《ARM 系列处理器应用技术完全手册》

作者：华清远见

第 5 章 数据传送指令

专业始于专注 卓识源于远见

5.1 MOV 指令

1．指令编码格式

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。
指令的编码格式如图 5.1 所示。

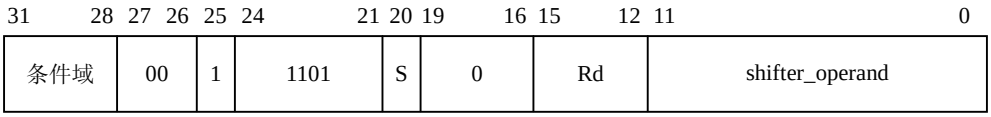


图 5.1 MOV 指令编码格式

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。

2．指令的语法格式

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

① <cond>
为指令编码中的条件域。它指示 MOV 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。
② S（bit[20]）
如果 S=1，MOV 指令更新 CPSR 中条件标志位的值；如果 S=0，MOV 指令不更新 CPSR 中条件标志位的值。当更新状态寄存器 CPSR 中的条件标志位时，有两种情况。

- 如果指令中的目标寄存器<Rd>不是为 r15，指令根据传送的数值设置 CPSR 中的 N 位和 Z 位（如果数据在传送前需要移位，则根据移位后的数值设置），并根据移位器的进位值设置 CPSR 的 C 位。标志位 V 和其他位不受影响。
- 如果指令中的目标寄存器<Rd>为 r15，则当前处理器模式对应的 SPSR 的值复制到 CPSR 寄存器中，对于用户模式和系统模式，由于没有相应的 SPSR，指令执行的结果不可预知。

③ <Rd>
确定目标寄存器。
④ <shifter_operand>
确定操作数，为目标寄存器传送数据。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
   Rd=shifter_operand
   If  S==1 and Rd==r15  then
     CPSR=SPSR
```

```
Else  if  S==1  then
    N Flag = Rd[31]
    Z Flag = If  Rd==0  then 1  else 0
    C Flag = shifter_carry_out
    V Flag = unaggected
```

4．指令举例

【例 5.1】MOV 指令
MOV 指令把一个数 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。

```
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
MOVS   PC, R14             ; 退出到调用者并恢复标志位
```

5．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 2^n 可以用左移 n 位来实现。
- 当 PC 寄存器（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以及从子程序返回，代替指令“B，BL”。

注意　在体系结构 v4 和 v5 以上的版本，必须使用 BX 指令代替 MOV　PC，LR 指令，因为 BX 指令可以自动实现 ARM 和 Thumb 的切换。

- 当 PC 寄存器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。

5.2　MVN 指令

1．指令编码格式

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。
指令的编码格式如图 5.2 所示。

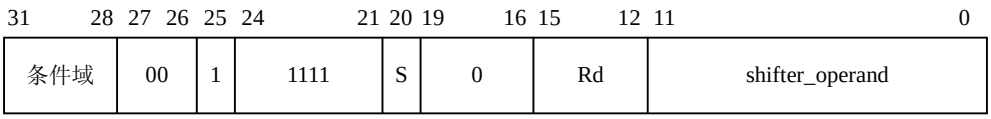


图 5.2　MVN 指令编码格式

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。

2．指令的语法格式

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

① <cond>

为指令编码中的条件域。它指示 MVN 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② S（bit[20]）

如果 S=1，MVN 指令更新 CPSR 中条件标志位的值；如果 S=0，MVN 指令不更新 CPSR 中条件标志位的值。当更新状态寄存器 CPSR 中的条件标志位时，有两种情况。

- 如果指令中的目标寄存器<Rd>不是为 r15，指令根据传送的数值设置 CPSR 中的 N 位和 Z 位（如果数据在传送前需要移位，则根据移位后的数值设置），并根据移位器的进位值设置 CPSR 的 C 位。标志位 V 和其他位不受影响。
- 如果指令中的目标寄存器<Rd>为 r15，则当前处理器模式对应的 SPSR 的值复制到 CPSR 寄存器中，对于用户模式和系统模式，由于没有相应的 SPSR，指令执行的结果不可预知。

③ <Rd>

确定目标寄存器。

④ <shifter_operand>

确定操作数，为目标寄存器传送数据。

3．指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
   Rd = NOT(shifter_operand)
   If  S==1 and Rd==r15  then
      CPSR = SPSR
   Else  if  S==1  then
      N Flag = Rd[31]
      Z Flag = If  Rd==0  then 1  else 0
      C Flag = shifter_carry_out
      V Flag = unaggected
```

4．指令举例

【例 5.2】MVN 指令

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。

注意 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
```

5．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。

5.3 单寄存器的 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拷贝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。

5.3.1 字数据传送指令

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 5.1 总结了所有单寄存器的 Load/Store 指令。

表 5.1　　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]}

1．LDR 指令

（1）指令编码格式

LDR 指令用于从内存中将一个 32 位的字读取到目标寄存器。

指令的编码格式如图 5.3 所示。

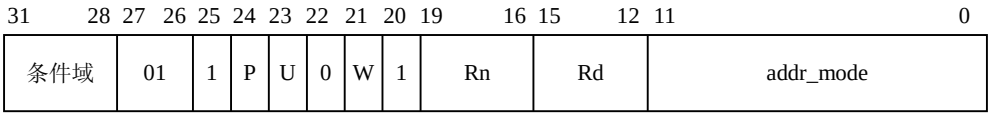


图 5.3　LDR 指令编码格式

LDR 指令根据<addr_mode>所确定的地址模式将一个 32 位字读取到指令中的目标寄存器<Rd>。如果指令中的寻址方式确定的地址不是字对齐的，则读出的数值要进行循环右移。所移位数为寻址方式确定的地址 bits[1：0]8 的倍，也就是说处理器将取到的数值作为字的最低位处理。

如果设置了 L 位，则进行装载，否则进行存储。

如果设置了 P 位，则使用预先变址寻址，否则使用过后变址寻址。

如果设置了 U 位，则给出的偏移量被加到基址寄存器上，否则从中减去偏移量。

如果设置了 B 位，传送内存的一个字节，否则传送一个字。这在助记符末尾添加后缀“B”，如 MOV r7，r5 变为 MOVB r7，r5。

W 位的解释依赖于使用的地址模式。

- 对于预先变址寻址，设置 W 位强制把它用做地址转换的最终地址写回基址寄存器中（例如，传送的副作用是 Rn:= Rn +/−offset。这在汇编器中表示为给指令加上后缀“!”。)。
- 对于过后变址寻址，地址总是写回，设置 W 位指示在进行传送之前强制地址转换。这在汇编器中表示为给指令加上后缀“T”。

当 PC 作为 LDR 的目的寄存器<Rd>时，从存储器取得的数据将被当作目标地址值，程序将跳转到目标地址开始执行。

（2）指令的语法格式

