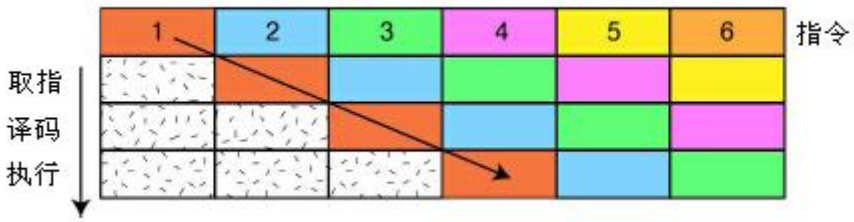
第一章 ARM7 内核

ARM 的设计精髓是结构简单。ARM7 内核采用了精简指令集计算机（RISC）设计思想，所用逻辑门数较少，硅片面积小，但具有高性能、低功耗的特点，这使得 ARM7 成为嵌入式系统的理想选择。ARM7 内核包括 ARM7TDMI（-S）、ARM720T 等，ARM7TDMI 处理器内核已经许可给许多世界顶级半导体公司，它是第一个包括 Thumb 指令集、快速乘法指令和嵌入式 ICE 调试技术的内核。

完整的 ARM 内核包括很多内容，如体系结构、指令系统、存储结构、调试接口等。本章将用较少的篇幅，从应用和编程者的角度，对 ARM7 内核的特点作一个精简的描述。

1.1 流水线（Pipeline）

ARM7 内核是冯• 诺伊曼体系结构，数据和指令使用同一条总线。ARM7 CPU 的核心是一条指令流水线，流水线用来处理从程序存储器中取出的指令。ARM7 使用的是三级流水线结构，执行 ARMv4 指令集（ISA）。



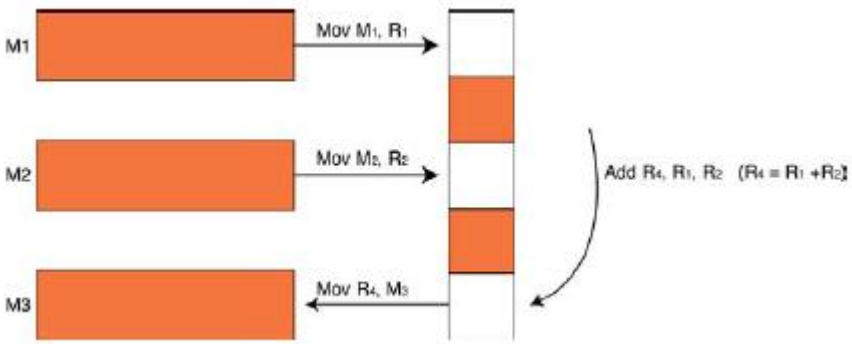
三级流水线结构是流水线技术中较简单的实现形式，并且不会出现多级流水线中会遇到的写前读（read-before-write）等数据相关问题。流水线的三级结构是由硬件实现的，在执行一条指令的同时，译码另一条指令并读取第三条指令。流水线技术大大提高了 CPU 的指令吞吐率，使得大多数 ARM 指令可以在一个时钟周期内完成。在执行无跳转的线性代码时，流水线的效率最高。当出现分支时，流水线将被清空，需要重新填满才能恢复到全速执行。后面我们将看到 ARM 指令集中许多有趣的特性，这些特性使得代码中小的跳转可以被平滑掉，从而使指令流可以更好地通过流水线。由于流水线是 CPU 的一部分，它对程序员是透明的。但必须记住：PC 指向的是正在被执行指令的前 8 个字节位置，当计算与 PC 有关的地址时，这一情况需要注意。例如指令：

0x4000 LDR PC, [PC, #4]

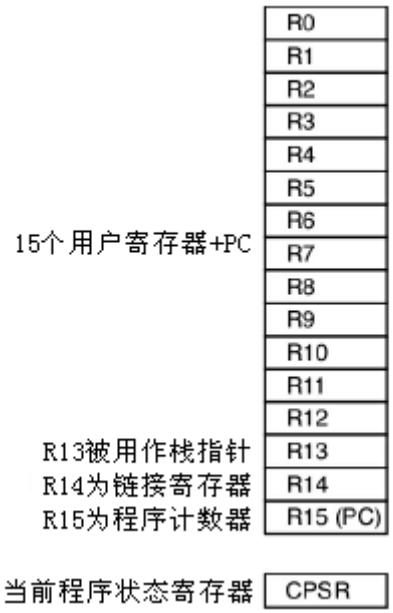
会把地址 PC+4 的内容载入 PC，但由于 PC 指向的是正在执行指令的前 8 个字节，所以地址 0x400C 中的内容会被载入 PC，而不是像你可能以为的那样载入地址 0x4004 中的内容。

1.2 寄存器

ARM7 是 Load-Store（装载-存储）结构的，所以为了执行任何数据处理指令，首先要把数据从存储器中装载到寄存器中，在数据处理指令执行结束后，需要的话数据再被保存到存储器中。

