嵌入式系统原理与设计

第6章 ARM指令集及汇编程序设计

6.1.4 ARM处理器的指令集

ARM处理器支持两类指令集：

- 32位的ARM指令集
- 16位Thumb指令集

Thumb指令集是ARM指令集的子集

ARM体系结构采用这种设计方式的目的在于减少程序代码需要的存储空间，因为完成同样功能的Thumb指令只需要2个字节，而ARM指令需要4个字节。

6.1.4 ARM处理器的指令集

ARM指令集和Thumb指令集的不同点

项目	ARM指令	Thumb指令
指令工作标志	CPSR的T位=0	CPSR的T位=1
操作数寻址方式	大多数指令为3地址	大多数指令为2地址
指令长度	32位	16位
内核指令	58条	30条
条件执行	大多数指令	只有分支指令
数据处理指令	访问桶形移位器和ALU	独立的桶形移位器和ALU指令
寄存器使用	15个通用寄存器+PC	8个通用低寄存器+2个高寄存器+PC
程序状态寄存器	特权模式下可读可写	不能直接访问
异常处理	能够全盘处理	不能处理

6.1.5 ARM指令的条件码

ARM指令可以条件执行，也就是根据CPSR中的条件标志位来决定是否执行某条指令。当条件满足时执行该指令，条件不满足时该指令被当作一条NOP指令（不完成任何实际操作，相当于在流水线中插入一个气泡）。

条件执行是ARM指令体系结构的特色之一，也特别有用。

例如，如下的C语言代码，如果变量a分配给R0寄存器，变量b分配给R1寄存器，如何用ARM汇编指令来实现？

```
if(a>b)   a++;
else      b++;
```

6.1.5 ARM指令的条件码

如下的C语言代码，如果变量a分配给R0寄存器，变量b分配给R1寄存器，如何用ARM汇编指令来实现?

```
if(a>b)   a++;
else      b++;
```

```
CMP    R0, R1              ;比较R0和R1的值，即比较a和b的大小
ADDHI  R0, R0, #1          ;如果R0>R1，执行该语句，即a++
ADDLS  R1, R1, #1          ;如果R0≤R1，执行该语句，即b++
```

6.1.5 ARM指令的条件码

各条件码的含义和助记符

条件码	条件码助记符	含义	CPSR中条件标志位值
0000	EQ	相等	Z=1
0001	NE	不相等	Z=0
0010	CS/HS	无符号数大于/等于	C=1
0011	CC/LO	无符号数小于	C=0
0100	MI	负数	N=1
0101	PI	非负数	N=0
0110	VS	上溢出	V=1
0111	VC	没有上溢出	V=0
1000	HI	无符号数大于（higher）	C=1且Z=0
1001	LS	无符号数小于等于	C=0且Z=1
1010	GE	带符号数大于等于	N=1且V=1或N=0且V=0
1011	LT	带符号数小于	N=1且V=0或N=0且V=1
1100	GT	带符号数大于	Z=0且N=V
1101	LE	带符号数小于/等于	Z=1或N!=V
1110	AL	无条件执行	
1111	NV	该指令无条件执行	ARM v5及以上版本

6.1.6 ARM指令分类

ARM指令集可以分为5大类：

- 数据处理指令：使用片内算术逻辑部件（ALU）、桶形移位器和乘法器完成数据算术、逻辑、移位和乘法等运算。
- 分支跳转指令：控制程序执行流程、以及完成ARM代码和Thumb代码的切换。
- 存储器访问指令：控制存储器和寄存器之间的数据传送，包括从存储器取数到寄存器（称为Load）和将寄存器中的数存到存储器（称为Store）。
- 协处理器指令：用于控制外部协处理器，以开放和统一的方式扩展指令集的片外功能。
- 杂类指令：包括中断调用、状态寄存器的读写等。

6.2 ARM指令集

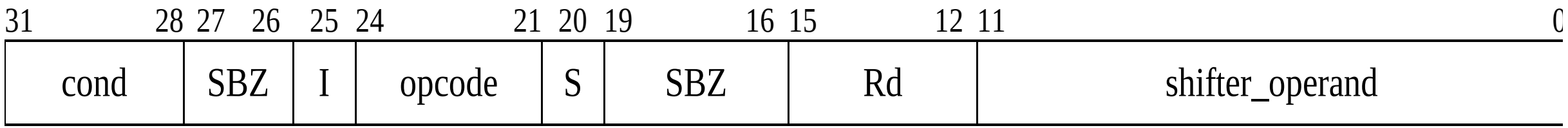
/02

6.2.1 数据处理指令

1. 数据传送指令

数据传送指令用于向寄存器传入一个数，属于2操作数指令，一个源操作数，一个目的操作数，包括MOV和MVN两条指令，指令语法格式如下：

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



6.2.1 数据处理指令

1. 数据传送指令

【例6-1】数据传送指令的用法举例。

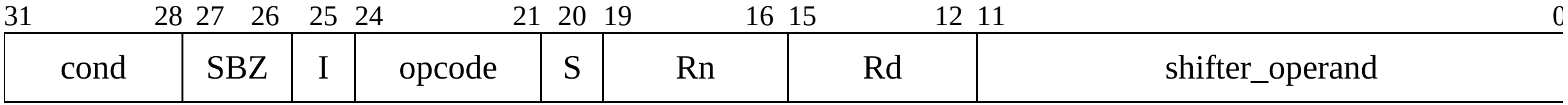
```
MOV  R1, R2                ; R1=R2
MOV  R1, #10               ; R1=10
MOV  PC, LR                ; PC=LR, 该指令用于从子程序返回
MOV  R2, R2 LSL #2         ; R2=R2<<2, 该指令用于实现纯移位操作
MVN  R1, #0xFF             ;将16进制数0xFF按位取反传送到R1，即R1=0xFFFFFF00
MOVEQ  R1, #10             ;条件标志Z=1时R1=10，否则R1不变
```

6.2.1 数据处理指令

2. 算术逻辑运算指令

该类指令用于加、减算术运算和与、或、异或等逻辑运算，属于3操作数指令，两个源操作数，一个目的操作数。指令语法格式如下：

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



6.2.1 数据处理指令（有错误）

1）ADD 加法指令

【例6-2】ADD指令的用法举例。

```
ADD Rx, Rx, #1                  ;Rx=Rx+1
ADD Rd, Rx, Rx, LSL #n          ;Rd= Rx*(2n+1)
ADD Rs, PC, #offset             ;生成基于PC的跳转指针（似乎这个）
```

6.2.1 数据处理指令

2）ADC 带进位加法指令

ADC指令和ADD指令联合使用，可以实现两个64位的操作数相加。例如，寄存器R0和R1中放置一个64位的源操作数，其中R0中放置低32位数值；寄存器R2和R3中放置另一个64位的源操作数，其中R2中放置低32位数值；则下面的指令序列实现了两个64位操作数的加法操作，运算结果存放在寄存器R5和R4中，其中R4保存低32位数值。

```
ADDS  R4, R0, R2          ;加低端的字
ADC   R5, R1, R3          ;加高端的字，带进位
```

6.2.1 数据处理指令

3）SUB 减法指令

【例6-3】SUB指令的用法举例。

```
SUB  R0, R1, R2                    ;R0=R1-R2
SUB  R0,R1,#256                    ;R0=R1-256
```

6.2.1 数据处理指令

4）SBC 带借位减法指令

SBC指令和SUBS指令联合使用，可以实现两个64位的操作数相减。例如寄存器R0和R1中放置一个64位的源操作数，其中R0中放置低32位数值，寄存器R2和R3中放置另一个64位的源操作数，其中R2中放置低32位数值，则下面的指令序列实现了两个64位操作数的减法操作。

```
SUBS  R4, R0, R2
SBC  R5, R1, R3
```

6.2.1 数据处理指令

5）RSB 逆向减法指令

【例6-4】RSB指令的用法举例。

```
RSB  R0, R1, #0                    ;Rd=-R1
RSB  R0, R1, R2                    ;Rd=R2-R1
RSB  Rd, Rx, Rx, LSL #n            ;Rd=Rx *(2n-1)
```

6.2.1 数据处理指令

6）RSC 带借位逆向减法指令

【例6-5】RSC指令的用法举例。

下面的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与R3中。其中R0与R2中放低32位值。

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

6.2.1 数据处理指令

7）AND 逻辑与操作指令

指令功能为Rd=Rn ＆ shifter_operand（按位与，如果shifter_operand为立即数，要扩展成32位）。

如果S=1，还要根据操作的结果更新CPSR中相应的条件标志位。

8）ORR 逻辑或操作指令

指令功能为Rd=Rn | shifter_operand（按位或）。

如果S=1，还要根据操作的结果更新CPSR中相应的条件标志位，更新方法与AND指令相同。

6.2.1 数据处理指令

9）EOR 逻辑异或操作指令

指令功能为Rd=Rn ^ shifter_operand（按位异或，即相异为1，相同为0）。

如果S=1，还要根据操作的结果更新CPSR中相应的条件标志位，更新方法与AND指令相同。

10）BIC 位清除指令

BIC指令用于清除寄存器<Rn>的某些位，并把结果保存到目标寄存器<Rd>中。

如果S=1，还要根据操作的结果更新CPSR中相应的条件标志位，更新方法与AND指令相同。

6.2.1 数据处理指令

【例6-6】AND、ORR、EOR和BIC指令的用法举例。

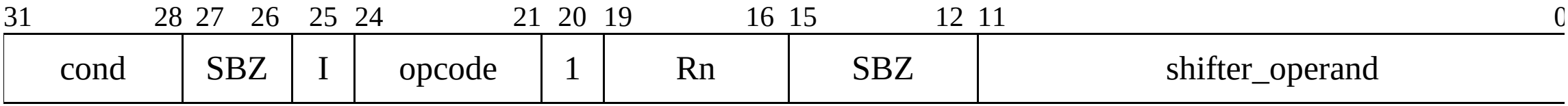
```
AND    R0, R0, #3          ;该指令保持R0的0、1位，其它位清零
ORR    R0, R0, #3          ;该指令将R0的0、1位置1，其它位保持不变
EOR    R0, R0, #3          ;该指令反转R0的0、1位，其它位保持不变
BIC    R0, R0, #3          ;该指令将R0中的0、1位置0，其余位保持不变
```

6.2.1 数据处理指令

3. 比较测试指令

这类指令属于2操作数指令，有两个源操作数，没有目的寄存器，因而不保存运算结果，只用于更新CPSR中相应的条件标志位（N、Z、C和V位），包括CMP、CMN、TST、TEQ指令。指令语法格式如下：

