

“黑色经典”系列之《ARM 系列处理器应用技术完全手册》



第 2 章 ARM 体系结构

华清远见

2.1 ARM 体系结构的特点

ARM 内核采用精简指令集结构（RISC，Reduced Instruction Set Computer）体系结构。RISC 技术产生于上世纪 70 年代。其目标是设计出一套能在高时钟频率下单周期执行、简单而有效的指令集，RISC 的设计重点在于降低硬件执行指令的复杂度，这是因为软件比硬件容易提供更大的灵活性和更高的智能。与其相对的传统复杂指令级计算机（CISC）则更侧重于硬件执行指令的功能性，使 CISC 指令变得更复杂。

RISC 的设计思想主要有以下特性。

- Load/Store 体系结构。

Load/Store 体系结构也称为寄存器/寄存器体系结构或者 RR 系统结构。在这类机器中，操作数和运算结果不是通过主存储器直接取回而是借用大量标量和矢量寄存器来取回的。与 RR 体系结构相反，还有一种存储器/存储器体系结构，在这种体系结构中，源操作数的中间值和最后的运算结果是直接从主存储器中取回的。这类机器的缩写符号是 SS 体系结构。

- 固定长度指令。

固定长度指令使得机器译码变得比较容易。由于指令简单，需要更多的指令来完成相同的工作，但是随着存储器存取速度的提高，处理器可以更快地执行较大代码段（即大量指令）。

- 硬联控制。

RISC 机以硬联控制指令为特点，而 CISC 的微代码指令则相反。使用 CISC（常常是可变长度的）指令集时处理器的语义效率最大，而简单指令往往容易被机器翻译。像 CISC 那样通过执行较少指令来完成工作未必省时，因为还要包括微代码译码所需要的时间。因此，由硬件实现指令在执行时间方面提供了更好的平衡。除此之外，还节省了芯片上用于存储微代码的空间并且消除了翻译微代码所需的时间。

- 流水线。

指令的处理过程被拆分为几个更小的、能够被流水线并行执行的单元。在理想情况下，流水线每周期前进一步，可获得更高的吞吐率。

- 寄存器。

RISC 处理器拥有更多的通用寄存器，每个寄存器都可存放数据或地址。寄存器可为所有的数据操作提供快速的局部存储访问。

表 2.1 总结了 RISC 和 CISC 之间主要的区别。

表 2.1 RISC 和 CISC 之间主要的区别

指 标	RISC	CISC
指令集	一个周期执行一条指令，通过简单指令的组合实现复杂操作；指令长度固定	指令长度不固定，执行需要多个周期
流水线	流水线每周期前进一步	指令的执行需要调用微代码的一个微程序
寄存器	更多通用寄存器	用于特定目的的专用寄存器

Load/Store 结构	独立的 Load 和 Store 指令完成数据在寄存器和外部存储器之间的传输	处理器能够直接处理存储器中的数据
---------------	--	------------------

为了使 ARM 指令集能够更好地满足嵌入式应用的需要，ARM 指令集和单纯的 RISC 定义有以下几方面的不同。

- 一些特定指令的周期数可变

并非所有的 ARM 指令都是单周期的。例如，多寄存器转载/存储的 Load/Store 指令的周期数就不确定，必须根据被传送的寄存器个数而定。如果是访问连续的存储器地址，就可以改善性能，因为连续的存储器访问通常比随机访问要快。同时，代码密度也得到了提高，因为在函数的起始和结尾，多个寄存器的传输是很常用的操作。

- 内嵌桶形移位器产生更复杂的指令

内嵌桶形移位器是一个硬件部件，在一个输入寄存器被一条指令使用之前，内嵌桶形移位器可以处理该寄存器中的数据。它扩展了许多指令的功能，改善了内核的性能，提高了代码密度。

- Thumb 指令集

ARM 处理器根据 RISC 原理设计，但是由于各种原因，在低代码密度上它比其他多数 RISC 要好一些，然而它的代码密度仍不如某些 CISC 处理器。在代码密度重要的场合，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。

- 条件执行

只有当某个特定条件满足时指令才会被执行。这个特性可以减少分支指令数目，从而改善性能，提高代码密度。

- DSP 指令

一些功能强大的数字信号处理（DSP）指令被加入到标准的 ARM 指令中，以支持快速的 16×16 位乘法操作及饱和运算。在某些应用中，传统的方法需要微处理器加上 DSP 才能实现。这些增强指令，使得 ARM 处理器也能够满足这些应用的需要。

综上所述，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；
- 每条指令都条件执行；
- 多寄存器的 Load/Store 指令；
- 能够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 指令集也当作 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。

