《嵌入式系统技术与设计》

作者：华清远见

第 2 章 ARM 体系结构与指令集

本章目标

本章将要介绍 ARM 体系结构、ARM 处理器的工作模式及常用指令集等。通过本章的学习，希望读者能够了解 ARM 处理器内部的主要工作单元、基本工作原理，掌握常用指令集，并为以后的程序设计打下基础。本章主要内容：

- ARM 体系结构的特点
- 寄存器组织
- ARM 存储
- ARM 处理器的寻址方式
- ARM 处理器的工作模式
- 流水线
- 异常
- ARM 处理器的指令集

2.1 ARM 体系结构的特点

ARM 内核采用 RISC 体系结构。RISC 技术的主要特点参见 1.3 节。

ARM 体系结构的主要特征如下：（在本书后面的章节中将对这些特征做详细讲解）

- 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 ARM 处理器工作模式

ARM 处理器共有 7 种工作模式，如表 2-2 所示。

表 2-2　　ARM 处理器的工作模式

处理器工作模式	简　写	描　述
用户模式（User）	usr	正常程序执行模式，大部分任务执行在这种模式下
快速中断模式（FIQ）	fiq	当一个高优先级（fast）中断产生时将会进入这种模式，一般用于高速数据传输和通道处理
外部中断模式（IRQ）	irq	当一个低优先级（normal）中断产生时将会进入这种模式，一般用于通常的中断处理
特权模式（Supervisor）	svc	当复位或软中断指令执行时进入这种模式，是一种供操作系统使用的保护模式
数据访问中止模式（Abort）	abt	当存取异常时将会进入这种模式，用于虚拟存储或存储保护
未定义指令中止模式（Undef）	und	当执行未定义指令时进入这种模式，有时用于通过软件仿真协处理器硬件的工作方式
系统模式（System）	sys	使用和 User 模式相同寄存器集的模式，用于运行特权级操作系统任务

除用户模式外的其他 6 种处理器模式称为特权模式（Privileged Modes）。在这些模式下，程序可以访问所有的系统资源，也可以任意地进行处理器模式切换。其中以下 5 种又称为异常模式：

（1）快速中断模式（FIQ）；

（2）外部中断模式（IRQ）；

（3）特权模式（Supervior）；

（4）数据访问中止模式（Abort）；

（5）未定义指令中止模式（Undef）。

处理器模式可以通过软件控制进行切换，也可以通过外部中断或异常处理过程进行切换。

大多数的用户程序运行在用户模式下。当处理器工作在用户模式时，应用程序不能够访问受操作系统保护的一些系统资源，应用程序也不能直接进行处理器模式切换。当需要进行处理器模式切换时，应用程

序可以产生异常处理，在异常处理过程中进行处理器模式切换。这种体系结构可以使操作系统控制整个系统资源的使用。

当应用程序发生异常中断时，处理器进入相应的异常模式。在每一种异常模式中都有一组专用寄存器以供相应的异常处理程序使用，这样就可以保证在进入异常模式时用户模式下的寄存器（保存程序运行状态）不被破坏。

2.3 寄存器组织

ARM 处理器有如下 37 个 32 位长的寄存器：

（1）30 个通用寄存器；

（2）6 个状态寄存器：1 个 CPSR（Current Program Status Register，当前程序状态寄存器），5 个 SPSR（Saved Program Status Register，备份状态寄存器）；

（3）1 个 PC（Program Counter，程序计数器）。

ARM 处理器共有 7 种不同的处理器模式，在每一种处理器模式中有一组相应的寄存器组。表 2-3 列出了 ARM 处理器的寄存器组织概要。

表 2-3 寄存器组织概要

User	FIQ	IRQ	SVC	Undef	Abort
R0	User mode R0～R7,R15 和 CPSR	User mode R0～R12,R15 和 CPSR	User mode R0～R12,R15 和 CPSR	User mode R0～R12,R15 和 CPSR	User mode R0～R12,R15 和 CPSR
R1					
R2					
R3					
R4					
R5					
R6					
R7					
R8	R8				
R9	R9				
R10	R10				
R11	R11				
R12	R12				
R13(SP)	R13(SP)	R13	R13	R13	R13
R14(LR)	R14(LR)	R14	R14	R14	R14
R15(PC)					
CPSR					
	SPSR	SPSR	SPSR	SPSR	SPSR

当前处理器的模式决定着哪组寄存器可操作，任何模式都可以存取下列寄存器：

（1）相应的 R0～R12；

（2）相应的 R13（the Stack Pointer，SP，栈指向）和 R14（the Link Register，LR，链路寄存器）；

（3）相应的 R15（PC）；

（4）相应的 CPSR。

特权模式（除 System 模式）还可以存取相应的 SPSR。

2.3.1 通用寄存器

通用寄存器根据其分组与否可分为以下 2 类。

- 未分组寄存器（the Unbanked Register），包括 R0～R7。
- 分组寄存器（the Banked Register），包括 R8～R14。

➢ 未分组寄存器

未分组寄存器包括 R0～R7。顾名思义，在所有处理器模式下对于每一个未分组寄存器来说，指的都是同一个物理寄存器。未分组寄存器没有被系统用于特殊的用途，任何可采用通用寄存器的应用场合都可以使用未分组寄存器。但由于其通用性，在异常中断所引起的处理器模式切换时，其使用的是相同的物理寄存器，所以也就很容易使寄存器中的数据被破坏。

➢ 分组寄存器

R8～R14 是分组寄存器，它们每一个访问的物理寄存器取决于当前的处理器模式。

对于分组寄存器 R8～R12 来说，每个寄存器对应两个不同的物理寄存器。一组用于除 FIQ 模式外的所有处理器模式，而另一组则专门用于 FIQ 模式。这样的结构设计有利于加快 FIQ 的处理速度。不同模式下寄存器的使用，要使用寄存器名后缀加以区分。例如，当使用 FIQ 模式下的寄存器时，寄存器 R8 和寄存器 R9 分别记为 R8_fiq、R9_fiq；当使用用户模式下的寄存器时，寄存器 R8 和 R9 分别记为 R8_usr、R9_usr 等。在 ARM 体系结构中，R8～R12 没有任何指定的其他的用途，所以当 FIQ 中断到达时，不用保存这些通用寄存器，也就是说，FIQ 处理程序可以不必执行保存和恢复中断现场的指令，从而可以使中断处理过程非常迅速。所以 FIQ 模式常被用来处理一些时间紧急的任务，如 DMA 处理。

对于分组寄存器 R13 和 R14 来说，每个寄存器对应 6 个不同的物理寄存器。其中的一个是用户模式和系统模式公用的，而另外 5 个分别用于 5 种异常模式。访问时需要指定它们的模式。名字形式如下：

（1）R13_<mode>

（2）R14_<mode>

其中，<mode>可以是以下几种模式之一：usr、svc、abt、und、irp 及 fiq。

R13 寄存器在 ARM 处理器中常用作堆栈指针，称为 SP。当然，这只是一种习惯用法，并没有任何指令强制性的使用 R13 作为堆栈指针，用户完全可以使用其他寄存器作为堆栈指针。而在 Thumb 指令集中，有一些指令强制性的将 R13 作为堆栈指针，如堆栈操作指令。

每一种异常模式拥有自己的 R13。异常处理程序负责初始化自己的 R13，使其指向该异常模式专用的栈地址。在异常处理程序入口处，将用到的其他寄存器的值保存在堆栈中，返回时，重新将这些值加载到寄存器。通过这种保护程序现场的方法，异常不会破坏被其中断的程序现场。

寄存器 R14 又被称为连接寄存器（Link Register，LR），在 ARM 体系结构中具有下面两种特殊的作用。

（1）每一种处理器模式用自己的 R14 存放当前子程序的返回地址。当通过 BL 或 BLX 指令调用子程序时，R14 被设置成该子程序的返回地址。在子程序返回时，把 R14 的值复制到程序计数器（PC）。典型的做法是使用下列两种方法之一。

① 执行下面任何一条指令。

```
MOV  PC, LR
BX  LR
```

② 在子程序入口处使用下面的指令将 PC 保存到堆栈中。

```
STMFD  SP!, {<register>,LR}
```

在子程序返回时，使用如下相应的配套指令返回。

```
LDMFD  SP!, {<register>,PC}
```

（2）当异常中断发生时，该异常模式特定的物理寄存器 R14 被设置成该异常模式的返回地址，对于有些模式 R14 的值可能与返回地址有一个常数的偏移量（如数据异常使用 SUB PC，LR，#8 返回）。具体的返回方式与上面的子程序返回方式基本相同，但使用的指令稍微有些不同，以保证当异常出现时正在执行的程序的状态被完整保存。

R14 也可以被用作通用寄存器使用。

2.3.2 状态寄存器

当前程序状态寄存器（Current Program Status Register，CPSR）可以在任何处理器模式下被访问，它包含下列内容：

- ALU（Arithmetic Logic Unit，算术逻辑单元）状态标志的备份；
- 当前的处理器模式；
- 中断使能标志；
- 设置处理器的状态（只在 4T 架构）。

每一种处理器模式下都有一个专用的物理寄存器作备份程序状态寄存器（Saved Program Status Register，SPSR）。当特定的异常中断发生时，这个物理寄存器负责存放当前程序状态寄存器的内容。当异常处理程序返回时，再将其内容恢复到当前程序状态寄存器。

CPSR 寄存器（和保存它的 SPSR 寄存器）中的位分配如图 2-1 所示。

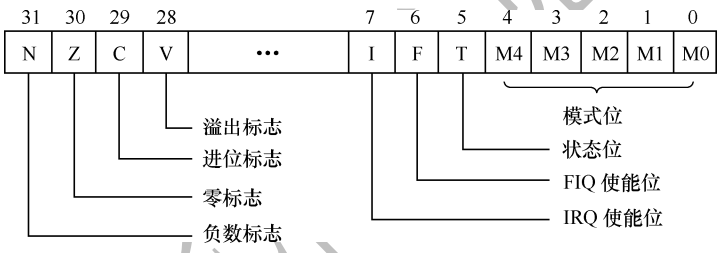

图 2-1 程序状态寄存器格式

➢ 标志位。

N（Negative）、Z（Zero）、C（Carry）和 V（oVerflow）通称为条件标志位。这些条件标志位会根据程序中的算术指令或逻辑指令的执行结果进行修改，而且这些条件标志位可由大多数指令检测以决定指令是否执行。

在 ARM 4T 架构中，所有的 ARM 指令都可以条件执行，而 Thumb 指令却不能。

各条件标志位的具体含义如下。

- N

本位设置成当前指令运行结果的 bit[31]的值。当两个由补码表示的有符号整数运算时，N = 1 表示运算的结果为负数；N = 0 表示结果为正数或零。

- Z

Z = 1 表示运算的结果为零，Z = 0 表示运算的结果不为零。

- C

下面分 4 种情况讨论 C 的设置方法。

① 在加法指令中（包括比较指令 CMN），当结果产生了进位，则 C = 1，表示无符号数运算发生上溢出；其他情况下 C = 0。

② 在减法指令中（包括比较指令 CMP），当运算中发生错位（即无符号数运算发生下溢出），则 C = 0；其他情况下 C = 1。

③ 对于在操作数中包含移位操作的运算指令（非加/减法指令），C 被设置成被移位寄存器最后移出去的位。

④ 对于其他非加/减法运算指令，C 的值通常不受影响。

- V

下面分两种情况讨论 V 的设置方法。

① 对于加/减运算指令，当操作数和运算结果都是以二进制的补码表示的带符号的数时，V = 1 表示符号位溢出。

② 对于非加/减法指令，通常不改变标志位 V 的值（具体可参照 ARM 指令手册）。

尽管以上 C 和 V 的定义看起来颇为复杂，但使用时在大多数情况下用一个简单的条件测试指令即可，不需要程序员计算出条件码的精确值即可得到需要的结果。

目的寄存器是 R15 的带“位设置”的算术和逻辑运算指令，也可以将 SPSR 的值复制到 CPSR 中，这种操作主要用于从异常中断程序中返回。

用 MSR 指令向 CPSR/SPSR 写进新值。

目的寄存器位 R15 的 MRC 协处理器指令通过这条指令可以将协处理器产生的条件标志位的值传送到 ARM 处理器。

在中断返回时，使用 LDR 指令的变种指令可以将 SPSR 的值复制到 CPSR 中。

➢ Q 标志位

在带 DSP 指令扩展的 ARM v5 及更高版本中，bit[27]被指定用于指示增强的 DAP 指令是否发生了溢出，因此也就被称为 Q 标志位。同样，在 SPSR 中 bit[27]也被称为 Q 标志位，用于在异常中断发生时保存和恢复 CPSR 中的 Q 标志位。

