《嵌入式系统原理与应用技术》

袁志勇 王景存
章登义 刘树波

北京: 北京航空航天大学出版社, 2009.11

PPT教学课件

Ch3 ARM指令系统

3.1 ARM指令集

3.2 ARM汇编伪指令与伪操作

ARM指令集总体可分为以下几类

- 数据处理指令：数据传输指令，算术指令，逻辑指令，比较指令，乘法指令，前导零计数
- 程序状态访问指令：MRS和MSR
- 分支/跳转指令：B、BL和BX
- 访存指令：单数据访存指令，多数据访存指令，信号量操作指令
- 异常中断产生指令：SWI和BKPT
- 协处理器指令

3.1 ARM指令集

ARM嵌入式微处理器是基于精简指令集计算机(RISC)原理而设计的，指令集和相关译码机制较为简单。ARM9具有32位ARM指令和16位Thumb指令。

3.1.1 ARM指令分类及格式

ARM嵌入式微处理器的指令集是加载、存储型的，即指令集中仅能处理寄存器中的数据，而且处理结果都要写回寄存器中，而对存储器的访问则需要通过专门的加载、存储指令来完成。ARM指令可分为以下6类：

- 数据处理指令：数据传送指令，算术指令，逻辑指令，比较指令，乘法指令，前导零计数
- 程序状态访问指令：MRS和MSR
- 分支指令：B、BL和BX
- 访存指令：单数据访存指令，多数据访存指令，数据交换指令
- 异常产生指令：SWI和BKPT
- 协处理器指令：CDP、LDC、STC、MCR、MRC

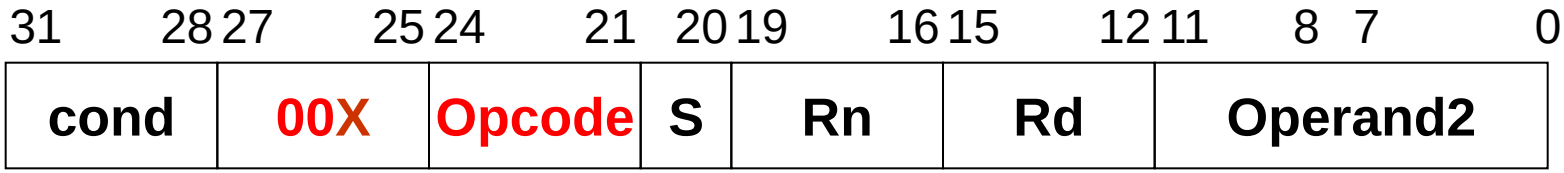


图3.1 ARM数据处理类指令编码格式

这里以ARM数据处理类指令为例，说明ARM指令格式。ARM数据处理类指令编码基本格式见图3.1所示。

ARM数据处理指令基本格式如下：

<Opcode>{<cond>}{s} <Rd>, <Rn>, <Operand2>

其中，<>内的项是必须的，{}内的项是可选的。如<Opcode>是指令助记符，是必须的，而{<cond>}是指令执行条件，是可选的，如果不写则使用默认条件AL(无条件执行)。

- cond: 指令的条件码。
- Opcode: 指令操作码(有16种编码,对应于16条指令,见下页)。
- s: 操作是否影响cpsr。
- Rn: 表示第1个操作数的寄存器编码。
- Rd: 目标寄存器编码。
- Operand2: 第2操作数(立即数/寄存器/寄存器移位)。
- X:1-第2操作数是立即数寻址, 0-第2操作数是寄存器寻址

按照图3.1所示的编码格式，操作码Opcode (未含乘法指令)所对应的指令助记符及其含义如下：

- **Opcode: bit[24:21] Operation codes(未含乘法指令)**

0000 = AND-Rd: = Op1 AND Op2
0001 = EOR-Rd: = Op1 EOR Op2
0010 = SUB-Rd: = Op1-Op2
0011 = RSB-Rd: = Op2-Op1
0100 = ADD-Rd: = Op1+Op2
0101 = ADC-Rd: = Op1+Op2+C
0110 = SBC-Rd: = OP1-Op2+C-1
0111 = RSC-Rd: = Op2-Op1+C-1
1000 = TST-set condition codes on Op1 AND Op2
1001 = TEQ-set condition codes on OP1 EOR Op2
1010 = CMP-set condition codes on Op1-Op2
1011 = CMN-set condition codes on Op1+Op2
1100 = ORR-Rd: = Op1 OR Op2
1101 = MOV-Rd: =Op2
1110 = BIC-Rd: = Op1 AND NOT Op2
1111 = MVN-Rd: = NOT Op2