```
LDR{<cond>}  <Rd>，<addr_mode>
```

① <cond>

为指令编码中的条件域。它指示 LDR 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <addr_mode>

它确定了指令编码中的 I、P、U、W、Rn 和<addr_mode>位。所有的寻址模式中，都会确定一个基址寄存器 Rn。

（3）指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
   If  address[1:0] == 0b00  then
       Value = Memory[address,4]
Else  if  address[1:0] == 0b01  then
    Value = Memory[address,4]  Rotate_Right 8
Else  if  address[1:0] == 0b10  then
    Value = Memory[address,4]  Rotate_Right 16
Eles  /* address[1:0] == 0b11*/
```

```
        Value = Memory[address,4]  Rotate_Right 24
    If  (Rd is R15)  then
        If (architecture version 5 or above)  then
               PC = value AND 0xfffffffe
               T Bit = value[0]
        Else
               PC = value AND 0xfffffffc
    Else
        Rd = value
```

（4）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±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
```

注意 地址对齐问题：大多数情况下，必须保证用于 32 位传送的地址是 32 位对齐的。

2．STR 指令

（1）指令编码格式

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。

指令的编码格式如图 5.4 所示。

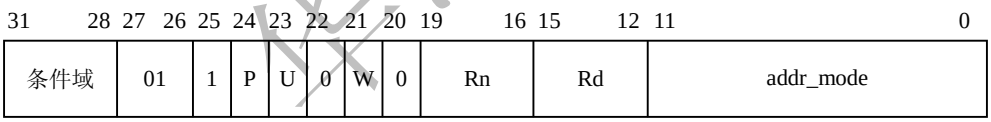


图 5.4　STR 指令编码格式

（2）指令的语法格式

```
STR{<cond>}  <Rd>，<addr_mode>
```

① <cond>

为指令编码中的条件域。它指示 STR 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <addr_mode>

它确定了指令编码中的 I、P、U、W、Rn 和<addr_mode>位。所有的寻址模式中，都会确定一个基址寄存器 Rn。

（3）指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Memory[address，4]=Rd
```

（4）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
           …
```

5.3.2 字节数据传送指令（LDRB/STRB）

1．LDRB 指令

（1）指令编码格式

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>。指令的编码格式如图 5.5 所示。

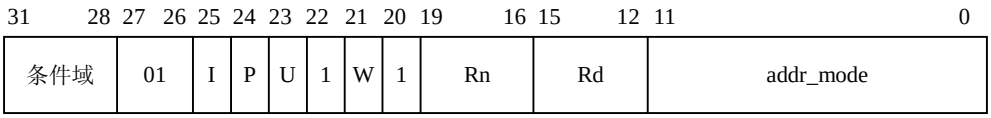


图 5.5 LDRB 指令编码格式

注意 LDRB 指令加载一个内存地址的 8 位字节到一个通用寄存器中。寄存器的高位数据补 0。

（2）指令的语法格式

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

① <cond>

为指令编码中的条件域。它指示 LDRB 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <addr_mode>

它确定了指令编码中的 I、P、U、W、Rn 和<addr_mode>位。所有的寻址模式中，都会确定一个基址寄存器 Rn。

（3）指令操作的伪代码

```
if ConditionPassed{cond} then
   Rd = Memory[address,1]
```

注意 当 PC 作为位基地址出现在指令中时，指令中将会使用 PC 相关地址，使用这种方法可以编写自己的位置无关（position-independ）指令。

2．STRB 指令

（1）指令编码格式

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。指令的编码格式如图 5.6 所示。

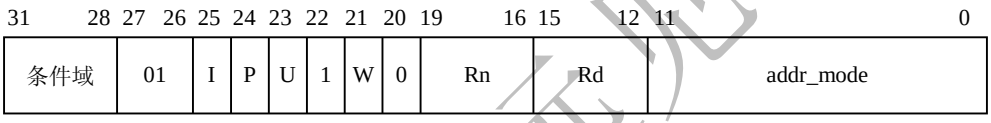


图 5.6 STRB 指令编码格式

（2）指令的语法格式

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

① <cond>

为指令编码中的条件域。它指示 STRB 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

注意 当 PC 作为目标寄存器<Rd>出现在指令中时，指令的执行结果不可预知。

③ <addr_mode>

它确定了指令编码中的 I、P、U、W、Rn 和<addr_mode>位。所有的寻址模式中，都会确定一个基址寄存器 Rn。

（3）指令操作的伪代码