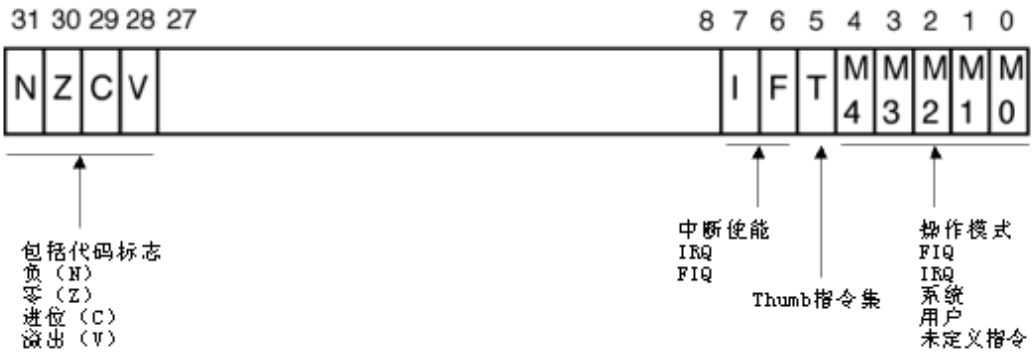


ARM 的中央寄存器集是 16 个用户寄存器 R0 – R15。这些寄存器均是 32 位宽度，R0 – R12 没有其他特殊功能，寄存器 R13 – R15 在 CPU 中有特殊功能。R13 被用作栈指针（stack pointer, SP）。R14 被称为链接寄存器（link register, LR），当调用一个函数时返回地址被自动保存到链接寄存器，在函数返回时有效。这使得快速进入和返回“叶”函数（不调用其他函数的函数）成为可能。如果函数是分支的一部分（即该函数将调用另一个函数），链接寄存器必须入栈（R13）。R15 是程序计数器（program counter, PC）。有趣的是，许多指令也可以在 R13 – R15 中执行，就像它们是标准的用户寄存器。



当前程序状态寄存器

除了 16 个用户寄存器外，还有一个 32 位宽的寄存器——当前程序状态寄存器（CPSR）。CPSR 包括一组反映和控制 ARM7 CPU 操作的标志（Flags）。



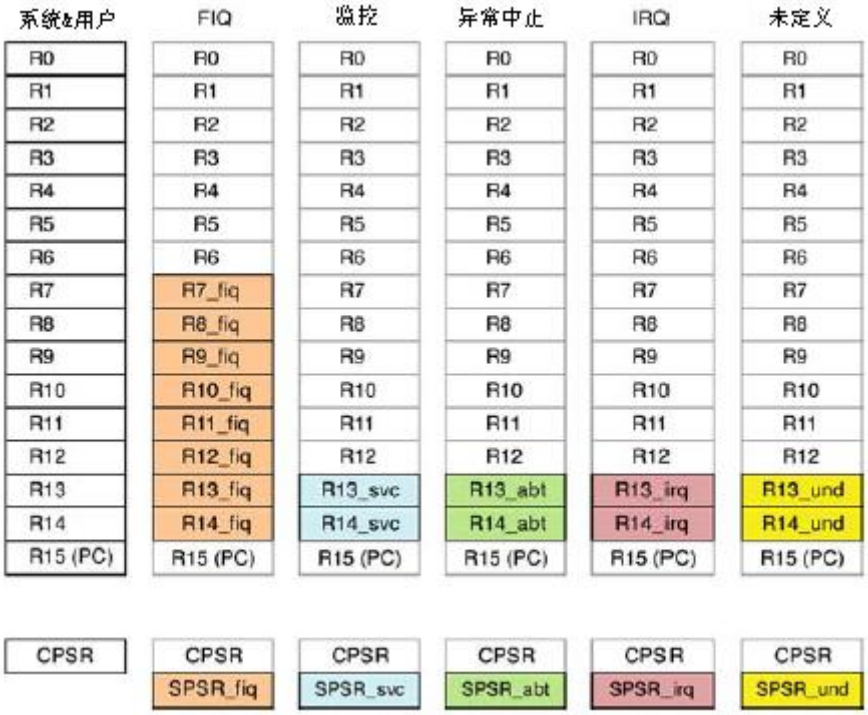
当前程序状态寄存器包括表示数据处理操作结果的代码标志和设置操作模式和使能中断的用户标志。T 位仅用于参考，表示 Thumb 指令集。

CPSR 中的最高四位是可由 **CPU** 设置的条件码。条件码指示数据处理、操作结果的状态。我们可从条件码中看出数据处理操作是否产生了负值，零，进位或溢出。**CPSR** 的最低 **8** 位是可由用户程序设置或清除的标志。位 **7** 和位 **8** 分别为 **I** 和 **F** 位，用于使能和禁止两个对于 **ARM7 CPU** 来讲为外部信号的中断源。位 **5** 为 **THUMB** 位，指示执行 **Thumb** 指令集。

ARM7 CPU 可以执行两个指令集——**32** 位宽的 **ARM** 指令集和 **16** 位宽的 **THUMB** 指令集，**T** 位指示正在执行的是哪一个指令集。用户代码不能通过置位或清除这一位来改变当前的指令集，下面我们会讲到正确的进入/切换方式。最后五位是模式位，**ARM7** 有 **7** 种不同的操作模式。用户的应用程序

通常运行于用户模式，可访问寄存器组 **R0 – R15** 以及 **CPSR**。但是，为了响应诸如中断、存储器错误，或者软件中断指令之类的异常，处理器将改变运行模式。当改变模式时，寄存器 **R0－R12** 和 **R15** 的作用不变，但 **R13**（**LR**）和 **R14**（**SP**）被相应模式特有的寄存器替代。这意味着每一个模式都有它自己的栈寄存器和链接寄存器。此外，快速中断模式（**FIQ**）有自己的 **R7－R12** 寄存器，这意味着不需要将寄存器入栈就可以快速进入 **FIQ** 中断。

除了用户模式外的所有模式都有一个额外的寄存器——“程序状态保存寄存器”（**SPSR**）。当应用程序在用户模式下运行时，如果发生了一个异常，**CPU** 将改变操作模式，并将 **CPSR** 的当前内容保存到 **SPSR**。执行完异常代码后返回时，**SPSR** 的内容被恢复到 **CPSR**，以允许应用程序继续正常执行。具体操作模式描述如下。