2.2 流水线

2.2.1 流水线的概念与原理

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 译码以鉴别它是属于哪一条指令（dec）；
- ③ 从指令中提取指令的操作数（这些操作数往往存在于寄存器中）（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，则访问存储器以存储数据（mem）；
- ⑥ 将结果写回到寄存器堆（res）。

并不是所有的指令都需要上述每一个步骤，但是，多数指令需要其中的多个步骤。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指令结束之前就开始，那么在每一步骤内处理器只有少部分的硬件在使用。

有一种方法可以明显改善硬件资源的使用率和处理器的吞吐量，这就是当前一条指令结束之前就开始执行下一条指令，即通常所说的流水线（Pipeline）技术。流水线是 RISC 处理器执行指令时采用的机制。使用流水线，可在取下一条指令的同时译码和执行其他指令，从而加快执行的速度。可以把流水线看作是汽车生产线，每个阶段只完成专门的处理器任务。

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 2.1 说明了这个过程。从原理上说，这样的流水线应该比没有重叠的指令执行快 6 倍，但由于硬件结构本身的一些限制，实际情况会比理想状态差一些。

2.2.2 流水线的分类

从 Acorn Computer 公司在 1983~1985 年间开发的第一个 3μm 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，这些处理器都是采用 3 级流水线，而这一时期 CMOS 工艺的发展，几乎将特征尺寸减少了一个数量级。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。

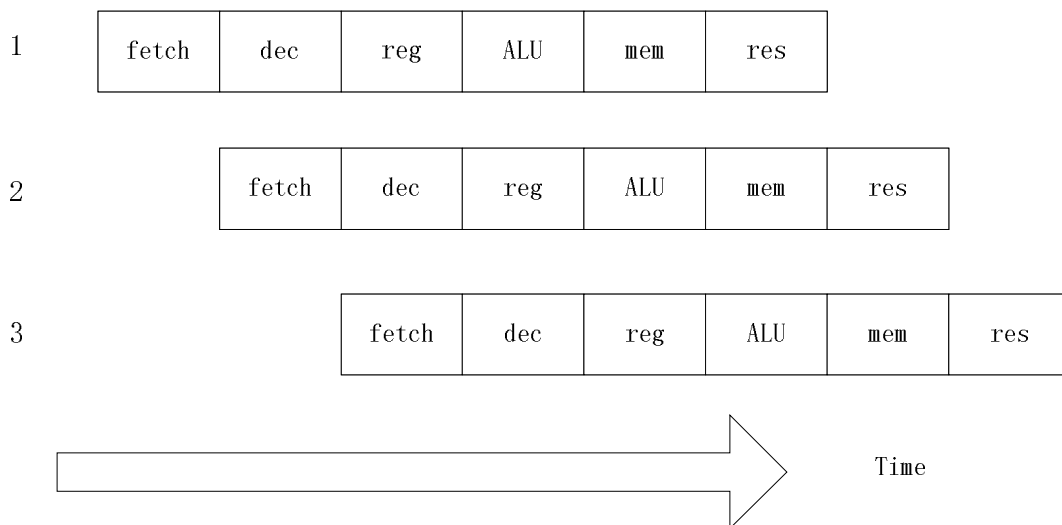


图 2.1 流水线的指令执行过程

从 1995 年以来，ARM 公司推出了几个新的 ARM 核。它们采用 5 级流水线和哈佛架构，获得了显著的高性能。例如，ARM9 增加了存储器访问段和回写段，这使得 ARM9 的处理能力可达到平均 1.1 Dhrystone¹ MISP/MHz，与 ARM7 相比，指令吞吐量提高了约 13%。

在许多高性能处理器内部，一级 Cache 一般都设置有两个，其中一个是指令 Cache，另一个是数据 Cache。这样可以减少取指令和读操作数的访问冲突，这种结构被称为哈佛架构。

注意 把主存储器分成两个独立编址的存储器，一个专门存放指令，称为指令存储器，简称指令存；另一个专门存放操作数，称为数据存储器，简称数存。两个存储器可以同时访问，这样就解决了取指令和读操作数的冲突。如果在此基础上规定在执行指令阶段产生的运算结果只写到通用寄存器中，不写到主存，那么取指令、分析指令和执行指令就可以同时进行。

ARM10 更是把流水线增加到 6 级。ARM10 的平均处理能力达到 1.3 Dhrystone MISP/MHz，与 ARM7 相比，指令吞吐量提高了约 34%。