<opcode>{cond} <Rn>, <shifter_operand>



opcode=CMP|CMN|TST|TEQ，其对应的二进制码分别为1010|1011|1000|1001

6.2.1 数据处理指令

1）CMP 比较指令

CMP指令从寄存器<Rn>中减去< shifter_operand >表示的数值，根据操作的结果更新CPSR中相应的条件标志位，后面的指令就可以根据CPSR中相应的条件标志位来判断是否执行。

指令功能的伪码如下：

```
if ConditionPassed(cond) then   /*cond条件成立*/
  alu_out=Rn-shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = NOT Borrowfrom(Rn-shifter_operand)
    V Flag = OverflowFrom(Rn-shifter_operand)
```

6.2.1 数据处理指令

2）CMN 基于相反数的比较指令

CMN指令将寄存器<Rn>中的值加上< shifter_operand >表示的数值，根据操作的结果更新CPSR中相应的条件标志位，后面的指令就可以根据CPSR中相应的条件标志位来判断是否执行。

指令功能的伪码如下：

```
if ConditionPassed(cond) then   /*cond条件成立*/
alu_out=Rn+shifter_operand
N Flag = alu_out[31]
Z Flag = if alu_out == 0 then 1 else 0
C Flag = CarryFrom(Rn+shifter_operand)
V Flag = OverflowFrom(Rn+shifter_operand)
```

6.2.1 数据处理指令

3）TST 位测试指令

TST指令将寄存器<Rn>的值与< shifter_operand >表示的数值按位做逻辑与操作，根据操作的结果更新CPSR中相应的条件标志位。

指令功能的伪码如下：

```
if ConditionPassed(cond) then   /*cond条件成立*/
alu_out=Rn AND shifter_operand
N Flag = alu_out[31]
Z Flag = if alu_out == 0 then 1 else 0
C Flag =shifter_carry_out
V Flag = unaffected
```

6.2.1 数据处理指令

4）TEQ 相等测试指令

TEQ指令将寄存器<Rn>的值与<shifter_operand >表示的数值按位做逻辑异或操作，根据操作的结果更新CPSR中相应的条件标志位。其中，条件标志位的更新方法同TST指令。

【例6-7】比较测试指令用法举例（设R0=8，R1=4，R2=10）

```
CMP    R1, R0              ;根据R1-R0的结果设置CPSR标志位，N=1，Z=C=V=0
CMN  R1, #100              ;根据R1+100的结果设置CPSR标志位，N=Z=C=V=0
TST    R1, #3              ;测试R1的第0、1位是否为1，N=C=V=0，Z=1
TST    R1, R2, LSL #1      ;测试R1的第2、4位是否为1，N=Z=C=V=0
TEQ    R1, R2              ;测试R1与R2是否相等，N=Z=C=V=0
TEQ    R1, #4              ;测试R1与4是否相等，N =C=V=0，Z=1
```

6.2.1 数据处理指令

4. 乘法指令与乘加指令

- ARMv5T支持的乘法指令与乘加指令共有6条，可根据运算结果分为32位和64位两类。
- 指令中所有的操作数、目的寄存器必须为通用寄存器，不能对操作数使用立即数或被移位的寄存器。
- 目的寄存器和操作数1必须是不同的寄存器。
- 乘法指令和乘加指令包括MUL、MLA、SMULL、SMLAL、UMULL、UMLAL共6条。
- 对于乘法指令，源寄存器或目的寄存器不能为R15寄存器，否则执行结果不可预测。

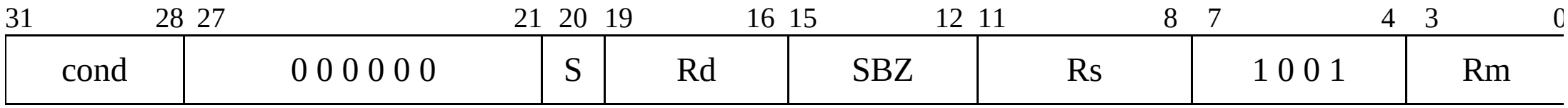
6.2.1 数据处理指令

1）MUL 32位乘法指令

MUL指令实现两个32位的数（可以为无符号数，也可以为有符号数）的乘法，并将低32位结果存放到一个32位的寄存器中。

如果S=1，可以根据运算结果设置CPSR寄存器中相应的条件标志位。

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



注：由于两个32位的数相乘结果为64位，而MUL指令仅仅保存了64位结果的低32位，所以对于有符号的和无符号的操作数来说，MUL指令执行的结果相同。

6.2.1 数据处理指令

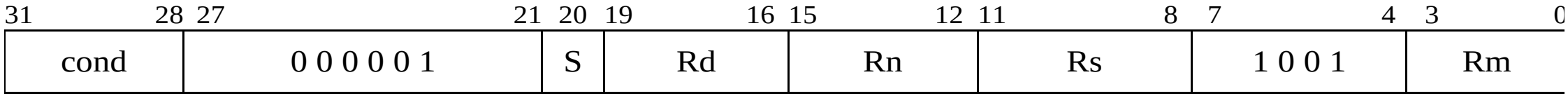
2）MLA 32位带加数的乘法指令

指令功能为Rd=(Rm*Rs)[31:0]+Rn。

MLA指令实现两个32位的数（可以为无符号数，也可为有符号数）的乘积，再将乘积加上第3个操作数，并将结果存放到一个32位的寄存器中。

如果S=1，根据运算结果更新CPSR寄存器中N和Z标志位，更新方法与MUL相同。

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



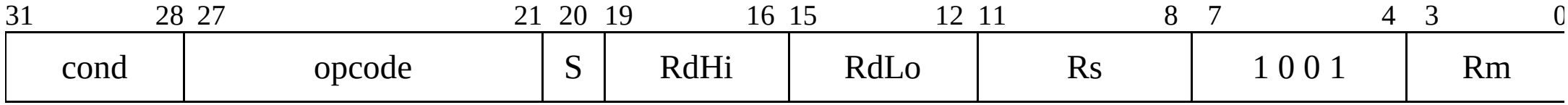
6.2.1 数据处理指令

3）SMULL、SMLAL、UMULL和UMLAL 64位乘法指令

SMULL、SMLAL、UMULL和UMLAL为4条64位的乘法指令，指令语法格式如下：

<opcode>{cond}{S} <RdLo>, <RdHi>, <Rm>,<Rs>

其中：RdHi寄存器存放乘积结果的高32位数据，RdLo寄存器存放乘积结果的低32位数据。



opcode=SMULL|SMLAL|UMULL|UMLAL，其对应的二进制码分别为0000110 |0000111|0000100|0000101。

6.2.1 数据处理指令

SMULL 64位有符号数乘法指令

SMULL指令实现两个32位的有符号数的乘积，乘积结果的高32位存放到一个32位的寄存器的<RdHi>中，乘积结果的低32位存放到另一个32位的寄存器<RdLo>中。

如果S=1，可以根据运算结果设置CPSR寄存器中N和Z条件标志位。

指令操作的伪代码如下：

```
if ConditionPassed(cond) then   /*cond条件成立*/
RdHi=(Rm*Rs)[63:32]
RdLo=(Rm*Rs)[31:0]
if S==1 then
N Flag =RdHi[31]
Z Flag = if (RdHi==0)and(RdLo == 0) then 1 else 0
C Flag =unaffected
V Flag = unaffected
```

6.2.1 数据处理指令

SMLAL 64位带加数的有符号数乘法指令

SMLAL指令将<Rm>和<Rs>两个32位的有符号数的乘积结果与<RdHi>和<RdLo>中的64位数相加，结果的高32位存放到<RdHi>的寄存器中，结果的低32位存放到另一个寄存器<RdLo>中。

UMULL 64位无符号数乘法指令

UMULL指令实现两个32位的无符号数的乘积，乘积结果的高32位存放到一个32位的寄存器<RdHi>中，乘积结果的低32位存放到另一个32位的寄存器<RdLo>中。

6.2.1 数据处理指令

UMLAL　64位带加数的无符号数乘法指令

UMLAL指令将两个32位的无符号数的64位乘积结果与<RdHi>和<RdLo>中的64位无符号数相加，结果的高32位存放到<RdHi>寄存器中，结果的低32位存放到另一个寄存器<RdLo>中。

6.2.1 数据处理指令

【例6-8】典型乘法指令的用法举例

```
MUL     R0, R1, R2          ;R0=(R1*R2)[31:0]
MULS    R0, R1, R2          ;R0=(R1*R2)[31:0]，同时设置CPSR中N位和Z位
MLA     R0, R1, R2, R3      ;R0=(R1*R2)[31:0]+R3
SMULL   R1, R2, R3, R4      ;R1=(R3*R4) [31:0]，R2=(R3*R4)[63:32]
SMLAL   R0, R1, R2, R3      ;R0=(R2*R3)[31:0]+R0，R1=(R2*R3)[63:32]+R1
UMULL   R1, R2, R3, R4      ;R1=(R3*R4) [31:0]，R2=(R3*R4)[63:32]
UMLAL   R1, R2, R3, R4      ;R1=(R3*R4)[31:0]+R1，R2=(R3*R4)[64:32]+R2
```

6.2.2 存储器访问指令

RISC处理器的一大特色就是Load/Store架构，即只有Load/Store指令才能访问存储器。

◆ Load指令用于从内存中读取数据放入寄存器中，

◆ Store指令用于将寄存器中的数据保存到内存。

ARM的Load/Store指令包括如下3类：

（1）单寄存器传输指令：单向在寄存器和存储器之间传输一个数据

（2）多寄存器传输指令：单向在寄存器和存储器之间传输多个数据

（3）数据交换指令：双向在存储器和寄存器之间交换一个数据，既有取数据、又有存数据

6.2.2 存储器访问指令

1. 单寄存器传输指令

1）LDR/LDRB/STR/STRB指令功能

指令	功能
LDR	从存储器中将一个32位的字数据传送到目的寄存器中
LDRB	从存储器中将一个8位的无符号字节数据传送到目的寄存器中的低8位，高24位补0
STR	将目的寄存器中的32位数据存储到存储器中
STRB	将目的寄存器中的低8位数据存储到存储器中

注：当程序计数器PC作为LDR指令的目的寄存器时，指令从存储器中读取的字数据被当做指令地址，从而可以实现程序流程的跳转

6.2.2 存储器访问指令

指令的语法格式

<opcode>{<cond>}{B} <Rd>,<addressing_mode>

opcode=LDR|STR。addressing_mode用来指定存储器的地址，与I、P、U和W位的设置相关，一般为Rn和addr_mode运算的结果。

6.2.2 存储器访问指令

1. 单寄存器传输指令

2）LDRH/LDRSH/LDRSB/STRH指令功能

指令	功能
LDRH	从存储器中将一个16位的无符号半字数据传送到目的寄存器低16位，高16位补0
LDRSH	从存储器中将一个16位的有符号半字数据传送到目的寄存器低16位，高16位补符号位
LDRSB	从存储器中将一个8位的有符号字节数据传送到目的寄存器低8位，高24位补符号位
STRH	将目的寄存器中的低16位数据存储到存储器中

6.2.2 存储器访问指令

【例6-9】设R1=0x9008、R2=0xA5B45A4B、R3=0xFFFFFFFC，存储器内容如图6-1所示（16进制表示，小端访问），在不考虑指令前后影响条件下，下面每条指令执行后的寄存器或存储器的值为多少?