```
if ConditionPassed{cond} then
   Memory[address,1] = Rd[7:0]
```

5.3.3 半字数据传送指令（LDRH/STRH）

1．LDRH 指令

（1）指令编码格式

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。

指令的编码格式如图 5.7 所示。

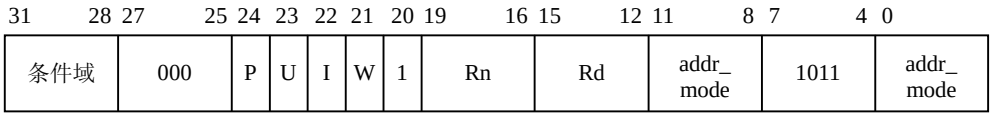


图 5.7　LDRH 指令的编码格式

（2）指令的语法格式

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

① <cond>

为指令编码中的条件域。它指示 LDRH 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

注意　如果 PC 作为目标寄存器，指令的执行结果不可预知。

③ <addr_mode>

它确定了指令编码中的 I、P、U、W、Rn 和<addr_mode>位。所有的寻址模式中，都会确定一个基址寄存器 Rn。

（3）指令操作的伪代码

```
if  ConditionPassed{cond}  then
   if address[0]==0
       data=Memory[address,2]
   else  /*address[0]==1*/
       data=unpredictable
   Rd=data
```

注意　在包含系统控制协处理器的芯片应用中，如果定义了地址对齐检测，当 bit[0]!=0 时，将发生地址对齐异常。

2．STRH 指令

（1）指令编码格式

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。

指令的编码格式如图 5.8 所示。

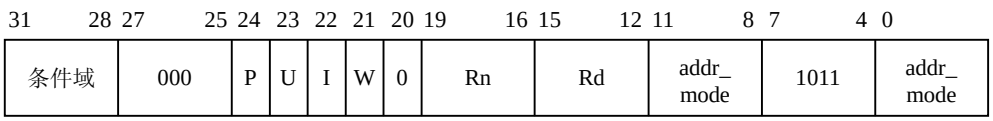


图 5.8　STRH 指令的编码格式

（2）指令的语法格式

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

① <cond>

指令编码中的条件域。它指示 STRH 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

注意 如果 PC 作为目标寄存器，指令的执行结果不可预知。

③ <addr_mode>

它确定了指令编码中的 I、P、U、W、Rn 和<addr_mode>位。所有的寻址模式中，都会确定一个基址寄存器 Rn。

（3）指令操作的伪代码

```
if  ConditionPassed{cond}  then
   if address[0]==0
       data=Rd[15:0]
   else  /*address[0]==1*/
       data=unpredictable
   Memory[address,2]=data
```

5.3.4 用户模式字数据传送指令（LDRT/STRT）

1．LDRT 指令

（1）指令编码格式

LDRT 指令用于从内存中将一个 32 位的字读取到目标寄存器。

指令的编码格式如图 5.9 所示。

LDRT 指令根据<addr_mode>所确定的地址模式将一个 32 位字读取到指令中的目标寄存器<Rd>。如果指令中的寻址方式确定的地址不是字对齐的，则读出的数值要进行循环右移。所移位数为寻址方式确定的地址 bits[1：0]的 8 倍。也就是说处理器将取到的数值作为字的最低位处理。

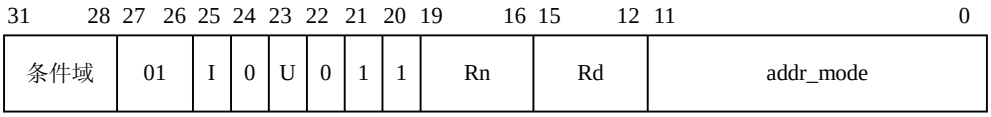


图 5.9 LDRT 指令编码格式

当处理器在特权模式下使用此指令时，内存系统将该操作当作一般用户模式下得内存访问指令。

注意 指令的编码格式中，P 位指定位 “0”，也就是说 LDRT 指令的寻址方式为固定寻址方式，即后索引编码寻址（post_indexed_addressing_mode）。

（2）指令的语法格式

```
LDR{<cond>}T  <Rd>，<post_indexed_addressing_mode>
```

① <cond>

为指令编码中的条件域。它指示 LDRT 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <post_indexex_address_mode>

使用后索引地址模式寻址。

> **注意** 后索引地址模式中 P=0 并且 W=0(即 bit[21]=0、bit[24]=0)。但此指令 P=0 并且 W=1(即 bit[21]=1、bit[24]=0)。但实际的寻址操作是一样的。

（3）指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
   If  address[1:0]==0b00
        Rd=Memory[address,4]
   Else  if  address[1:0]==0b01
        Rd=Memory[address,4]  Rotate_Right  8
   Else  if  address[1:0]==0b10
        Rd=Memory[address,4]  Rotate_Right  16
   Else  address[1:0]==0b11
        Rd=Memory[address,4]  Rotate_Right  24
```

2．STRT 指令

（1）指令编码格式

STRT 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。

当处理器在特权模式下执行此指令时，内存系统将该操作当作一般用户模式下的内存访问操作。

指令的编码格式如图 5.10 所示。

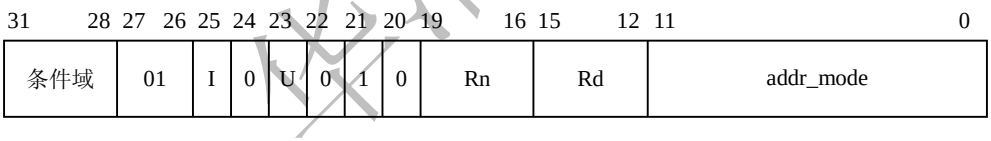


图 5.10　STR 指令编码格式

（2）指令的语法格式

```
STR{<cond>}T  <Rd>，<post_indexed_addressing_mode>
```

① <cond>

为指令编码中的条件域。它指示 STRT 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <post_indexed_address_mode>