在 ARM v5 以前的版本及 ARM v5 的非 E 系列处理器中，Q 标志位没有被定义。属于待扩展的位。

➢ 控制位

CPSR 的低 8 位（I、F、T 及 M[4：0]）统称为控制位。当异常发生时，这些位的值将发生相应的变化。另外，如果在特权模式下，也可以通过软件编程来修改这些位的值。

① 中断禁止位

I = 1，IRQ 被禁止。

F = 1，FIQ 被禁止。

② 状态控制位

T 位是处理器的状态控制位。

T = 0，处理器处于 ARM 状态（即正在执行 32 位的 ARM 指令）。

T = 1，处理器处于 Thumb 状态（即正在执行 16 位的 Thumb 指令）。

当然，T 位只有在 T 系列的 ARM 处理器上才有效，在非 T 系列的 ARM 版本中，T 位将始终为 0。

③ 模式控制位

M[4：0]作为位模式控制位，这些位的组合确定了处理器处于哪种状态。表 2-4 列出了其具体含义。只有表中列出的组合是有效的，其他组合无效。

表 2-4　　状态控制位 M[4：0]

M[4：0]	处理器模式	可以访问的寄存器
0b10000	User	PC，R14～R0，CPSR
0b10001	FIQ	PC，R14_fiq～R8_fiq，R7～R0，CPSR，SPSR_fiq
0b10010	IRQ	PC，R14_irq～R13_irq，R12～R0，CPSR，SPSR_irq

0b10011	Supervisor	PC，R14_svc～R13_svc，R12～R0，CPSR，SPSR_svc
0b10111	Abort	PC，R14_abt～R13_abt，R12～R0，CPSR，SPSR_abt
0b11011	Undefined	PC，R14_und～R13_und，R12～R0，CPSR，SPSR_und
0b11111	System	PC，R14～R0，CPSR（ARM v4 及更高版本）

2.3.3 程序计数器

程序计算器 R15 又被记为 PC。它有时可以和 R0～R14 一样用作通用寄存器，但很多特殊的指令在使用 R15 时有些限制。当违反了这些指令的使用限制时，指令的执行结果是不可预知的。

程序计数器在下面两种情况下用于特殊的目的。

- 读程序计数器。
- 写程序计数器。

2.4 流 水 线

2.4.1 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（decode）；

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，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）。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。

2.4.2 流水线的分类

➢ 3 级流水线 ARM 组织

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

- 取指令（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（excute）：处理指令并将结果写回寄存器。

图 2-2 所示为 3 级流水线指令的执行过程。

图 2-2　3 级流水线

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 1 条指令。

➢ 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。有两种方法来提高性能。

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系统性能，效果并不明显。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。

在 ARM9TDMI 中使用了典型的 5 级流水线，5 级流水线包括下面的流水线级。

- 取指令（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 1 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 1 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 2-3 列出了 5 级流水线指令的执行过程。

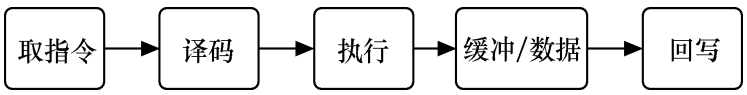

图 2-3　5 级流水线

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。

➢ 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 1.3DMIPS/MHz。CPU 性能评估采用合成测试程序，较流行的有 Whetstone 和 Dhrystone 两种。Dhrystone 主要用于测整数计算能力，计算单位就是 DMIPS。DMIPS 可以理解为处理器单位时间内执行处理整数的指令的百万次数。而因为处理器的性能与工作频率密切相关，在不同工作频率下测算出的 DMIPS 是不同的，所以通常使用 DMIPS/MHz 作为标准，评估各个处理器的结构优劣和性能高低。图 2-4 所示为 6 级流水线上指令的执行过程。

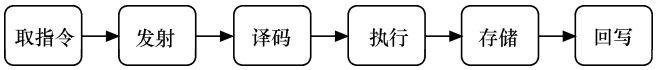

图 2-4　6 级流水线

2.4.3　影响流水线性能的因素

➢ 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 2-1】所示。

【例 2-1】　互锁指令操作。

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
```

从例子中可以看出，流水线的操作产生中断，因为第 1 条指令的结果在第 2 条指令取数时还没有产生。第 2 条指令必须停止，直到结果产生为止。

➢ 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不被丢弃。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。

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。

2.5　ARM 存储系统

ARM 存储系统有非常灵活的体系结构，可以适应不同的嵌入式应用系统的需要。ARM 存储器系统可以使用简单的平板式地址映射机制（就像一些简单的单片机一样，地址空间的分配方式是固定的，系统中各部分都使用物理地址），也可以使用其他技术提供功能更为强大的存储系统。例如：

- 系统可能提供多种类型的存储器件，如 Flash、ROM、SRAM 等；
- Cache 技术；
- 写缓存技术（write buffers）；

- 虚拟内存和 I/O 地址映射技术。

大多数的系统通过下面的方法之一可实现对复杂存储系统的管理。

- 使用 Cache，缩小处理器和存储系统速度差别，从而提高系统的整体性能。
- 使用内存映射技术实现虚拟空间到物理空间的映射。这种映射机制对嵌入式系统非常重要。通常嵌入式系统程序存放在 ROM/Flash 中，这样系统断电后程序能够得到保存。但是，通常 ROM/Flash 与 SDRAM 相比，速度慢很多，而且基于 ARM 的嵌入式系统中通常把异常中断向量表放在 RAM 中。利用内存映射机制可以满足这种需要。在系统加电时，将 ROM/Flash 映射为地址 0，这样可以进行一些初始化处理；当这些初始化处理完成后将 SDRAM 映射为地址 0，并把系统程序加载到 SDRAM 中运行，这样很好地满足嵌入式系统的需要。
- 引入存储保护机制，增强系统的安全性。
- 引入一些机制保证将 I/O 操作映射成内存操作后，各种 I/O 操作能够得到正确的结果。在简单存储系统中，不存在这样问题。而当系统引入了 Cache 和 write buffer 后，就需要一些特别的措施。

在 ARM 系统中，要实现对存储系统的管理通常是使用协处理器 CP15，它通常也被称为系统控制协处理器（System Control Coprocessor）。

ARM 的存储器系统是由多级构成的，可以分为：内核级、芯片级、板卡级、外设级。图 2-5 所示为存储器的层次结构。

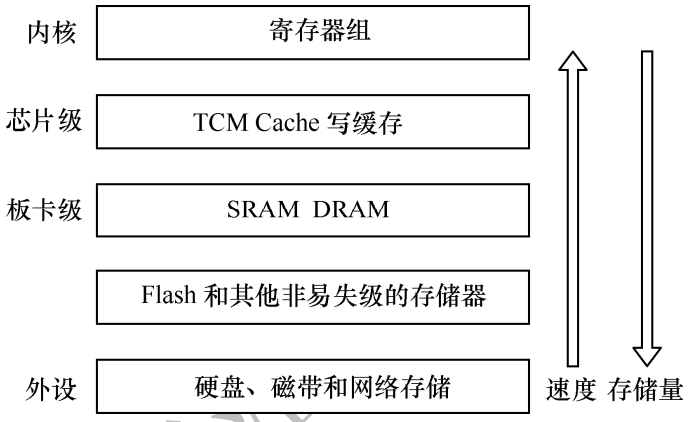

图 2-5 存储器的层次结构

每级都有特定的存储介质，下面对比各级系统中特定存储介质的存储性能。

① 内核级的寄存器。处理器寄存器组可看作是存储器层次的顶层。这些寄存器被集成在处理器内核中，在系统中提供最快的存储器访问。典型的 ARM 处理器有多个 32 位寄存器，其访问时间为 ns 量级。

② 芯片级的紧耦合存储器 TCM。为弥补 Cache 访问的不确定性增加的存储器。TCM 是一种快速 SDRAM，它紧挨内核，并且保证取指和数据操作的时钟周期数，这一点对一些要求确定行为的实时算法是很重要的。TCM 位于存储器地址映射中，可作为快速存储器来访问。

③ 芯片级的片上 Cache 存储器的容量在 8KB～32KB 之间，访问时间大约为 10ns。高性能的 ARM 结构中，可能存在第二级片外 Cache，容量为几百 KB，访问时间为几十 ns。

⑤ 板卡级的 DRAM。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 100ns。

⑥ 外设级的后援存储器，通常是硬盘，可能从几百 MB 到几个 GB，访问时间为几十 ms。

2.5.1 协处理器（CP15）

ARM 处理器支持 16 个协处理器。在程序执行过程中，每个协处理器忽略属于 ARM 处理器和其他协处理器的指令。当一个协处理器硬件不能执行属于它的协处理器指令时，将产生一个未定义指令异常中断，在该异常中断处理程序中，可以通过软件模拟该硬件操作。例如，如果系统不包含向量浮点运算器，则可以选择浮点运算软件模拟包来支持向量浮点运算。CP15，即通常所说的系统控制协处理器（System Control Coprocesssor），它负责完成大部分的存储系统管理。除了 CP15 外，在具体的各种存储管理机制中可能还会用到其他的一些技术，如在 MMU 中除了 CP15 外，还使用了页表技术等。

在一些没有标准存储管理的系统中，CP15 是不存在的。在这种情况下，针对 CP15 的操作指令将被视为未定义指令，指令的执行结果不可预知。

CP15 包含 16 个 32 位寄存器，其编号为 0～15。实际上对于某些编号的寄存器可能对应多个物理寄存器，在指令中指定特定的标志位来区分这些物理寄存器。这种机制有些类似于 ARM 中的寄存器，当处于不同的处理器模式时，某些相同编号的寄存器对应于不同的物理寄存器。

CP15 中的寄存器可能是只读的，也可能是只写的，还有一些是可读可写的。在对协处理器寄存器进行操作时，需要注意以下几个问题：

- 寄存器的访问类型（只读/只写/可读可写）；
- 不同的访问引发的不同功能；
- 相同编号的寄存器是否对应不同的物理寄存器；
- 寄存器的具体作用。

2.5.2 存储管理单元（MMU）

在创建多任务嵌入式系统时，最好有一个简单的方式来编写、装载及运行各自独立的任务。目前大多数的嵌入式系统不再使用自己定制的控制系统，而使用操作系统来简化这个过程。较高级的操作系统采用基于硬件的存储管理单元（MMU）来实现上述操作。

MMU 提供的一个关键服务是使各个任务作为各自独立的程序在其自己的私有存储空间中运行。在带 MMU 的操作系统控制下，运行的任务无须知道其他与之无关的任务的存储需求情况，这就简化了各个任务的设计。

MMU 提供了一些资源以允许使用虚拟存储器（将系统物理存储器重新编址，可将其看成一个独立于系统物理存储器的存储空间）。MMU 作为转换器，将程序和数据的虚拟地址（编译时的连接地址）转换成实际的物理地址，即在物理主存中的地址。这个转换过程允许运行的多个程序使用相同的虚拟地址，而各自存储在物理存储器的不同位置。

这样存储器就有两种类型的地址：虚拟地址和物理地址。虚拟地址由编译器和连接器在定位程序时分配；物理地址用来访问实际的主存硬件模块（物理上程序存在的区域）。

2.5.3 高速缓冲存储器（Cache）

Cache 是一个容量小但存取速度非常快的存储器，它保存最近用到的存储器数据拷贝。对于程序员来说，Cache 是透明的。它自动决定保存哪些数据、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特性。所谓局部性就是指在任何特定的时间，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）。

Cache 经常与写缓存器（write buffer）一起使用。写缓存器是一个非常小的先进先出（FIFO）存储器，位于处理器核与主存之间。使用写缓存的目的是，将处理器核和 Cache 从较慢的主存写操作中解脱出来。当 CPU 向主存储器做写入操作时，它先将数据写入到写缓存区中，由于写缓存器的速度很高，这种写入操作的速度也将很高。写缓存区在 CPU 空闲时，以较低的速度将数据写入到主存储器中相应的位置。

通过引入 Cache 和写缓存区，存储系统的性能得到了很大的提高，但同时也带来了一些问题。例如，由于数据将存在于系统中的不同的物理位置，可能造成数据的不一致性；由于写缓存区的优化作用，可能有些写操作的执行顺序不是用户期望的顺序，从而造成操作错误。

2.6 异常

异常或中断是用户程序中最基本的一种执行流程和形态。本节主要对 ARM 架构下的异常中断做详细说明。

2.6.1 异常的种类

ARM 体系结构中，存在 7 种异常处理。当异常发生时，处理器会把 PC 设置为一个特定的存储器地址。这一地址放在被称为向量表（vector table）的特定地址范围内。向量表的入口是一些跳转指令，跳转到专门处理某个异常或中断的子程序。

存储器映射地址 0x00000000 是为向量表（一组 32 位字）保留的。在有些处理器中，向量表可以选择定位在存储空间的高地址（从偏移量 0xffff0000 开始）。一些嵌入式操作系统，如 Linux 和 Windows CE 就利用了这一特性。

表 2-5 列出了 ARM 的 7 种异常。

表 2-5 ARM 的 7 种异常

异 常 类 型	处理器模式	执行低地址	执行高地址
复位异常（Reset）	特权模式	0x00000000	0xFFFF0000
未定义指令异常（Undefined interrupt）	未定义指令中止模式	0x00000004	0xFFFF0004
软中断异常（Software Abort）	特权模式	0x00000008	0xFFFF0008
预取异常（Prefetch Abort）	数据访问中止模式	0x0000000C	0xFFFF000C
数据异常（Data Abort）	数据访问中止模式	0x00000010	0xFFFF0010
外部中断请求（IRQ）	外部中断请求模式	0x00000018	0xFFFF0018
快速中断请求（FIQ）	快速中断请求模式	0x0000001C	0xFFFF001C

异常返回时，SPSR 内容恢复到 CPSR，连接寄存器 R14 的内容恢复到程序计数器（PC）。

2.6.2 异常的优先级

每一种异常按表 2-6 中设置的优先级得到处理。

表 2-6 异常优先级

优 先 级		异 常
最高	1	复位异常
	2	数据中止
	3	快速中断请求
	4	外部中断请求
	5	预取指令异常
	6	软中断
最低	7	未定义指令

异常可以同时发生，处理器则按表 2-6 中设置的优先级顺序处理异常。例如，处理器上电时发生复位异常，复位异常的优先级最高，所以当产生复位时，它将优先于其他异常得到处理。同样，当一个数据访问中止异常发生时，它将优先于除复位异常外的其他所有异常而得到处理。

优先级最低的 2 种异常是软件中断和未定义指令异常。因为正在执行的指令不可能既是一条 SWI 指令，又是一条未定义指令，所以软件中断异常和未定义指令异常享有相同的优先级。

2.6.3 构建异常向量表

异常处理向量表如图 2-6 所示。

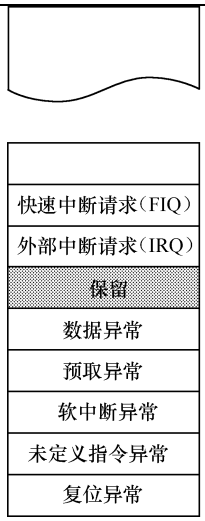

图 2-6　异常处理向量表

异常返回时，SPSR 内容恢复到 CPSR，连接寄存器 R14 的内容恢复到程序计数器（PC）。

➢ 复位异常

当处理器的复位引脚有效时，系统产生复位异常中断，程序跳转到复位异常中断处理程序处执行。复位异常中断通常用在下面两种情况下：

- 系统上电；
- 系统复位。

复位异常中断处理程序将进行一些初始化工作，内容与具体系统相关。下面是复位异常中断处理程序的主要功能。

- 设置异常中断向量表。
- 初始化数据栈和寄存器。
- 初始化存储系统，如系统中的 MMU 等。
- 初始化关键的 I/O 设备。
- 使能中断。
- 处理器切换到合适的模式。
- 初始化 C 变量，跳转到应用程序执行。

➢ 未定义指令异常

当 ARM 处理器执行协处理器指令时，它必须等待一个外部协处理器应答后，才能真正执行这条指令。若协处理器没有响应，则发生未定义指令异常。

未定义指令异常可用于在没有物理协处理器的系统上，对协处理器进行软件仿真，或通过软件仿真实现指令集扩展。例如，在一个不包含浮点运算的系统中，CPU 遇到浮点运算指令时，将发生未定义指令异常中断，在该未定义指令异常中断的处理程序中可以通过其他指令序列仿真浮点运算指令。

仿真功能可以通过下面步骤实现。

① 将仿真程序入口地址链接到向量表中未定义指令异常中断入口处（0x00000004 或 0xffff0004），并保存原来的中断处理程序。

② 读取该未定义指令的 bits[27：24]，判断其是否是一条协处理器指令。如果 bits[27：24]值为 0b1110 或 0b110x，该指令是一条协处理器指令；否则，由软件仿真实现协处理器功能，可以同过 bits[11：8]来判断要仿真的协处理器功能（类似于 SWI 异常实现机制）。

③ 如果不仿真该未定义指令，程序跳转到原来的未定义指令异常中断的中断处理程序执行。

➢ 软中断

软中断（SWI）异常发生时，处理器进入特权模式，执行一些特权模式下的操作系统功能。

➢ 预取指令异常

预取指令异常是由系统存储器报告的。当处理器试图去取一条被标记为预取无效的指令时，发生预取指令异常。

如果系统中不包含 MMU 时，指令预取异常中断处理程序只是简单地报告错误并退出。若包含 MMU，引起异常的指令的物理地址被存储到内存中。

➢ 数据访问中止异常

数据访问中止异常是由存储器发出数据中止信号，它由存储器访问指令 Load/Store 产生。当数据访问指令的目标地址不存在或者该地址不允许当前指令访问时，处理器产生数据访问中止异常。

当数据访问中止异常发生时，寄存器的值将根据以下规则进行修改。

① 返回地址寄存器 R14 的值只与发生数据异常的指令地址有关，与 PC 值无关。

② 如果指令中没有指定基址寄存器回写，则基址寄存器的值不变。

③ 如果指令中指定了基址寄存器回写，则寄存器的值和具体芯片的 Abort Models 有关，由芯片的生产商指定。

④ 如果指令只加载一个通用寄存器的值，则通用寄存器的值不变。

⑤ 如果是批量加载指令，则寄存器中的值是不可预知的值。

⑥ 如果指令加载协处理器寄存器的值，则被加载寄存器的值不可预知。

➢ 外部中断请求

当处理器的外部中断请求（IRQ）引脚有效，而且 CPSR 寄存器的 I 控制位被清除时，处理器产生外部中断 IRQ 异常。系统中各外部设备通常通过该异常中断请求处理器服务。

➢ 快速中断请求

当处理器的快速中断请求（FIQ）引脚有效且 CPSR 寄存器的 F 控制位被清除时，处理器产生快速中断请求 FIQ 异常。

2.6.4 异常响应流程

➢ 判断处理器状态

当异常发生时，处理器自动切换到 ARM 状态，所以在异常处理函数中要判断在异常发生前处理器是 ARM 状态还是 Thumb 状态。这可以通过检测 SPSR 的 T 位来判断。

通常情况下，只有在 SWI 处理函数中才需要知道异常发生前处理器的状态。所以在 Thumb 状态下，调用 SWI 软中断异常必须注意以下两点。

① 发生异常的指令地址为(lr-2)，而不是(lr-4)。

② Thumb 状态下的指令是 16 位的，在判断中断向量信号时使用半字加载指令 LDRH。

下面的例子显示了一个标准的 SWI 处理函数，在函数中通过 SPSR 的 T 位判断异常发生前的处理器状态。