地址	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00009000	70	71	72	73	80	81	82	83	00	00	FF	FF	4B	B4	B4	4B
00009010	A0	A1	A2	A3	B0	B1	B2	B3	FF	00	00	00	A5	A5	5A	5A

6.2.2 存储器访问指令

LDR	R0, [R1]	R0←Mem[R1, 4]，即R0=0xFFFF0000
LDR	R0, [R1, #12]	R0←Mem[R1+12, 4]，即R0=0xB3B2B1B0
LDR	R0, [R1, R3]	R0←Mem[R1+R3, 4]，即R0=0x83828180
LDR	R0, [R1, #16]!	R0←Mem[R1+16, 4]，R1←R1+16，即R0=0xFF，R1=0x9018
LDR	R0, [R1], #16	R0←Mem[R1, 4]，R1←R1+16，即R0=0xFFFF0000，R1=0x9018
LDRB	R0, [R1, #2]	R0←ZeroExtend(Mem[R1+2, 1])，即R0=0xFF
STR	R2, [R1, #4]	Mem [R1+4,4]←R2，即地址0x900C开始4个字节为4B,5A,B4和A5
STRB	R2, [R1, #9]!	Mem[R1+9,1]←R2[7:0]，R1←R1+9，即地址0x9011处为4B,R1=0x9011

6.2.2 存储器访问指令

LDRH R0, [R1]	R0←ZeroExtend(Mem[R1, 2])，即R0=0
LDRH R0, [R1, #2]	R0←ZeroExtend(Mem[R1+2, 2])，即R0=0xFFFF
LDRSH R0, [R1, R3]	R0←SignExtend(Mem[R1+R3, 2])，即R0=0xFFFF8180
LDRSH R0, [R1], #2	R0←SignExtend(Mem[R1, 2])，R1←R1+2，即R0=0，R1=0x900A
LDRSB R0, [R1, #3]	R0←SignExtend(Mem [R1+3, 1])，即R0=0xFFFFFFFF
STRH R2,[R1,#6]	Mem[R1+6]←R2[15:0]，即地址0x900E开始2个字节为4B, 5A

6.2.2 存储器访问指令

2. 批量数据加载/存储指令

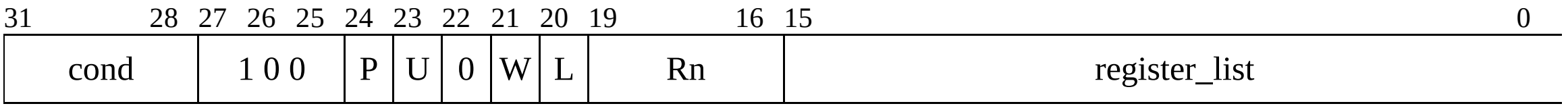
ARM处理器支持批量数据加载/存储指令，可以一条指令实现在地址连续的存储器单元和多个寄存器之间传送多个字数据。

具体来说，批量数据加载指令LDM可以从地址连续的存储器中读取多个字数据，传送到指令中指定的多个寄存器；批量数据存储指令STM可以将指令中寄存器列表中的各个寄存器的值写入到地址连续的存储器中。

LDM/STM指令的常见用途是将多个寄存器的内容入栈或出栈。

6.2.2 存储器访问指令

指令的编码格式



- cond为条件码。
- P为变址方式指示位，1代表前变址，0代表后变址。
- U为运算指示位，1代表做加法，0代表做减法。
- W为回写指示位，0代表地址不回写到Rn，1代表回写到Rn。
- L为加载指示位，1代表为LDM指令，0代表为STM指令。
- Rn为基址寄存器。
- register_list为寄存器列表，R0-R15每个寄存器1位，总共16位。

6.2.2 存储器访问指令

指令的语法格式

LDM|STM {<cond>} <addr_mode> <Rn>{!},<register_list>{^}

addr_mode，用来指示存储器地址如何进行变化，必须为以下8种情况

用于为数据块传送操作，
称为块拷贝寻址模式；
IA： 每次传送后地址加4；
IB： 每次传送前地址加4；
DA：每次传送后地址减4；
DB：每次传送前地址减4；

用于堆栈操作，
称为堆栈寻址模式。
FD：满递减堆栈；
ED：空递减堆栈；
FA：满递增堆栈；
EA：空递增堆栈。

6.2.2 存储器访问指令

2. 批量数据加载/存储指令

在进行数据复制时，即从存储器中把一段数据复制到另外一个位置时，一般采用块拷贝寻址模式。此时，需要先设置好源指针和目标指针，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。

在进行程序现场保护时，即要把多个寄存器的值放入堆栈，或现场恢复时，即从堆栈取出多个寄存器的值时，一般采用堆栈寻址模式。此时，需要先设置好堆栈指针（SP），然后使用堆栈寻址指令STMFD/LDMFD、STMED/LDMED、STMFA/LDMFA和STMEA/LDMEA实现堆栈操作。

6.2.2 存储器访问指令

【例6-10】设R4=0x8010，分析下面块拷贝STM指令执行后的R4及存储器的值（不考虑指令执行相互间的影响）。

```
STMIA       R4!, {R0-R3}
STMIB       R4!, {R0-R3}
STMDA       R4!, {R0-R3}
STMDB       R4!, {R0-R3}
```

6.2.2 存储器访问指令

【例6-10】设R4=0x8010，分析下面块拷贝STM指令执行后的R4及存储器的值（不考虑指令执行相互间的影响）。

```
STMIA        R4!, {R0-R3}
```

将寄存器列表中的{R0-R3}存入到基址寄存器R4指向的存储器中，每次传送后地址加4，存储器值如图6-2(a)所示，最后的地址写入基址寄存器R4中，R4=0x8020。

	8010H	R0
	8014H	R1
	8018H	R2
	801CH	R3
R4→	8020H	

6.2.2 存储器访问指令

【例6-10】设R4=0x8010，分析下面块拷贝STM指令执行后的R4及存储器的值（不考虑指令执行相互间的影响）。

```
STMIB        R4!, {R0-R3}
```

将寄存器列表中的{R0-R3}存入到基址寄存器R4指向的存储器中，每次传送前地址加4，存储器值如图6-2(b)所示，最后的地址写入基址寄存器R4中，R4=0x8020。

	8010H	
	8014H	R0
	8018H	R1
	801CH	R2
R4→	8020H	R3

6.2.2 存储器访问指令

【例6-10】设R4=0x8010，分析下面块拷贝STM指令执行后的R4及存储器的值（不考虑指令执行相互间的影响）。

```
STMDA        R4!, {R0-R3}
```

将寄存器列表中的{R0-R3}存入到基址寄存器R4指向的存储器中，每次传送后地址减4，存储器值如图6-2(c)所示，最后的地址写入基址寄存器R4中，R4=0x8000。

R4→	8000H	
	8004H	R0
	8008H	R1
	800CH	R2
	8010H	R3

6.2.2 存储器访问指令

【例6-10】设R4=0x8010，分析下面块拷贝STM指令执行后的R4及存储器的值（不考虑指令执行相互间的影响）。

```
STMDB        R4!, {R0-R3}
```

将寄存器列表中的{R0-R3}存入到基址寄存器R4指向的存储器中，每次传送前地址减4，存储器值如图6-2(d)所示，最后的地址写入基址寄存器R4中，R4=0x8000。

R4→	8000H	R0
	8004H	R1
	8008H	R2
	800CH	R3
	8010H	

6.2.2 存储器访问指令

【例6-11】设SP=0x8010，分析下面STM指令执行后的SP及存储器的值（不考虑指令执行相互间的影响）。

```
STMEA     SP!, {R0-R3}
STMFA     SP!, {R0-R3}
STMED     SP!, {R0-R3}
STMFD     SP!, {R0-R3}
```

6.2.2 存储器访问指令

【例6-11】设SP=0x8010，分析下面STM指令执行后的SP及存储器的值（不考虑指令执行相互间的影响）。

STMEA　　SP!, {R0-R3}

将寄存器列表中的{R0-R3}存入到基址寄存器SP指向的存储器中，每次传送后地址加4，存储器值如图6-3(a)所示，最后的地址写入基址寄存器SP中，SP=0x8020。

	8010H	R0
	8014H	R1
	8018H	R2
	801CH	R3
SP→	8020H	

6.2.2 存储器访问指令

【例6-11】设SP=0x8010，分析下面STM指令执行后的SP及存储器的值（不考虑指令执行相互间的影响）。

```
STMFA        SP!, {R0-R3}
```

将寄存器列表中的{R0-R3}存入到基址寄存器SP指向的存储器中，每次传送前地址加4，存储器值如图6-3(b)所示，最后的地址写入基址寄存器SP中，SP=0x8020。

	8010H	
	8014H	R0
	8018H	R1
	801CH	R2
SP→	8020H	R3

6.2.2 存储器访问指令

【例6-11】设SP=0x8010，分析下面STM指令执行后的SP及存储器的值（不考虑指令执行相互间的影响）。

STMED　　SP!, {R0-R3}

将寄存器列表中的{R0-R3}存入到基址寄存器SP指向的存储器中，每次传送后地址减4，存储器值如图6-3(c)所示，最后的地址写入基址寄存器SP中，SP=0x8000。

SP→	8000H	
	8004H	R0
	8008H	R1
	800CH	R2
	8010H	R3

6.2.2 存储器访问指令

【例6-11】设SP=0x8010，分析下面STM指令执行后的SP及存储器的值（不考虑指令执行相互间的影响）。

STMFD　　SP!, {R0-R3}

将寄存器列表中的{R0-R3}存入到基址寄存器SP指向的存储器中，每次传送前地址减4，存储器值如图6-3(d)所示，最后的地址写入基址寄存器SP中，SP=0x8000。