使用后索引地址模式寻址，参见 LDRT 指令。

（3）指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
   Memory[address,4]=Rd
```

5.3.5 用户模式字节数据传送指令（LDRBT/STRBT）

1．LDRBT 指令

（1）指令编码格式

LDRBT 指令根据<post_indexed_addressing_mode>地址模式将一个 8 位字节读取到指令中的目标寄存器<Rd>。

当处理器在特权模式下执行此指令时，内存系统将该操作当作一般用户模式下的内存访问操作。

指令的编码格式如图 5.11 所示。

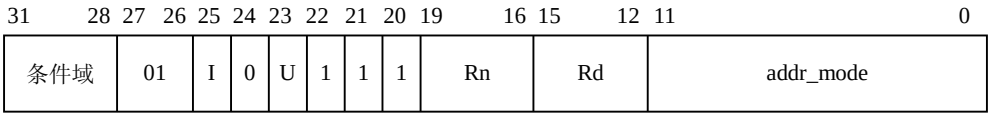


图 5.11 LDRBT 指令编码格式

注意 LDRBT 指令加载一个内存地址的 8 位字节到一个通用寄存器中。寄存器的高位数据补 0。

（2）指令的语法格式

```
LDR{<cond>}BT  <Rd>，<post_indexed_addressing_mode>
```

① <cond>

为指令编码中的条件域。它指示 LDRBT 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <post_indexed_addressing_mode>

使用后索引地址模式寻址，参见 LDRT 指令。

（3）指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
   Rd=Memory[address,1]
```

2．STRBT 指令

（1）指令编码格式

STRBT 指令用于将一个 8 位的字节数据写入到指令中指定的内存单元。

当处理器在特权模式下执行此指令时，内存系统将该操作当作一般用户模式下的内存访问操作。

指令的编码格式如图 5.12 所示。

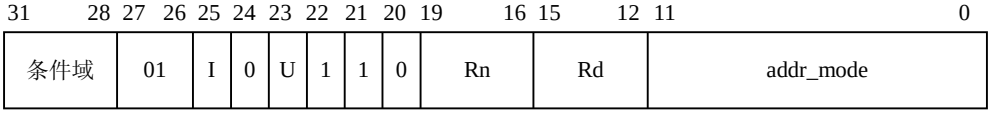


图 5.12 STRBT 指令编码格式

（2）指令的语法格式

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

① <cond>

为指令编码中的条件域。它指示 LDRBT 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <post_indexed_addressing_mode>

使用后索引地址模式寻址，参见 LDRT 指令。

（3）指令操作的伪代码

指令操作的伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Memory[address，1]=Rd[7:0]
```

5.3.6 有符号的字节/半字数据传送指令（LDRBT/STRBT）

1．LDRSB 指令

（1）指令编码格式

LDRSB 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>。

> **注意** LDRSB 与 LDRB 指令的不同之处在于它将寄存器的高 24 位设置成该字节数据的符号位的值（即将该 8 位字节数据进行符号位扩展，生成 32 位字数据）。

指令的编码格式如图 5.13 所示。

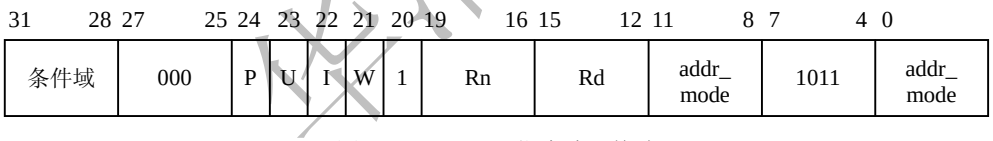


图 5.13 LDRSB 指令编码格式

（2）指令的语法格式

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

① <cond>

为指令编码中的条件域。它指示 LDRSB 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <addr_mode>

它确定了指令编码中的 I、P、U、W、Rn 和<addr_mode>位。所有的寻址模式中，都会确定一个基址寄存器 Rn。

（3）指令操作的伪代码

```
If  ConditionPassed{cond}  then
   data=Memory[address,1]
Rd=SignExtend{data}
```

2．LDRSH 指令

（1）指令编码格式

LDRSH 指令根据<addr_mode>所确定的地址模式将一个 16 位半字读取到指令中的目标寄存器<Rd>。

> **注意** LDRSH 与 LDRH 指令的不同之处在于它将寄存器的高 16 位设置成该字节数据的符号位的值（即将该 16 位字节数据进行符号位扩展，生成 32 位字数据）。

指令的编码格式如图 5.14 所示。

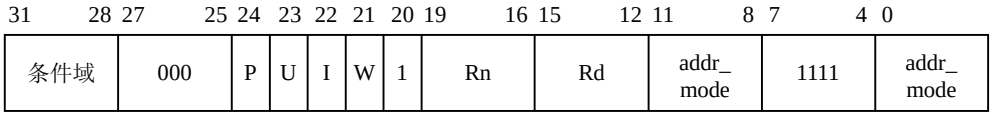


图 5.14　LDRSH 指令编码格式

（2）指令的语法格式

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

① <cond>

为指令编码中的条件域。它指示 LDRSH 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

确定使用哪个通用寄存器作为目标寄存器。

③ <addr_mode>

它确定了指令编码中的 I、P、U、W、Rn 和<addr_mode>位。所有的寻址模式中，都会确定一个基址寄存器 Rn。

（3）指令操作的伪代码

