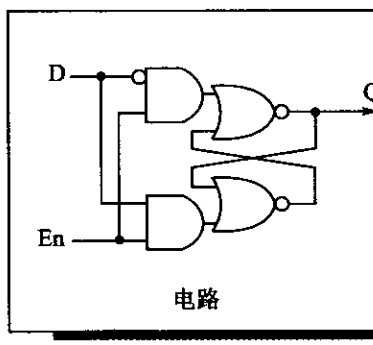
R	S	Q_{t+1}
0	0	Q_t
0	1	1
1	0	1
1	1	未定义

时序表

图 A.4 R-S 触发器及其时序表

透明锁存器

透明(或 D 型)锁存器有一个数据输入(D)和一个使能信号(En)。只要 En 为高,输出就随着输入 D 变化,当 En 保持为低电平时,输出就一直保持 En 变低之前的值。可以用 R-S 触发器构造这种锁存器,用 D 和 En 来产生 R 和 S,如图 A.5 所示。



电路

En	D	Q_{t+1}
0	0	Q_t
0	1	Q_t
1	0	0
1	1	1

时序表

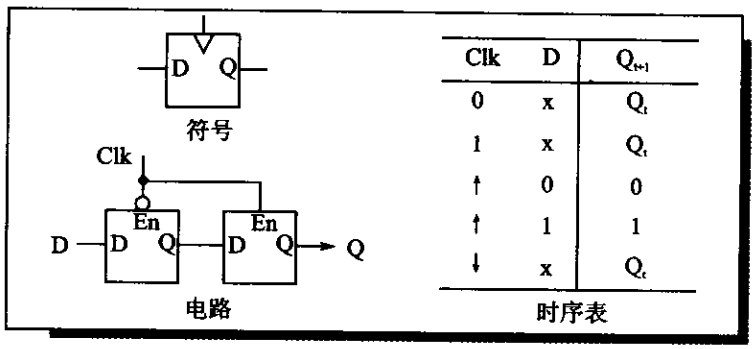
图 A.5 D 型锁存器电路及其时序表

从原理上说,只要级间的组合逻辑足够慢,就可以用 D 型锁存器构造任何时序电路。在 En 施加很短的正脉冲,就能使下一个状态的数据通过锁存器,然后在组合逻辑来不及响应新数据之前保持它。但是,实际上这种方法非常难于构造一个可靠的电路。

沿触发锁存器

有各种方法来构造更加可靠的锁存电路。其中多数方法要求每个信号在每个时钟周期通过两个透明锁存器。最简单的方法是使第二个锁存器与第一个串连,并用反相的使能信号来控制。任何时候总有一个锁存器保持,另一个锁存器透明。在时钟沿,若第一个锁存器不透明而第二个锁存器透明,则输入数据传输到输出。这个输出将保持

整个周期，直到下个周期的同样时钟沿为止。因此，这是一种**沿触发锁存器**。它的逻辑符号、电路及时序表如图 A.6 所示。（时序表中输入列的 x 表示输入不影响输出。）



Clk	D	Q_{t+1}
0	x	Q_t
1	x	Q_t
↑	0	0
↑	1	1
↓	x	Q_t

时序表

图 A.6　D 型沿触发锁存器的符号、电路和时序表

沿触发锁存器需要很仔细地设计，因为它不是简单的组合逻辑电路，它的功能与电路单元的动态特性密切相关。如果像上面所讲的那样用两个串连的透明锁存器来构造沿触发锁存器，则必须避免各种竞争条件。但是，使用好的工具，可以设计出可靠的锁存器。一旦有了可靠的锁存器，就可以构造任意复杂的时序电路了。