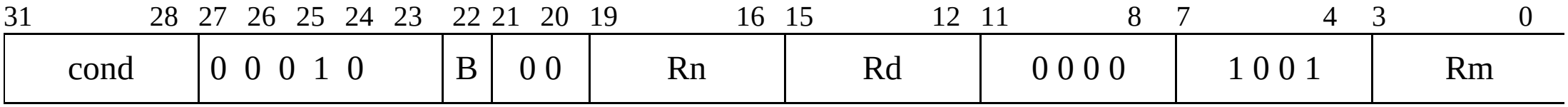
SP→	8000H	R0
	8004H	R1
	8008H	R2
	800CH	R3
	8010H	

6.2.2 存储器访问指令

3、数据交换指令

数据交换指令能在存储器和寄存器之间实现双向数据传输，既有存数据，也有加载数据。数据交换指令包括SWP和SWPB两条，SWP实现字数据（32位）交换，SWPB实现字节数据（8位）交换。

指令的编码格式



B=1代表SWPB指令，B=0代表SWP指令。

6.2.2 存储器访问指令

【例6-12】SWP指令用法举例。

SWP　R0,R1,[R2]	将R2所指向的存储器中的字数据传送到R0，同时将R1中的字数据传送到R2所指向的存储单元
SWPB R0,R1,[R2]	将R2所指向的存储器中的字节数据传送到R0，R0的高24位清零，同时将R1的低8位数据传送到R2所指向的存储单元

6.2.3 跳转指令

跳转指令用于改变程序的顺序执行流程，实现流程跳转。

在ARMv5指令集中有两种方式可以实现程序的跳转：

- 跳转指令；
- 直接向程序计数器PC（R15）中写入目标地址值。可以实现在4GB的地址空间中任意跳转，这种跳转指令又称为长跳转。

6.2.3 跳转指令

ARM的跳转指令可以在当前指令向前或向后的32MB的地址空间跳转。这类跳转指令有如下4种：

1）B{cond}，最基本的跳转指令。

2）BX{cond}，带状态切换的跳转指令。

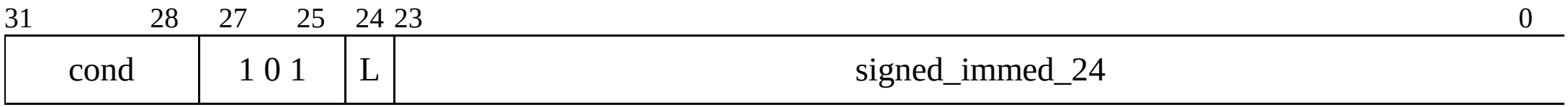
3）BL{cond}，带返回的跳转指令。

4）BLX{cond}，带返回和状态切换的跳转指令。

6.2.3 跳转指令

1. B/BL指令

指令的编码格式



L=1，代表为BL指令，否则为B指令。

signed_immed_24为有符号的24位立即数，代表相对当前PC的偏移数。

6.2.3 跳转指令

1. B/BL指令

指令的语法格式

B{L} {cond} <target_address> 跳转的范围大致为-32MB~+32MB

L决定是否保存返回地址。当有L时，将紧跟在跳转指令之后指令的地址保存到LR寄存器中；当没有L时，跳转指令之后指令的地址将不会保存到LR寄存器中。

target_address为指令跳转的目标地址，其计算方法如下：

- ✧ 将指令中的24位带符号的二进制立即数扩展为30位。
- ✧ 将此30位数左移两位以形成32位值。
- ✧ 将得到的值与PC寄存器的值做加法，即得到跳转的目标地址。

6.2.3 跳转指令

【例6-13】B及BL指令的使用。

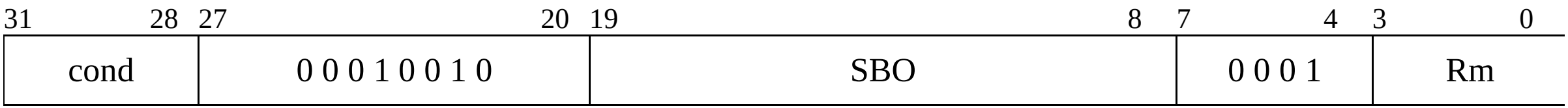
```
B  WAITA              ;程序跳转到标号WAITA处执行
B  0x1234             ;程序跳转到绝度地址0x1234处
BL  Label             ;先将下一条指令的地址保存到LR，再跳转到Label处
```

6.2.3 跳转指令

2. BX指令

BX（Branch and Exchange）指令跳转到指令中指定的目标地址，目标地址处的指令可以是ARM指令，也可以是 Thumb指令，具体由寄存器Rm的bit[0]决定。

指令的编码格式



6.2.3 跳转指令

2. BX指令

指令的语法格式

BX{<cond>} Rm

其中，Rm寄存器中为跳转的目标地址。

当Rm寄存器的bit[0]=0时，目标地址处的指令为ARM指令；

当bit[0]=1时，目的地址处的指令为Thumb指令。

6.2.3 跳转指令

指令操作的伪代码：

```
if ConditionPassed(cond) then
  CPSR T bit = Rm[0]
  PC = Rm AND 0xFFFFFFFE
```

指令的使用：

```
ADRL R0, ThumbFun+1          ;将Thumb程序的入口地址加1存入R0
BX   R0                      ;跳转到R0指定的地址
```

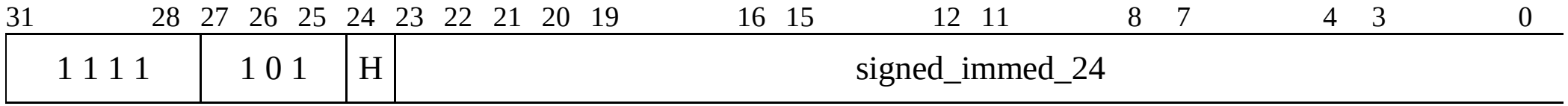
6.2.3 跳转指令

3. BLX指令

（1）BLX(1)指令

BLX(1)是无条件执行指令。BLX(1)指令一定会跳转到指令中指定的目标地址，并将程序状态切换为 Thumb状态，同时将返回地址保存到LR寄存器中。通常，BLX(1)用于在ARM程序中调用Thumb指令的子程序。

指令的编码格式



6.2.3 跳转指令

3. BLX指令

（1）BLX(1)指令

BLX(1)指令的语法格式

BLX <target_address>

target_address为指令跳转的目标地址，其计算方法如下：

1）将指令中的24位带符号的二进制立即数扩展为30位。

2）将此30位数左移两位以形成32位值。

3）将上步得到的32位数的bit[1]设置为H（H即指令编码中的bit[24]）。

4）将上步得到的32位数与PC寄存器的内容做加法，结果即为跳转的目标地址。

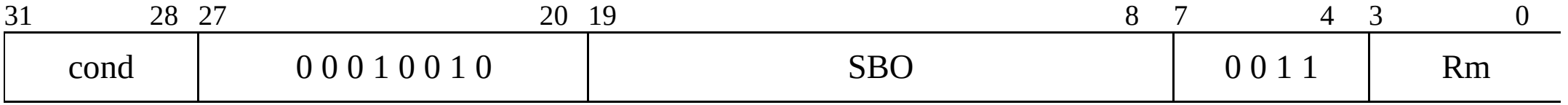
6.2.3 跳转指令

3. BLX指令

（2）BLX(2)指令

BLX(2)指令从ARM指令集跳转到指令中指定的目标地址，目标地址的指令可以是ARM指令，也可以是Thumb指令。目标地址放在寄存器Rm中，目标地址处的指令类型由Rm的bit[0]决定。同时，该指令也将返回地址保存到LR寄存器中。

指令的编码格式



6.2.3 跳转指令

3. BLX指令

（2）BLX(2)指令

指令的语法格式

BLX{<cond>} Rm

其中， Rm寄存器中为跳转的目标地址。

当Rm的bit[0]=0时，目标地址处的指令为ARM指令；

当bit[0]=1时，目标地址处的指令为Thumb指令。

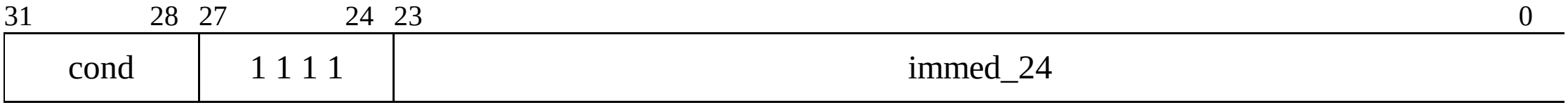
当Rm为R15时，会产生不可预知的结果。

6.2.4 杂项指令

（1）SWI（软中断指令）

SWI用于产生软件中断，ARM通过这种机制实现在用户模式对操作系统中特权模式的程序的调用。

指令的编码格式



指令的语法格式

SWI{<cond>} <immed_24>

6.2.4 杂项指令

（1）SWI（软中断指令）

指令　SWI{<cond>} <immed_24>参数传递的方法：

- 指令中24位的立即数指定了用户请求的服务类型，参数通过通用寄存器传递。
- 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的数值决定，参数通过其他的通用寄存器传递。

指令的使用：

```
MOV   R0, #34          ;通过寄存器R0传递参数
SWI    12              ;产生软件中断，调用操作系统编号为12的系统例程
```

6.2.4 杂项指令

（2）状态寄存器读指令MRS

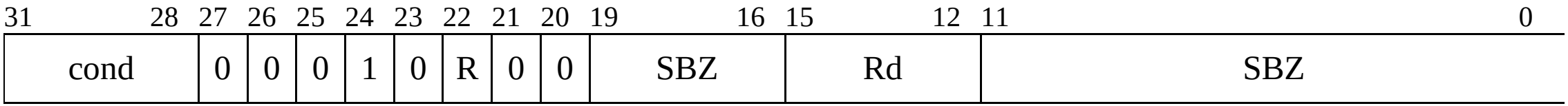
MRS指令用于将程序状态寄存器的内容传送到通用寄存器中。

- 当需要改变程序状态寄存器的内容时，可用MRS指令将程序状态寄存器的内容读入通用寄存器，修改后再写回程序状态寄存器。
- 当在异常中断或进程切换时，需要保存当前程序状态寄存器值，可先用MRS指令读出程序状态寄存器的值，然后保存。