```
T_bit EQU 0x20              ; bit 5. SPSR 中的 ARM/Thumb 状态位
:
SWIHandler
STMFD sp!, {R0-R3,R12,lr}       ; 寄存器压栈，保护程序现场
MRS R0, spsr                ; 读 SPSR 寄存器，判断异常发生前的处理器状态
TST R0, #T_bit              ; 检测 SPSR 的 T 位，判断异常发生前是否为 Thumb 状态
LDRNEH R0,[lr,#-2]          ; 如果是 Thumb 状态，使用半字加载指令读取发生异常的指令地址
BICNE R0,R0,#0xFF00         ; 提取中断向量号
LDREQ R0,[lr,#-4]               ; 如果是 ARM 状态，使用字加载指令，读取发生异常的指令地址
BICEQ R0,R0,#0xFF000000     ; 提取中断向量号并将中断向量号存入 R0
; R0 存储中断向量号
CMP R0, #MaxSWI             ; 判断中断是否超出范围
LDRLS pc, [pc, R0, LSL#2]       ; 如果未超出范围，跳转到软中断向量表 Switable
B SWIOutOfRange             ; 如果超出范围，跳转到软中断越界处理程序
switable
DCD do_swi_1
DCD do_swi_2
:
do_swi_1
; 1 号软中断处理函数
LDMFD sp!, {R0-R3,R12,pc}^  ; Restore the registers and return
                            ; 恢复寄存器并返回
do_swi_2                    ; 2 号软中断处理函数
:
```

➢ 向量表

前面介绍向量表时提到，每一个异常发生时总是从异常向量表开始跳转。最简单的一种情况是向量表里面的每一条指令直接跳向对应的异常处理函数。其中快速中断处理函数 FIQ_handler()可以直接从地址 0x1C 处开始，省下一条跳转指令，如图 2-7 所示。

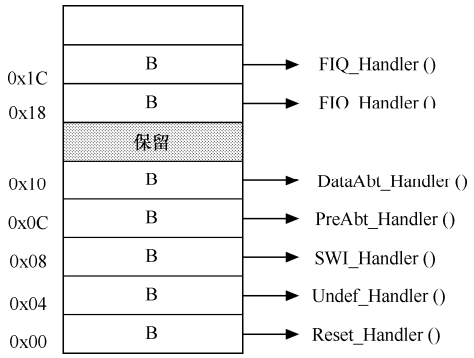

图 2-7 异常处理向量表

但跳转指令 B 的跳转范围为−32～32MB，但很多情况下不能保证所有的异常处理函数都定位在向量的 32MB 范围内，需要更大范围的跳转，而且由于向量表空间的限制，只能由一条指令完成。具体实现方法有下面两种。

（1）MOV PC，＃imme_value

这种方法将目标地址直接赋值给 PC。但这种方法受格式限制不能处理任意立即数。这个立即数由一个 8 位数值循环右移偶数位得到。

（2）LDR PC，[PC+offset]

把目标地址先存储在某一个合适的地址空间，然后把这个存储器单元的 32 位数据传送给 PC 来实现跳转。

这种方法对目标地址值没有要求。但是，存储目标地址的存储器单元必须在当前指令的−4～4KB 空间范围内。

2.6.5 从异常处理程序中返回

当一个异常处理返回时，一共有 3 件事情需要处理：通用寄存器的恢复、状态寄存器的恢复及 PC 指针的恢复。通用寄存器的恢复采用一般的堆栈操作指令即可，下面重点介绍状态寄存器的恢复及 PC 指针的恢复。

➢ 恢复被中断程序的处理器状态

PC 和 CPSR 的恢复可以通过一条指令来实现，下面是 3 个例子。

- MOVS PC，LR
- SUBS PC，LR，＃4
- LDMFD SP!，{PC}^

这几条指令是普通的数据处理指令，特殊之处在于它们把 PC 作为目标寄存器，并且带了特殊的后缀“S”或“^”。其中“S”或“^”的作用就是使指令在执行时同时完成从 SPSR 到 CPSR 的拷贝，达到恢复状态寄存器的目的。

➢ 异常的返回地址

异常返回时，另一个非常重要的问题就是返回地址的确定。前面提到过，处理器进入异常时会有一个保存 LR 的动作，但是该保持值并不一定是正确中断的返回地址。下面以一个简单的指令执行流水状态图（见图 2-8）来对此加以说明。

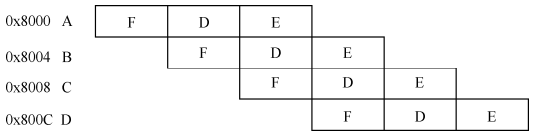

图 2-8 3 级流水线示例

在 ARM 架构中，PC 值指向当前执行指令地址加 8。也就是说，当执行指令 A（地址 0x8000）时，PC 等于 0x8000 + 8 = 0x8008，即等于指令 C 的地址。假设指令 A 是 BL 指令，则当执行时，会把 PC 值（0x8008）保存到 LR 寄存器。但是，接下来处理器会对 LR 进行一次自动调整，使 LR = LR − 0X04。所以，最终保存在 LR 里面的是图 2-7 中所示的 B 指令地址。所以当从 BL 返回时，LR 里面正好是正确的返回地址。

同样的跳转机制在所有的 LR 自动保存操作中都存在。当进入中断响应时，处理器对保存的 LR 也进行一次自动调整，并且跳转动作也是 LR = LR − 0x04。由此，就可以对不同异常类型的返回地址依次比较。

假设在指令 B 处（地址 0x8004）发生了异常，进入异常响应后，LR 经过跳转保存的地址值应该是 C 的地址 0x8008。

（1）软中断异常

如果发生软中断异常，即指令 B 为 SWI 指令，从 SWI 中断返回后下一条执行指令就是 C，正好是 LR 寄存器保存的地址，所以只有直接把 LR 恢复给 PC 即可。

（2）IRQ 或 FIQ 异常

如果发生的是 IRQ 或 FIQ 异常，因为外部中断请求中断了正在执行的指令 B，当中断返回后，需要重新回到 B 指令执行，也就是说，返回地址应该是 B（0x8004），需要把 LR 减 4 送 PC。

（3）Data Abort 数据中止异常

在指令 B 处进入数据异常的响应，但导致数据异常的原因却应该是上一条指令 A。当中断处理程序恢复数据异常后，要回到 A 重新执行导致数据异常的指令，因此返回地址应该是 LR 加 8。

为方便起见，表 2-7 列出了异常和返回地址的关系。

表 2-7　　异常和返回地址

异　常	地　址	用　途
复位	—	复位没有定义 LR
数据中止	LR－8	指向导致数据中止异常的指令
FIQ	LR－4	指向发生异常时正在执行的指令
IRQ	LR－4	指向发生异常时正在执行的指令
预取指令中止	LR－4	指向导致预取指令异常的那条指令
SWI	LR	执行 SWI 指令的下一条指令
未定义指令	LR	指向未定义指令的下一条指令

2.7　ARM 处理器的寻址方式

ARM 指令集可以分为跳转指令、数据处理指令、程序状态寄存器传输指令、Load/Store 指令、协处理器指令和异常中断产生指令。根据使用的指令类型不同，指令的寻址方式分为数据处理指令寻址方式和内存访问指令寻址方式。

2.7.1　数据处理指令寻址方式

数据处理指令的基本语法格式如下：

