《ARM 嵌入式体系结构与接口技术》

作者：华清远见

第 3 章 ARM 微处理器的指令系统

本章目标

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。本章主要介绍利用 ARM 汇编语言。主要内容如下：

- ARM 处理器的寻址方式
- ARM 处理器的指令集

3.1 ARM 处理器的寻址方式

ARM 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。

3.1.1 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 3-1 所示。

表 3-1 <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）。如图 3-1 所示

图 3-1 立即数表示方法

下面列举了一些有效的立即数：

0xFF、0x104、0xFF0、0xFF00、0xFF000、0xFF000000、0xF000000F

下面是一些无效的立即数：

0x101、0x102、0xFF1、0xFF04、0xFF003、0xFFFFFFFF、0xF000001F

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,#0xFF00      ;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
```

3.1.2　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。

1．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 3-2 所示。

表 3-2　　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）

上表中：“!”表示完成数据传输后要更新基址寄存器。

2．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 3-3 所示。

表 3-3　　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）

3．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 3-4 所示。

表 3-4　　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

4．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 3-5 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 3-5　　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

5．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>
```

3.2　ARM 处理器的指令集

3.2.1　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 3-6 所示。

表 3-6　　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）

1．MOV 指令

MOV 指令是最简单的 ARM 指令，执行的结果就是把一个数 N 送到目标寄存器 Rd，其中 N 可以是寄存器，也可以是立即数。

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。

（1）指令的语法格式

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

（2）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
```

（3）指令的使用

MOV 指令主要完成以下功能。

① 将数据从一个寄存器传送到另一个寄存器。

② 将一个常数值传送到寄存器中。

③ 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。

④ 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”。

⑤ 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．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。

（1）指令的语法格式

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

（2）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
```

（3）指令的使用

MVN 指令主要完成以下功能：

① 向寄存器中传送一个负数。

② 生成位掩码（Bit Mask）。

③ 求一个数的反码。

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。

（1）指令的语法格式

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

（2）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
```

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。

（1）指令的语法格式

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

（2）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
```

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。

（1）指令的语法格式

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

（2）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
```

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。

（1）指令的语法格式

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

（2）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
```

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。

（1）指令的语法格式

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

（2）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)
```

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。

（1）指令的语法格式

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

（2）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
```

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。

（1）指令的语法格式

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

（2）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
```

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。

（1）指令的语法格式

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

（2）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
```

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。

（1）指令的语法格式

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

（2）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
```

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“逻辑异或”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。

（1）指令的语法格式

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

（2）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。

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。

（1）指令的语法格式

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

（2）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（其值可能为大、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。

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。

（1）指令的语法格式

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

（2）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
```

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。

（1）指令的语法格式

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

（2）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
```

16．BIC 位清零指令

BIC（Bit Clear）位清零指令，将寄存器 Rn 的值与第 2 个源操作数 shifter_operand 的值的反码按位做“逻辑与”操作，结果保存到 Rd 中。

（1）指令的语法格式

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

（2）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
```

3.2.2 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 3-7 为各种形式乘法指令的功能。

表 3-7　　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。

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。

（1）指令的语法格式

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

（2）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．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。

（1）指令的语法格式

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

（2）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
```

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。

（1）指令的语法格式

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

（2）指令举例

下面指令完成(R1，R0) = R5 × R8 操作。

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

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。

（1）指令的语法格式

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

（2）指令举例

下面的指令完成(R1，R0) = R5 × R8+(R1，R0)操作。

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

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。

（1）指令的语法格式

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

（2）指令举例

下面的指令完成(R3，R2) = R7 × R6 操作。

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

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。

（1）指令的语法格式

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

（2）指令举例

下面的指令完成(R3，R2) = R7 × R6 +(R3，R2)操作。

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

3.2.3 Load/Store 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。

3.2.3.1 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 3-8 列出了所有单寄存器的 Load/Store 指令。

表 3-8　　单寄存器 Load/Store 指令

指　令	作　用	操　作

LDR	把存储器中的一个字装入一个寄存器	Rd←mem32[address]
STR	将寄存器中的字保存到存储器	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]}

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。

（1）指令的语法格式

LDR{<cond>}　<Rd>,<addr_mode>

（2）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．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。

（1）指令的语法格式

STR{<cond>}　<Rd>,<addr_mode>

（2）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
            …
```

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>

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>

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>

6．STRH 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>

3.2.3.2 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 3-9 列出了多寄存器的 Load/Store 内存访问指令。

表 3-9　　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]

1．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 Ri 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．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 >^

3、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，其中前面 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，然后使用块复制寻址指令 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 3-10 所示。

表 3-10 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

【举例】 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
```

【举例】 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
```

3.2.3.3 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 3-11 所示。

表 3-11 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。

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
```

3.2.4　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 3-12 所示。

表 3-12　　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。

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复制到 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。

（1）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。

① BX　R14（如果体系结构支持 BX 指令）。

② MOV　PC，R14。

③ 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]。

（2）程序举例

（1）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．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 位复制到 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。

（1）指令的语法格式

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

① 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。

② 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
```

（2）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;
```

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。

（1）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。

（2）指令的使用

① 从 Thumb 状态返回到 ARM 状态，使用 BX 指令。

```
BX  R14
```

② 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}
```

3.2.5 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 3-13 所示。

表 3-13　　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。

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。

（1）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。

（2）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．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

```
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。

3．程序状态寄存器指令的应用

【举例】 使能 IRQ 中断。

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

【举例】 禁止 IRQ 中断。

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

【举例】 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
```

3.2.6 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。

（1）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。

（2）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。

（3）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 3-14 列出了所有协处理器处理指令。

表 3-14　　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

3.2.7　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 3-15 为 ARM 异常产生指令。

表 3-15　　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

1．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．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。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。

3.3 本 章 小 结

本章在第 2 章的基础上，介绍了 ARM 处理器的寻址方式及 ARM 处理器的指令集。ARM 处理器的寻址方式包括：数据处理指令寻址方式和内存访问指令寻址方式；ARM 处理器的指令集包括：数据操作指令、乘法指令、load/store 指令、跳转指令、状态操作指令、协处理器指令、异常产生指令。

3.4 思 考 题

3-1 用 ARM 汇编实现下面列出的操作

a) r0=15

b) r0=r1/16（有符号数）

c) r1=r2*3

d) r0=-r0

3-2 BIC 指令的作用是？

3-3 执行 SWI 指令时会发生什么？

3-4 B、BL、BX 指令的区别？

3-5 下面哪个数据可以作为数据操作指令的有效立即数

a) 0x101 b）0x1f8 c)0xf000000f d)0x08000012 e) 0x104

3-6 ARM 在哪些工作工作模式下可以修改 CPSR 寄存器？

3-7 写一个程序，判断 R0 的值大于 0x50，则将 R1 的值减去 0x10，并把结果送给 R0。

3-8 编写一段 ARM 汇编程序，实现数据块复制，将 R0 指向的 8 个字的连续数据保存到 R1 指向的一段连续的内存单元。

联系方式