6.2.4 杂项指令

（2）状态寄存器读指令MRS

指令的编码格式



指令的语法格式　　**MRS{<cond>} <Rd>, PSR**

Rd为通用寄存器，

PSR为当前状态寄存器CPSR或备份状态寄存器SPSR，由R位（即bit[22]）指定。

6.2.4 杂项指令

（3）状态寄存器写指令MSR

MSR指令用于将通用寄存器的内容或一个立即数传送到状态寄存器特定域中，即用于恢复状态寄存器的内容或者改变状态寄存器的内容。

在使用时，建议在MSR指令中指明将要操作的域。

指令的语法格式

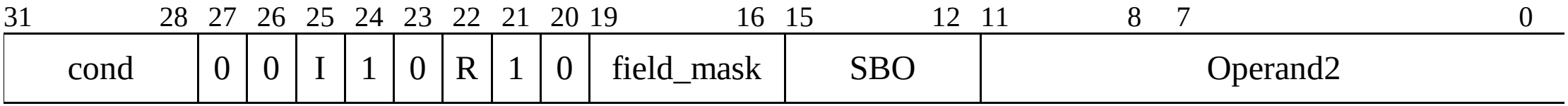
MSR{<cond>} PSR _<fields>, <Operand2>

注：PSR=CPSR|SPSR，fields=f|s|x|c，Operand2= #immediate|Rm

6.2.4 杂项指令

（3）状态寄存器写指令MSR

MSR指令的编码格式



I位（即bit[25]）用来指定Operand2的形式，I=0则Operand2来源于Rm寄存器；

I=1则Operand2由4位循环右移立即数rotate_imm_4和8位立即数imm_8构成；

R位用来指定是哪个状态寄存器，用法同MRS指令；

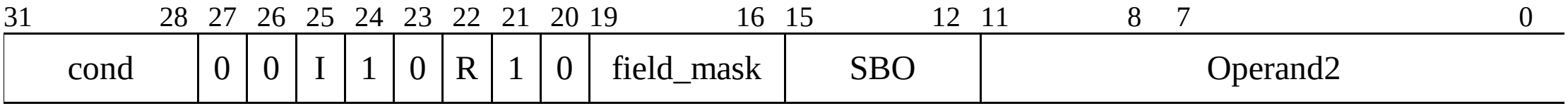
field_mask用来指定操作的域。

field_mask（即bit[19:16]）每位依次代表操作状态寄存器的标志位域（PSR[31:24]）、状态位域（PSR[23:16]）、扩展位域（PSR[15:8]）和控制位域（PSR[7:0]）

6.2.4 杂项指令

（3）状态寄存器写指令MSR

MSR指令的编码格式



I位（即bit[25]）用来指定Operand2的形式，I=0则Operand2来源于Rm寄存器；

I=1则Operand2由4位循环右移立即数rotate_imm_4和8位立即数imm_8构成；

R位用来指定是哪个状态寄存器，用法同MRS指令；

field_mask用来指定操作的域。

field_mask（即bit[19:16]）每位依次代表操作状态寄存器的标志位域（PSR[31:24]）、状态位域（PSR[23:16]）、扩展位域（PSR[15:8]）和控制位域（PSR[7:0]）

6.2.4 杂项指令

（3）状态寄存器写指令MSR

【例6-14】MRS和MSR指令使用举例。

```
MRS R0, CPSR                ; Read the CPSR
BIC R0, R0, #0xF0000000     ; Clear the N, Z, C and V bits
MSR CPSR_f, R0              ; Update the flag bits in the CPSR
                            ; N, Z, C and V flags now all clear
MRS R0, CPSR                ; Read the CPSR
ORR R0, R0, #0x80           ; Set the interrupt disable bit
MSR CPSR_c, R0              ; Update the control bits in the CPSR
                            ; interrupts (IRQ) now disabled
MRS R0, CPSR                ; Read the CPSR
BIC R0, R0, #0x1F           ; Clear the mode bits
ORR R0, R0, #0x11           ; Set the mode bits to FIQ mode
MSR CPSR_c, R0              ; Update the control bits in the CPSR
                            ; now in FIQ mode
```

6.3 Thumb指令集

/03

6.3.1 Thumb指令集概述

Thumb指令集是对ARM指令集重新编码得到的子集，它旨在增强使用16位或更窄数据总线实现的ARM处理器的性能和提供比ARM指令集更好的代码密度。

ARM指令集的T变种同时包含32位的ARM指令集和16位Thumb指令集。

在ARMv6及以上指令版本中，Thumb指令集支持是必需的。

所有的Thumb数据处理指令仍然支持32位的操作，指令和数据的寻址空间也仍然是32位。

6.3.1 Thumb指令集概述

- 大多数Thumb指令是无条件执行的
- 大多数Thumb指令采用2地址格式
- 要实现相同的程序功能，所需的Thumb指令的条数要比ARM指令多。
- Thumb代码所需的存储空间为ARM代码的60%~70%。
- Thumb代码使用的指令数比ARM代码多30%~40%。
- 若使用16位的存储器，Thumb代码比ARM代码快40%~50%。
- 与ARM代码相比，使用Thumb代码，存储器的功耗会降低约30%。

6.3.2 Thumb指令集编码

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Shift by immediate	0	0	0	opcode [1]		immediate					Rm			Rd		
Add/subtract register	0	0	0	1	1	0	opc	Rm			Rn			Rd		
Add/subtract immediate	0	0	0	1	1	1	opc	immediate			Rn			Rd		
Add/subtract/compare/move immediate	0	0	1	opcode		Rd / Rn		immediate								
Data-processing register	0	1	0	0	0	0	opcode				Rm / Rs			Rd / Rn		
Special data processing	0	1	0	0	0	1	opcode [1]		H1	H2	Rm			Rd / Rn		
Branch/exchange instruction set [3]	0	1	0	0	0	1	1	1	L	H2	Rm			SBZ		
Load from literal pool	0	1	0	0	1	Rd			PC-relative offset							
Load/store register offset	0	1	0	1	opcode			Rm			Rn			Rd		
Load/store word/byte immediate offset	0	1	1	B	L	offset					Rn			Rd		
Load/store halfword immediate offset	1	0	0	0	L	offset					Rn			Rd		
Load/store to/from stack	1	0	0	1	L	Rd			SP-relative offset							
Add to SP or PC	1	0	1	0	SP	Rd			immediate							

6.3.2 Thumb指令集编码

Load/store multiple	1	1	0	0	L	Rn			register list							
Conditional branch	1	1	0	1	cond [2]				offset							
Undefined instruction	1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	x
Software interrupt	1	1	0	1	1	1	1	1	immediate							
Unconditional branch	1	1	1	0	0	offset										
BLX suffix [4]	1	1	1	0	1	offset										0
Undefined instruction	1	1	1	0	1	x	x	x	x	x	x	x	x	x	x	1
BL/BLX prefix	1	1	1	1	0	offset										
BL suffix	1	1	1	1	1	offset										

6.3.3 Thumb指令集举例

部分Thumb指令和ARM指令的对照表

Thumb指令	ARM指令	说明	操作
MOV Rd,#expr8 MOV Rd,Rm	MOV Rd, operand2	数据传送	Rd←#expr8 Rd←Rm
MVN Rd,Rm	MVN Rd, operand2	数据非传送	Rd←(~Rm)

ADD Rd,Rn,#expr3
ADD Rd,#expr8
ADD Rd,Rn,Rm
ADD Rd,Rm
ADD Rd,Rp,#expr8*4
ADD SP,#expr7*4
	ADD Rd,Rn,operand2	加法运算	Rd←Rn+#expr3
			Rd←Rd+expr8
			Rd←Rn+Rm
			Rd←Rd+Rm
			Rd←SP|PC+expr8*4
			SP←SP+expr7*4

6.3.3 Thumb指令集举例

部分Thumb指令和ARM指令的对照表

SUB Rd,Rn,#expr3
SUB Rd,#expr8
SUB Rd,Rn,Rm
SUB SP,#expr7*4
	SUB Rd, Rn, operand2	减法运算	Rd←Rn-#expr3
			Rd←Rd-expr8
			Rd←Rn-Rm
			SP←SP-expr7*4

ADC Rd,Rm	ADC Rd, Rn,operand2	带进位加法	Rd←Rd+Rm+C
SBC Rd,Rm	SBC Rd, Rn, operand2	带进位减法	Rd←Rd-Rm-(NOT C)
AND Rd,Rm	AND Rd, Rn,operand2	逻辑与	Rd←Rd&Rm
ORR Rd,Rm	ORR Rd, Rn, operand2	逻辑或	Rd←Rd|Rm
EOR Rd,Rm	EOR Rd, Rn, operand2	逻辑异或	Rd←Rd^Rm
BIC Rd,Rm	BIC Rd, Rn, operand2	位清除	Rd←Rd&(~Rm)

6.3.3 Thumb指令集举例

部分Thumb指令和ARM指令的对照表

CMP Rn,#expr8 CMP Rn,Rm	CMP Rn, operand2	比较	状态标志←Rn-#expr8 状态标志←Rn-Rm
CMN Rn,Rm	CMN Rn, operand2	负数比较	状态标志←Rn+Rm
TST Rn,Rm	TST Rn, operand2	位测试	状态标志←Rn&Rm
MUL Rn,Rm	MUL Rd, Rm, Rs	32位乘法	Rd←Rd*Rm

6.3.3 Thumb指令集举例

部分Thumb指令和ARM指令的对照表

LDR Rd,[Rn,#expr5*4] **LDR Rd,[Rn,Rm]** **LDR Rd,[Rp,#expr8*4]**	LDR Rd, addressing	加载字数据	Rd←[Rn,#expr5*4] Rd←[Rn,Rm] Rd←[PC|SP,#expr8*4]
STR Rd,[Rn,#expr5*4] **STR Rd,[Rn,Rm]** **STR Rd,[SP,#expr8*4]**	STR Rd, addressing	存储字数据	[Rn,#expr5*4] ←Rd [Rn,Rm] ←Rd [SP,#expr8*4] ←Rd

6.4 ARM指令的寻址方式

/04

6.4.1 寻址方式的类型

1. 立即数寻址

立即数寻址是一种特殊的寻址方式，操作数就在指令编码中给出，只要取出指令也就得到了操作数，故而这个操作数被称为立即数。如下面指令：

```
MOV  R3, #0x3A                ; 将十六进制数3a放到寄存器R3中，即R3=0x3A
```

在上面的指令中，第2个源操作数即为立即数，实际使用时以“#”符号作为前缀。十六进制的立即数在“#”后面加“0x”，以二进制表示的立即数在“#”后面加“%”，以十进制表示的立即数直接跟在“#”后。

6.4.1 寻址方式的类型

2. 寄存器寻址

操作数的值在寄存器中，指令中的地址码字段指出的是寄存器编号，指令执行时直接取出寄存器值来操作。寄存器寻址是各类处理器经常采用的一种寻址方式。在下面所示的指令中，R2即为寄存器寻址。

```
MOV  R1, R2          ; 将R2的数值放到R1中
ADD  R0, R1, R2      ; 将R1和R2中的数值相加，然后赋值给R0
```