注意 虽然 ARM9 和 ARM10 的流水线不同，但它们都使用了与 ARM7 相同的流水线执行机制，因此 ARM7 上的代码也可以在 ARM9 和 ARM10 上运行。

1. 3 级流水线 ARM 组织

3 级流水线 ARM 组织如图 2.2 所示，其主要的组成如下：

① 处理器状态寄存器堆（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。另外还有附加的可以访问 PC 的一个读端口和一个写端口。

注意 PC 的附加写端口可以在取指地址增加后更新 PC，读端口可以在数据地址发出之后从新开始取指。

¹ Dhrystone 是测量处理器运算能力的最常见基准程序之一，Dhrystone 的计量单位为每秒计算多少次 Dhrystone。

- ② 桶形移位寄存器（Barrel Shifter）。它可以把一个操作数移位或循环移位任意位数。
- ③ ALU。完成指令集要求的算术或逻辑功能。

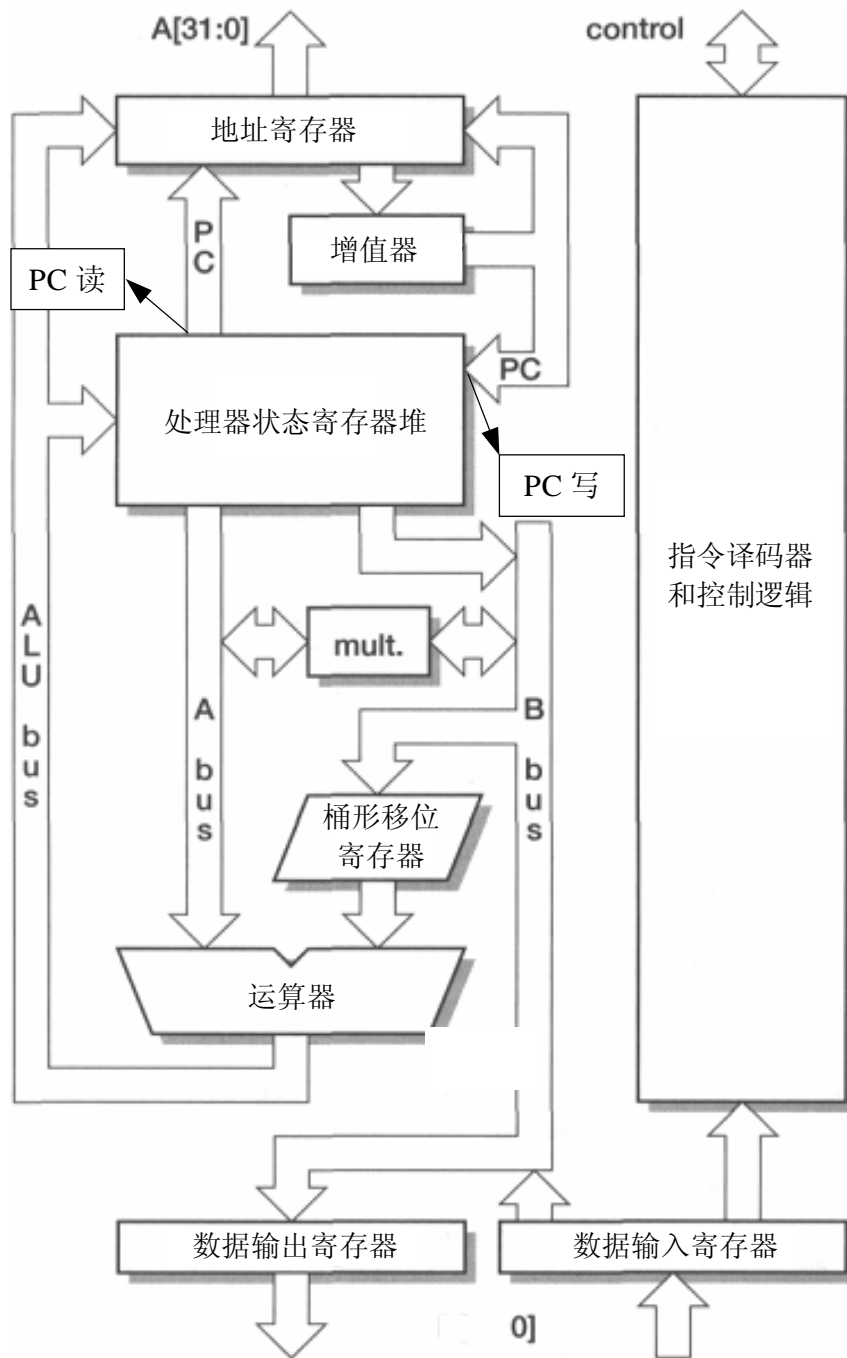


图 2.2 3 级流水线 ARM 的组织

- ④ 地址寄存器（Address Register）和增值器（Incrementer）。可选择和保存所用的存储

器地址并在需要时产生顺序地址。

⑤ 数据输出寄存器（data-out register）和数据输入寄存器（data-in register）。用于保存传输到存储器和从存储器输出的数据。

⑥ 指令译码器和相关的控制逻辑（instruction decode and control）。

例 2.1 显示了一条单周期指令在流水线上的执行过程。

【例 2.1】

ADD r1, r2

指令在流水线上的执行过程如图 2.3 所示。

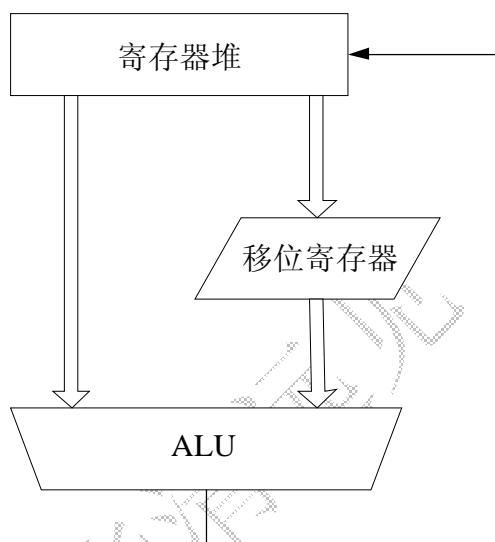


图 2.3 单周期指令在流水线上的执行过程