ARM7 CPU 有六种用于处理异常的操作模式。由阴影标明的寄存器是当操作模

式改变时使能的寄存器。SPSR 寄存器用于在改变模式时保存 CPSR 的副本。

1.3 异常

当发生异常时，**CPU** 将改变操作模式，**PC** 被强制进入异常向量。向量表开始是处于零地址的复位异常向量，然后每四个字节为一个异常向量。

异常	模式	地址
复位	监控	**0x00000000**
未定义指令	未定义	**0x00000004**
软件中断（**SWI**）	监控	**0x00000008**
预取异常终止（指令预取存储器异常终止）	异常终止	**0x0000000C**
数据异常终止（数据访问存储器异常终止）	异常终止	**0x00000010**
IRQ（中断）	**IRQ**	**0x00000018**
FIQ（快速中断）	**FIQ**	**0x0000001C**

注意：由于在地址 **0x00000014** 处缺少了一个向量，向量表中有一个缺口。这一位置在早期的 **ARM** 体系结构中有使用，所以 **ARM7** 中保留它，以确保不同 **ARM** 体系结构之间的兼容性。

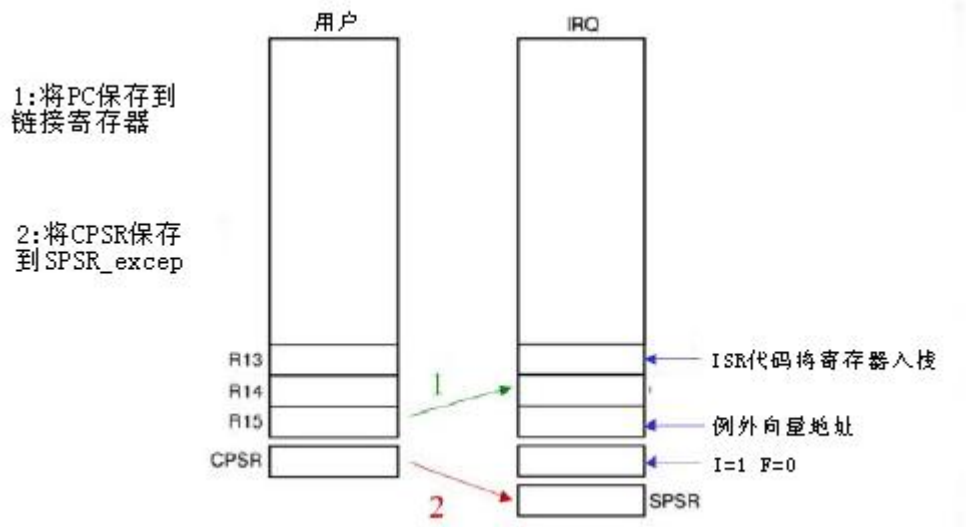
如果发生多重异常，则按下表所示确定优先级。

优先级	异常

最高 1	复位
2	数据异常终止
3	FIQ
4	IRQ
5	预取异常终止
最低 6	未定义指令 SWI

每个异常有一个固定的优先级。片上外围部件（片上外设）由 FIQ 和 IRQ 二组中断服务。每一个外围部件的优先级还可以在相应的组内被指定。

当一个异常发生时，比如说 IRQ 异常，内核将进行以下操作：首先下一条将要执行的指令的地址（PC + 4）被保存到链接寄存器（LR），然后 CPSR 被复制到将要进入的异常模式的 SPSR（即 SPSR_irq），接着将异常模式的中断向量地址写入 PC，IRQ 模式的向量地址为 0x00000018。同时，CPU 切换到 IRQ 模式，这将导致寄存器 R13 和 R14 被 IRQ_R13 和 IRQ_R14 所取代。在进入 IRQ 模式时，CPSR 的 I 位被置位，使 IRQ 中断被禁止。如果想要 IRQ 中断嵌套，用户程序必须使能 IRQ 中断，并将链接寄存器的内容入栈以保存原来的返回地址。用户程序将从异常中断向量，跳转到该异常的服务程序 ISR，所以用户程序要做的第一件事就是将 ISR 将会用到的寄存器 R0 – R12 压入 IRQ 堆栈。这些事完成后，用户程序就可以开始进行异常处理了。



当用户代码完成异常处理后，CPU 必须回到用户模式继续执行。但是 ARM 指令集中不包括“返回”或者“从中断返回”指令，所以必须通过普通指令控制 PC。由于有多种不同的返回情况，返回还是比较复杂的。

首先，考虑软件中断 SWI 指令。在这一情况下 SWI 指令被执行，下一指令的地址被存储在链接寄存器。异常被处理后，只需将链接寄存器的内容移入 PC 中，即可从异常返回并继续执行。为了使 CPU 返回到用户模式，需要用到 move 指令的一个修改版本，即 MOVS。所以对于一个软件中断，返回指令为：

```
MOVS    R15, R14        ；将链接寄存器的内容移入 PC 中并切换模式
```

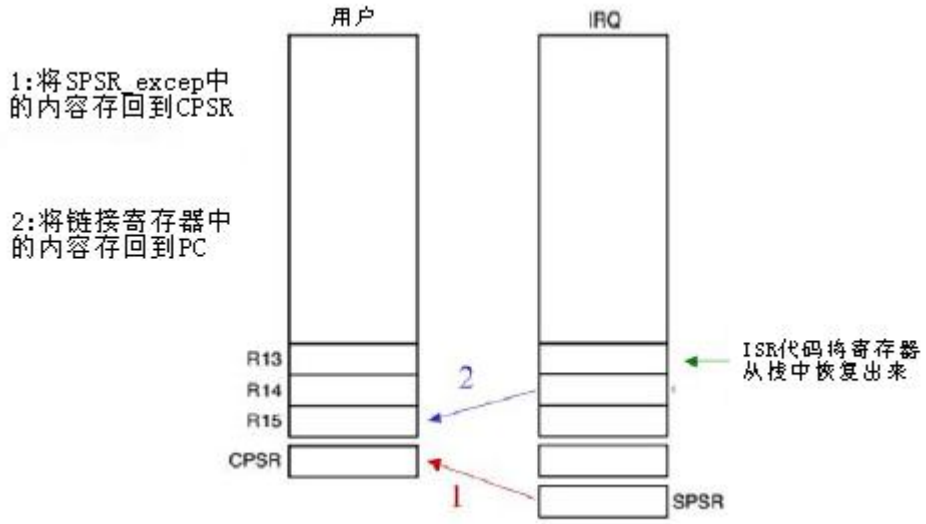
但是，当 FIQ 和 IRQ 异常发生时，当前正在被译码的指令被抛弃并进入异常处理。当代码从异常返回时，链接寄存器中的地址是被抛弃的指令地址加 4。为了从正确的位置继续执行，我们需要将链接寄存器中的内容减 4。在这种情况下，我们使用减法指令将链接寄存器中的值减 4，然后将结果存到 PC 中。就象 move 指令一样，减法指令也有一个切换操作模式的特殊形式。对于 IRQ，FIQ 和预取异常终止，返回指令为：

```
SUBS    R15，R14，#4  ；
```