寄存器

用一组沿触发（或等价的）锁存器在一个时钟周期内共同地存储一个以二进制数表示的状态，就称为**寄存器**。灵活的寄存器连接着自主运行的时钟，并有一个时钟使能控制输入端。用控制逻辑可以根据每个周期的需要来确定是否更新寄存器的内容。建造这种寄存器的一个简单方法是在沿触发锁存器的共用时钟线上加一个门。如果使用下降沿触发的锁存器，那么加一个“与”门，只要在时钟变高之前将使能输入降为低电平并且在时钟再次变低之前保持为低，就可以消除整个高的时钟脉冲。（这就是使能输入的**建立和保持条件**。）如果使能信号本身就是由同一时钟控制的类似寄存器产生的，那么它将满足这些限制。寄存器电路如图 A.7 所示。

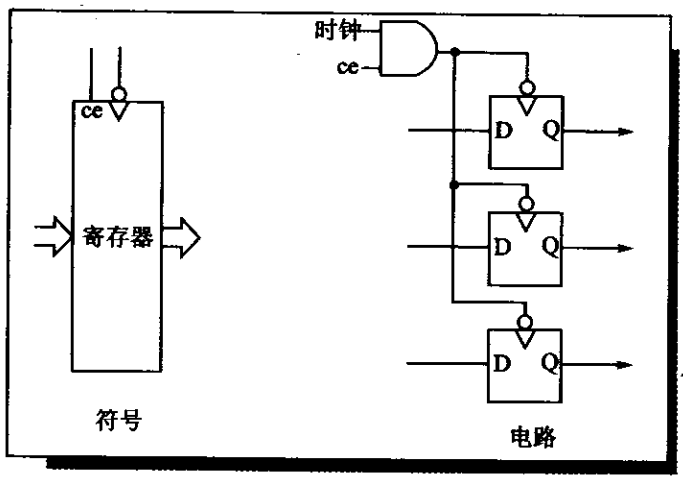


图 A.7　有时钟使能控制信号的寄存器

术　语

ACM（Association for Computing Machinery）：计算机协会。美国的计算机专业协会。

Acorn Computers Limited：于 1983 年—1985 年间开发了 ARM 处理器的英国公司。该公司还开发了 BBC 微计算机。在英国，这个机型曾广泛地应用于教育领域。它还开发了基于 ARM 的阿基米德系列计算机。该机型继 BBC 微计算机之后进入学校市场。

ADC（Analogue to Digital Converter）：模/数转换器。一种电路，它将一定范围内连续变化的输入信号转换为一组 n 位二进制数，将输入近似为该范围内 2^n 个离散量之一。

ALU（Arithmetic-Logic Unit）：算术-逻辑单元。处理器中执行算术（加、减，有时还包括乘、除）和逻辑（移位、位与，等等）操作的部件。

AMBA（Advanced Microcontroller Bus Architecture）：一种公开的片上总线标准。片上总线用来连接复杂嵌入式系统芯片的各个模块。

AMULET：指 Manchester 大学开发的、采用 ARM 体系结构的异步处理器的原型机，是一种采用低功耗技术的异步微处理器。

ANSI（American National Standards Institute）：美国国家标准学会。美国的标准化团体，它制定了许多计算机界广泛使用的约定，如 ASCII 和 ANSI 标准 C 等。

APCS（ARM Procedure Call Standard）：ARM 过程调用标准。ARM 公司制定的调用规范，使不同编译器产生的（或用汇编语言编写的）过程可以互相调用。

ARM：以前的 Advanced RISC Machine 公司（再以前是 Acorn RISC Machine 公司）。现在只使用缩写的名称而不用扩展的形式了。32 位微处理器就是根据 RISC 的原理设计的。

ARM Limited：1990 年为开发 ARM 技术从 Acorn 公司分离出来的公司。总部设在英国的剑桥，但在世界的几个区域进行设计和销售。

ASCII（American Standard for Computer Information Interchange）：美国计算机信息交换标准。是用 7 位二进制数（在今天的计算机中常常扩展到 8 位）表示可打印字符及打印控制字符的标准方法。

ASIC（Application-Specific Integrated Circuit）：专用集成电路。为特定的应用而设计的大规模集成电路，通常是为特定的用户设计的。