在 ADD 指令中，需要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。在指令执行过程中，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。然后将增值后的数据拷贝到寄存器堆的 r15（程序计数器），同时还拷贝到地址寄存器，作为下一次取指的地址。

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：

- 取指（fetch）：从寄存器装载一条指令。
- 译码（decode）：识别被执行的指令，并为下一个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，不占用数据通路。
- 执行（execute）：处理指令并将结果写回寄存器。

图 2.4 显示了 3 级流水线指令执行过程。

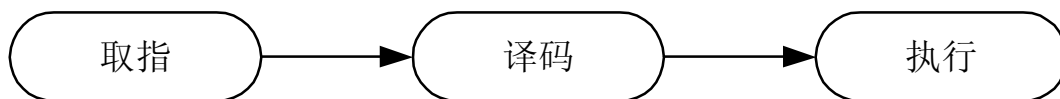


图 2.4 3 级流水线

注意 在任一时刻，可能有 3 种不同的指令占有这 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，流水线使得平均每个时钟周期能完成 1 条指令。但 1 条指令需要 3 个时钟周期来完成，因此，有 3 个时钟周期的延时（latency），但吞吐率（throughput）是每个周期一条指令。例 2.2 通过一个简单的例子说明了流水线的机制。

【例 2.2】

指令序列为：

```
ADD r1 r2
SUB r3 r2
CMP r1 r3
```

流水线指令序列如图 2.5 所示。

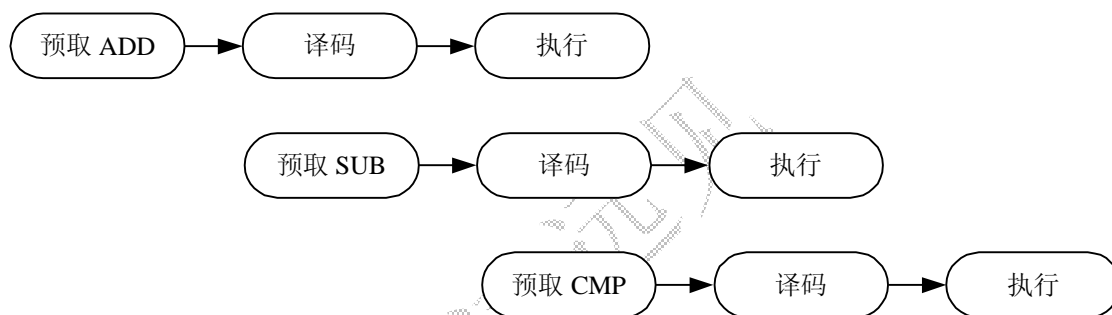


图 2.5 流水线指令顺序

在第一个周期，内核从存储器取出指令 ADD；在第二个周期，内核取出指令 SUB，同时对 ADD 译码；在第三个周期，指令 SUB 和 ADD 都沿流水线移动，ADD 被执行，而 SUB 被译码，同时又取出 CMP 指令。可以看出，流水线使得每个时钟周期都可以执行一条指令。