- **条件码的位数和位置：**每条ARM指令包含4位条件码域<cond>,它占用指令编码的最高4位[31:28]。
- **条件码的表示：**条件编码共 2^4 ＝16 种，其中，15种用于指令的条件码。每种条件码用2个英文缩写字符表示。
- **带条件指令的执行：**ARM处理器根据指令的执行条件是否满足，决定当前指令是否执行。

  只有在cpsr中的条件标志位满足指定的条件时，指令才会被执行。不符合条件的代码依然占用一个时钟周期（相当于一个NOP指令）。

- **条件码的书写方法：**条件码的位置在指令助记符的后面（因此也称为条件后缀）。

  例如：　　MOVEQ　R0，R1

教材P50 表3.1 指令条件码

条件码	助记符	含　义	标 志
0000	EQ	相等	Z=1
0001	NE	不相等	Z=0
0010	CS/HS	无符号数大于或等于	C=1
0011	CC/LO	无符号数小于	C=0
0100	MI	负数	N=1
0101	PL	非负数	N=0
0110	VS	溢出	V=1
0111	VC	没有溢出	V=0
1000	HI	无符号数大于	C=1且Z=0
1001	LS	无符号数小于或等于	C=0或Z=1
1010	GE	有符号数大于或等于	N=V
1011	LT	有符号数小于	N!=V
1100	GT	有符号数大于	Z=0且N=V
1101	LE	有符号数小于或等于	Z=1或N!=V
1110	AL	无条件执行	任意
1111	保留	v5以下版本总执行,v5及以上版本有用	

(1) ARM数据处理指令的功能

主要完成寄存器中数据的算术和逻辑运算操作。

(2) ARM数据处理指令的特点

--操作数来源：所有的操作数要么来自寄存器，要么来自立即数，不会来自存储器。

--操作结果：如果有结果，则结果一定是为32位宽、或64位宽(长乘法指令)，并且放在一个或两个寄存器中，不会写入存储器。

--有第2个操作数(除了乘法指令)Operand2：切记其三种形式：立即数、寄存器、寄存器移位。

--乘法指令的操作数：全部是寄存器。

3.1.2 ARM指令寻址方式

所谓**寻址方式**就是处理器根据指令中给出的地址信息来寻找操作数物理地址的方式。目前ARM处理器支持几种常见的寻址方式。

1. 寄存器寻址

寄存器寻址是指所需要的值在寄存器中，指令中地址码给出的是寄存器编号，即寄存器的内容为操作数。

例：

```
ADD  R0, R1, R2   ；R0←R1+R2
```

2. 立即寻址

立即寻址是一种特殊的寻址方式，指令中在操作码字段后面的地址码部分不是操作数地址，而是操作数本身。

例：

ADD R3, R3, #10 ； R3←R3+10

立即数要以“#”号作前缀，以十进制数10为例：它的16进制立即数为#0xa;它的2进制立即数为#0b1010。

关于立即数的构成，可参考图3.2。

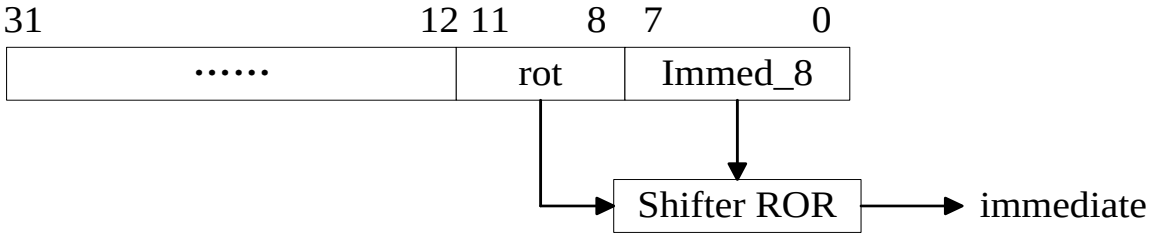


图3.2 立即数构成示意图

从图3.2可知，有效的立即数可以表示为：

<immediate>=Immed_8 循环右移2 × rot位

由于4 位rot移位值的取值 (0~15)乘于2，得到一个范围在0~30、步长为 2 的移位值。　　因此，ARM中的立即数又称为8位位图。我们只需记住一条准则：“最低8位Immed_8移动偶数位”得到立即数。只有通过此构造方法得到的立即数才是合法的。

下面是三条带有立即数的MOV 指令及对应的机器码，请注意机器码中的立即数计算：

```
MOV  R0, #0xF200        ; E3A00CF2, 0xF200 =0xF2循环右移(2×C)
MOV  R1, #0x110000      ; E3A01811, 0x110000 =0x11循环右移(2×8)
MOV  R4, #0x12800       ; E3A04B4A, 0x12800 =0x4A循环右移(2×B)
```