ASSP（Application-Specific Standard Part）：专用标准部件。为特定的应用而设计并由半导体制造商作为标准元件销售的大规模集成电路。通常，这些芯片本来是作为 ASIC 开发的，但是，后来为适应市场的需求，制造商与原来的用户达成协议，将这些芯片供应其它用户。

BBC（The British Broadcasting Corporation）：英国广播公司，英国的公共无线电与电视广播公司。BBC 的资金来源主要是特许费。在英国，这是电视拥有者的义务。他们的责任包括公众教育。在 1982 年他们根据 BBC 微计算机摄制了一部叫做“计算机程序”的通俗系列片。

BCD（Binary Coded Decimal）：一种用二进制形式对十进制的每位数字编码的数的表示法。

BCS（The British Computer Society）：英国计算机协会。英国的计算机专业协会。

C：广泛用于通用和嵌入式系统开发的 C 编程语言。

CAM（Content Addressable Memory）：按内容寻址的存储器。这种存储器包含若干不同的数据项。访问时，用一个数据值和存储的所有数据项进行比较，看是否匹配。如果有一个数据项匹配，就按匹配处的地址

输出；否则，CAM 就发出未命中的信号。它也可称为相联存储器，是相联 Cache 或 TLB 的重要部件。

CISC(Complex Instruction Set Computer)：复杂指令集计算机。这个术语是与 RISC 同时产生的，用来描述早期没有 RISC 特征的体系结构。典型的 CISC 具有指令长度可变的小型机式的指令集，有多种寻址模式，支持存储器-存储器操作及多种数据类型。

CMOS(Complementary Metal Oxide Semiconductor)：这是现代集成电路的主流技术。它在同一芯片上结合使用 NMOS 和 PMOS 场效应晶体管。这种技术使两个方向都得到有源信号的驱动，因而能够快速开关，而在非开关时功耗接近于零。

Codec(Coder-decoder)：编码-译码器。一种电子系统，它能将输入(例如模拟语音信号)转换成数字化的编码形式，也能做反向的转换。这个术语也可以用于压缩-解压器(compressor-decompressor)。

CPI(Cycles Per Instruction)：计算机效率的量度。用执行典型代码的时钟周期数除以指令数来计算。

CPSR(Current Program Status Register)：当前程序状态寄存器。ARM 寄存器，它包括条件码位、中断禁止标志位及处理器操作模式位。

CPU(The Central Processing Unit)：中央处理单元。这个术语用来指计算机中的处理器，但语义不太严密。它可能仅指整数核，也可能包括片上 MMU 和 Cache，还可以包括主存储器。在本书中，它的使用仅限于那些包括 Cache 存储器的处理器核。还用这个术语来描述由处理器、Cache 和 MMU(如果存在)组成的系统。

DAC(Digital-to-Analogue Converter)：数/模转换器。一种电路，它将通常在 n 条线上以二进制形式表示的数字信号转换为单一线上代表 2^n 个值中一个值的模拟信号。

DECT(Digital European Cordless Telephone)：一种欧洲的无绳电话标准。语音通过无线电以数字形式在手机及本地基站之间传输。

DRAM(Dynamic Random Access Memory)：动态随机访问存储器。是随机存储器中每位价格最低的一种，在大多数计算机系统中用做主存储器。数据以电荷形式储存在电容上，并会在几 ms 内泄漏掉(因此称为动态)。为了长期保存，DRAM 必须定期刷新，这就是在信号失效之前读取数据，然后再重新写入，以便恢复全电荷。

DSP(Digital Signal Processor)：数字信号处理器。一种可编程的处理器，其组织结构优化为适于处理连续的数字数据流。与之对照的是如 ARM 这样的通用处理器，它们优化的目标是控制(判定)操作。