当执行多条指令时，流水线的执行不一定会如图 2.5 那么规则，图 2.6 显示了有 STR 指令的流水线状态。

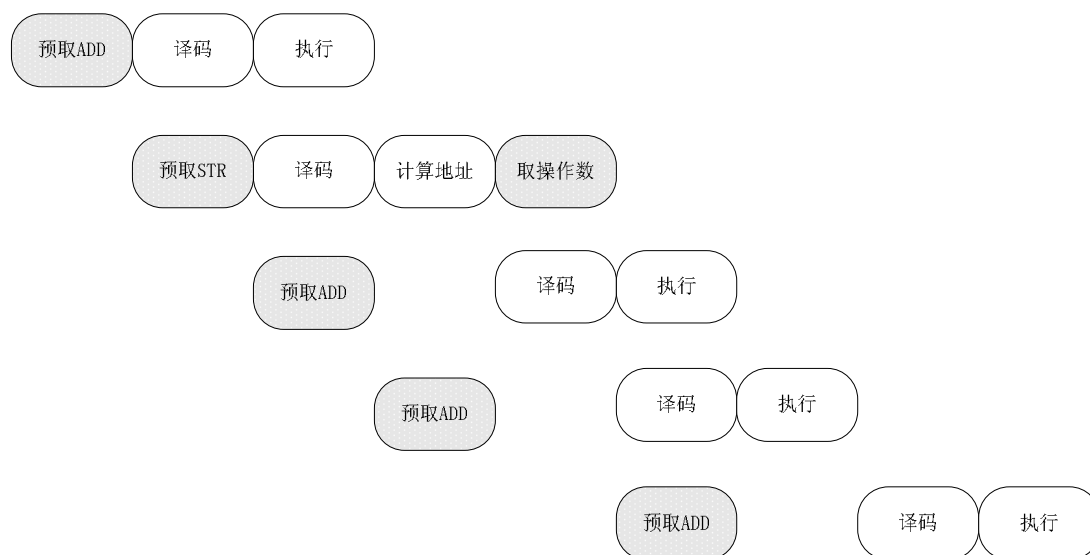


图 2.6 含有存储器访问指令的流水线状态

图 2.6 中在单周期指令 ADD 后出现了一条数据存储器指令 STR。访问主存储器的指令用阴影表示，可以看出在每个周期都使用了存储器。同样，在每一个周期也使用了数据通路。在执行周期、地址计算和数据传输周期，数据通路都是被占用的。在译码周期，译码逻辑负责产生下一周期用到的数据通路的控制信号。

注意

对于 STR 这种存储器访问指令，实际是在地址计算时由译码逻辑产生下一周期数据传输所需要的数据通路控制信号。

在图 2.6 中的指令序列中，处理器的每个逻辑单元在每个指令都是活动的。可以看出流水线的执行与存储器访问密切相关。存储器访问限制了程序执行必须花费的指令周期数。

ARM 的流水线执行模式导致了一个结果，就是程序计数器 PC（对使用者而言为 r15）必须在当前指令执行前计数。例如，指令在其第一个周期为下条指令取指，这就意味着 PC 必须指向当前指令的后 8 个字节（其后的第 2 条指令）。

当程序中必须用到 PC 时，程序员要特别注意这一点。大多数正常情况下，不用考虑这一点，它由汇编器或编译器自动处理这些细节。

例 2.3 显示了流水线下程序计数器 PC 的使用情况。

【例 2.3】

指令序列为：

```
0x8000 LDR pc, [pc, #0]
0x8004 NOP
0x8008 DCD jumpAddress
```

当指令 LDR 处于执行阶段时， $pc = address + 8$ 即 0x8008。

2. 5 级流水线 ARM 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。执行一个给定的程序需要的时间由下式决定：

$$T_{\text{prog}} = (N_{\text{inst}} \times \text{CPI}) / f_{\text{clk}}$$

式中：

N_{inst} ：表示在程序中执行的 ARM 指令数；

CPI：表示每条指令的平均时钟周期；

f_{clk} ：表示处理器的时钟频率。


因为对给定程序（假设使用给定的优化集并用给定的编译器来编译） N_{inst} 是常数，所以，仅有两种方法来提供性能。

第一，提高时钟频率。时钟频率的提高，必然引起指令执行周期的缩短，所以要求简化流水线每一级的逻辑，流水线的级数就要增加。

第二，减少每条指令的平均指令周期数 CPI。这就要求重新考虑 3 级流水线 ARM 中多于 1 个流水线周期的实现方法，以便使其占有较少的周期，或者减少因指令相关造成的流水线停顿，也可以将两者结合起来。