6.4.1 寻址方式的类型

3. 寄存器间接寻址

寄存器间接寻址指令中的地址码给出的是一个通用寄存器的编号，所需的操作数保存在以寄存器的值作为地址的存储单元中，即寄存器为操作数的地址指针。如下面的指令：

```
LDR R1, [R2]                    ; 将R2指向的存储单元的数据读出保存在R1中
```

6.4.1 寻址方式的类型

4. 基址变址寻址

基址变址寻址方式就是将寄存器（该寄存器一般称为基址寄存器）的内容与指令中给出的地址偏移量相加/减，从而得到一个操作数的有效地址。基址寻址用于访问基址附近的存储单元，常用于查表、数组操作、功能部件寄存器访问等。寄存器间接寻址是偏移量为0的基址加偏移寻址。如下面的指令：

```
LDR  R0, [R1, #4]   ; 将寄存器R1的内容加上4形成操作数的有效地址
LDR  R0, [R1], #4   ; 将寄存器R1的值作为内存地址加载第2个操作数
                    ; 到R0；加载完成后，R1的值加4保存。
LDR  R0, [R1, R2]   ; 将R1+R2的值作为操作数的地址
```

6.4.1 寻址方式的类型

5. 多寄存器寻址

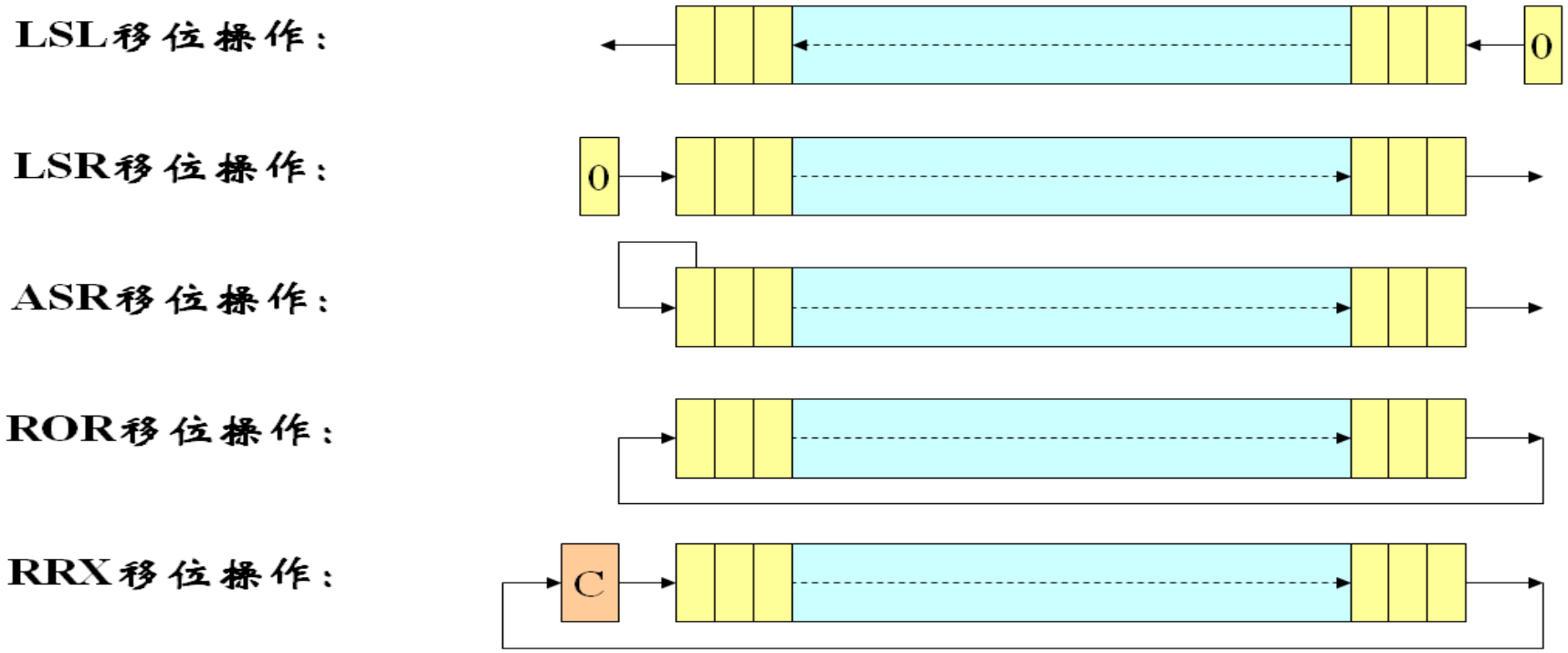
一次可以传送几个寄存器的值，允许一条指令传送16个寄存器的任何子集。

```
LDMIA     R1!, {R2-R7}   ; {R2-R7}即为多寄存器寻址
STMIA     R0!,{R2-R7 }   ; {R2-R7}即为多寄存器寻址
```

6.4.1 寻址方式的类型

6. 寄存器移位寻址

寄存器移位寻址是ARM指令集特有的寻址方式。当第2个操作数是寄存器移位方式时，第2个寄存器操作数在与第1个操作数结合之前，选择进行移位操作。



6.4.1 寻址方式的类型

6. 寄存器移位寻址

寄存器移位选址举例如下：

```
MOV     R0, R1, LSL #3      ;“R1, LSL #3”代表对R1进行逻辑左移3位操作
ANDS    R1, R1, R2, LSR #3  ;“R2, LSR #3”代表对R2进行逻辑右移3位操作
```

6.4.1 寻址方式的类型

7. 相对寻址

与基址变址寻址方式类似，但相对寻址由程序计数器PC提供基准地址，指令中的地址标号作为偏移量，两者相加后得到的地址即为操作数的有效地址。

```
BL    SUBR1                    ;调用到SUBR1子程序
      BEQ  LOOP                ;条件跳转到LOOP标号处
      ...
LOOP                                    ; LOOP标号
      MOV  R6, #1
      ...
SUBR1                                   ; 子程序SUBR1的入口标号
      ...
```

6.4.1 寻址方式的类型

8. 堆栈寻址

堆栈是一种数据结构，按先进后出（First In Last Out，FILO）的方式工作，使用一个称为堆栈指针的专用寄存器（ARM指令中，通常都采用R13寄存器作为堆栈寄存器SP）指示当前的操作位置，堆栈指针总是指向栈顶。

根据堆栈的生成方式，堆栈又可以分为递增堆栈（Ascending Stack）和递减堆栈（Decending Stack）。当堆栈由低地址向高地址生成时，称为递增堆栈；当堆栈由高地址向低地址生成时，称为递减堆栈。

根据堆栈指针指向的地址是否存有有效数据，堆栈又可以分为满堆栈（Full Stack）和空堆栈（Empty Stack）。当堆栈指针指向的地址存有有效数据时，称为满堆栈；否则，堆栈指针指向下一个要放入的空位置，称为空堆栈（Empty Stack）。

6.4.1 寻址方式的类型

8. 堆栈寻址

1) 满递增堆栈（FA）：堆栈指针指向最后压入的数据，且由低地址向高地址生成；

2) 满递减堆栈（FD）：堆栈指针指向最后压入的数据，且由高地址向低地址生成；

3) 空递增堆栈（EA）：堆栈指针指向下一个将要放入数据的空位置，且由低地址向高地址生成；

4) 空递减堆栈（ED）：堆栈指针指向下一个将要放入数据的空位置，且由高地址向低地址生成。

```
LDMFA SP!, {R2-R7}        ; FA指定满递增堆栈方式
STMEA SP!,{R2-R7 }        ; EA指定空递增堆栈方式
```

6.4.1 寻址方式的类型

9. 块拷贝寻址

块拷贝寻址与堆栈寻址类似，主要用于LDM/STM指令中的存储器地址的变化方式，只不过此时基址寄存器一般为R0-R12中的一个。

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。

```
LDMIA R0!, {R2-R7}          ; IA指定每次传送后地址加4
STMIB R6!,{R2-R7 }          ; IB指定每次传送前地址加4
```

6.4.2 具体寻址方式

1. 数据处理指令的第2操作数的具体形式

1）#<immediate>

采用立即数寻址方式指定shifter_operand。此时，立即数immediate可以为32位立即数。

```
MOV R4, #0x8000000A          ; #0x8000000A为合法立即数，可由8位的0xA8
                             ;循环右移4位得到
ADD R1, R2, #0x3F0           ; #0x3F0可以由0x3F循环右移28位得到
ADD R1, R2, #0x3FF           ; #0x3FF为非法立即数，此条指令汇编会报错
```

6.4.2 具体寻址方式

2）<Rm>

采用寄存器寻址方式指定shifter_operand。此时，12位shifter_operand的编码为：

bits[3: 0]=Rm，其它8位全部为0；shifter_operand为Rm的值。

3）<Rm>, LSL #<shift_imm>

采用寄存器移位寻址方式指定shifter_operand。这种寻址方式中，是对Rm的值逻辑左移shift_imm位。此时，12位shifter_operand的编码为：

bits[11:7]= shift_imm，bits[3:0]=Rm，bits[6:4]=000。

6.4.2 具体寻址方式

4）<Rm>, LSL <Rs>

采用寄存器移位寻址方式指定shifter_operand。这种寻址方式中，是对Rm的值逻辑左移，左移多少位由Rs的内容决定。此时，12位shifter_operand的编码为：

bits[11:8]=Rs，bits[3:0]=Rm，bits[7:4]=0001。

5）<Rm>, LSR #<shift_imm>

采用寄存器移位寻址方式指定shifter_operand。这种寻址方式中，是对Rm的值逻辑右移shift_imm位。此时，12位shifter_operand的编码为：

bits[11:7]= shift_imm，bits[6:4]=010，bits[3:0]=Rm。

6.4.2 具体寻址方式

6）<Rm>, LSR <Rs>

采用寄存器移位寻址方式指定shifter_operand。这种寻址方式中，是对Rm的值逻辑右移，右移多少位由Rs的内容决定。此时，12位shifter_operand的编码为：

bits[11:8]=Rs，bits[3:0]=Rm，bits[7:4]=0011。

7）<Rm>, ASR #<shift_imm>

采用寄存器移位寻址方式指定shifter_operand。这种寻址方式中，是对Rm的值算术右移shift_imm位。此时，12位shifter_operand的编码为：

bits[11:7]= shift_imm，bits[6:4]=100，bits[3:0]=Rm。

6.4.2 具体寻址方式

8）<Rm>, ASR <Rs>

采用寄存器移位寻址方式指定shifter_operand。这种寻址方式中，是对Rm的值算术右移，右移多少位由Rs的内容决定。此时，12位shifter_operand的编码为：

bits[11:8]=Rs，bits[3:0]=Rm，bits[7:4]=0101。

9）<Rm>, ROR #<shift_imm>

采用寄存器移位寻址方式指定shifter_operand。这种寻址方式中，是对Rm的值循环右移shift_imm位。此时，12位shifter_operand的编码为：

bits[11:7]= shift_imm，bits[6:4]=110，bits[3:0]=Rm。

6.4.2 具体寻址方式

10) <Rm>, ROR <Rs>

采用寄存器移位寻址方式指定shifter_operand。这种寻址方式中，是对Rm的值循环右移，右移多少位由Rs的内容决定。此时，12位shifter_operand的编码为：bits[11:8]=Rs，bits[3:0]=Rm，bits[7:4]=0111。