EDO(Extended Data Out)：扩展数据输出。DRAM 的特殊接口标准，用于构建高性能存储器系统。

EEPROM(亦为 **E^2PROM**)(Electrically Erasable and Programmable Read Only Memory)：电可擦除可编程只读存储器。一种可以编程并通过施加适当电信号来擦除的 ROM。类似于 Flash 存储器，但使用不同的技术，一般没有 Flash 灵活。

EPLD(Electrically-Programmable Logic Device)：电可编程逻辑器件。一种通用的逻辑芯片，它有大量的门，其连接由片上存储器单元的状态定义。这些单元可以重新编程以改变器件的逻辑功能。

EPROM(Electrically-Programmable Read Only Memory)：电可编程只读存储器。一种可通过施加适当电信号来编程的只读存储器。通常它可以使用紫外光擦除，并可多次编程。

Flash：一种类型的电可编程只读存储器。它可以被电擦除和编程。它广泛地用来在便携式系统中存储不经常变化的程序和数据，并越来越多地用做非挥发性文件存储。

FPA(Floating-Point Accelerator)：浮点加速器。用于加速浮点操作的附加硬件，例如 ARM FPA10。

FPASC(Floating-Point Accelerator Support Code)：浮点加速器支持码。这是在带有 FPA10 浮点加速器的 ARM 系统中运行的软件。FPA10 以硬件实现 ARM 浮点指令集的子集，它需要 FPASC 来处理其余的指令。

FPE(Floating-Point Emulator)：浮点仿真器。这是在不包含 FPA10 浮点加速器的 ARM 系统中运行的软件，用于支持 ARM 浮点指令集。

FPGA(Field-Programmable Gate Array)：现场可编程门阵列。一种通用的逻辑芯片。它有大量的门，其连接由片上的可编程元件(例如反熔丝)的状态定义。

FPSR(Floating-Point Status Register)：浮点状态寄存器。ARM 浮点体系结构中的用户可见的寄存器，它控制各种选项并指示错误条件。

FPU(Floating-Point Unit)：浮点单元。在处理器中执行浮点操作的元件。

FSM(Finite State Machine)：有限状态机。一种时序数字电路。它具有内部状态，其输出及下一状态是其输入及当前状态的组合逻辑函数。

GSM(Global System for Mobile communications)：移动通信全球系统。一种用于欧洲、亚洲和世界其他地区的移动电话数字标准。

IC(Integrated Circuit)：集成电路。一种半导体器件，在它上面用同一工艺印制了若干(多达数百万)晶体管。有时称 IC 为“芯片”。

IDE(Integrated Drive Electronics)：一种硬盘驱动接口，其中所有的操作都由驱动器内的电路完成，主机接口则由数据总线和少数地址线组成。

IEE(The Institution of Electrical Engineers)：电气工程师协会。英国电子(和电气)工程师的专业协会。IEE 在计算机领域主办会议和出版期刊。

IEEE(The Institute of Electrical and Electronics Engineers, Inc)：电气和电子工程师协会。美国电子和电气工程师的专业协会。其成员不限于美国国民。IEEE 非常积极地在计算机领域主办国际会议和出版期刊，经常与 ACM(计算机协会)合作。

I²C(Inter-IC)：一种用于印制板上集成电路互联的串行总线标准。它对于连接小型 CMOS RAM 和 RTC 芯片特别有用。这些芯片即使在系统的其他部分关断时也不断电。

I/O(Input/Output)：输入/输出。指计算机与外界环境之间通过外围设备传输数据的行为。

IrDA(Infra-red Data Association)：红外数据协会。一个致力于工业界红外通信协议标准化的组织。经常将符合协会标准的系统接口称为 IrDA。

ISDN(Integrated Services Digital Network)：一项数字电话的国际标准。规定语音作为 64 Kbit/s 的数据流传送。此标准还规定了控制协议。