3 级流水线 ARM 核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周期来改善系统性能，效果是不明显的。为了改善 CPI，存储器系统必须在每个时钟周期中给出多于一个的数据。方法是在每个时钟周期从单个存储器中给出多于 32 位数据，或者为指令或数据分别设置存储器。

基于以上原因，较高性能的 ARM 核使用了 5 级流水线，而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，进而可以使用更高的时钟频率，分开的指令和数据存储器使核的 CPI 明显减少。

 **注意** 分开的指令和数据存储器。一般是分开的 Cache 连接到统一的指令和数据存储器上。

在 ARM9TDMI 中使用了典型的 5 级流水线。ARM9TDMI 的组织结构如图 2.7 所示。

5 级流水线包括下面的流水线级：

- 取指（fetch）：从存储器中取出指令，并将其放入指令流水线。
 - 译码（decode）：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。
 - 执行（execute）：将其中一个操作数移位，并在 ALU 中产生结果。如果指令是 Load 或 Store 指令，则在 ALU 中计算存储器的地址。
 - 缓冲/数据（buffer/data）：如果需要则访问数据存储器，否则 ALU 只是简单地缓冲一个时钟周期。
 - 回写（write-back）：将指令的结果回写到寄存器堆，包括任何从寄存器读出的数据。
- 图 2.8 显示了 5 级流水线指令的执行过程。