对于数据终止异常，异常将在导致异常的指令的下一条指令后产生。在这种情况下，我们进入数据异常终止 **ISR**，解决存储器的问题后，重新执行导致异常后面的指令，所以我们需要将 **PC** 的内容退回两条指令，即被抛弃的指令和导致异常的指令。换句话说，需要将链接寄存器的内容减 **8**，再存入 **PC**。对于数据终止异常，返回指令如下：

```
SUBS    R15，R14，#8  ；
```

一旦返回指令被执行，修改后的链接寄存器内容被移入 **PC**，**CPU** 进入用户模式且 **SPSR** 被恢复到 **CPSR**。在 **FIQ** 或者 **IRQ** 异常情况下，返回后相应的中断也被使能。这将退出特权模式并回到用户代码准备继续执行。

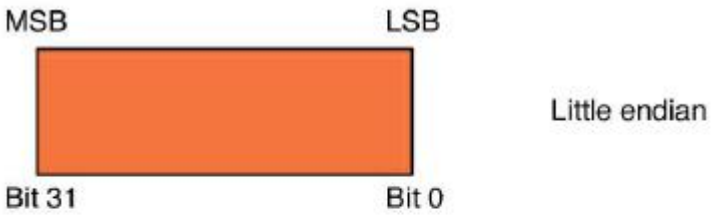


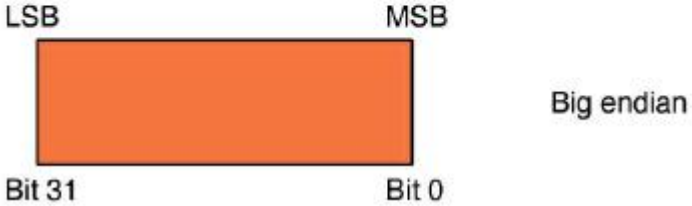
1.4 ARM7 指令集

现在我们已对 **ARM7** 的体系结构、编程模型和操作模式有所了解，接下来看一下它的指令集。虽然我这里的例程大多是以 **C** 写的，我们的目标也不是要成为一个 **ARM** 汇编程序专家，但对于开发高效代码来说，理解机器代码是非常重要的。在开始介绍 **ARM7** 指令集前，我们先介绍几个相关的重要技术问题和术语。

ARM7 CPU 有两种指令集：**32** 位宽的 **ARM** 指令集和 **16** 位宽的 **THUMB** 指令集。下面的一节中，**ARM** 指的是 **32** 位指令集，而 **ARM7** 表示 **CPU**。

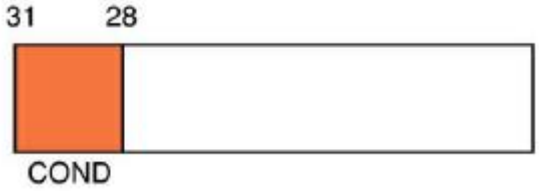
ARM7 可以被设计成*大端*（***big-endian***）或者*小端*（***little-endian***）处理器，即 **MSB** 位于最高位或者最低位。用户必须确保他们设置了正确的存储格式，否则代码或数据就会颠倒。





ARM 指令集最有趣的特点之一，就是所有的指令都可以被有条件的执行，简称*条件执行*。在以往的微控制器中，只有*条件转移指令*或*位测试和设置*一类的指令是条件指令。但是在 **ARM** 指令集中，

操作码的最高四位与 CPSR 中的条件码进行比较，如果它们彼此不匹配，指令将不被执行，而是以 NOP 指令（无操作）通过流水线。



正因为如此，可能会由于执行一条影响了 CPSR 中条件码的数据处理指令，下一条指令就会或不会被执行。一般的汇编指令，如 MOV 或 ADD，可被预置 16 个条件助记符，它们定义了被测试的条件码状态：

词缀	标志	意义
EQ	Z 位置位	等于
NE	Z 位清零	不等于
CS	C 位置位	无符号大于或等于
CC	C 位清零	无符号小于
MI	N 位置位	负
PL	N 位清零	正或零
VS	V 位置位	溢出
VC	V 位清零	未溢出
HI	C 位置位且 Z 位清零	无符号大于
LS	C 位清零且 Z 位置位	无符号小于或等于
GE	N 等于 V	大于或等于
LT	N 不等于 V	小于
GT	Z 位清零且（N 等于 V）	大于
LE	Z 位置位或（N 不等于 V）	小于或等于
AL	（忽略）	总是

例如：

```
EQMOV    R1，  #0x00800000
```

只有当上一条数据处理指令最后的结果为相等从而置位了 CPSR 中的 Z 标志时，才会将 0x00800000 装载到 R1。指令条件执行的目的，是保证指令可以平滑的通过流水线。每当遇到一个分支或跳转时，流水线都会被清空并重新填入，这将影响流水线的效率。实际操作中，在强制 NOP 指令进入流水线和传统的条件转移和重填流水线之间，有一个开销的平衡点，这一平衡点为三条指令，所以一个小的分支，比如：