又如，0xFF、0x104(其8位图为0x41)、0xFF0、0xFF00是合法的立即数； 0x101、0x102、0xFF1是非法的立即数。

问： MOV R0,#0x104的机器码是什么？

3. 寄存器移位寻址

寄存器移位寻址方式是ARM指令集中所特有的，第二个寄存器操作数在与第一个操作数结合之前，选择进行移位操作。在寄存器移位寻址中，移位的位数可以用立即数或寄存器方式表示。

例：

```
ADD R3, R2, R1, LSL #3
; R3←R2+8×R1(即R1中的值向左移3位，与R2中的值相加，结果存入R3)
MOV R0,R1,ROR R2     ;R0←R1循环右移R2位
```

ARM中有一桶式移位器，途经它的操作数在被使用前能够被移位或循环移位任意位数，这在处理列表、表格和其他复杂数据结构时非常有用。

ARM中常用的几种移位操作指令：

(1)算术右移ASR (Arithmetic Shift Right)

存储第二操作数的寄存器算术右移。算术移位的操作数是带符号数，完成移位时应该保持操作数的符号不变。因此，当被移位的操作数为正数时，寄存器的高端空出位补0；当被移位的操作数为负数时，寄存器的高端空出位补1。

(2)逻辑左移LSL (Logical Shift Left)

存储第二操作数的寄存器逻辑左移。寄存器中的高端送至C标志位，低端空出位补0。

(3)逻辑右移LSR (Logical Shift Right)

存储第二操作数的寄存器逻辑右移。寄存器中的高端空出位补0。

(4)循环右移ROR (ROtate Right)

存储第二操作数的寄存器循环右移。从寄存器低端移出的位填入到寄存器高端的空出位上。

(5)扩展的循环右移RRX (Rotate Right eXtended)

存储第二操作数的寄存器进行带进位位的循环右移。每右移一位，寄存器中高端空出位用原C标志位的值填充。

若移位的位数由5位立即数(取值范围0-31)给出，就叫作立即数控制移位方式(immediate specified shift)；若移位的位数由通用寄存器(不能是R15)的低5位决定，就叫做寄存器控制移位方式(register specified shift) 。

关于寄存器控制移位方式，有如下两点需要说明：

- 移位的寄存器不能是PC，否则会产生不可预知的结果。
- 使用寄存器控制移位方式有额外代价（overhead），需要更多的周期才能完成指令，因为ARM没有能力一次读取3个寄存器。

立即数控制移位方式则没有上述问题。

4．寄存器间接寻址

寄存器间接寻址是指指令中的地址码给出的是某一通用寄存器的编号，在被指定的寄存器中存放操作数的有效地址，而操作数则存放在该地址对应的存储单元中，即寄存器为地址指针。

例：

LDR R0, [R1] ； R0←[R1]

5．变址寻址

变址寻址(或基址变址寻址)就是将基址寄存器的内容与指令中给出的偏移量相加，形成操作数有效地址。变址寻址用于访问基址附近的单元，包括基址加偏移和基址加索引寻址。寄存器间接寻址是偏移量为0的基址加偏移寻址。

基址加（或减）最大4KB的偏移量得到操作数的有效地址。

例：

LDR R0, [R1, #4] ; R0←[R1+4]

有三种加偏移量(偏移地址)的变址寻址方式:

（1）**前变址方式**(pre-indexed)

先将基地址加上偏移量，生成操作数地址，再做指令指定的操作。该方式不修改基址寄存器。以上的例子即为此方式。

（2）**自动变址方式**(auto-indexed)

先将基地址加上偏移量，生成操作数地址，再做指令指定的操作；然后再自动修改基址寄存器。

例:

LDR R0, [R1, #4]!　　; R0←[R1＋4], R1←R1＋4

说明：！表示**写回或更新基址寄存器**。

（3）**后变址方式**(post-indexed)

先将基址寄存器作为操作数地址，完成指令操作后，再将基地址加上偏移量修改基址寄存器。即先用基地址传数，然后再修改基地址(基址+偏移)。

例:

STR R0, [R1], #12　　; [R1]←R0, R1←R1＋12

这里R1是基址寄存器。

!: Writes back the base register (set the W bit) if ! is present.