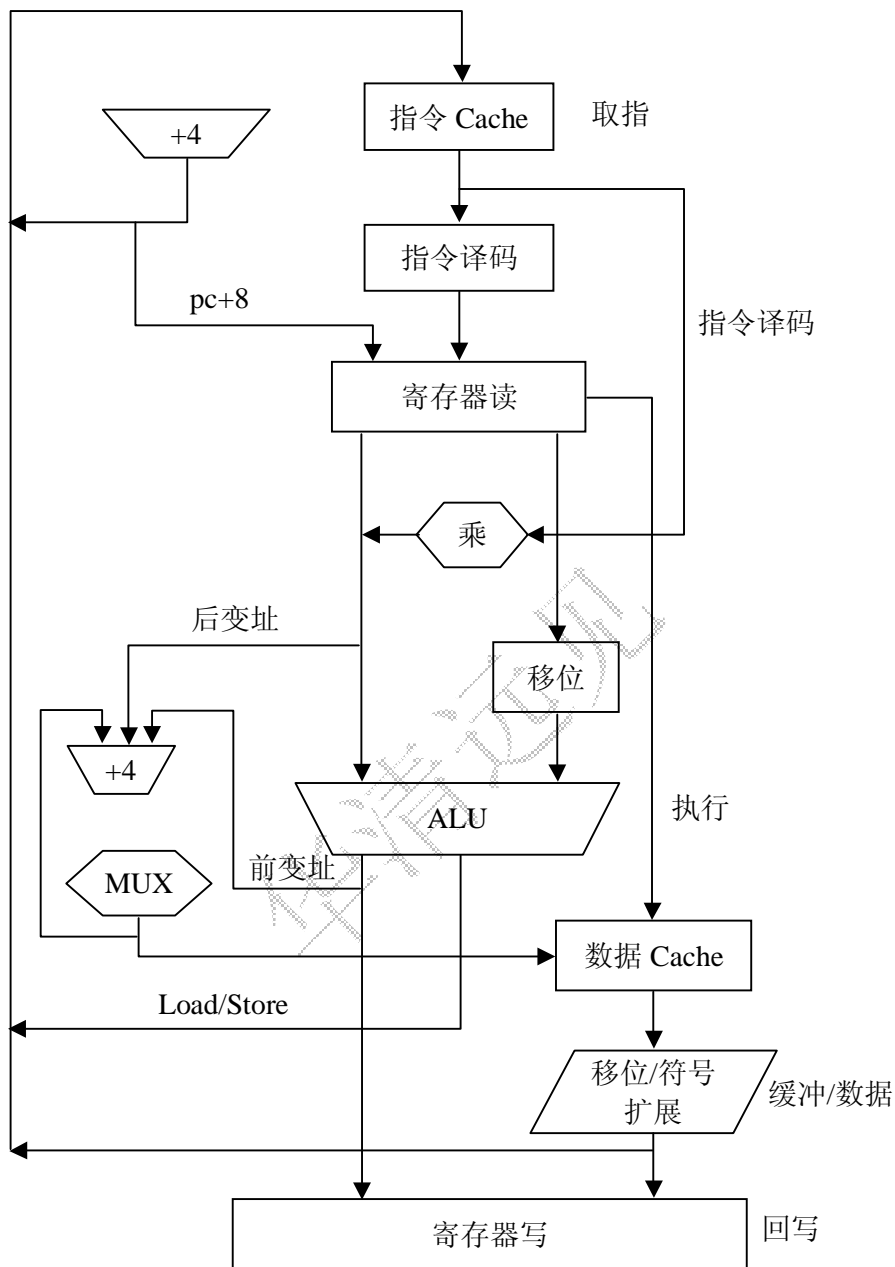


图 2.7 5 级流水线的组织结构

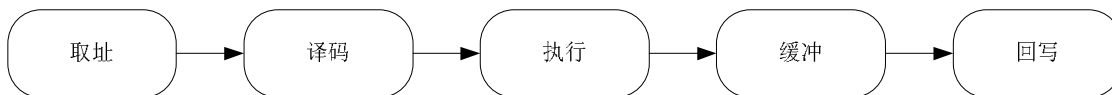


图 2.8 5 级流水线

在程序执行过程中，PC 值是基于 3 级流水线操作特性的。5 级流水线中提前 1 级来读取指令操作数，得到的值是不同的（PC+4 而不是 PC+8）。这产生的代码不兼容是不容许的。但 5 级流水线 ARM 完全仿真 3 级流水线的行为。在取指级增加的 PC 值被直接送到译码级的寄存器，穿过两极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。

3. 6 级流水线 ARM 组织

在 ARM10 中，将流水线的级数增加到 6 级，使系统的平均处理能力达到了 1.3Dhrystone MIPS/MHz。图 2.9 显示了 6 级流水线上指令的执行过程。

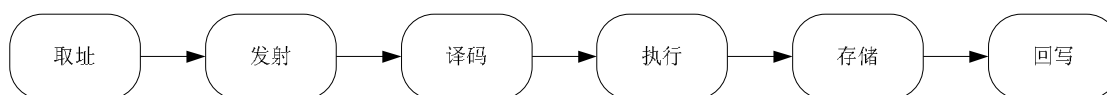


图 2.9 6 级流水线

2.2.3 影响流水线性能的因素

1. 互锁

在典型的程序处理过程中，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。如例 2.4 所示。

【例 2.4】

有如下指令序列：

```
LDR r0,[r0,#0]
ADD r0,r0,r1    ;在 5 级流水线上产生互锁
```

从例 2.4 中可以看出，流水线的操作产生中断，因为第一条指令的结果在第二条指令取数时还没有产生。第二条指令必须停止，直到结果产生为止。

2. 跳转指令

跳转指令也会破坏流水线的行为，因为后续指令的取指步骤受到跳转目标计算的影响，因而必须推迟。但是，当跳转指令被译码时，在它被确认是跳转指令之前，后续的取指操作已经发生。这样一来，已经被预取进入流水线的指令不得被丢弃。如果跳转目标的计算是在 ALU 阶段完成的，那么，在得到跳转目标之前已经有两条指令按原有指令流读取。

解决的办法是，如果有可能最好早一些计算转移目标，当然这需要硬件支持；如果转移指令具有固定格式，那么可以在解码阶段预测跳转目标，从而将跳转的执行时间减少到单个周期。但要注意，由于条件跳转与前一条指令的条件码结果有关，在这个流水线中，还会有条件转移的危险。

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

级数越多，问题就越严重。对于相对简单的处理器，使用 3~5 级流水线效果最好。

显然，只有当所有指令都依照相似的步骤执行时，流水线的效率达到最高。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。

2.3 ARM 存储器

ARM 处理器内核广泛应用于嵌入式系统和其他行业应用中。为了适应不同系统的需要，ARM 采用了灵活多样的存储管理体系。从平板式内存映射到灵活方便的 MMU 内存管理单元，用户可以根据自己的需要使用不同的存储管理策略。

在 ARM 体系结构中可使用的存储管理策略包括：

- 多类型的存储单元（可以使用 SDRAM、FLASH 等）；
- Cache；
- 写缓存；
- 虚拟内存地址。

另外，内存映射 I/O 机制可以使开发者灵活、方便地增加大量外设。

可以通过下面的几种方法实现对存储系统的管理：

- 使能 Cache，加快存储器的访问速度；
- 启动虚拟地址到物理地址的映射；
- 使用“域管理”策略，对存储单元的访问进行保护；
- 对 I/O 映射地址空间的访问加以限制。