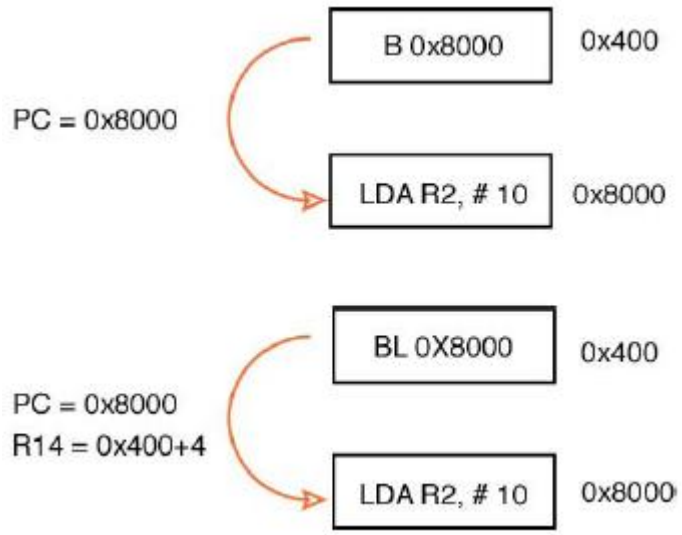
```
if ( x < 100 )
{
   x++;
}
```

如果使用 ARM 指令集的条件执行将获得最高的效率。

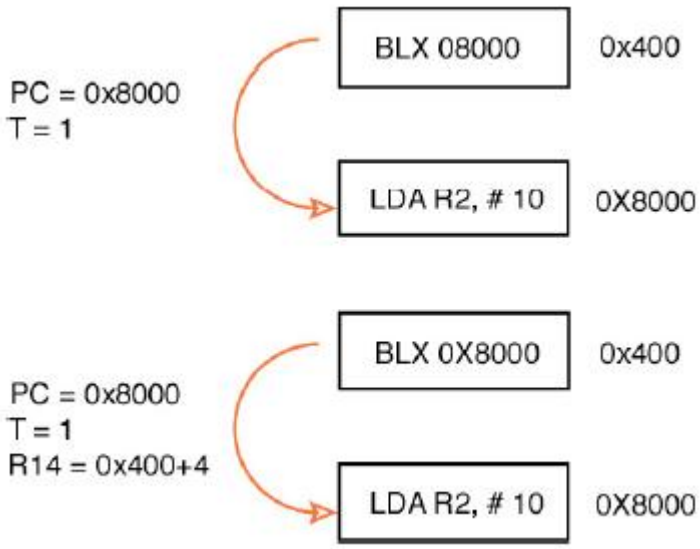
ARM 指令集的主要指令组可以分为五类：分支、数据处理、数据传输、软件中断和乘法。

1.4.1 分支（Branching）

基本的分支指令允许在 32MB 空间范围内向前或向后跳转。分支指令的一个修改版本——分支*链接*——在进行相同的跳转同时，还将当前 PC 地址加 4 字节存入链接寄存器。



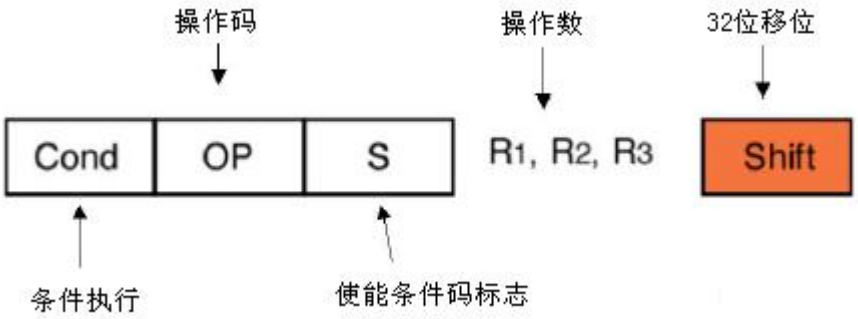
所以，*分支链接指*令用于调用一个将其返回地址存入链接寄存器的函数，并且这一分支指令可用于根据链接寄存器的内容在函数结束时正确返回。通过使用条件码我们可以执行传统的条件分支和条件函数调用。分支指令还有另外两个选择，“*分支交换*”和“*分支链接交换*”。这两个指令执行与上述指令相同分支操作，并完成从 **ARM** 指令集切换到 **THUMB** 指令集，反之亦然。



这是唯一可以使用的切换指令集的方法，因为直接操作 **CPSR** 中的 **T** 位将会导致不可预测的结果。

1.4.2 数据处理

所有数据处理指令的通用形式如下图所示。每个指令有一个结果寄存器和两个操作数。第一个操作数必须是寄存器，但第二个操作数可以是寄存器或立即数。



另外，**ARM7** 内核还有一个移位器，它允许第二个操作数在同一个指令周期内进行最多达 **32** 位的移位操作。**S** 位用于控制条件码。如果该位被置位，条件码将根据指令结果被修改。如果 **S** 位被清除，条件码将不会进行更新。但是，如果指定 **PC**（**R15**）为结果寄存器且 **S** 位置位，将导致当前模式的 **SPSR** 被复制 **CPSR**。这用于在异常处理结束时恢复 **PC** 并切换回原来的模式。当处于用户模式时不要尝试这样做，因为用户模式下没有 **SPSR**，所以结果将不可预测。

助记符	含意
AND	逻辑位与
EOR	逻辑异或
SUB	减
RSB	反减
ADD	加
ADC	带进位加
SBC	带进位减
RSC	带进位反减
TST	测试
TEQ	测试相等
CMP	比较
CMN	否定比较
ORR	逻辑位或
MOV	移动
BIC	位清除
MVN	否定移动

这些特点带来了丰富的数据处理功能，这些指令可用于建立一个非常高效的程序，但同时也为编译器设计者制造了一场“噩梦”——编译器复杂化。**ARM** 高效指令的一个典型的例子如下：

```
if ( Z == 1 ) R1 = R2 + ( R3 * 4)
```