```
If  ConditionPassed{cond}  then
   if  address[0]==0
       data=Memory[address,2]
  else  /*address[0] ==1*/
       data=UNPREDICTABLE
  Rd=SignExtend{data}
```

5.4　多寄存器 Load/Store 内存访问指令

多寄存器 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 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 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。

> **注意** 多寄存器 Load/Store 内存访问指令会增加中断延时，因为 ARM 通常不会打断正在执行的指令去响应中断，而必须等到指令执行完。也就是说，如果一个中断在多寄存器 Load/Store 内存访问指令执行期间产生，那么处理器在多寄存器 Load/Store 内存访问指令执行完后才对中断响应。

表 5.2 总结了多寄存器 Load/Store 内存访问指令

表 5.2　　多寄存器 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]

5.4.1　多寄存器内存字数据传送指令

1．LDM（1）指令

（1）指令编码格式

LDM（1）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。

指令的编码格式如图 5.15 所示。

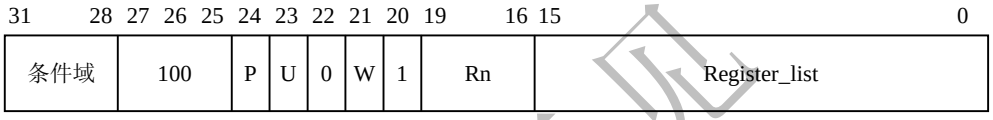


图 5.15　LDM（1）指令编码格式

（2）指令的语法格式

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

① <cond>

为指令编码中的条件域。它指示 LDM（1）指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <address_mode>

指令的寻址方式。确定编码格式中的 P、U 和 W 位。

③ <Rn>

确定寻址模式所使用的基址寄存器。

如果 r15 作为指令的基址寄存器，指令的执行结果不可预知。

④ ！

设置指令编码格式中的 W 位。它使指令执行后将操作数的内存地址写入基址寄存器<Rn>中；如果！被忽略，W 位为 0，指令执行完后，不修改基址寄存器的值。

注意　如果基址寄存器包含在指令列表中，当指令执行完后，基址寄存器的值是新加载进的特定内存地址的值。也就是说，即使指令没有出现在指令列表中，基址寄存器的值也可能被修改。

⑤ <registers>

被加载的寄存器列表。不同的寄存器之间用“,”隔开。完整的寄存器列表包含在“{}”中。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高地址单元。

注意　无论寄存器在寄存器列表“{}”中如何排列，都将遵循上述规则。

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。

（3）指令操作的伪代码

指令操作伪代码如下面程序段所示。

```
If  ConditionPass{cond}  then
    Address=start_address

    For  i=0  to 14
       If  register_list[i]==1  then
          Ri=Memory[address,4]
          Address=address+4

    If  register_list[15]==1  then
          Value = Memory[address,4]
          If(architecture version 5 or above)  then
               Pc= value  AND  0xfffffffe
               T bit=value[0]
          Else
               Pc= value  AND  0xfffffffc
          Address=address+4
    Assert end_address=address+4
```

2．STM（1）指令

（1）指令编码格式

STM（1）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、数据栈操作以及进入子程序时保存相关寄存器的操作。

指令编码格式如图 5.16 所示。

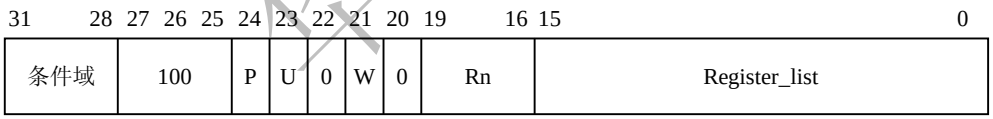


图 5.16　STM（1）指令编码格式

（2）指令的语法格式

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

① <cond>

为指令编码中的条件域。它指示 STM（1）指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <address_mode>

指令的寻址方式。确定编码格式中的 P、U 和 W 位。

③ <Rn>

确定寻址模式所使用的基址寄存器。

如果 r15 作为指令的基址寄存器，指令的执行结果不可预知。

④ ！

设置指令编码格式中的 W 位。它使指令执行后将操作数的内存地址写入基址寄存器<Rn>中；如果！被忽略，W 位为 0，指令执行完后，不修改基址寄存器的值。

⑤ <registers>

被加载的寄存器列表。不同的寄存器之间用“,”隔开。完整的寄存器列表包含在“{}”中。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高地址单元。

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。

（3）指令操作的伪代码

指令操作伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Address=Start_address
    For  i=0  to  15
         If  register_list[i]==1
         Memory[address,4]=Ri
         Address=address+4
    Assert  end_address==address-4
```

5.4.2 用户模式多寄存器内存字数据传送指令

1．LDM（2）指令

（1）指令编码格式

LDM（2）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。

注意 与 LDM（1）指令不同，PC 不能包含在寄存器列表中。

指令的编码格式如图 5.17 所示。

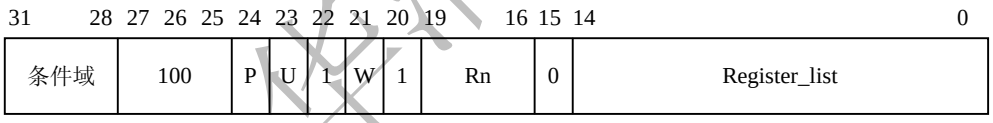


图 5.17 LDM（2）指令编码格式

（2）指令的语法格式

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

① <cond>

为指令编码中的条件域。它指示 LDM（2）指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <address_mode>