```
<opcode> {<cond>} {S} <Rd>,<Rn>,<shifter_operand>
```

其中，<shifter_operand>有 11 种形式，如表 2-8 所示。

表 2-8　　<shifter_operand>的寻址方式

	语　法	寻 址 方 式
1	#<immediate>	立即数寻址
2	<Rm>	寄存器寻址
3	<Rm>, LSL　#<shift_imm>	立即数逻辑左移
4	<Rm>, LSL　<Rs>	寄存器逻辑左移
5	<Rm>, LSR　#<shift_imm>	立即数逻辑右移
6	<Rm>, LSR　<Rs>	寄存器逻辑右移
7	<Rm>, ASR　#<shift_imm>	立即数算术右移
8	<Rm>, ASR　<Rs>	寄存器算术右移
9	<Rm>, ROR　#<shift_imm>	立即数循环右移

10	<Rm>, ROR <Rs>	寄存器循环右移
11	<Rm>, RRX	寄存器扩展循环右移

数据处理指令寻址方式可以分为以下几种。

① 立即数寻址方式。

② 寄存器寻址方式。

③ 寄存器移位寻址方式。

1．立即数寻址方式

指令中的立即数是由一个 8bit 的常数移动 4bit 偶数位（0，2，4，…，26，28，30）得到的。所以，每一条指令都包含一个 8bit 的常数 X 和移位值 Y，得到的立即数＝X 循环右移（2×Y）。

下面列举了一些有效的立即数：

0xFF、0x104、0xFF0、0x FF00、0x FF000、0x FF000000、0x F000000F

下面是一些无效的立即数：

0x101、0x102、0x FF1、0x FF04、0x FF003、0x FFFFFFFF、0x F000001F

下面是一些应用立即数的指令：

```
MOV  R0，＃0              ;送 0 到 R0
ADD  R3，R3，＃1          ;R3 的值加 1
CMP  R7，＃1000           ;R7 的值和 1000 比较
BIC  R9，R8，＃0x FF00        ;将 R8 中 8～15 位清零，结果保存在 R9 中
```

2．寄存器寻址方式

寄存器的值可以被直接用于数据操作指令，这种寻址方式是各类处理器经常采用的一种方式，也是一种执行效率较高的寻址方式，如：

```
MOV  R2，R0               ;R0 的值送 R2
ADD  R4，R3，R2           ;R2 加 R3，结果送 R4
CMP  R7，R8               ;比较 R7 和 R8 的值
```

3．寄存器移位寻址方式

寄存器的值在被送到 ALU 之前，可以事先经过桶形移位寄存器的处理。预处理和移位发生在同一周期内，所以有效地使用移位寄存器，可以增加代码的执行效率。

下面是一些在指令中使用了移位操作的例子：

```
ADD  R2,R0,R1,LSR  #5
MOV  R1,R0,LSL  #2
RSB  R9,R5,R5,LSL  #1
SUB  R1,R2,R0,LSR ＃4
MOV  R2,R4,ROR  R0
```

2.7.2　内存访问指令寻址方式

内存访问指令的寻址方式可以分为以下几种。

① 字及无符号字节的 Load/Store 指令的寻址方式。

② 杂类 Load/Store 指令的寻址方式。
③ 批量 Load/Store 指令的寻址方式。
④ 协处理器 Load/Store 指令的寻址方式。

➢ 字及无符号字节的 Load/Store 指令的寻址方式

字及无符号字节的 Load/Store 指令语法格式如下：

```
LDR|STR{<cond>}{B}{T}  <Rd>，<addressing_mode>
```

其中，<addressing_mode>共有 9 种寻址方式，如表 2-9 所示。

表 2-9　　字及无符合字节的 Load/Store 指令的寻址方式

	格　　式	模　　式
1	[Rn，#±<offset_12>]	立即数偏移寻址 （Immediate offset）
2	[Rn，±Rm]	寄存器偏移寻址 （Register offset）
3	[Rn，Rm，<shift>#< offset_12>]	带移位的寄存器偏移寻址 （Scaled register offset）
4	[Rn，#±< offset_12>]!	立即数前索引寻址 （Immediate pre-indexed）
5	[Rn，±Rm]!	寄存器前索引寻址 （Register post-indexed）
6	[Rn，Rm，<shift>#< offset_12>]!	带移位的寄存器前索引寻址 (Scaled register pre-indexed)
7	[Rn]，#±< offset_12>	立即数后索引寻址 (Immediate post-indeded)
8	[Rn]，±<Rm>	寄存器后索引寻址 (Register post-indexed)
9	[Rn]，±<Rm>，<shift>#< offset_12>	带移位的寄存器后索引寻址 (Scaled register post-indexed)

➢ 杂类 Load/Store 指令的寻址方式

使用该类寻址方式的指令的语法格式如下：

```
LDR|STR{<cond>}H|SH|SB|D  <Rd>，<addressing_mode>
```

使用该类寻址方式的指令包括（有符号/无符号）半字 Load/Store 指令、有符号字节 Load/Store 指令和双字 Load/Store 指令。

该类寻址方式分为 6 种类型，如表 2-10 所示。

表 2-10　　杂类 Load/Store 指令的寻址方式

	格　　式	模　　式
1	[Rn，#±<offset_8>]	立即数偏移寻址 （Immediate offset）
2	[Rn，±Rm]	寄存器偏移寻址 （Register offset）
3	[Rn，#±< offset_8>]!	立即数前索引寻址 （Immediate pre-indexed）
4	[Rn，±Rm]!	寄存器前索引寻址 （Register post-indexed）

5	[Rn]，#±< offset_8>	立即数后索引寻址 (Immediate post-indeded)
6	[Rn]，±<Rm>	寄存器后索引寻址 (Register post-indexed)

➢ 批量 Load/Store 指令寻址方式

批量 Load/Store 指令将一片连续内存单元的数据加载到通用寄存器组中或将一组通用寄存器的数据存储到内存单元中。

批量 Load/Store 指令的寻址模式产生一个内存单元的地址范围，指令寄存器和内存单元的对应关系满足这样的规则，即编号低的寄存器对应于内存中低地址单元，编号高的寄存器对应于内存中的高地址单元。

该类指令的语法格式如下：

```
LDM|STM{<cond>}<addressing_mode>  <Rn>{!}，<registers><^>
```

该类指令的寻址方式如表 2-11 所示。

表 2-11　　批量 Load/Store 指令的寻址方式

	格　　式	模　　式
1	IA（Increment After）	后递增方式
2	IB（Increment Before）	先递增方式
3	DA（Decrement After）	后递减方式
4	DB（Decrement Before）	先递减方式

➢ 堆栈操作寻址方式

堆栈操作寻址方式和批量 Load/Store 指令寻址方式十分类似。但对于堆栈的操作，数据写入内存和从内存中读出要使用不同的寻址模式，因为进栈操作（pop）和出栈操作（push）要在不同的方向上调整堆栈。

下面详细讨论如何使用合适的寻址方式实现数据的堆栈操作。

根据不同的寻址方式，将堆栈分为以下 4 种。

① Full 栈：堆栈指针指向栈顶元素（last used location）。

② Empty 栈：堆栈指针指向第一个可用元素（the first unused location）。

③ 递减栈：堆栈向内存地址减小的方向生长。

④ 递增栈：堆栈向内存地址增加的方向生长。

根据堆栈的不同种类，将其寻址方式分为以下 4 种。

① 满递减 FD（Full Descending）。

② 空递减 ED（Empty Descending）。

③ 满递增 FA（Full Ascending）。

④ 空递增 EA（Empty Ascending）。

表 2-12 列出了堆栈的寻址方式和批量 Load/Store 指令寻址方式的对应关系。

表 2-12　　堆栈寻址方式和批量 Load/Store 指令寻址方式对应关系

批量数据寻址方式	堆栈寻址方式	L 位	P 位	U 位
LDMDA	LDMFA	1	0	0
LDMIA	LDMFD	1	0	1
LDMDB	LDMEA	1	1	0
LDMIB	LDMED	1	1	1
STMDA	STMED	0	0	0

STMIA	STMEA	0	0	1
STMDB	STMFD	0	1	0
STMIB	STMFA	0	1	1

➢ 协处理器 Load/Store 寻址方式

协处理器 Load/Store 指令的语法格式如下：

```
<opcode>{<cond>}{L}  <coproc>，<CRd>，<addressing_mode>
```

2.8 ARM 处理器的指令集

2.8.1 数据操作指令

数据操作指令是指对存放在寄存器中的数据进行操作的指令。主要包括数据传送指令、算术指令、逻辑指令、比较与测试指令及乘法指令。

如果在数据处理指令前使用 S 前缀，指令的执行结果将会影响 CPSR 中的标志位。数据处理指令如表 2-13 所示。

表 2-13 数据处理指令列表

助 记 符	操 作	行 为
MOV	数据传送	
MVN	数据取反传送	
AND	逻辑与	Rd：=Rn AND op2
EOR	逻辑异或	Rd：=Rn EOR op2
SUB	减	Rd：=Rn − op2
RSB	翻转减	Rd：=op2 − Rn
ADD	加	Rd：=Rn + op2
ADC	带进位的加	Rd：=Rn + op2 + C
SBC	带进位的减	Rd：=Rn− op2 + C − 1
RSC	带进位的翻转减	Rd：=op2 − Rn + C − 1
TST	测试	Rn AND op2 并更新标志位
TEQ	测试相等	Rn EOR op2 并更新标志位
CMP	比较	Rn−op2 并更新标志位
CMN	负数比较	Rn+op2 并更新标志位
ORR	逻辑或	Rd：=Rn OR op2
BIC	位清 0	Rd：=Rn AND NOT（op2）

2.8.1.1 MOV 指令

MOV 是最简单的 ARM 指令，结果是把一个数 N 送到目标寄存器 Rd，其中 N 可以是寄存器也可是立即数。

MOV 指令多用于设置初始值或者在寄存器间传送数据。

MOV 指令将移位码（shifter_operand）表示的数据传送到目的寄存器 Rd，并根据操作的结果更新 CPSR 中相应的条件标志位。

➢ 指令的语法格式

```
MOV{<cond>}{S}   <Rd>，<shifter_operand>
```

➢ 指令举例

```
MOV     R0, R0          ; R0 = R0... NOP 指令
MOV     R0, R0, LSL#3   ; R0 = R0 * 8
```

如果 R15 是目的寄存器，将修改程序计数器或标志。这用于被调用的子函数结束后返回到调用代码，方法是把连接寄存器的内容传送到 R15。

```
MOV     PC, R14     ; 退出到调用者，用于普通函数返回，PC 即是 R15
MOVS    PC, R14     ; 退出到调用者并恢复标志位，用于异常函数返回
```

➢ 指令的使用

MOV 指令主要完成以下功能。

- 将数据从一个寄存器传送到另一个寄存器。 ● 将一个常数值传送到寄存器中。
- 实现无算术和逻辑运算的单纯移位操作，操作数乘以 2^n 可以用左移 n 位来实现。
- 当 PC（R15）用作目的寄存器时，可以实现程序跳转。如“MOV PC，LR”，所以这种跳转可以实现子程序调用及从子程序返回，代替指令“B，BL”。
- 当 PC 作为目标寄存器且指令中 S 位被设置时，指令在执行跳转操作的同时，将当前处理器模式的 SPSR 寄存器的内容复制到 CPSR 中。这种指令“MOVS PC LR”可以实现从某些异常中断中返回。

2.8.1.2 MVN 指令

MVN 是反相传送（Move Negative）指令。它将操作数的反码传送到目的寄存器。

MVN 指令多用于向寄存器传送一个负数或生成位掩码。

MVN 指令将 shifter_operand 表示的数据的反码传送到目的寄存器 Rd。并根据操作的结果更新 CPSR 中相应的条件标志位。

➢ 指令的语法格式