可以被编译为：**EQADDS R1, R2, R3, LSL #2**　一条指令！

1.4.3 数据传输

一、单寄存器装载/存储

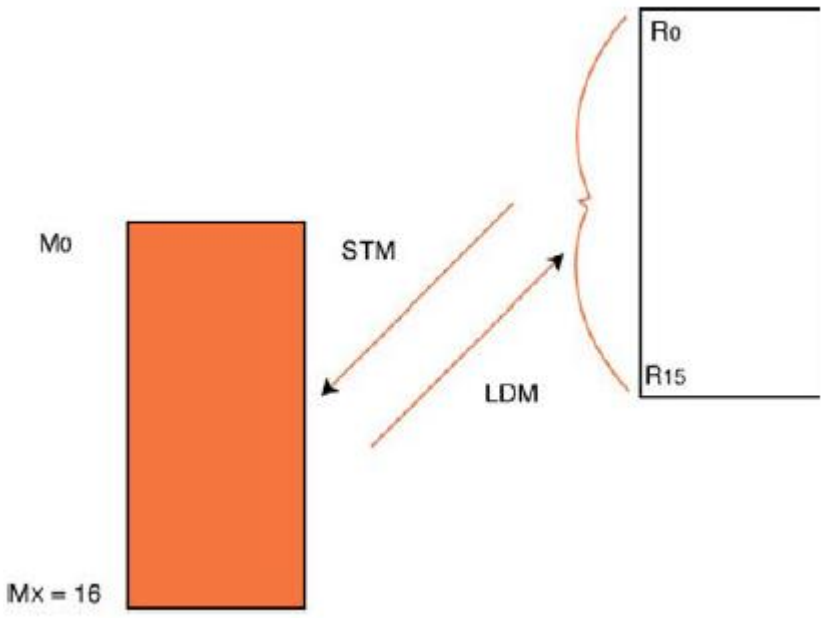
下面一组指令为单寄存器数据传输指令。**ARM7 CPU** 有可以在存储器和寄存器间传送有符号和无符号字、半字和字节的 **Load-Store** 指令。

助记符	含意
LDR	载入字
LDRH	载入半字
LDRSH	载入有符号半字
LDRB	载入字节
LRDSB	载入有符号字节
STR	存储字
STRH	存储半字
STRSH	存储有符号半字
STRB	存储字节
STRSB	存储有符号半字

由于寄存器集是完全正交的，可以将一个 **32** 位的值装入 **PC**，强制程序跳转到处理器地址空间的任何位置。如果目标地址超出跳转指令的范围，可以将一个存储的常量装载到 **PC**。

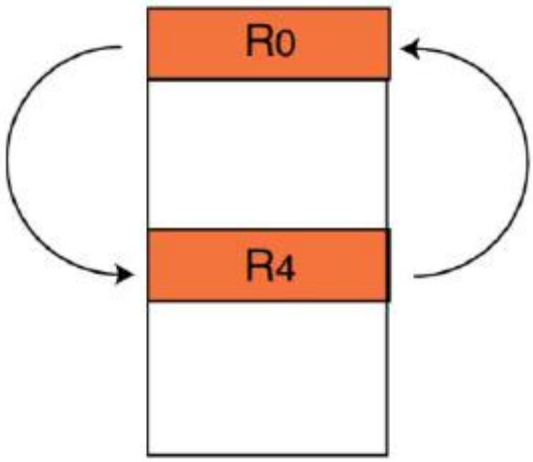
二、多寄存器装载/存储

除了装载和存储单个寄存器的值，**ARM** 还有装载和存储多个寄存器的指令。只用一条指令，整个寄存器组或者一个被选择的子集就可以被复制到存储器。



三、交换指令

ARM 指令集通过*交换指*令提供对 **OS** 实时信号量的支持。交换指令将把在寄存器间或寄存器和存储器间交换一个字作为一个原子操作，这可以防止重要的数据交换被异常中断。

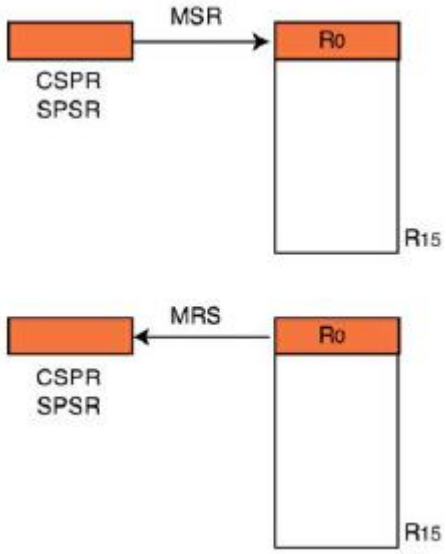


交换指令允许交换两个寄存器的内容。这需要两个周期但却被当作一个原子指令对待，所以交换不可以被中断打断。

C 语言无法使用这一指令，编译器的库中有内部函数支持这一指令。

四、修改状态寄存器

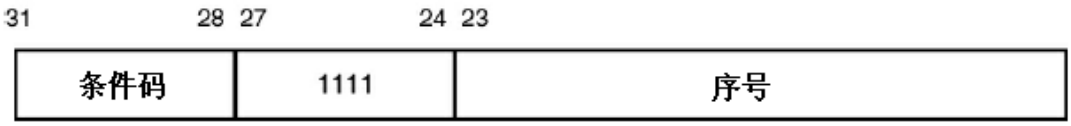
正如在 **ARM7** 体系结构一节中所讲述的，**CPSR** 和 **SPSR** 是 **CPU** 寄存器，但不是主寄存器组的一部分。只有两条 **ARM** 指令可以直接操作这些寄存器。**MSR** 和 **MRS** 指令支持将 **CPSR** 或 **SPSR** 中的内容保存到一个选定的寄存器中，或将该寄存器的值赋给 **CPSR** 或 **SPSR**。例如，为了禁止 **IRQ** 中断，**CPSR** 的内容必须被保存到一个寄存器，“**I**”位必须通过将其内容和 **0x00000080** 相‘与’来置位，以禁止中断，然后 **CPSR** 必须被一个新的值所编程。