11) <Rm>, RRX

采用寄存器移位寻址方式指定shifter_operand。这种寻址方式是将[C flag, Rm]一起循环右移一位。

6.4.2 具体寻址方式

2.字和无符号字节Load/Store指令的内存操作数的具体形式

1) [<Rn>, #+/-<offset_12>]{!}

{!}是可选位，若有！，则指令编码的W=1，否则为0。因此，此种语法格式包括W=0或1两种具体方式。12位addr_mode的编码为：bits[11:0]=offset_12。内存地址address计算方式为：

```
if U == 1 then
        address = Rn + offset_12
else /* U == 0 */
        address = Rn - offset_12
if W==1
        Rn = address
```

6.4.2 具体寻址方式

2.字和无符号字节Load/Store指令的内存操作数的具体形式

2) [<Rn>, +/-<Rm>]{!}

{!}是可选位，若有！，则指令编码的W=1，否则为0。因此，此种语法格式包括W=0或1两种具体方式。12位addr_mode的编码为：bits[3:0]=Rm，其它位全部为0。内存地址address计算方式为：

```
if U == 1 then
        address = Rn + Rm
else /* U == 0 */
        address = Rn – Rm
if W==1
        Rn = address
```

6.4.2 具体寻址方式

2.字和无符号字节Load/Store指令的内存操作数的具体形式

3) [<Rn>, +/-<Rm>, <shift> #<shift_imm>]{!}

{!}是可选位，若有！，则指令编码的W=1，否则为0。因此，此种语法格式包括W=0或1两种具体方式。

12位addr_mode的编码为：bits[11:7]=shift_imm，bits[6:5] =shift，bit[4] =0，bits[3:0]=Rm。此处shift_imm用来指定移位值，shift用来指定移位方式，Rm用来指定移位源。

具体的语法格式包括下面5种：

[<Rn>, +/-<Rm>, LSL #<shift_imm>]{!}

[<Rn>, +/-<Rm>, LSR #<shift_imm>]{!}

[<Rn>, +/-<Rm>, ASR #<shift_imm>]{!}

[<Rn>, +/-<Rm>, ROR #<shift_imm>]{!}

[<Rn>, +/-<Rm>, RRX] {!}

6.4.2 具体寻址方式

2.字和无符号字节Load/Store指令的内存操作数的具体形式

4) [<Rn>], #+/-<offset_12>

这种寻址模式的12位addr_mode的编码为：bits[11:0]=offset_12，其对应的内存地址address计算方式为：

```
address = Rn
if U == 1 then
        Rn = Rn + offset_12
else /* U == 0 */
        Rn = Rn - offset_12
```

6.4.2 具体寻址方式

2.字和无符号字节Load/Store指令的内存操作数的具体形式

5) [<Rn>], +/-<Rm>

这种寻址模式的12位addr_mode的编码与“[<Rn>，+/-<Rm>]{!}”的addr_ mode的编码完全一样，差别在于整个指令编码的bit[24]和bit[21]位。

当LDR/ LDRB/STR/STRB采用这种寻址模式时，其bit[24]=0，代表后变址；

此时bit[21]也强制为0。这种寻址模式的内存地址address及Rn的变化方式为：

```
address = Rn
if U == 1 then
        Rn = Rn + Rm
else /* U == 0 */
        Rn = Rn – Rm
```

6.4.2 具体寻址方式

2.字和无符号字节Load/Store指令的内存操作数的具体形式

6) [<Rn>], +/-<Rm>, <shift> #<shift_imm>

这种寻址模式的12位addr_mode的编码与“[<Rn>，+/-<Rm>，<shift> #<shift_imm>]{!}”相同。

差别在于先用Rn的值作为地址进行数据加载/存储，然后再改变Rn的值，

Rn的改变方法与“[<Rn>, +/-<Rm>, <shift> #<shift_imm>]!”雷同。

6.4.2 具体寻址方式

3. 半字/有符号字节Load/Store指令的内存操作数的具体形式

ARM半字/有符号字节Load/Store的寻址方式也是基于基址变址寻址，但实际应用中，又有6种不同的具体形式，用于计算内存操作数的地址。

1) [<Rn>, #+/-<offset_8>]
2) [<Rn>, +/-<Rm>]
3) [<Rn>, #+/-<offset_8>]!
4) [<Rn>, +/-<Rm>]!
5) [<Rn>], #+/-<offset_8>
6) [<Rn>], +/-<Rm>

6.4.2 具体寻址方式

4. LDM/STM指令的内存操作数的具体形式

块拷贝寻址	堆栈寻址	L位	P位	U位	说明
LDMDA (Decrement After)	LDMFA (Full Ascending)	1	0	0	先取后减
LDMIA (Increment After)	LDMFD (Full Descending)	1	0	1	先取后加
LDMDB (Decrement Before)	LDMEA (Empty Ascending)	1	1	0	先减后取
LDMIB (Increment Before)	LDMED (Empty Descending)	1	1	1	先加后取
STMDA (Decrement After)	STMED (Empty Descending)	0	0	0	先存后减
STMIA (Increment After)	STMEA (Empty Ascending)	0	0	1	先存后加
STMDB (Decrement Before)	STMFD (Full Descending)	0	1	0	先减后存
STMIB (Increment Before)	STMFA (Full Ascending)	0	1	1	先加后存

6.5 ARM伪指令与伪操作

05

6.5.1 ARM伪指令

在ARM汇编语言程序里，有一些特殊指令助记符，这些助记符与指令系统的助记符不同，没有相对应的操作码，通常称这些特殊指令助记符为伪指令。

- 伪指令不是真实的ARM指令集中的指令，它们是为了汇编编程方便而定义的。
- 伪指令可以像真实ARM指令一样使用，但在汇编时这些指令将被等效的一条或多条真实ARM指令所代替。
- 伪指令仅在汇编过程中起作用，一旦汇编结束，伪指令的使命就完成。

6.5.1 ARM伪指令

1. ADR/ADRL伪指令

伪指令ADR和ADRL都是将基于PC相对偏移的地址值或基于寄存器相对偏移的地址值加载到寄存器中。在汇编时，ADR伪指令被替换成一条合适的指令，ADRL伪指令被替换成两条合适的指令。

ADR/ADRL伪指令的语法格式为：

ADR{L}{cond}　Rd, expr

6.5.1 ARM伪指令

1. ADR/ADRL伪指令

ADR与ADRL的区别在于，ADR为小范围的地址读取，ADRL是中范围的地址读取，它们的地址读取范围如表所示。

地址值的对齐方式	ADR的地址读取范围	ADRL的地址读取范围
地址值非字对齐	-255~255B	-64~64KB
地址值字对齐	-1020~1020B	-256~256KB

6.5.1 ARM伪指令

1. ADR/ADRL伪指令

（1）用ADR伪指令加载地址，实现查表。

```
        …
        ADR    R0, DISP_TAB          ;加载标签DISP_TAB的地址到R0
        LDRB   R1, [R0, R2]          ;使用R2作为参数，进行查表
        …
DISP_TAB
        DCB 0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90
```

6.5.1 ARM伪指令

1. ADR/ADRL伪指令

（2）ADRL伪指令举例

```
…
ADRL   R1, Delay        ；加载标签Delay的地址到R1
…
Delay
MOV    R0, r14
```

6.5.1 ARM伪指令

2. LDR伪指令

LDR伪指令用于加载32位的立即数或一个地址值到指定寄存器。

在汇编编译源程序时，LDR伪指令被编译器替换成一条合适的指令。

LDR伪指令的语法格式：

LDR{cond} Rd, =[expr | label-expr]

6.5.1 ARM伪指令

2. LDR伪指令的使用

```
    ...
      LDR      R1, =InitStack                    ;加载标签InitStack的地址值到R1
      ...
InitStack
      LDR      SP, =0xFF005000    ;加载32位立即数0xFF005000到SP
       ...
```

6.5.1 ARM伪指令

3. NOP伪指令

NOP伪指令在汇编时将会被代替成ARM中的空操作，即该条指令执行不改变处理器任何状态，比如可用“MOV R0, R0”指令实现NOP伪指令。

通常，NOP用于延时操作。

6.5.1 ARM伪指令

3. NOP伪指令的使用

```
MOV    R1, #0x1234
Delay
NOP                          ;空操作
NOP
NOP
SUBS   R1, R1, #1            ;循环次数减一
BNE    Delay                 ;如果循环没有结束，跳转Delay继续
MOV    PC, LR                ;子程序返回
```

6.5.2 ARM伪操作

在ARM汇编语言程序中，有一些特殊助记符，这些助记符没有相应的操作码，它们所完成的操作称为伪操作。

在汇编源程序设计中，伪操作的作用是为完成汇编程序作各种准备工作。

1. 符号定义伪操作

- GBLA（LCLA）伪操作声明一个全局的（局部的）算术变量，并将其初始化为0
- GBLL（LCLL）伪操作声明一个全局的（局部的）逻辑变量，并将其初始化成{FALSE}
- GBLS（LCLS）伪操作声明一个全局的（局部的）串变量，并将其初始化为空串" "

6.5.2 ARM伪操作

1. 符号定义伪操作

- SETA用于对算术变量赋值
- SETL用于对逻辑变量赋值
- SETS用于对字符串变量赋值

```
Test1 SETA 0xaa          ;将该算术变量赋值为0xaa
Test2 SETL {TRUE}        ;将该逻辑变量赋值为真
Test3 SETS “Testing”     ;将该字符串变量赋值为 “Testing”
```