标准的对 ARM 处理器的存储管理是使用协处理器 CP15 来实现的。ARM 体系的存储系统将在第 15 章详细介绍。

2.4 I/O 管理

ARM 系统完成 I/O 功能的标准方法是使用存储器映射 I/O。这种方法使用特定的存储器地址。当从这些地址加载或向这些地址存储时，它们提供 I/O 功能。某些 ARM 系统也可能有直接存储器访问（DMA，Direct Memory Access）硬件。

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以下 5 种寄存器。


- ① 发送数据寄存器（只写）：写入这个位置的数据被送往串行线。
- ② 接受数据寄存器（只读）：保存从串行线送来的数据。
- ③ 控制寄存器（读/写）：设置数据速率，管理 RTS（请求发送）和其他类似信号。
- ④ 中断使能寄存器（读/写）：控制中断的硬件事件。
- ⑤ 状态寄存器（读/写）：指示读数据是否有效、写缓存是否满等。

要接受数据，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存器中并进行错误检测。

应该注意的是，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（read-sensitive）的。

当涉及读敏感寄存器时，编程必须小心。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，不能在用户模式下访问 I/O 寄存器。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。

 **注意** 在 ARM 编程中，通常将存储器的 I/O 区域标记为非 Cache 区（uncacheable），并绕过 Cache 访问。通常 Cache 与读敏感（read-sensitive）器件相互排斥。显示帧缓存器（DisplayFrame Buffers）也需要仔细考虑，通常也设为不可 Cache 的。

2.5 ARM 开发调试方法

用户选用 ARM 处理器开发嵌入式系统时，选择合适的开发工具可以加快开发进度，节省开发成本。因此一套含有编辑软件、编译软件、汇编软件、链接软件、调试软件、工程管理及函数库的集成开发环境（IDE）一般来说是必不可少的，如 ARM 公司的 RealView 开发环境。至于嵌入式实时操作系统、评估板等其他开发工具则可以根据应用软件规模和开发计划选用。

使用集成开发环境开发基于 ARM 的应用软件，包括编辑、编译、汇编、链接等工作全部在 PC 机上即可完成，调试工作则需要配合其他的模块或产品方可完成。目前常见的调试方法有以下几种。

1. 指令集模拟器

部分集成开发环境提供了指令集模拟器，可方便用户在 PC 机上完成一部分简单的调试工作。但是，由于指令集模拟器与真实的硬件环境相差很大，因此即使用户使用指令集模拟器调试通过的程序也有可能无法在真实的硬件环境下运行，用户最终必须在硬件平台上完成整个应用的开发。

2. 驻留监控软件

驻留监控软件（Resident Monitors）是一段运行在目标板上的程序，集成开发环境中的调试软件通过以太网口、并行端口、串行端口等通信端口与驻留监控软件进行交互，由调试软件发布命令通知驻留监控软件控制程序的执行、读写存储器、读写寄存器、设置断点等。

利用驻留监控软件是一种比较低廉有效的调试方式，不需要任何其他的硬件调试和仿真设备。ARM 公司的 Angel 就是该类软件，大部分嵌入式实时操作系统也采用该类软件进行调试，不同的是在嵌入式实时操作系统中，驻留监控软件是作为操作系统的任务存在的。

驻留监控软件的不便之处在于它对硬件设备的要求比较高，一般在硬件稳定之后才能进

行应用软件的开发，同时它占用目标板上的一部分资源，而且不能对程序的全速运行进行完全仿真，所以对一些要求严格的情况不是很适合。

3. JTAG 仿真器

JTAG 仿真器也称为 JTAG 调试器，是通过 ARM 芯片的 JTAG 边界扫描口进行调试的设备。JTAG 仿真器比较便宜，连接比较方便，通过现有的 JTAG 边界扫描口与 ARM 处理器核通信，属于完全非插入式（即不使用片上资源）调试。它无需目标存储器，不占用目标系统的任何端口，而这些是驻留监控软件所必需的。另外，由于 JTAG 调试的目标程序是在目标板上执行，仿真更接近于目标硬件，因此，许多接口问题，如高频操作限制、AC 和 DC 参数不匹配、电缆长度的限制等被最小化了。使用集成开发环境配合 JTAG 仿真器进行开发是目前采用最多的一种调试方式。

4. 在线仿真器

在线仿真器使用仿真头完全取代目标板上的 ARM 处理器，可以完全仿真 ARM 芯片的行为，提供更进一步的调试功能。但这类仿真器为了能够全速仿真时钟速度高于 100MHz 的处理器，通常必须采用极其复杂的设计和工艺，因而其价格比较昂贵。在线仿真器通常用在 ARM 的硬件开发中，在软件的开发中较少使用。其价格高昂是在线仿真器难以普及的原因。