MSR 和 **MRS** 指令可以在除用户模式外的所有处理器模式执行，所以，只可能在特权模式下改变处理的操作模式，或者使能/禁止中断。一旦进入用户模式，除非通过一个异常、复位、**FIQ**、**IRQ** 或者 **SWI** 指令，否则不能离开用户模式。

1.4.4 软件中断

软件中断指令（**SWI**）在执行时产生一个异常，迫使处理器进入监控模式并使 **PC** 跳转到 **0x00000008**。如同其他的 **ARM** 指令，**SWI** 指令的最高四位包括条件执行代码，接下来是操作码，指令的剩余部分为空。但可以通过对这些不使用的位进行编码，在进入软件中断后，软件中断程序可以检查这些位，然后决定执行哪些代码。所以，可以使用 **SWI** 指令进入受保护的模式，来运行特权代码或者进行操作系统调用。



例如，汇编指令：

```
SWI     #3
```

将值 **3** 编码到 **SWI** 指令中不用的位中。在 **SWI** 的 **ISR** 例程中，我们可以用下面的伪代码检查 **SWI** 指令：

```
switch ( *(R14 - 4 ) & 0x00FFFFFF )     //将链接寄存器中存储的地址回滚 4 字节
{
   //对于结果
   case ( SWI - 1 ):
                        … …
```

根据不同编译器，用户可能需要自己实现这一功能，也可能由编译器实现。

1.4.5 乘累加（MAC）单元

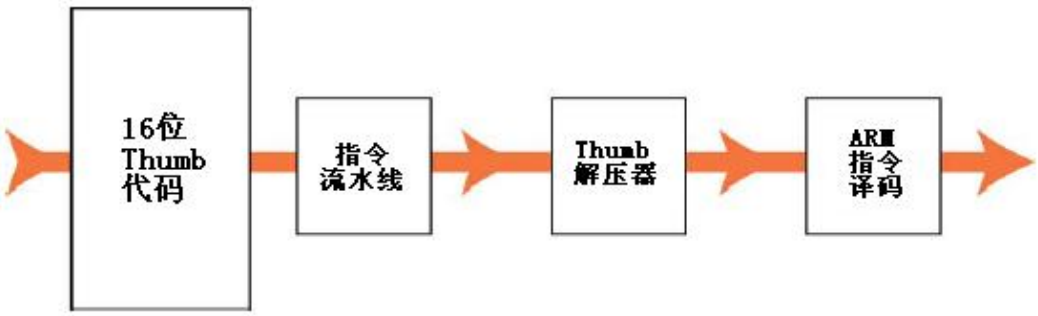
除了快速移位器，**ARM7** 还有一个集成的硬件*乘-累加器*（***MAC***）。**MAC** 支持整数和长整数乘法。整数乘法指令支持两个 **32** 位整数的乘法并将结果保存在一个 **32** 位的寄存器（模 **32**）中。一个*乘累加指*令得到相同乘积后，再将其加到一个累加和上。长整数乘法允许两个 **32** 位的数相乘，并将 **64** 位的结果存到两个寄存器中。**ARM** 核通常也提供*长乘累加*操作。

助记符	意义	结果
MUL	乘	**32** 位结果

MULA	乘累加	32 位结果
UMULL	无符号乘	64 位结果
UMLAL	无符号乘累加	64 位结果
SMULL	有符号乘	64 位结果
SMLAL	有符号乘累加	64 位结果

1.5 THUMB 指令集

虽然 ARM7 是 32 位的处理器，但它也有 16 位的指令集，叫做 ***THUMB***。THUMB 指令集是 ARM 指令集的一个压缩精简版本。它主要是针对 C 编程和 16 位存储器系统的。



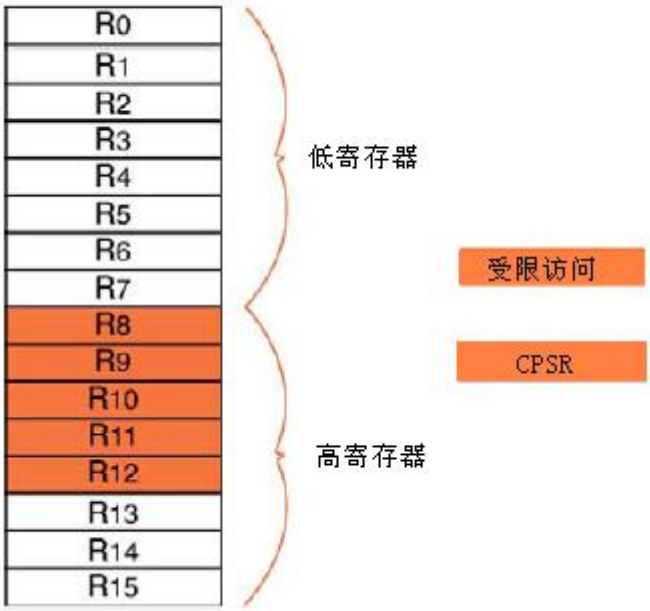
Thumb 指令集对于提高代码密度以适用单片 ARM7 微控制器来说非常重要。

Thumb 允许指令以 16 位的格式存储，到内核后扩展到 ARM 指令然后再执行。虽然相对于 32 位的 ARM 指令来讲，THUMB 指令会导致系统性能下降，但它会取得一个相对很高的代码密度（存储空间小）。所以，为了建立一个适合于小的单片微控制器的应用，将代码编译为 ARM 和 THUMB 函数的混合体是非常重要的，这种处理被叫做 ARM-Thumb *交互工作*（***interworking***）。所有 ARM 编译器均支持这种工作方式。使用 THUMB 指令集编译代码，大约可以节约 30%的代码空间，但如果同样的代码编译为 ARM 指令集，运行速度将快 40%（在 32 位存储器系统中）。