```
MNV{<cond>}{S}   <Rd>，<shifter_operand>
```

➢ 指令举例

MVN 指令和 MOV 指令相同，也可以把一个数 N 送到目标寄存器 Rd，其中 N 可以是立即数，也可以是寄存器。

这是逻辑非操作而不是算术操作，这个取反的值加 1 才是它的取负的值。

```
    MVN     R0, #4              ; R0 = -5
    MVN     R0, #0              ; R0 = -1
```

➢ 指令的使用

MVN 指令主要完成以下功能：

- 向寄存器中传送一个负数。
- 生成位掩码（Bit Mask）。
- 求一个数的反码。

2.8.1.3 AND 指令

AND 指令将 shifter_operand 表示的数值与寄存器 Rn 的值按位（bitwise）做逻辑与操作，并将结果保存到目标寄存器 Rd 中，同时根据操作的结果更新 CPSR 寄存器。

➢ 指令的语法格式

```
AND{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

➢ 指令举例

（1）保留 R0 中的 0 位和 1 位，丢弃其余的位。

```
AND     R0, R0, #3
```

（2）R2=R1&R3。

```
AND    R2,R1,R3
```

（3）R0=R0&0x01，取出最低位数据。

```
ANDS    R0,R0,#0x01
```

2.8.1.4 EOR 指令

EOR（Exclusive OR）指令将寄存器 Rn 中的值和 shifter_operand 的值执行按位“异或”操作，并将执行结果存储到目的寄存器 Rd 中，同时根据指令的执行结果更新 CPSR 中相应的条件标志位。

➢ 指令的语法格式

```
EOR{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

➢ 指令举例

（1）反转 R0 中的位 0 和 1。

```
EOR     R0, R0, #3
```

（2）将 R1 的低 4 位取反。

```
EOR     R1,R1,#0x0F
```

（3）R2 = R1∧R0。

```
EOR     R2,R1,R0
```

（4）将 R5 和 0x01 进行逻辑异或，结果保存到 R0，并根据执行结果设置标志位。

```
EORS     R0，R5，＃0x01
```

2.8.1.5 SUB 指令

SUB（Subtract）指令从寄存器 Rn 中减去 shifter_operand 表示的数值，并将结果保存到目标寄存器 Rd 中，并根据指令的执行结果设置 CPSR 中相应的标志位。

- **指令的语法格式**

```
SUB{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

- **SUB 指令举例：**

（1）R0 = R1 − R2。

```
SUB     R0, R1, R2
```

（2）R0 = R1 − 256。

```
SUB     R0, R1, #256
```

（3）R0 = R2− (R3<<1)。

```
SUB     R0, R2, R3,LSL#1
```

2.8.1.6 RSB 指令

RSB（Reverse Subtract）指令从寄存器 shifter_operand 中减去 Rn 表示的数值，并将结果保存到目标寄存器 Rd 中，并根据指令的执行结果设置 CPSR 中相应的标志位。

- **指令的语法格式**

```
RSB{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

- **RSB 指令举例：**

下面指令序列可以求一个 64 位数值的负数。64 位数放在寄存器 R0 与 R1 中，其负数放在 R2 和 R3 中。其中 R0 与 R2 中放低 32 位值。

```
RSBS     R2，R0，＃0
RSC      R3，R1，＃0
```

2.8.1.7 ADD 指令

ADD 指令将寄存器 shifter_operand 的值加上 Rn 表示的数值，并将结果保存到目标寄存器 Rd 中，并根据指令的执行结果设置 CPSR 中相应的标志位。

➢ 指令的语法格式

```
ADD{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

➢ ADD 指令举例：

```
ADD    R0, R1, R2            ; R0 = R1 + R2
ADD    R0, R1, #256      ; R0 = R1 + 256
ADD    R0, R2, R3,LSL#1  ; R0 = R2 + (R3 << 1)
```

2.8.1.8 ADC 指令

ADC 指令将寄存器 shifter_operand 的值加上 Rn 表示的数值，再加上 CPSR 中的 C 条件标志位的值，将结果保存到目标寄存器 Rd 中，并根据指令的执行结果设置 CPSR 中相应的标志位。

➢ 指令的语法格式

```
ADC{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

➢ ADC 指令举例：

ADC 指令将把两个操作数加起来，并把结果放置到目的寄存器中。它使用一个进位标志位，这样就可以做比 32 位大的加法。下面的例子将加两个 128 位的数。

128 位结果：寄存器 R0、R1、R2 和 R3

第一个 128 位数：寄存器 R4、R5、R6 和 R7

第二个 128 位数：寄存器 R8、R9、R10 和 R11。

```
  ADDS    R0, R4, R8            ;加低端的字
  ADCS    R1, R5, R9            ;加下一个字，带进位
  ADCS    R2, R6, R10           ;加第三个字，带进位
  ADCS    R3, R7, R11           ;加高端的字，带进位
```

2.8.1.9 SBC 指令

SBC（Subtract with Carry）指令用于执行操作数大于 32 位时的减法操作。该指令从寄存器 Rn 中减去 shifter_operand 表示的数值，再减去寄存器 CPSR 中 C 条件标志位的反码［NOT（Carry flag）］。并将结果保存到目标寄存器 Rd 中，并根据指令的执行结果设置 CPSR 中相应的标志位。

➢ 指令的语法格式

```
SBC{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

➢ SBC 指令举例

下面的程序使用 SBC 实现 64 位减法，（R1，R0）-（R3，R2），结果存放到（R1，R0）

```
SUBS    R0，R0，R2
SBCS    R1，R1，R3
```

2.8.1.10 RSC 指令

RSC（Reverse Subtract with Carry）指从寄存器 shifter_operand 中减去 Rn 表示的数值，再减去寄存器 CPSR 中 C 条件标志位的反码［NOT（Carry Flag）］，并将结果保存到目标寄存器 Rd 中，并根据指令的执行结果设置 CPSR 中相应的标志位。

➢ 指令的语法格式

```
RSC{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

➢ RSC 指令举例

下面程序使用 RSC 指令实现求 64 位数值的负数。

```
RSBS    R2，R0，＃0
RSC     R3，R1，＃0
```

2.8.1.11 TST 测试指令

TST（Test）测试指令用于将一个寄存器的值和一个算术值进行比较。条件标志位根据两个操作数做“逻辑与”后的结果设置。

➢ 指令的语法格式

```
TST{<cond>}  <Rn>,<shifter_operand>
```

➢ TST 指令举例

TST 指令类似于 CMP 指令，不产生放置到目的寄存器中的结果。而是在给出的两个操作数上进行操作并把结果反映到状态标志上。使用 TST 指令来检查是否设置了特定的位。操作数 1 是要测试的数据字而操作数 2 是一个位掩码。经过测试后，如果匹配则设置 Zero 标志，否则清除它。与 CMP 指令一样，该指令不需要指定 S 后缀。

下面的指令测试在 R0 中是否设置了位 0

```
  TST    R0, #%1
```

2.8.1.12 TEQ 指令

TEQ（Test Equivalence）指令用于将一个寄存器的值和一个算术值做比较。条件标志位根据两个操作数做“逻辑或”后的结果设置。以便后面的指令根据相应的条件标志来判断是否执行。

- **指令的语法格式**

```
TEQ{<cond>}  <Rn>,<shifter_operand>
```

- **TEQ 指令举例：**

下面的指令是比较 R0 和 R1 是否相等，该指令不影响 CPSR 中的 V 位和 C 位。

```
TEQ     R0,R1
```

TST 指令与 EORS 指令的区别在于 TST 指令不保存运算结果。使用 TEQ 进行相等测试，常与 EQ 和 NE 条件码配合使用，当两个数据相等时，条件码 EQ 有效；否则条件码 NE 有效。

2.8.1.13 CMP 指令

CMP（Compare）指令使用寄存器 Rn 的值减去 operand2 的值，根据操作的结果更新 CPSR 中相应的条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。

- **指令的语法格式**

```
CMP{<cond>}  <Rn>,<shifter_operand>
```

- **CMP 指令举例：**

CMP 指令允许把一个寄存器的内容与另一个寄存器的内容或立即值进行比较，更改状态标志来允许进行条件执行。它进行一次减法，但不存储结果，而是正确地更改标志位。标志位表示的是操作数 1 与操作数 2 比较的结果（其可能为大、小、相等）。如果操作数 1 大于操作数 2，则此后的有 GT 后缀的指令将可以执行。

显然，CMP 不需要显式地指定 S 后缀来更改状态标志。

（1）下面的指令是比较 R1 和立即数 10 并设置相关的标志位。

```
CMP     R1,#10
```

（2）下面的指令是比较寄存器 R1 和 R2 中的值并设置相关的标志位。

```
CMP     R1,R2
```

通过上面的例子可以看出，CMP 指令与 SUBS 指令的区别在于 CMP 指令不保存运算结果，在进行两个数据大小判断时，常用 CMP 指令及相应的条件码来进行操作。

2.8.1.14 CMN 指令

CMN（Compare Negative）指令使用寄存器 Rn 的值减去 operand2 的负数值（加上 operand2），根据操作的结果更新 CPSR 中相应的条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。

➢ 指令的语法格式

```
CMN{<cond>}  <Rn>,<shifter_operand>
```

➢ CMN 指令举例：

CMN 指令将寄存器 Rn 中的值加上 shifter_operand 表示的数值，根据加法的结果设置 CPSR 中相应的条件标志位。寄存器 Rn 中的值加上 shifter_operand 的操作结果对 CPSR 中条件标志位的影响，与寄存器 Rn 中的值减去 shifter_operand 的操作结果的相反数对 CPSR 中条件标志位的影响有细微差别。当第 2 个操作数为 0 或者为 0x80000000 时两者结果不同。比如下面两条指令。

```
CMP     Rn，#0
CMN     Rn，#0
```

第 1 条指令使标志位 C 值为 1，第 2 条指令使标志位 C 值为 0。

下面的指令使 R0 值加 1，判断 R0 是否为 1 的补码，若是，则 Z 置位。

```
CMN     R0，#1
```

2.8.1.15 ORR 指令

ORR（Logical OR）为逻辑或操作指令，它将第 2 个源操作数 shifter_operand 的值与寄存器 Rn 的值按位做“逻辑或”操作，结果保存到 Rd 中。

➢ 指令的语法格式

```
ORR{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

➢ ORR 指令举例：

（1）设置 R0 中位 0 和 1。

```
ORR     R0, R0, #3
```

（2）将 R0 的低 4 位置 1。

```
ORR     R0,R0,#0x0F
```

（3）使用 ORR 指令将 R2 的高 8 位数据移入到 R3 的低 8 位中。

```
MOV     R1，R2，LSR #4
ORR     R3，R1，R3，LSL #8
```

2.8.1.16 BIC 位清零指令

BIC（Bit Clear）位清零指令，将寄存器 Rn 的值与第二源操作数<shifter_operand>的值的反码按位做“逻辑与”操作，结果保存到 Rd 中。

➢ 指令的语法格式

```
BIC{<cond>}{S}  <Rd>,<Rn>,<shifter_operand>
```

➢ BIC 指令举例

（1）清除 R0 中的位 0、1 和 3。保持其余的不变。

```
BIC     R0, R0, #0x1011
```

（2）将 R3 的反码和 R2 逻辑与，结果保存到 R1 中。

```
BIC     R1，R2，R3
```

2.8.2 乘法指令

ARM 乘法指令完成两个数据的乘法。两个 32 位二进制数相乘的结果是 64 位的积。在有些 ARM 的处理器版本中，将乘积的结果保存到两个独立的寄存器中。另外一些版本只将最低有效 32 位存放到一个寄存器中。

无论是哪种版本的处理器，都有乘—累加的变型指令，将乘积连续累加得到总和。而且有符号数和无符号数都能使用。对于有符号数和无符号数，结果的最低有效位是一样的。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这两种情况。

表 2-14 为各种形式乘法指令的功能。

表 2-14　　各种形式乘法指令

操作码[23:21]	助　记　符	意　　义	操　　作
000	MUL	乘（保留 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

其中：

① “RdHi：RdLo”是由 RdHi（最高有效 32 位）和 RdLo（最低有效 32 位）连接形成的 64 位数，“[31：0]”只选取结果的最低有效 32 位。