JTAG(Joint Test Action Group)：制定基于串行接口的测试标准的委员会。许多 ARM 芯片使用这个标准。

LCD(Liquid Crystal Display)：液晶显示。一种显示技术，由于其低重量和低功耗而应用于大多数便携式设备，例如膝上型计算机和 PDA。

LRU(Least-Recently Used)：指一种算法，用来确定在相联 Cache 或 TLB 中将哪一个值释放，以便为新值准备空间。

MFLOPS(Millions of Floating-Point Operations Per Second)：计算机执行浮点操作性能的量度。

MIPS(Millions of Instructions Per Second)：处理器执行指令的速率的量度。由于不同的指令集中每一条指令包含的语意不同，根据它们本身的 MIPS 值来比较两个处理器是没有意义的。人们试图用基准程序来为有效的比较提供一个基础。Dhrystone MIPS 是(可以证明的)更加有效的规格化测定。

MMU(Memory Management Unit)：存储器管理单元。是处理器的一部分，它使用存储器中的页表将虚拟地址转换为物理地址。它包括查表硬件和 TLB。

Modem(Modulator-demodulator)：调制/解调器。一种电子系统或子系统，它将数字数据转换为(例如)可以在常规的有声电话线上传送的形式。还可以由接收到的音频信号恢复出相似的数字数据。简单的调制/解调器使用一种音频代表 0，用另一种频率代表 1。但是，今天的高速调制/解调器使用更加复杂的调制技术。

NMOS(N-type Metal Oxide Semiconductor)：早于 CMOS 的一种半导体技术，曾用于制造某些 8 位和 16 位微处理器。它支持的逻辑系列具有有源下拉和无源上拉的输出。即使不开关时，逻辑门也有低输出拉动电流。将 NMOS 晶体管与 PMOS 晶体管一起使用，就形成 CMOS。NMOS 晶体管比 PMOS 晶体管更有

效。这就是为什么 NMOS 曾取代 PMOS 作为微处理器的优选技术。在 CMOS 中两者结合起来，尽管制造工艺更复杂了，但无论在速度和功耗效率上都有很大的提高。

OS(Operating System)：操作系统。系统中的核心代码，它管理应用代码，处理调度、资源分配和保护等等。

PC(Program Counter)：程序计数器。处理器中的寄存器，它保存待读取的下一条指令的地址。通常应根据上下文区分本术语的这种用途和下面一种用途。

PC(Personal Computer)：个人计算机。这个术语尽管很普通，但它现在用来指与 IBM PC 兼容的台式计算机。这就是说，除了其他部件，它使用拥有 Intel x86 指令集体系结构的处理器。

PCMCIA(Personal Computer Memory Card International Association)：个人计算机存储器卡国际委员会。负责 PC 机(及其他便携式设备)插接卡的物理形式和接口标准的组织。该标准并不限于其名称中所述的存储器卡，多种外设接口卡也遵从该标准。PCMCIA 卡简称为“PC 卡”。

PDA(Personal Digital Assistant)：个人数字助理。该术语最初指 Apple Newton，但是现在用于所有掌上计算机。

PLA(Programmable Logic Array)：可编程逻辑阵列。实现复杂多输出组合逻辑功能的伪规则集成电路。

PLL(Phase-Locked Loop)：锁相环。使用另一基准时钟信号来产生时钟信号的电路。

PMOS(P-type Metal Oxide Semiconductor)：早于 CMOS 和 NMOS 的一种半导体技术，曾用于制造早期 4 位微处理器。它支持的逻辑系列具有有源上拉和无源下拉的输出。即使不开关时，逻辑门也有高输出拉动电流。将 NMOS 晶体管与 PMOS 晶体管一起使用就形成 CMOS。

PSR(Program Status Register)：程序状态寄存器。处理器中用于保存条件码、中断禁止位和操作模式位等各种信息位的寄存器。在 ARM 中，对于每个非用户模式都有一个 CPSR(当前程序状态寄存器)和一个 SPSR(保存程序状态寄存器)。