6．多寄存器寻址

多寄存器寻址是指一条指令可以传送多个寄存器的值，允许一条指令传送16个寄存器的任何子集。

例：

```
LDMIA  R1, {R0, R2, R5}
; R0←[R1], R2←[R1+4], R5←[R1+8]
```

上面指令的含义是将R1所指向的连续3个存储单元中的内容分别送到寄存器R0,R2,R5中。

由于传送的数据项总是32位的字，基址R1应该字对准。

7．堆栈寻址

堆栈是一种按特定顺序进行存取的存储区，这种特定顺序即是“先进后出”或“后进先出”。堆栈寻址是隐含的，它使用一个专门的寄存器（堆栈指针）指向一块存储器区域。栈指针所指定的存储单元就是堆栈的栈顶。

堆栈可分为两种：

- 向上生长：又称递增(Ascending)堆栈，即地址向高地址方向生长。
- 向下生长：又称递减(Decending)堆栈，即地址向低地址方向生长。

若SP指向最后压入的堆栈的有效数据单元，称为满堆栈(Full Stack)；若SP指向下一个数据项放入的空单元，称为空堆栈(Empty stack)。

ARM处理器支持上面4种类型的堆栈工作方式：

- 满递增堆栈FA (Full Ascending)：堆栈指针指向最后压入的数据单元，且由低地址向高地址生成；
- 满递减堆栈FD (Full Descending)：堆栈指针指向最后压入的数据单元，且由高地址向低地址生成；
- 空递增堆栈EA (Empty Ascending)：堆栈指针指向下一个将要放入数据的空单元，且由低地址向高地址生成；
- 空递减堆栈ED (Empty Descending)：堆栈指针指向下一个将要放入数据的空单元，且由高地址向低地址生成；