② 简单的赋值由“：=”表示。

③ 累加（将右边加到左边）是由“+=”表示。

各个乘法指令中的位 S（参考下文具体指令的语法格式）控制条件码的设置会产生以下结果：

① 对于产生 32 位结果的指令形式，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。

② 对于产生 32 位结果的指令形式，如果 Rd 等于零，则标志位 Z 置位；对于产生长结果的指令形式，RdHi 和 RdLo 同时为零时，标志位 Z 置位。

③ 将标志位 C 设置成无意义的值。

④ 标志位 V 不变。

2.8.2.1 MUL 指令

MUL（Multiply）32 位乘法指令将 Rm 和 Rs 中的值相乘，结果的最低 32 位保存到 Rd 中。

➢ 指令的语法格式

```
MUL{<cond>}{S}   <Rd>，<Rm>，<Rs>
```

➢ 指令举例

（1）R1 = R2 × R3。

```
MUL    R1, R2, R3
```

（2）R0 = R3 × R7，同时设置 CPSR 中的 N 位和 Z 位。

```
MULS    R0, R3, R7
```

2.8.2.2 MLA 乘—累加指令

MLA（Multiply Accumulate）32 位乘—累加指令将 Rm 和 Rs 中的值相乘，再将乘积加上第 3 个操作数，结果的最低 32 位保存到 Rd 中。

➢ 指令的语法格式

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

➢ 指令举例

下面指令完成 R1 = R2×R3 + 10 的操作。

```
MOV     R0, #0x0A
MLA     R1, R2, R3, R0
```

2.8.2.3 UMULL 指令

UMULL（Unsigned Multiply Long）为 64 位无符号乘法指令。它将 Rm 和 Rs 中的值做无符号数相乘，结果的低 32 位保存到 RsLo 中，高 32 位保存到 RdHi 中。

➢ 指令的语法格式

```
UMULL{<cond>}{S}   <RdLo>，<RdHi>，<Rm>，<Rs>
```

➢ 指令举例

下面指令完成(R1，R0) = R5 × R8 操作。

```
UMULL    R0, R1, R5, R8;
```

2.8.2.4 UMLAL 指令

UMLAL（Unsigned Multiply Accumulate Long）为 64 位无符号长乘—累加指令。指令将 Rm 和 Rs 中的值做无符号数相乘，64 位乘积与 RdHi、RdLo 相加，结果的低 32 位保存到 RsLo 中，高 32 位保存到 RdHi 中。

- 指令的语法格式

```
UMALL{<cond>}{S}   <RdLo>，<RdHi>，<Rm>，<Rs>
```

- 指令举例

下面的指令完成(R1，R0) = R5 × R8+(R1，R0)操作。

```
UMLAL    R0, R1, R5,R8;
```

2.8.2.5 SMULL 指令

SMULL（Signed Multiply Long）为 64 位有符号长乘法指令。指令将 Rm 和 Rs 中的值做有符号数相乘，结果的低 32 位保存到 RsLo 中，高 32 位保存到 RdHi 中。

- 指令的语法格式

```
SMULL{<cond>}{S}   <RdLo>，<RdHi>，<Rm>，<Rs>
```

- 指令举例

下面的指令完成(R3，R2) = R7 × R6 操作。

```
SMULL    R2, R3, R7,R6;
```

2.8.2.6 SMLAL 指令

SMLAL（Signed Multiply Accumulate Long）为 64 位有符号长乘—累加指令。指令将 Rm 和 Rs 中的值做有符号数相乘，64 位乘积与 RdHi、RdLo 相加，结果的低 32 位保存到 RsLo 中，高 32 位保存到 RdHi 中。

- 指令的语法格式

```
SMLAL{<cond>}{S}   <RdLo>，<RdHi>，<Rm>，<Rs>
```

➢ 指令举例

下面的指令完成(R3，R2) = R7 × R6 +(R3，R2)操作。

```
SMLAL    R2, R3, R7,R6;
```

2.8.3 Load/Stor 指令

Load/Store 内存访问指令在 ARM 寄存器和存储器之间传送数据。ARM 指令中有 3 种基本的数据传送指令。

1．单寄存器 Load/Store 指令（Single Register）

这些指令在 ARM 寄存器和存储器之间提供更灵活的单数据项传送方式。数据项可以是字节、16 位半字或 32 位字。

2．多寄存器 Load/Store 内存访问指令

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复制存储器中的一块数据。

3．单寄存器交换指令（Single Register Swap）

这些指令允许寄存器和存储器中的数值进行交换，在一条指令中有效地完成 Load/Store 操作。它们在用户级编程中很少用到。它的主要用途是在多处理器系统中实现信号量（Semaphores）的操作，以保证不会同时访问公用的数据结构。

单寄存器的 Load/Store 指令

这种指令用于把单一的数据传入或者传出一个寄存器。支持的数据类型有字节（8 位）、半字（16 位）和字（32 位）。

表 2-15 列出了所有单寄存器的 Load/Store 指令。

表 2-15　　单寄存器 Load/Store 指令

指　令	作　用	操　作
LDR	把一个字装入一个寄存器	Rd←mem32[address]
STR	将存储器中的字保存到寄存器	Rd→mem32[address]
LDRB	把一个字节装入一个寄存器	Rd←mem8[address]
STRB	将寄存器中的低 8 位字节保存到存储器	Rd→mem8[address]
LDRH	把一个半字装入一个寄存器	Rd←mem16[address]
STRH	将寄存器中的低 16 位半字保存到存储器	Rd→mem16[address]
LDRBT	用户模式下将一个字节装入寄存器	Rd←mem8[address] under user mode

STRBT	用户模式下将寄存器中的低 8 位字节保存到存储器	Rd→mem8[address] under user mode
LDRT	用户模式下把一个字装入一个寄存器	Rd←mem32[address]under user mode
STRT	用户模式下将存储器中的字保存到寄存器	Rd→mem32[address]]under user mode
LDRSB	把一个有符号字节装入一个寄存器	Rd←sign{mem8[address]}
LDRSH	把一个有符号半字装入一个寄存器	Rd←sign{mem16[address]}

2.8.3.1 LDR 指令

LDR 指令用于从内存中将一个 32 位的字读取到目标寄存器。

➢ 指令的语法格式

```
LDR{<cond>}  <Rd>，<addr_mode>
```

➢ 指令举例

```
LDR  R1，[R0，#0x12]      ;将 R0+12 地址处的数据读出，保存到 R1 中（R0 的值不变）
LDR  R1，[R0]                 ;将 R0 地址处的数据读出，保存到 R1 中（零偏移）
LDR  R1，[R0，R2]         ;将 R0+R2 地址的数据读出，保存到 R1 中（R0 的值不变）
LDR  R1，[R0，R2，LSL #2] ;将 R0+R2×4 地址处的数据读出，保存到 R1 中（R0、R2 的值不变）
LDR  Rd，label            ;label 为程序标号，label 必须是当前指令的-4～4KB 范围内
LDR  Rd，[Rn]，＃0x04     ;Rn 的值用作传输数据的存储地址。在数据传送后，将偏移量 0x04 与
                           Rn 相加，结果写回到 Rn 中。Rn 不允许是 R15
```

2.8.3.2 STR 指令

STR 指令用于将一个 32 位的字数据写入到指令中指定的内存单元。

➢ 指令的语法格式

```
STR{<cond>}  <Rd>，<addr_mode>
```

➢ 指令举例

LDR/STR 指令用于对内存变量的访问、内存缓冲区数据的访问、查表、外围部件的控制操作等，若使用 LDR 指令加载数据到 PC 寄存器，则实现程序跳转功能，这样也就实现了程序散转。

① 变量访问

```
NumCount  EQU  0x40003000     ;定义变量 NumCount
LDR  R0，=NumCount            ;使用 LDR 伪指令装载 NumCount 的地址到 R0
LDR  R1，[R0]                     ;取出变量值
ADD  R1，R1，＃1              ;NumCount=NumCount+1
```

```
STR   R1，[R0]                ;保存变量
```

② GPIO 设置

```
GPIO_BASE  EQU   0xe0028000   ;定义 GPIO 寄存器的基地址
…
LDR    R0,=GPIO_BASE
LDR    R1,=0x00ffff00         ;将设置值放入寄存器
STR    R1,[R0,#0x0C]          ;IODIR=0x00ffff00，IOSET 的地址为 0xE0028004
```

③ 程序散转

```
…
MOV  R2,R2,LSL #2                ;功能号乘以 4，以便查表
LDR  PC,[PC,R2]                  ;查表取得对应功能子程序地址并跳转
NOP
FUN_TAB  DCD   FUN_SUB0
         DCD   FUN_SUB1
         DCD   FUN_SUB2
           …
```

2.8.3.3 LDRB 指令

LDRB 指令根据 addr_mode 所确定的地址模式将一个 8 位字节读取到指令中的目标寄存器 Rd。

指令的语法格式：

```
LDR{<cond>}B  <Rd>，<addr_mode>
```

2.8.3.4 STRB 指令

STRB 指令从寄存器中取出指定的 8 位字节放入寄存器的低 8 位，并将寄存器的高位补 0。
指令的语法格式：

```
STR{<cond>}B  <Rd>，<addr_mode>
```

2.8.3.5 LDRH 指令

LDRH 指令用于从内存中将一个 16 位的半字读取到目标寄存器。
如果指令的内存地址不是半字节对齐的，指令的执行结果不可预知。
指令的语法格式：

```
LDR{<cond>}H  <Rd>，<addr_mode>
```

2.8.3.6 TRH 指令

STRH 指令从寄存器中取出指定的 16 位半字放入寄存器的低 16 位，并将寄存器的高位补 0。
指令的语法格式：

```
STR{<cond>}H  <Rd>，<addr_mode>
```

多寄存器的 Load/Store 内存访问指令

多寄存器的 Load/Store 内存访问指令也叫批量加载/存储指令，它可以实现在一组寄存器和一块连续的内存单元之间传送数据。LDM 用于加载多个寄存器，STM 用于存储多个寄存器。多寄存器的 Load/Store 内存访问指令允许一条指令传送 16 个寄存器的任何子集或所有寄存器。

多寄存器的 Load/Store 内存访问指令主要用于现场保护、数据复制和参数传递等。

表 2-16 列出了多寄存器的 Load/Store 内存访问指令。

表 2-16　　多寄存器的 Load/Store 内存访问指令

指　令	作　用	操　作
LDM	装载多个寄存器	{Rd}*^{N}←mem32[start address+4*N]
STM	保存多个寄存器	{Rd}*^{N}→mem32[start address+4*N]

2.8.3.7　LDM 指令

LDM 指令将数据从连续的内存单元中读取到指令中指定的寄存器列表中的各寄存器中。

当 PC 包含在 LDM 指令的寄存器列表中时，指令从内存中读取的字数据将被作为目标地址值，指令执行后程序将从目标地址处开始执行，从而实现了指令的跳转。

指令的语法格式：

```
LDM{<cond>}<addressing_mode>  <Rn>{!},  <registers>
```

寄存器 R0～R15 分别对应于指令编码中 bit[0]～bit[15]位。如果 R*i* 存在于寄存器列表中，则相应的位等于 1，否则为 0。

LDM 指令将数据从连续的内存单元中读取到指令中指定的寄存器列表中的各寄存器中。

指令的语法格式：

```
LDM{<cond>}<addressing_mode>  <Rn>,  <registers_without_pc>
```

2.8.3.8　STM 指令

STM 指令将指令中寄存器列表中的各寄存器数值写入到连续的内存单元中。主要用于块数据的写入、数据栈操作及进入子程序时保存相关寄存器的操作。

指令的语法格式：

```
STM{<cond>}<addressing_mode>  <Rn>{!},  <registers>
```

STM 指令将指令中寄存器列表中的各寄存器数值写入到连续的内存单元中。主要用于块数据的写入、数据栈操作及进入子程序时保存相关寄存器等操作。

指令的语法格式：

```
STM{<cond>}<addressing_mode>  <Rn>,  <registers >
```

数据传送指令应用

LDM/STM 批量加载/存储指令可以实现在一组寄存器和一块连续的内存单元之间传输数据。LDM 为加载多个寄存器，STM 为存储多个寄存器。允许一条指令传送 16 个寄存器的任何子集或所有寄存器。指令格式如下：

```
LDM{cond}<模式>  Rn{!}，regist{  }
STM{cond}<模式>  Rn{!}，regist{  }
```

LDM/STM 的主要用途有现场保护、数据复制和参数传递等。其模式有 8 种，前面 4 种用于数据块的传输，后面 4 种是堆栈操作，如下所示。

（1）IA：每次传送后地址加 4。
（2）IB：每次传送前地址加 4。
（3）DA：每次传送后地址减 4。
（4）DB：每次传送前地址减 4。
（5）FD：满递减堆栈。
（6）ED：空递增堆栈。
（7）FA：满递增堆栈。
（8）EA：空递增堆栈。