PSU(Power Supply Unit)：电源单元。提供系统所需稳定电源电压的电子线路，通常使用市电电源或电池电源。

RAM(Random Access Memory)：随机访问存储器。这是一个不当的用词，因为 ROM 也是随机访问的。RAM 指的是在计算机中用于存储程序及数据的读-写存储器。也用这个术语来指用来构造这类存储器以及 Cache 和 TLB 等结构的半导体元件。

RISC(Reduced Instruction Set Computer)：精简指令计算机。其体系结构具有某些特征的一类计算机。这些特征基于 Patterson (U. C. Berkeley)、Ditzel (Bell 实验室) and Hennessy (Stanford 大学)1980 年阐释的原理。

ROM(Read-Only Memory)：只读存储器。计算机中存储固定程序的存储器。也用来指可以用于这种用途的半导体器件。可对照 RAM。

RS232：异步串行通信的特定标准，使用它可连接调制解调器、打印机，并与其它计算机通信。

RTC(Real-Time Clock)：实时时钟。计算机用来计算时间和日期等信息的时钟源。通常它是一个带有低频晶体振荡器的、电池支持的小型系统。即使计算机本身关断，它也会全时运行。

RTL(Register Transfer Level)：寄存器传输级。硬件系统的一种抽象级，例如在处理器中，将多位的数据看做总线和寄存器之间的数据流。

RTOS(Real-Time Operating System)：实时操作系统。这种操作系统支持那些必须满足外部时序限制的程序。它们通常是较小的(几千字节)，并适用于嵌入式系统。

SDLC(Synchronous Data Link Controller)：同步数据链接控制器。一种外围器件，它将计算机连接到时钟控制的串行接口，并支持一种或数种标准协议。

SoC(System-on-Chip)：片上系统。含有电子系统全部功能，包括处理器、存储器及外围器件的单片集成电路。直到本书写作时，大多数 SoC 除了其片上的存储器之外，还需要片外的存储器资源，但其它的系统部件都在片上。

SPSR(Saved Program Status Register)：保存程序状态寄存器。一种 ARM 寄存器，当出现异常时用来保存

CPSR 的值。

SRAM(Static Random Access Memory)：静态随机访问存储器。这种形式的 RAM 比 DRAM 贵，它将数据保存在不需要刷新的触发器中。SRAM 的访问时间比 DRAM 的短，可以无限期并几乎无功耗地保持其内容。在处理器芯片中，它用于实现大多数 RAM 功能，例如 Cache 和 TLB 存储器，在某些小型嵌入式系统中还可以用做主存储器。

TLB(Translation Look-aside Buffer)：地址转换后备缓冲器，存储最近使用过的页表项的 Cache。它可以避免每次访问存储器时搜索页表的开销。

UART(Universal Asynchronous Receiver/Transmitter)：通用异步收发器。处理器总线与串行线(一般使用 RS232 信号协议)接口的外围设备。

USB(Universal Serial Bus)：通用串行总线。最近 PC 机上支持各种外设连接的标准接口。它使用高速串行协议和允许器件在机器运行时连接和断开(即热插拔)的电气接口。

VHDL(VHSIC Hardware Description Language)：超高速集成电路硬件描述语言(其中 VHSIC 代表 Very High-Speed Integrated Circuit)。一种在行为级或结构级描述硬件的标准语言。大多数半导体设计工具公司都支持这种语言。

VLSI(Very Large Scale Integration)：超大规模集成。在单个芯片上集成大量晶体管的工艺。人们曾企图根据晶体管数的数量级将芯片划分为 SSI、MSI、LSI(小规模、中规模及大规模集成)和 VLSI 等等，但工艺技术进步的速度超过创造新术语的速度。新术语 ULSI(Ultra Large Scale Integration，甚大规模集成)还没有获得广泛应用，现在就有可能被废除。

VLSI Technology, Inc：公司名称，有时简称为 VLSI。该公司制造了 Acorn Computers 公司设计的第一个 ARM 芯片，并于 1990 年与 Acorn 公司及 Apple 公司一起建立了独立的 ARM 公司。VLSI 是 ARM 公司的第一个半导体合伙人，制造了一系列基于 ARM 的 CPU 及系统芯片。它现在属于 Philips Semiconductors 公司。