THUMB 指令集更像一个传统的微控制器指令集。与 ARM 指令不同，除了条件分支指令，THUMB 指令不是条件执行的。数据处理指令为两地址格式，目标寄存器即是源寄存器之一。例如：

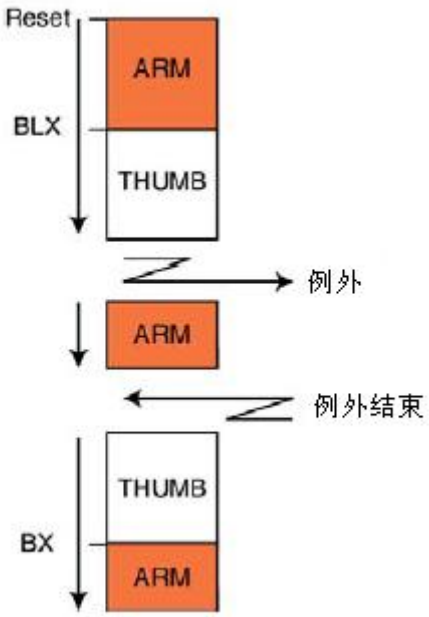
```
ARM 指令                THUMB 指令

ADD  R0, R0, R1         ADD R0, R1      ; R0 = R0 + R1
```

THUMB 指令集不能访问所有的寄存器。所有数据处理指令可访问 R0 – R7（这些被叫做‘低端寄存器’。）

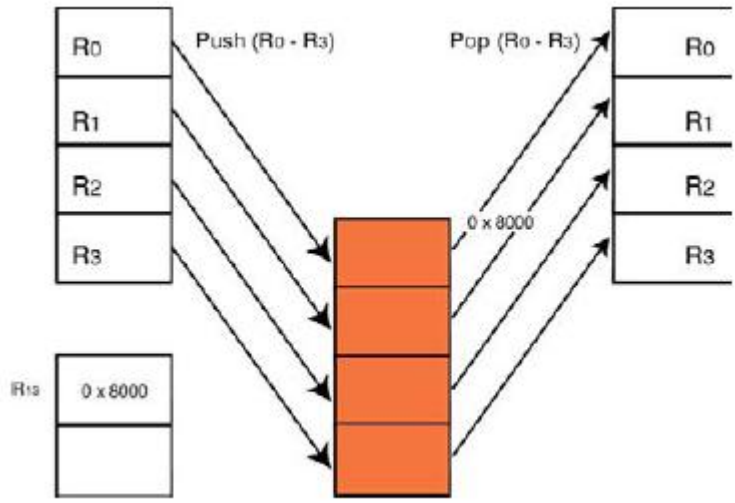


只有少数 **Thumb** 指令可以访问 **R8** – **R12**（‘高端寄存器’），它们是：**MOV**、**ADD** 和 **CMP**。

THUMB 指令集不包括 **MSR** 和 **MRS** 指令，所以在 **Thumb** 模式下不能直接修改 **CPSR** 和 **SPSR**。如果需要修改 **CPSR** 中的用户位，必须切换到 **ARM** 模式。可以通过使用 **BX** 和 **BLX** 指令来切换模式。另外，**CPU** 复位后，或者进入一个异常模式，会自动转换到 **ARM** 模式。



对于堆栈操作，**THUMB** 指令集有更加传统的堆栈指令 **PUSH** 和 **POP**。注意在指令中没有出现堆栈指针，这是因为在 Thumb 操作中，寄存器 r13 是固定作为堆栈指针用的，sp 是自动更新的。可操作的寄存器仅限于‘低端寄存器’r0~r7。PUSH 指令可以操作的寄存器还包括链接寄存器 lr，同样 POP 指令可以操作 pc。这为子程序的进入和退出提供了支持。堆栈指令仅支持*递减式满堆栈*操作。



THUMB 指令集也包括 **SWI** 指令，它的工作方式与在 **ARM** 指令集中类似，但是只包括 **8** 位未使用位，能给出最多 **255** 个 **SWI** 调用。

1.6 小结

至此，你应该对 **ARM7 CPU** 有了一个基本的了解，这也是使用 **ARM7** 必须的基础知识。若要了解更多的 **ARM7** 细节，可以参考其他书籍，包括 **ARM7** 用户手册（**ARM Reference Manual**），以及《**ARM** 嵌入式系统开发——软件设计与优化》，详见www.arm.com 。

第二章 软件开发

Dhrystone V2.1

Parameter	Compiler			
	Keil CA BETA	GNU V3.22	ARM ADS V1.2	IAR V4.11A
Execution Speed (µSeconds)	25.4	112.9	16.8	24.4
Dhrystones/sec	39,370.1	8,857.4	59,382.4	40,983.6
Total Code Size (bytes)	10,330	36,004	22,266	19,892
Stack Size (bytes)	208	852	608	356
Total Data Size (bytes)	10,256	11,912	10,256	10,269

All tests were performed under identical conditions using the Keil µVision Simulator.
The ARM device used was a Philips LPC2294 running at 60MHz in Thumb Mode.

Whetstone

Parameter	Compiler			
	Keil CA BETA	GNU V3.22	ARM ADS V1.2	IAR V4.11A
Execution Speed (seconds)	0.195308	2.461430	0.268623	0.635633
Whetstone (KWIPS)	5,263	406	3,846	1,587
Total Code Size (bytes)	8,492	44,428	28,516	16,316
Stack Size (bytes)	212	1,552	710	488
Total Data Size (bytes)	72	1,768	76	72

All tests were performed under identical conditions using the Keil µVision Simulator.
The ARM device used was a Philips LPC2294 running at 60MHz in Thumb Mode.