其中，寄存器 Rn 为基址寄存器，装有传送数据的初始地址，Rn 不允许为 R15；后缀“!”表示最后的地址写回到 Rn 中；寄存器列表 reglist 可包含多于一个寄存器或寄存器范围，使用“,”分开，如{R1，R2，R6～R9}，寄存器排列由小到大排列；“^”后缀不允许在用户模式下使用，只能在系统模式下使用。若在 LDM 指令用寄存器列表中包含有 PC 时使用，那么除了正常的多寄存器传送外，将 SPSR 复制到 CPSR 中，这可用于异常处理返回；使用“^”后缀进行数据传送且寄存器列表不包含 PC 时，加载/存储的是用户模式寄存器，而不是当前模式寄存器。

```
LDMIA  R0!，{R3~R9}       ;加载 R0 指向的地址上的多字数据，保存到 R3～R9 中，R0 值更新
STMIA  R1!，{R3～R9}      ;将 R3～R9 的数据存储到 R1 指向的地址上，R1 值更新
STMFD  SP!，{R0～R7，LR}  ;现场保存，将 R0～R7、LR 入栈
LDMFD  SP!，{R0～R7，PC}  ;恢复现场，异常处理返回
```

在进行数据复制时，先设置好源数据指针，然后使用块拷贝寻址指令 LDMIA/STMIA、LDMIB/STMIB、LDMDA/STMDA、LDMDB/STMDB 进行读取和存储。而进行堆栈操作时，则要先设置堆栈指针，一般使用 SP 然后使用堆栈寻址指令 STMFD/LDMFD、STMED/LDMED、STMEA/LDMEA 实现堆栈操作。

数据是存储在基址寄存器的地址之上还是之下，地址是存储第一个值之前还是之后、增加还是减少，如表 2-17 所示。

表 2-17　　多寄存器的 Load/Store 内存访问指令映射

		向上生长		向下生长	
		满	空	满	空
增加	之前	STMIB			LDMIB
		STMFA			LDMED
	之后		STMIA	LDMIA	
			STMEA	LDMFD	
增加	之前		LDMDB	STMDB	
			LDMEA	STMFD	
	之后	LDMDA			STMDA
		LDMFA			STMED

【例 2-1】　使用 LDM/STM 进行数据复制。

```
LDR  R0，=SrcData         ;设置源数据地址
LDR  R1，=DstData         ;设置目标地址
LDMIA  R0，{R2～R9}       ;加载 8 字数据到寄存器 R2～R9
STMIA  R1，{R2～R9}       ;存储寄存器 R2～R9 到目标地址
```

【例 2-2】　使用 LDM/STM 进行现场寄存器保护，常在子程序或异常处理使用。

```
SENDBYTE
     STMFD    SP!，{R0～R7，LR}       ;寄存器压栈保护
      …
     BL       DELAY                  ;调用 DELAY 子程序
      …
     LDMFD    SP!，{R0～R7，PC}       ;恢复寄存器，并返回
```

2.8.4 单数据交换指令

交换指令是 Load/Store 指令的一种特例，它把一个寄存器单元的内容与寄存器内容交换。交换指令是一个原子操作（Atomic Operation），也就是说，在连续的总线操作中读/写一个存储单元，在操作期间阻止其他任何指令对该存储单元的读/写。

交换指令如表 2-18 所示。

表 2-18 交换指令 SWP

指　令	作　用	操　作
SWP	字交换	tmp=men32[Rn] mem32[Rn]=Rm Rd=tmp
SWPB	字节交换	tmp=men8[Rn] mem8[Rn]=Rm Rd=tmp

1．SWP 字交换指令

SWP 指令用于将内存中的一个字单元和一个指定寄存器的值相交换。操作过程如下：假设内存单元地址存放在寄存器<Rn>中，指令将<Rn>中的数据读取到目的寄存器 Rd 中，同时将另一个寄存器<Rm>的内容写入到该内存单元中。当<Rd>和<Rm>为同一个寄存器时，指令交换该寄存器和内存单元的内容。

指令的语法格式：

```
SWP{<cond>}  <Rd>，<Rm>，[<Rn>]
```

2．SWPB 字节交换指令

SWPB 指令用于将内存中的一个字节单元和一个指定寄存器的低 8 位值相交换，操作过程如下：假设内存单元地址存放在寄存器<Rn>中，指令将<Rn>中的数据读取到目的寄存器 Rd 中，寄存器 Rd 的高 24 位设为 0，同时将另一个寄存器<Rm>的低 8 位内容写入到该内存字节单元中。当<Rd>和<Rm>为同一个寄存器时，指令交换该寄存器低 8 位内容和内存字节单元的内容。

指令的语法格式：

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

3．交换指令 SWP 应用

SWP 指令用于将一个内存单元（该单元地址放在寄存器 Rn 中）的内容读取到一个寄存器 Rd 中，同时将另一个寄存器 Rm 的内容写到该内存单元中，使用 SWP 可实现信号量操作。

指令的语法格式：

```
SWP{cond}B  Rd，Rm，[Rn]
```

其中，B 为可选后缀，若有 B，则交换字节；否则交换 32 位字。Rd 为目的寄存器，存储从存储器中加载的数据，同时，Rm 中的数据将会被存储到存储器中。若 Rm 与 Rn 相同，则为寄存器与存储器内容进行交换。Rn 为要进行数据交换的存储器地址，Rn 不能与 Rd 和 Rm 相同。

【例 2-3】 SWP 指令举例。

```
SWP  R1，R1，[R0]      ;将 R1 的内容与 R0 指向的存储单元内容进行交换
SWPB  R1，R2，[R0]          ;将 R0 指向的存储单元内容读取一字节数据到 R1 中（高 24 位清零），
                          并将 R2 的内容写入到该内存单元中（最低字节有效）
```

使用 SWP 指令可以方便地进行信号量操作。

```
12C_SEM      EQU     0x40003000
              …
12C_SEM_WAIT
          MOV      R0,#0
          LDR      R0,=12C_SEM
          SWP      R1,R1,[R0]   ;取出信号量，并将其设为 0
          CMP      R1,#0    ;判断是否有信号
          BEQ      12C_SEM_WAIT ;若没有信号则等待
```

2.8.5 跳转指令

跳转（B）和跳转连接（BL）指令是改变指令执行顺序的标准方式。ARM 一般按照字地址顺序执行指令，需要时使用条件执行跳过某段指令。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其他几种方式实现这个目的，但转移和转移连接指令是标准的方式。

跳转指令改变程序的执行流程或者调用子程序。这种指令使得一个程序可以使用子程序、if-then-else 结构及循环。执行流程的改变迫使程序计数器（PC）指向一个新的地址，ARMv5 架构指令集包含的跳转指令如表 2-19 所示。

表 2-19 ARMv5 架构跳转指令

助 记 符	说 明	操 作
B	跳转指令	pc←label
BL	带返回的连接跳转	pc←label(lr←BL 后面的第一条指令)
BX	跳转并切换状态	pc←Rm&0xfffffffe, T←Rm&1
BLX	带返回的跳转并切换状态	pc←lable, T←1 pc←Rm&0xfffffffe, T←Rm&1 lr←BL 后面的第一条指令

另一种实现指令跳转的方式是通过直接向 PC 寄存器中写入目标地址值，实现在 4GB 地址空间中任意跳转，这种跳转指令又称为长跳转。如果在长跳转指令之前使用“MOV LR”或“MOV PC”等指令，可以保存将来返回的地址值，也就实现了在 4GB 的地址空间中的子程序调用。

2.8.5.1 跳转指令 B 及带连接的跳转指令 BL

跳转指令 B 使程序跳转到指定的地址执行程序。带连接的跳转指令 BL 将下一条指令的地址拷贝到 R14（即返回地址连接寄存器 LR）寄存器中，然后跳转到指定地址运行程序。需要注意的是，这两条指令和目标地址处的指令都要属于 ARM 指令集。两条指令都可以根据 CPSR 中的条件标志位的值决定指令是否执行。

➢ 指令的语法格式

```
B{L}{<cond>}  <target_address>
```

BL 指令用于实现子程序调用。子程序的返回可以通过将 LR 寄存器的值复制到 PC 寄存器来实现。下面三种指令可以实现子程序返回。

- BX　R14（如果体系结构支持 BX 指令）。
- MOV　PC，R14。
- 当子程序在入口处使用了压栈指令：

```
STMFD  R13!，{<registers>，R14}
```

可以使用指令：

```
LDMFD  R13!，{<registers>，PC}
```

将子程序返回地址放入 PC 中。

ARM 汇编器通过以下步骤计算指令编码中的 signed_immed_24。

（1）将 PC 寄存器的值作为本跳转指令的基地址值。

（2）从跳转的目标地址中减去上面所说的跳转的基地址，生成字节偏移量。由于 ARM 指令是字对齐的，该字节偏移量为 4 的倍数。

（3）当上面生成的字节偏移量超过−33 554 432～+33 554 430 时，不同的汇编器使用不同的代码产生策略。

（4）否则，将指令编码字中的 signed_immed_24 设置成上述字节偏移量的 bits[25：2]。

➢ 程序举例

（1）程序跳转到 LABLE 标号处。

```
   B  LABLE；
   ADD  R1，R2，＃4
   ADD  R3，R2，＃8
   SUB  R3，R3，R1
LABLE
   SUB  R1，R2，#8
```

（2）跳转到绝对地址 0x1234 处。

```
B  0x1234
```

（3）跳转到子程序 func 处执行，同时将当前 PC 值保存到 LR 中。

```
BL  func
```

（4）条件跳转：当 CPSR 寄存器中的 C 条件标志位为 1 时，程序跳转到标号 LABLE 处执行。

```
BCC  LABLE
```

（5）通过跳转指令建立一个无限循环。

```
LOOP
   ADD  R1，R2，＃4
   ADD  R3，R2，＃8
   SUB  R3，R3，R1
   B  LOOP
```

（6）通过使用跳转使程序体循环 10 次。

```
   MOV  R0，＃10
LOOP
   SUBS  R0，＃1
   BNE   LOOP
```

（7）条件子程序调用示例。

```
…
CMP  R0，＃5                ;如果 R0<5
BLLT  SUB1                  ;则调用
BLGE  SUB2                  ;否则调用 SUB2
```

2.8.5.2 BX 带状态切换的跳转指令 BX

带状态切换的跳转指令（BX）使程序跳转到指令中指定的参数 Rm 指定的地址执行程序，Rm 的第 0 位拷贝到 CPSR 中 T 位，bit[31：1]移入 PC。若 Rm 的 bit[0]为 1，则跳转时自动将 CPSR 中的标志位 T 置位，即把目标地址的代码解释为 Thumb 代码；若 Rm 的位 bit[0]为 0，则跳转时自动将 CPSR 中的标志位 T 复位，即把目标地址代码解释为 ARM 代码。

➢ 指令的语法格式

```
BX{<cond>}  <Rm>
```

- 当 Rm[1：0]=0b10 时，指令的执行结果不可预知。因为在 ARM 状态下，指令是 4 字节对齐的。
- PC 可以作为 Rm 寄存器使用，但这种用法不推荐使用。当 PC 作为<Rm>使用时，指令“BX　PC”将程序跳转到当前指令下面第二条指令处执行。虽然这样跳转可以实现，但最好使用下面的指令完成这种跳转。

```
MOV   PC, PC
```

或，

```
ADD   PC, PC, #0
```

➢ 指令举例

（1）转移到 R0 中的地址，如果 R0[0]=1，则进入 Thumb 状态。

```
BX   R0;
```

（2）跳转到 R0 指定的地址，并根据 R0 的最低位来切换处理器状态。

```
ADRL   R0，ThumbFun+1 ;
BX   R0;
```

2.8.5.3 BXL 带状态切换的连接跳转指令 BLX

带连接和状态切换的跳转指令（Branch with Link Exchange，BLX）使用标号，用于使程序跳转到 Thumb 状态或从 Thumb 状态返回。该指令为无条件执行指令，并用分支寄存器的最低位来更新 CPSR 中的 T 位，将返回地址写入到连接寄存器 LR 中。

➢ 语法格式

```
BLX  <target_add>
```

其中，<target_add>为指令的跳转目标地址。该地址根据以下规则计算。

① 将指令中指定的 24 位偏移量进行符号扩展，形成 32 位立即数。

② 将结果左移两位。

③ 位 H（bit[24]）加到结果地址的第一位（bit[1]）。

④ 将结果累加进程序计数器（PC）中。

计算偏移量的工作一般由 ARM 汇编器来完成。这种形式的跳转指令只能实现−32～32MB 空间的跳转。

左移两位形成字偏移量，然后将其累加进程序计数器（PC）中。这时，程序计数器的内容为 BX 指令地址加 8 字节。位 H（bit[24]）也加到结果地址的第一位（bit[1]），使目标地址成为半字地址，以执行接下来的 Thumb 指令。计算偏移量的工作一般由 ARM 汇编器来完成。这种形式的跳转指令只能实现−32～32MB 空间的跳转。

➢ 指令的使用

- 从 Thumb 状态返回到 ARM 状态，使用 BX 指令。

```
BX  R14
```

- 可以在子程序的入口和出口增加栈操作指令。

```
PUSH  {<registers>，R14}
…
POP  {<registers>，PC}
```

2.8.6 状态操作指令

ARM 指令集提供了两条指令，可直接控制程序状态寄存器（Program State Register，PSR）。MRS 指令用于把 CPSR 或 SPSR 的值传送到一个寄存器；MSR 与之相反，把一个寄存器的内容传送到 CPSR 或 SPSR。这两条指令相结合，可用于对 CPSR 和 SPSR 进行读/写操作。程序状态寄存器指令如表 2-20 所示。