指令的寻址方式。确定编码格式中的 P 位和 U 位。此指令中 W 位指定为 0。

③ <Rn>

确定寻址模式所使用的基址寄存器。

如果 r15 作为指令的基址寄存器，指令的执行结果不可预知。

④ <registers_without_pc>^

被加载的寄存器列表。不同的寄存器之间用“,”隔开。完整的寄存器列表包含在“{}”中。此寄存器列表中不能包含 PC 寄存器。

如果 PC 包含在寄存器列表中，指令的执行结果不可预知。

其他细节可参考 LDM（1）指令。

（3）指令操作的伪代码

指令操作伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Address=start_address
    For  i=0  to  14
        If  register_list[i]==1
             Ri_usr=Memory[address,4]
             Address=address+4
    Assert  end_address == address-4
```

2．STM（2）指令

（1）指令编码格式

STM（2）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、数据栈操作以及进入子程序时保存相关寄存器等操作。

指令编码格式如图 5.18 所示。

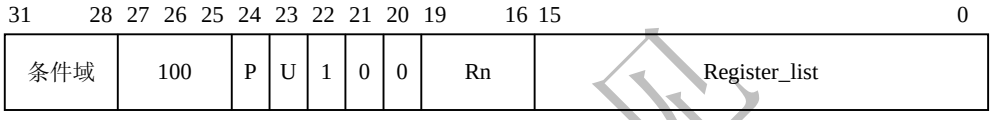


图 5.18　STM（2）指令编码格式

（2）指令的语法格式

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

① <cond>

为指令编码中的条件域。它指示 LDM（2）指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <address_mode>

指令的寻址方式。确定编码格式中的 P 位和 U 位。此指令中 W 位指定为 0。

③ <Rn>

确定寻址模式所使用的基址寄存器。

如果 r15 作为指令的基址寄存器，指令的执行结果不可预知。

④ <registers >^

寄存器列表。只能使用用户模式下的寄存器。

（3）指令操作的伪代码

指令操作伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Address=start_address
    For  i=0  to  15
         If  register_list[i] == 1
              Memory[address,4]=Ri_usr
              Address = address +4
     Assert  end_address == address-4
```

5.4.3　带状态寄存器的多寄存器内存字数据装载指令（LDM（3））

（1）指令编码格式

LDM（3）指令将数据从连续的内存单元中读取数据到寄存器列表中的各寄存器中。它同时将当前处理器模式对应的 SPSR 寄存器的内容复制到 CPSR 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。

在 ARM v5 及以上的版本和 T 系列的 ARM v4 版本中，SPSR 寄存器的 T 位将复制到 CPSR 寄存器的 T 位，该位决定目标地址处的程序状态。在以前的版本中程序继续执行在 ARM 状态。

指令的编码格式如图 5.19 所示。

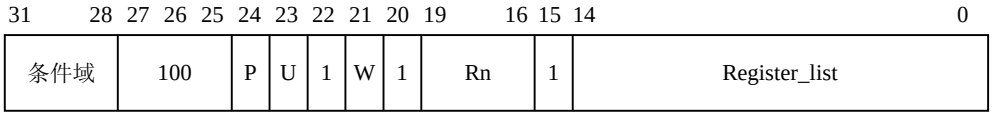


图 5.19　LDM（3）指令编码格式

（2）指令的语法格式

```
LDM{<cond>}<addressing_mode>  <Rn>{!},  <registers_and_pc>^
```

① <cond>

为指令编码中的条件域。它指示 LDM（1）指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <address_mode>

指令的寻址方式。确定编码格式中的 P、U 和 W 位。

③ <Rn>

确定寻址模式所使用的基址寄存器。

如果 r15 作为指令的基址寄存器，指令的执行结果不可预知。

④ ！

设置指令编码格式中的 W 位。它使指令执行后将操作数的内存地址写入基址寄存器<Rn>中；如果！被忽略，W 位为 0，指令执行完后，不修改基址寄存器的值。

注意　如果基址寄存器包含在指令列表中，当指令执行完后，基址寄存器的值是新加载进的特定内存地址的值。也就是说，即使指令没有出现在指令列表中，基址寄存器的值也可能被修改。

⑤ <registers_and_pc>^

寄存器列表。

注意　在本格式的指令中寄存器列表中必须包含 PC 寄存器。

被加载的寄存器列表。不同的寄存器之间用“,”隔开。完整的寄存器列表包含在“{}”中。编号低的寄存器对应于内存中的低地址单元，编号高的寄存器对应于内存中的高地址单元。

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。

该指令执行时将当前处理器模式下的 SPSR 值复制到 CPSR 中。指令的其他参数可参见 LDM（1）指令格式。

（3）指令操作的伪代码

指令操作伪代码如下面程序段所示。

```
If  ConditionPass{<cond>}  then
    Address=start_address
```

```
For i=0  to  14
     If  register_list[i]==1 then
          Ri=Memory[address,4]
          Address=address+4
CPSR=SPSR
Value=memory[address,4]
If {architecture version 4T, 5 or above} and {T bit ==1} then
Else
     Pc=value AND oxfffffffc
Address=address + 4
Assert end_address=address-4
```

5.4.4 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，如下所示。
前面 4 种用于数据块的传输，后面 4 种是堆栈操作。
（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，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 拷贝到 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。

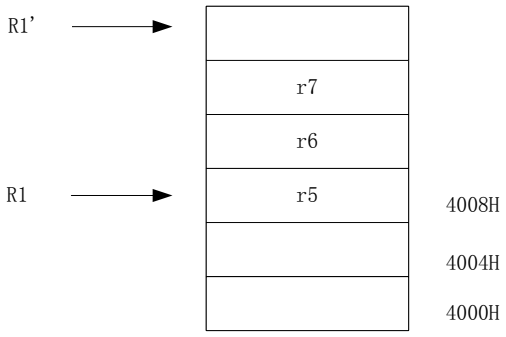
注意 地址对齐问题，在这些指令中，忽略地址位[1:0]。
批量加载/存储指令举例如下。

```
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
```