6.5.2 ARM伪操作

1. 符号定义伪操作

- RLIST伪指令可用于对一个通用寄存器列表定义名称，使用该伪指令定义的名称可在ARM指令LDM/STM中使用。

```
RegList RLIST {R0-R5，R8，R10}        ;将寄存器列表名称定义为RegList
LDMIA R11!, RegList                   ; LDM指令中使用此名称。
```

6.5.2 ARM伪操作

2. 数据定义伪操作

数据定义伪操作一般用于为特定的数据分配存储单元，同时可完成已分配存储单元的初始化

- ✧ DCB：用于分配一片连续的字节存储单元并用指定的数据初始化。
- ✧ DCW：用于分配一片连续的半字存储单元并用指定的数据初始化。
- ✧ DCD：用于分配一片连续的字存储单元并用指定的数据初始化。
- ✧ SPACE：用于分配一片连续的存储单元。

6.5.2 ARM伪操作

2. 数据定义伪操作

语法格式

标号 < DCB/DCW/DCD> 表达式

示例

```
TABLEI  DCB 0xC0,0xF9,0xA4,0xB0        ;分配连续4字节的存储单元并初始化
DataSpace SPACE 100                    ;分配连续100字节的存储单元并初始化为0
```

6.5.2 ARM伪操作

3. 汇编控制伪操作

汇编控制伪操作用于控制汇编程序的执行流程

（1）IF、ELSE、ENDIF

```
IF 逻辑表达式
        指令序列1
ELSE
        指令序列2
ENDIF
```

6.5.2 ARM伪操作

3. 汇编控制伪操作

汇编控制伪操作用于控制汇编程序的执行流程

(2) WHILE、WEND

```
WHILE 逻辑表达式
        指令序列
WEND
```

6.5.2 ARM伪操作

3. 汇编控制伪操作

汇编控制伪操作用于控制汇编程序的执行流程

3）MACRO、MEND

MACRO、MEND伪操作可以将一段代码定义为一个整体，称为宏指令，然后就可以在程序中通过宏指令多次调用该段代码。

MACRO

$标号 宏名 $参数1，$参数2，.....

指令序列

MEND

6.5.2 ARM伪操作

4. 其它伪操作

（1）AREA　　用于定义一个代码段或数据段

AREA Init, CODE, READONLY, ALIEN = 3　　声明Init代码段，只读，指令8字节对齐

（2）CODE32/CODE16

CODE16伪操作指示编译器，其后的指令序列为16位的Thumb指令。

CODE32伪操作指示编译器，其后的指令序列为32位的ARM指令。

6.5.2 ARM伪操作

4. 其它伪操作

(3) ENTRY、END

ENTRY和END伪操作分别用来指定汇编程序的入口点和汇编程序的结束点。

使用示例:

```
AREA Init，CODE，READONLY        ;声明Init代码段
ENTRY                            ;指定程序入口点
指令序列
END                              ;指定程序结尾，其后指令不汇编
```

6.5.2 ARM伪操作

4. 其它伪操作

(4) EQU

EQU伪操作用于为程序中的常量、标号等定义一个等效的字符名称，类似于C语言中的＃define关键字。

使用示例:

```
NUM EQU 50                          ;定义NUM的值为50
FIQAddr EQU 0x100, CODE32           ;定义FIQAddr的值为0x100，且该处为32
                                     位的ARM指令。
```

6.5.2 ARM伪操作

4. 其它伪操作

(5) EXPORT、IMPORT

EXPORT用于通知编译器在本源文件中声明的符号可以在其它文件中引用，IMPORT用于通知变压器当前符号的定义不在本源文件中。这两个符号在多文件的汇编编程中一般要配合使用。

使用示例：　　EXPORT/IMPORT 符号

6.6 ARM汇编程序设计

06

6.6.1 汇编语言结构

ARM汇编语言源程序中语句由指令、伪操作和宏指令组成。

在汇编语言程序设计中，每一条指令的助记符可以全部用大写、或全部用小写，但不能在一条指令中大、小写混用。

- 符号区分大小写，同名的大、小写符号会被认为是两个不同的符号。
- 符号在其作用范围内必须唯一。
- 自定义的符号名不能与系统的保留字相同。
- 符号名不应与指令或伪指令同名。

6.6.1 汇编语言结构

以下是一个汇编语言源程序的基本结构

```
AREA Init，CODE，READONLY ;声明Init代码段
ENTRY                     ;程序入口
Start                     ;标签Start，其值为下面这条指令的地址
LDR R0，=0x3FF5000        ;此处LDR指令为伪指令
MOV R1，#0xFF             ;立即数寻址
STR R1，[R0]
……
END                       ;指示汇编结束
```

6.6.2 汇编语言程序示例

【例6-14】内存拷贝示例。

```
        AREA Word, CODE, READONLY ;声明代码段
num     EQU   20                  ;定义常量num
        ENTRY                     ;指示程序入口点
        CODE32                    ;指示32位的ARM编码
Start   LDR   r0, =src            ;设置源指针
        LDR   r1, =dst            ;设置目的指针
        MOV   r2, #num            ;设置拷贝的字数
        MOV   r7, #0              ;r7置0
Loop    LDR   r3, [r0], #4                ;从源地址加载一个数
```

6.6.2 汇编语言程序示例

【例6-14】内存拷贝示例。

```
Loop   LDR    r3, [r0], #4          ;从源地址加载一个数
       BL     abs                   ;调用子程序求这个数的绝对值
       STR    r3, [r1], #4          ;将数据存到目的地址
       SUBS r2, r2, #1              ;拷贝数量减1
       BNE    Loop                  ;拷贝完成否，未完成到Loop
Stop   MOV   r0, #0x18               ;设置软中断功能号
       LDR    r1, =0x20026          ;设置软中断参数
       SWI    0x123456              ;调用SWI指示仿真完成
abs            CMP   r3, r7                                  ;比较r3和r7
```

6.6.2 汇编语言程序示例

【例6-14】内存拷贝示例。

```
abs     CMP   r3, r7                                ;比较r3和r7
        SUBLT         r3, r7, r3                    ;r3小于r7时执行减法
        MOV   pc, lr                                ;从子程序返回
        AREA BlockData, DATA, READWRITE             ;声明数据段
src     DCD   1,-2,-3,4,5,6,-7,8,-9,10,11,-12,-13,-14,15,16,-17,18,-19,20
                                                    ;用上述值初始化连续的20个字，首地址为src
dst     SPACE 80                                    ;用0初始化连续的80个字节，首地址为dst
```

6.6.2 汇编语言程序示例

【例6-15】冒泡排序示例。

冒泡排序的示例C代码如下：

```
#define N 5
int main(void)
{   int a[N] = {9,5,3,1,7};
    int  i, j, t;
    for (i = 0; i < N-1; i++)
    {  for (j = 0; j < N - i - 1; j++)
        {   if (a[j] > a[j + 1])
            {   t = a[j];
                a[j] = a[j + 1];
                a[j + 1] = t;
            }
        }
```

6.6.2 汇编语言程序示例

【例6-15】冒泡排序示例。

冒泡排序的示例C代码如下：

```
#define N 5
int main(void)
{   int a[N] = {9,5,3,1,7};
    int  i, j, t;
    for (i = 0; i < N-1; i++)
    {  for (j = 0; j < N - i - 1; j++)
        {   if (a[j] > a[j + 1])
            {   t = a[j];
                a[j] = a[j + 1];
                a[j + 1] = t;
            }
        }
```

6.6.2 汇编语言程序示例

【例6-15】冒泡排序示例。

```
AREA main, CODE, READONLY
N               EQU     5
                ENTRY
                CODE32
Start
                MOV   r1, #N
                SUB    r1, r1, #1
                MOV   r2, #0
```

6.6.2 汇编语言程序示例

【例6-15】冒泡排序示例。

```
LOOPi
        LDR    r0, =a
        MOV    r3, #0
LOOPj
        LDR    r4, [r0]
        LDR    r5, [r0,#4]
        CMP    r4, r5
        STRGT          r5, [r0]
```

6.6.2 汇编语言程序示例

【例6-15】冒泡排序示例。

```
        STRGT       r4, [r0,#4]
        SUB    r6, r1, r2
        ADD    r3, r3, #1
        CMP    r6, r3
        ADD    r0, r0, #4
        BGT    LOOPj
        ADD    r2, r2, #1
        CMP    r1, r2
        BGT    LOOPi
```

6.6.2 汇编语言程序示例

【例6-15】冒泡排序示例。

```
stop
        MOV   r0, #0x18
        LDR   r1, =0x20026
        SWI   0x123456
  AREA Array, DATA, READWRITE
a    DCD   9, 5, 3, 1, 7
     END
```

6.6.3 汇编语言与C/C++的混合编程

汇编语言与C/C++的混合编程通常有以下几种方式：

- 在C/C + + 代码中嵌入汇编指令。
- 在汇编程序和C/C + + 的程序之间进行变量的互访。
- 汇编程序、C/C + + 程序间的相互调用。

6.6.3 汇编语言与C/C++的混合编程

1. 在C/C + + 代码中嵌入汇编指令

C内嵌汇编的语法格式如下:

```
__asm__ [__volatile__]
(
代码列表
: 输出运算符列表
: 输入运算符列表
: 被更改资源列表
)
```

示例程序如下:

```
void test(void)
{
int tmp=5;
__asm (
"mov r4,%0\n" : :"r"(tmp) :"r4"
);
}
```

6.6.3 汇编语言与C/C++的混合编程

2. C/C++程序调用汇编程序

C文件代码：

```
extern void strcpy(char *d,const char
*s);     //指出该函数为外部函数
int main(void)
{
......
    strcpy(dest, src);
                        //调用汇编函数
......
}
```

S文件代码:

```
AREA Example, CODE, READONLY
EXPORT strcpy ; EXPORT对外声明汇编
                        中的函数，表示其可被调用
strcpy
    LDRB    r2, [r1], #1
    STRB    r2, [r0], #1
    CMP             r2, #0
    BNE             strcpy
    MOV             pc, lr
    END
```

6.6.3 汇编语言与C/C++的混合编程

3. 汇编程序调用C/C++程序

```
IMPORT Main                          ;通知编译器该标号为一个外部标号
AREA Init, CODE, READONLY            ;定义一个代码段
ENTRY                                ;定义程序的入口点
LDR R0, =0x3FF0000                   ;初始化系统配置寄存器
LDR R1, =0xE7FFFF80
STR R1, [R0]
LDR SP, =0x3FE1000                   ;初始化用户堆栈
BL Main                              ;跳转到Main（）函数处的C/C++代码执行
END                                  ;标识汇编程序的结束
```

第六章　完

开课学院：通信学院
授课老师：姚英彪