表 2-20　　程序状态寄存器指令

指　令	作　用	操　作
MRS	把程序状态寄存器的值送到一个通用寄存器	Rd=SPR
MSR	把通用寄存器的值送到程序状态寄存器或把一个立即数送到程序状态字	PSR[field]=Rm 或 PSR[field]=immediate

在指令语法中可看到一个称为 fields 的项，它可以是控制（C）、扩展（X）、状态（S）及标志（F）的组合。

2.8.6.1 MRS

MRS 指令用于将程序状态寄存器的内容传送到通用寄存器中。

在 ARM 处理器中，只有 MRS 指令可以将状态寄存器 CPSR 或 SPSR 读出到通用寄存器中。

➢ 指令的语法格式

```
MRS{cond}  Rd，PSR
```

其中，Rd 为目标寄存器，Rd 不允许为程序计数器（PC）。PSR 为 CPSR 或 SPSR。

➢ 指令举例

```
MRS  R1，CPSR          ;将 CPSR 状态寄存器读取，保存到 R1 中
```

```
MRS  R2，SPSR          ;将 SPSR 状态寄存器读取，保存到 R1 中
```

MRS 指令读取 CPSR，可用来判断 ALU 的状态标志及 IRQ/FIQ 中断是否允许等；在异常处理程序中，读 SPSR 可指定进入异常前的处理器状态等。MRS 与 MSR 配合使用，实现 CPSR 或 SPSR 寄存器的读—修改—写操作，可用来进行处理器模式切换，允许/禁止 IRQ/FIQ 中断等设置。另外，进程切换或允许异常中断嵌套时，也需要使用 MRS 指令读取 SPSR 状态值并保存起来。

2.8.6.2 MSR

在 ARM 处理器中，只有 MSR 指令可以直接设置状态寄存器 CPSR 或 SPSR。

（1）指令的语法格式如下。

```
MSR{cond}  PSR_field,#immed_8r
MSR{cond}  PSR_field,Rm
```

其中，PSR 是指 CPSR 或 SPSR。<fields>设置状态寄存器中需要操作的位。状态寄存器的 32 位可以分为 4 个 8 位的域（field）。bits[31：24]为条件标志位域，用 f 表示；bits[23：16]为状态位域，用 s 表示；bits[15：8]为扩展位域，用 x 表示；bits[7：0]为控制位域，用 c 表示；immed_8r 为要传送到状态寄存器指定域的立即数，8 位；Rm 为要传送到状态寄存器指定域的数据源寄存器。

（2）指令举例

```
MSR  CPSR_c，#0xD3   ;CPSR[7:0]=0xD3，切换到管理模式
MSR  CPSR_cxsf，R3   ;CPSR=R3
```

只有在特权模式下才能修改状态寄存器。

程序中不能通过 MSR 指令直接修改 CPSR 中的 T 位控制位来实现 ARM 状态/Thumb 状态的切换，必须使用 BX 指令来完成处理器状态的切换（因为 BX 指令属转移指令，它会打断流水线状态，实现处理器状态的切换）。MRS 与 MSR 配合使用，实现 CPSR 或 SPSR 寄存器的读—修改—写操作，可用来进行处理器模式切换及允许/禁止 IRQ/FIQ 中断等设置。

2.8.6.3 程序状态寄存器指令的应用

【例 2-4】 使能 IRQ 中断。

```
ENABLE_IRQ
        MRS    R0，CPSR
        BIC     R0，R0，#0x80
        MSR    CPSR_c，R0
        MOV    PC，LR
```

【例 2-5】 禁止 IRQ 中断。

```
DISABLE_IRQ
       MRS   R0,CPSR
       ORR   R0,R0,#0x80
       MSR   CPSR_c,R0
       MOV   PC,LR
```

【例 2-6】 堆栈指令初始化。

```
INITSTACK
```

```
        MOV   R0,LR              ;保存返回地址
;设置管理模式堆栈
        MSR   CPSR_c,#0xD3
        LDR   SP,StackSvc
;设置中断模式堆栈
        MSR   CPSR_c,#0xD2
        LDR   SP,StackSvc
```

2.8.7 协处理器指令

ARM 体系结构允许通过增加协处理器来扩展指令集。最常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 Cache 和存储管理单元的 CP15 寄存器。此外，还有用于浮点运算的浮点 ARM 协处理器，各生产商还可以根据需要开发自己的专用协处理器。

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

程序的控制流指令由 ARM 处理器来处理，所有协处理器指令只能同数据处理和数据传送有关。按照 RISC 的 Load/Store 体系原则，数据的处理和传送指令是被清楚分开的，所以它们有不同的指令格式。

ARM 处理器支持 16 个协处理器，在程序执行过程中，每个协处理器忽略 ARM 和其他协处理器指令。当一个协处理器硬件不能执行属于它的协处理器指令时，将产生一个未定义指令异常中断，在该异常中断处理过程中，可以通过软件仿真该硬件操作。如果一个系统中不包含向量浮点运算器，则可以选择浮点运算软件包来支持向量浮点运算。

ARM 协处理器可以部分地执行一条指令，然后产生中断。如除法运算除数为 0 和溢出，这样可以更好地处理运行时产生（run-time-generated）的异常。但是，指令的部分执行是由协处理器完成的，此过程对 ARM 来说是透明的。当 ARM 处理器重新获得执行时，它将从产生异常的指令处开始执行。

对某一个协处理器来说，并不一定用到协处理器指令中的所有的域。具体协处理器如何定义和操作完全由协处理器的制造商自己决定，因此，ARM 协处理器指令中的协处理器寄存器的标识符及操作助记符也有各种不同的实现定义。程序员可以通过宏定义这些指令的语法格式。

ARM 协处理器指令可分为以下 3 类。

- 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操作，它完成协处理器寄存器的状态改变。如浮点加运算，在浮点协处理器中两个寄存器相加，结果放在第 3 个寄存器中。这类指令包括 CDP 指令。
- 协处理器数据传送指令。这类指令从寄存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装入存储器。因为协处理器可以支持自己的数据类型，所以每个寄存器传送的字数与协处理器有关。ARM 处理器产生存储器地址，但传送的字节由协处理器控制。这类指令包括 LDC 指令和 STC 指令。
- 协处理器寄存器传送指令。在某些情况下，需要 ARM 处理器和协处理器之间传送数据。如一个浮点运算协处理器，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，比较结果必须传送到 ARM 的 CPSR 中。这类协处理器寄存器传送指令包括 MCR 和 MRC。

表 2-21 列出了所有协处理器处理指令。

表 2-21　　协处理器指令

助　记　符	操　　作
CDP	协处理器数据操作
LDC	装载协处理器寄存器
MCR	从 ARM 寄存器传数据到协处理器寄存器
MRC	从协处理器寄存器传数据到 ARM 寄存器
STC	存储协处理器寄存器

2.8.8 异常产生指令

ARM 指令集中提供了两条产生异常的指令，通过这两条指令可以用软件的方法实现异常。表 2-22 为 ARM 异常产生指令。

表 2-22 ARM 异常产生指令

助 记 符	含 义	操 作
SWI	软中断指令	产生软中断，处理器进入管理模式
BKPT	断点中断指令	处理器产生软件断点

2.8.1.1 软中断指令

软件中断指令（Software Interrupt，SWI）用于产生软中断，从而实现从用户模式变换到管理模式，CPSR 保存到管理模式的 SPSR 中，执行转移到 SWI 向量，在其他模式下也可以使用 SWI 指令，处理器同样切换到管理模式。

（1）指令的语法格式

```
SWI{<cond>}   <immed_24>
```

（2）指令举例

① 下面指令产生软中断，中断立即数为 0。

```
SWI   0;
```

② 产生软中断，中断立即数为 0x123456。

```
SWI   0x123456;
```

③ 使用 SWI 指令时，通常使用以下两种方法进行参数传递。

- 指令 24 位的立即数指定了用户请求的类型，中断服务程序的参数通过寄存器传递。

下面的程序产生一个中断号为 12 的软中断。

```
MOV   R0，#34    ;设置功能号为 34
SWI   12 ;产生软中断，中断号为 12
```

- 另一种情况，指令中的 24 位立即数被忽略，用户请求的服务类型由寄存器 R0 的值决定，参数通过其他寄存器传递。

下面的例子通过 R0 传递中断号，R1 传递中断的子功能号。

```
MOV   R0，#12              ;设置 12 号软中断
MOV   R1，#34              ;设置功能号为 34
SWI    0                   ;
```

④ 在 SWI 异常中断处理程序中，取出 SWI 立即数的步骤为：首先确定引起软中断的 SWI 指令是 ARM 指令还是 Thumb 指令，这可通过对 SPSR 访问得到；然后要确定该 SWI 指令的地址，这可通过访问 LR 寄存器得到；最后读出指令，分解立即数。

下面的例子为一个标准的 SWI 中断处理程序。

```
T_bit     EQU     0x20
SWI_Hander
          STMFD  SP!,{R0_R3,R12,LR}   ;保护现场
          MOV        R1，sp               ;设置参数指针
          MRS        R0，SPSR             ;读取 SPSR
          STMFD  SP!,{R0,R3}          ;保持 SPSR，R3 压栈保证字节对齐
          TST        R0,#T_bit            ;测试 T 标志位
          LDRNEH R0,[LR,#-2]          ;若为 Thumb 指令，读取指令码（16 位）
          BICNE  R0,R0,#0xff00            ;取得 Thumb 指令 8 位立即数
```

```
        LDREQ  R0,[LR,#-4]          ;若为 ARM 指令，读取指令码（32 位）
        BICNQ  R0,R0,#0xff00000     ;取得 ARM 指令的 24 位立即数
        ; R0 存储中断号
        ; R1 指向栈顶

    BL     C_SWI_Handler            ;调用主要的中断服务程序
    LDMFD   sp!, {R0, R3}               ;SPSR 出栈
    MSR     spsr_cf, R0              ;恢复 SPSR
    LDMFD   sp!, {R0-R3, R12, pc}^      ;保存寄存器并返回
```

中断服务程序的主要工作放在 C_SWI_Handler 中，由 C 语言完成，用 swich_case 结构判断中断类型。典型的程序如下。

```
void C_SWI_Handler( int swi_num, int *regs )
{
     switch( swi_num )
     {
     case   0:
          regs[0] = regs[0] * regs[1];
     break;

     case   1:
          regs[0] = regs[0] + regs[1];
     break;

     case   2:
          regs[0] = (regs[0] * regs[1]) + (regs[2] * regs[3]);
     break;

     case   3:
     {
          int w, x, y, z;

          w = regs[0];
          x = regs[1];
          y = regs[2];
          z = regs[3];

          regs[0] = w + x + y + z;
          regs[1] = w - x - y - z;
          regs[2] = w * x * y * z;
          regs[3] =(w + x) * (y - z);
     }
     break;

     }
}
```

2.8.8.2 断点中断指令

断点中断指令（BreakPoint，BKPT）产生一个预取异常（Prefetch Abort），它常被用来设置软件断点，在调试程序时十分有用。当系统中存在调试硬件时，该指令被忽略。

指令格式如下：

```
BKPT    <immediate>
```

要正确地使用 BKPT 指令，必须和具体的调试系统相结合。一般来说，BKPT 有两种使用方法。

（1）如果当前使用的系统调试硬件没有屏蔽 BKPT 指令，那么在此系统中预取指令异常和软件调试命令同时使用一个中断向量。这样当异常发生时，就要依靠系统自身来判断是真正地预取异常还是软件调试命令。也根据系统的不同，判断的方法也有所不同。

（2）如果当前的系统调试硬件屏蔽了 BKPT 指令，那么系统会跳过 BKPT 指令顺序执行该指令下面的程序代码。

2.9 本 章 小 结

本章对 ARM 处理器的体系结构、寄存器组织、流水线、ARM 存储、异常、ARM 处理器的寻址方式、ARM 处理器的指令集等内容进行了介绍，这些内容是 ARM 处理器理论的基本内容，是系统软硬件设计的基础。

2.10 思 考 题

1．说出 ARM 可以工作的模式的名字？
2．ARM 核有多少个寄存器？
3．什么寄存器用于存储 PC 和 LR 寄存器？
4．R13 通常用来存储什么？
5．哪种模式使用的寄存器最少？
6．CPSR 的哪一位反映了处理器的状态？
7．ARM 有哪几个异常类型
8．复位后，ARM 处理器处于何种模式、何种状态？
9．BIC 指令有什么作用？
10．当执行 SWI 指令时，会发生什么？

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-25590506

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218

广州地址：广州市天河区中山大道 268 号天河广场 3 层，电话：020-28916067