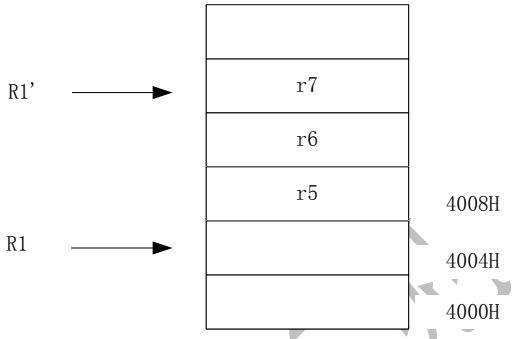
在进行数据复制时，先设置好源数据指针，然后使用块拷贝寻址指令 LDMIA/STMIA、LDMIB/STMIB、LDMDA/STMDA、LDMDB/STMDB 进行读取和存储。而进行堆栈操作时，则要先设置堆栈指针，一般使用 SP 然后使用堆栈寻址指令 STMFD/LDMFD、STMED/LDMED、STMEA/LDMEA 实现堆栈操作。
多寄存器传送指令如例 5.3 所示。其中 r1 为指令执行前的基址寄存器，r1’则为指令执行后的基址寄存器。

【例 5.3】多寄存器传送指令示意。

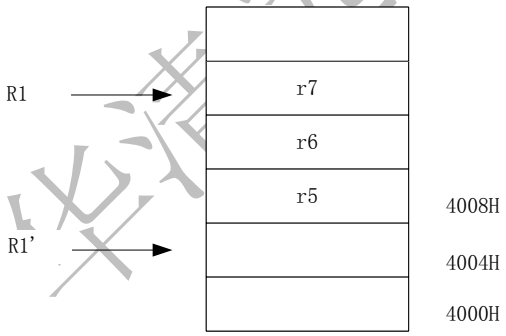
（1）STMIA　r1，{r5～r7}



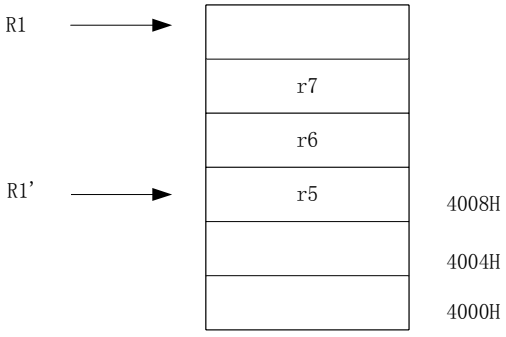
（2）STMIB　r1!，{r5～r7}



（3）STMDA　r1!，{r5～r7}



（4）STMDB　r1!，{r5～r7}



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 5.3 所示。

表 5.3　　多寄存器 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 5.4】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 5.5】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
```

5.5 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 5.4 所示。

表 5.4　　交换指令 SWP

指　令	作　用	操　作
SWP	字交换	Tmp=men32[Rn] Mem32[Rn]=Rm Rd=tmp

续表

指　令	作　用	操　作
SWPB	字节交换	Tmp=men8[Rn] Mem8[Rn]=Rm Rd=tmp

注意　交换指令在执行期间不能被其他任何指令或其他任何总线访问打断，在此期间系统占用总线（holds the bus），直至交换完成。

5.5.1 字交换指令 SWP

（1）指令编码格式

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。

指令的编码格式如图 5.20 所示。

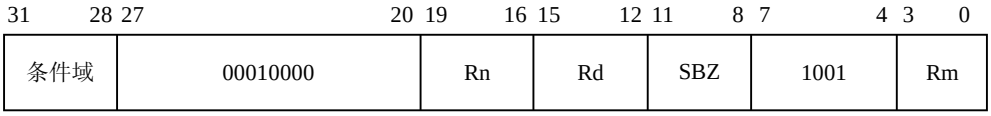


图 5.20　SWP 指令编码格式

（2）指令的语法格式

```
SWP{<cond>}  <Rd>，<Rm>，[<Rn>]
```

① <cond>

为指令编码中的条件域。它指示 SWP 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <Rd>

目标寄存器。

③ <Rm>

寄存器包含将要存储到内存中的数据。

④ <Rn>

寄存器中包含将要访问的内存地址。

（3）指令操作的伪代码

指令操作伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    If  Rn[1:0]==0b00  then
          Temp=memory[Rn,4]
    Else  if  Rn[1:0]==0b01  then
          Temp=memory[Rn,4]  Rotate_right 8
    Else  if  Rn[1:0]==0b10  then
          Temp=memory[Rn,4]  Rotate_right 16
    Else  /*  Rn[1:0]==0b11  */
          Temp=memory[Rn,4]  Rotate_right 24
    Memory[Rn,4]=Rm
    Rd=temp
```

5.5.2　字节交换指令 SWPB

（1）指令编码格式

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。

指令的编码格式如图 5.21 所示。

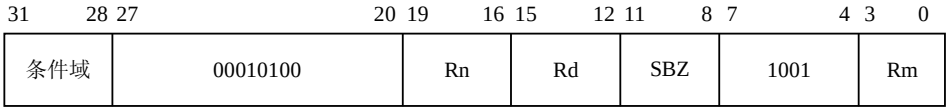


图 5.21　SWPB 指令编码格式

（2）指令的语法格式

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

① <cond>
为指令编码中的条件域。它指示 SWPB 指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。
② <Rd>
目标寄存器。
③ <Rm>
寄存器包含将要存储到内存中的数据。
④ <Rn>
寄存器中包含将要访问的内存地址。
（3）指令操作的伪代码
指令操作伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    Temp=Memory[Rn,1]
    Memory[Rn,1]=Rm[7:0]
    Rd=temp
```

5.5.3　交换指令 SWP 应用

寄存器和存储器交换指令 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。
格式如下。