VM(Virtual Memory)：虚拟存储器。程序运行所占用的地址空间由 MMU 映射到物理存储器。虚拟空间可以大于物理空间，部分虚拟空间可以分页到硬盘上，也可能不在任何位置。

VRAM(Video Random Access Memory)：一种带有片上移位寄存器的 DRAM，访问顺序数据时可以获得高带宽，以产生视频显示。

参考文献

ARM 读本

- http://www.arm.com/
  在 ARM 公司的网页上可找到 ARM 处理器的数据表及其他相关资料。
- Jaggar (ed.). ARM Architectural Reference Manual. Prentice Hall.
  ISBN 0-13-736299-4
  一本参考书，详述了 ARM 和 Thumb 指令及存储器管理结构等。在 ARM 业界称为“ARM ARM”。

Thumb

- Segars, Clarke and Goudge. Embedded Control Problems, Thumb, and the ARM7TDMI. IEEE Micro, 15(5), October 1995, pages 22～30.
  一篇论述 Thumb 指令集原理和性能的论文。

RISC 体系结构

- Patterson and Ditzel. The Case for the Reduced Instruction Set Computer. ACM SIGARCH Computer Architecture News, 8(6), October 1980, pages 25～33.
  一篇置疑处理器设计日益复杂化趋势的开创性论文。
- Katevenis. Reduced Instruction Set Computer Architectures for VLSI. MIT Press, Cambridge, MA, USA, 1985.
  ISBN 0-262-11103-9
  Berkeley RISC 设计及美国计算机协会博士论文奖获得者的详细报告。
- Hennessy and Patterson. Computer Architecture——A Quantitative Approach, 2nd edition. Morgan Kaufmann, San Francisco, CA, USA, 1990.
  ISBN 1-55860-329-8
  主流 RISC 设计的权威著作，由最初发现 RISC 的两个人写成。
- Patterson and Hennessy. Computer Organization and Design——The Hardware/Software Interface. Morgan Kaufmann, San Francisco, CA, USA, 1994.
  ISBN 1-55860-281-X

由相同作者(但不同排序)继《Computer Architecture——A Quantitative Approach》一书成功后写成的续篇。本书更详尽地涉及基础知识。

CMOS 设计

- Weste and Eshraighan. Principles of CMOS VLSI Design, A Systems Perspective, 2nd edition. Addison-Wesley, Reading, MA, USA, 1993.
  ISBN 0-201-53376-6
  关于晶体管级 CMOS 设计的权威著作,涉及物理版图、电路和系统。

自定时逻辑

- Proceedings of the IEEE, 87(2), February 1999, ISSN 0018-9219.
  这本关于异步设计的专刊收集了若干复杂异步设计的背景材料和细节,包括 AMULET2e。
- http://www.cs.man.ac.uk/async/
  异步逻辑的主页,它包含指导教材和对世界上活跃于异步逻辑研究领域的大多数团体的链接。

大端与小端

- Cohen. On Holy Wars and a Plea for Peace. Computer, 14(10), October 1981, IEEE Computer Society Press, pages 48～54.
  本文提出关于字节顺序的大端与小端问题,并仿照《格利佛游记》一书建立了大端与小端的术语。

C 语言

- Kernighan and Ritchie. The C Programming Language, 2nd edition. Prentice Hall, Englewood Cliffs, NJ, USA, 1988.
  ISBN 0-13-110362-8
  这是一本 C 语言的标准参考书。请注意第二版包括 ANSI 标准 C 语言,而第一版不包括。
- Koenig. C Traps and Pitfalls. Addison-Wesley, Reading, MA, USA, 1989.
  ISBN 0-201-17928-8
  本书讲述如何避免 C 编程中的标准陷阱,对各种水平的程序员都很有用。

索　引