```
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 5.6】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
```

5.6 程序状态寄存器指令

ARM 指令集提供了两条指令，可直接控制程序状态寄存器（PSR，Program State Register）。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。这两条指令结合，可用于对 CPSR 和 SPSR 进行读/写操作。
交换指令如表 5.5 所示。

表 5.5　　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。

注意　程序不能通过直接修改 CPSR 中的 T 位控制直接将程序状态切换到 Thumb 状态，必须通过 BX 等指令完成程序状态的切换。

5.6.1 MRS

（1）指令编码格式
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。
指令的编码格式如图 5.22 所示。

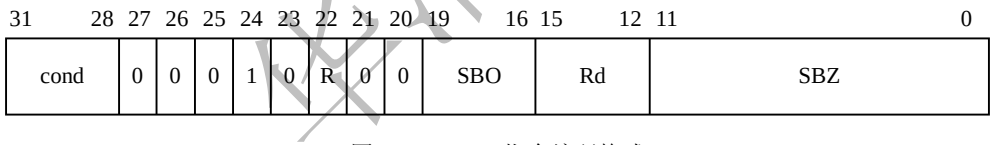


图 5.22　MRS 指令编码格式

当数据被移到通用寄存器以后，就可以对数据进行处理。
（2）指令的语法格式

```
MRS{<cond>}  <Rd>，CPSR
MRS{<cond>}  <Rd>，SPSR
```

① <cond>
为指令编码中的条件域。它指示 LDM（1）指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。
② <Rd>
目标寄存器。当 r15 被用作目标寄存器时，指令执行结果不可预知。
（3）指令操作的伪代码
指令操作伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
    If  R==1  then
        Rd=SPSR
    ELSE
        Rd=CPSR
```

注意
① 当操作码 opcode[11：0]≠0x000 时，指令的执行结果不可预知。
② 当操作码 opcode[19：16]≠0b1111 时，指令的执行结果不可预知。
③ 当在用户模式下对 SPSR 进行操作时，指令的执行结果不可预知。

5.6.2 MSR

（1）指令编码格式

MSR 指令用于将通用寄存器中的内容或立即数传送到程序状态寄存器中。因此指令的编码格式也有两种格式。

指令的源操作数为通用寄存器时编码格式如图 5.23 所示。

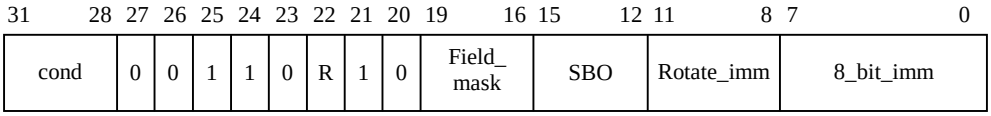


图 5.23 源操作数为通用寄存器的 MSR 指令编码

指令的源操作数为立即数时编码格式如图 5.24 所示。

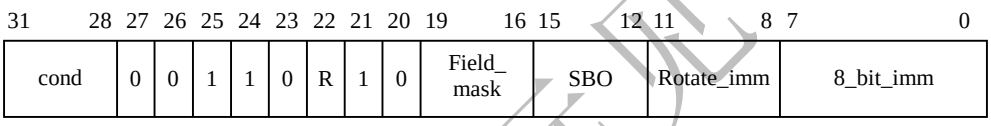


图 5.24 源操作数为立即数的 MSR 指令编码

（2）指令的语法格式

```
MSR{<cond>}  CPSR_<fields>, #<immediate>
MSR{<cond>}  CPSR_<fields>, #<Rm>
MSR{<cond>}  SPSR_<fields>, #<immediate>
MSR{<cond>}  SPSR_<fields>, #<Rm>
```

① <cond>

为指令编码中的条件域。它指示 LDM（1）指令在什么条件下执行。当<cond>忽略时，指令为无条件执行（cond=AL（Alway））。

② <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]状态位域，用 s 表示。
- bits[15：8]扩展位域，用 x 表示。
- bits[7：0]控制位域，用 c 表示。

③ < immediate >

表示将要传送到状态寄存器中的立即数。

（3）指令操作的伪代码

指令操作伪代码如下面程序段所示。

```
If  ConditionPassed{cond}  then
   If  opcode[25]==1
         Operand=8_bit_immediate  Rotate_Right {rotate_imm*2}
   Else  /*opcode[25]==0*/
         Operand=Rm
```

```
    If  R==0  then
          If  field_mask[0]==1  and  InAPrivilegedMode()  then
               CPSR[7:0]=operand[7:0]
          If  field_mask[1]==1  and  InAPrivilegedMode()  then
               CPSR[15:8]=operand[15:8]
          If  field_mask[2]==1  and  InAPrivilegedMode()  then
               CPSR[23:16]=operand[23:16]
          If  field_mask[2]==1  then
               CPSR[31:24]=operand[31:24]
    Else  /*R==1*/
          If  field_mask[0]==1  and  InAPrivilegedMode()  then
               SPSR[7:0]=operand[7:0]
          If  field_mask[1]==1  and  InAPrivilegedMode()  then
               SPSR[15:8]=operand[15:8]
          If  field_mask[2]==1  and  InAPrivilegedMode()  then
               SPSR[23:16]=operand[23:16]
          If  field_mask[2]==1  then
               SPSR[31:24]=operand[31:24]
```

5.6.3 程序状态寄存器指令应用

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。指令格式如下：

```
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。
MRS 指令举例如下。

```
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 的状态标志以及 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。
【例 5.7】使能 IRQ 中断

```
ENABLE_IRQ
        MRS    r0，CPSR
        BIC     r0，r0，＃0x80
        MSR    CPSR_c，r0
        MOV    PC，LR
```

【例 5.8】禁止 IRQ 中断

```
DISABLE_IRQ
       MRS   r0,CPSR
       ORR   r0,r0,#0x80
       MSR   CPSR_c,r0
       MOV   PC,LR
```

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。指令格式如下。

```
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。

MSR 指令举例如下。

```
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 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以及允许/禁止 IRQ/FIQ 中断等设置。

【例 5.9】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
```


联系方式