例：

STMFD sp!, {r4-r7, lr}　　；将 r4~r7、lr入栈，满递减堆栈

LDMFD sp!, {r4-r7, pc}　；数据出栈，放入r4~r7、pc寄存器

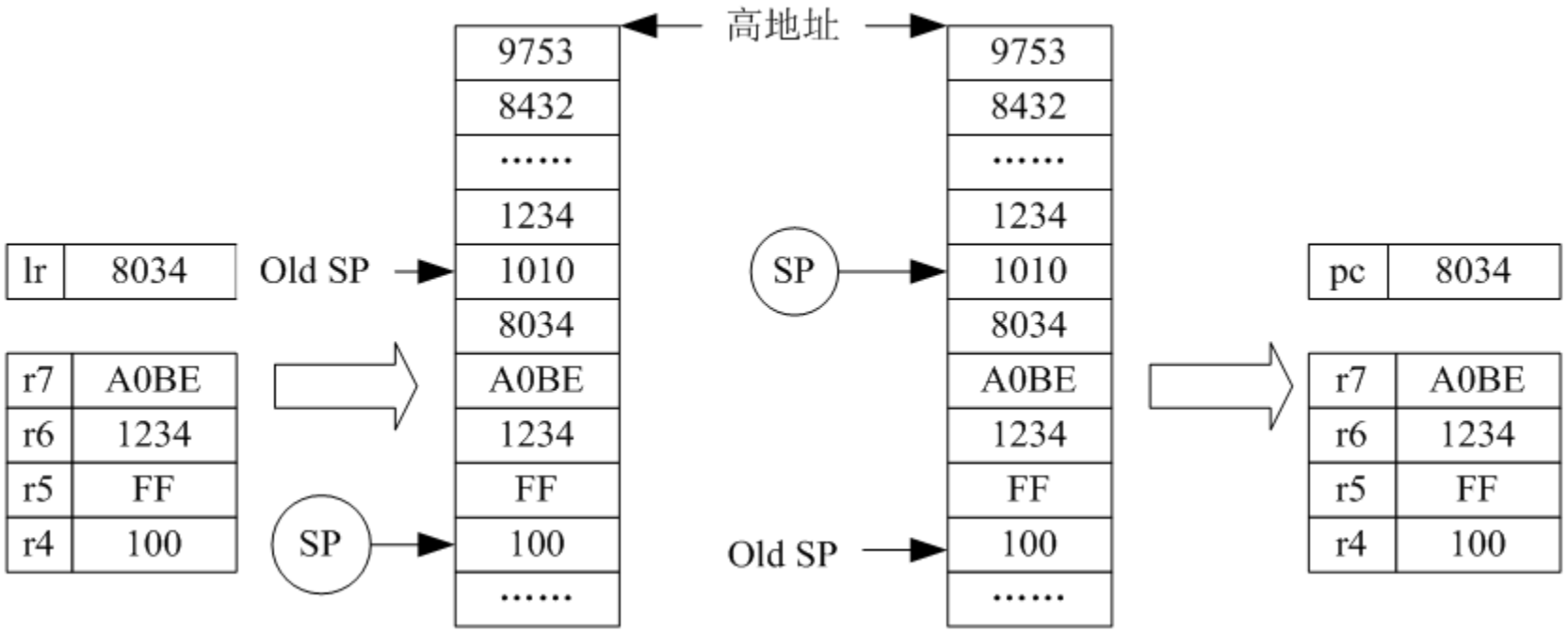


图3.3 本例堆栈操作示意图

满递减堆栈：STMFD指令相当于STMDB指令，LDMFD指令相当于LDMIA指令

8. 块拷贝(复制)寻址

块拷贝寻址是指把存储器中的一个数据块加载到多个寄存器中，或者是把多个寄存器中的内容保存到存储器中。

- 块拷贝寻址是多寄存器传送指令LDM/STM的寻址方式，因此也叫**多寄存器寻址**；
- 多寄存器传送指令用于把一块数据从存储器的某一位置拷贝到另一位置；
- 块拷贝指令的寻址操作取决于数据是存储在基址寄存器所指的地址之上还是之下、地址是递增还是递减，并与数据的存取操作有关；
- 块拷贝寻址操作中的寄存器，可以是R0~R15这16个寄存器的子集，或是所有寄存器。

几种块拷贝指令及其寻址操作说明如下：

LDMIA/STMIA：先传送，后地址加4(Increment After,事后递增)；
LDMIB/STMIB：先地址加4，后传送(Increment Before, 事先递增)；
LDMDA/STMDA：先传送，后地址减4 (Decrement After, 事后递减)；
LDMDB/STMDB：先地址减4，后传送(Decrement Before, 事先递减)。

例：

```
STMIA  r10, {r0, r1, r4};
STMIB  r10, {r0, r1, r4};
STMDA  r10, {r0, r1, r4};
STMDB  r10, {r0, r1, r4};
```

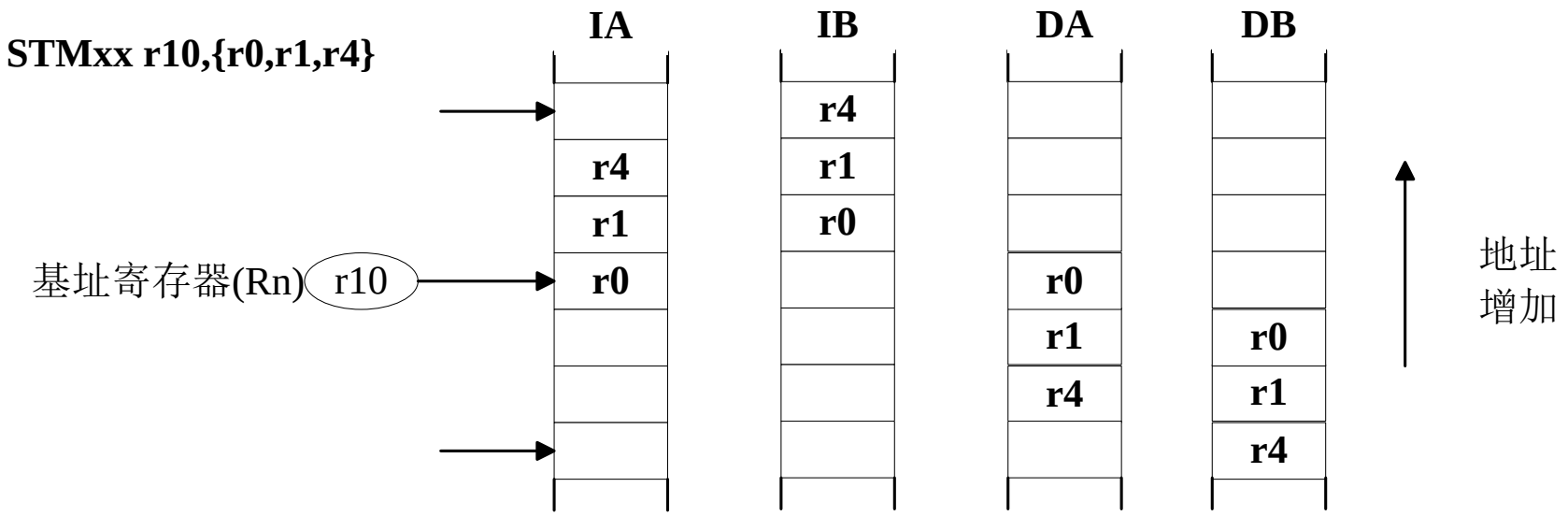


图3.4 本例的块拷贝寻址操作示意图

表3.2 多寄存器load和store指令的堆栈和块拷贝对照

栈生长 / 顶空满	地址增减次序	栈、块递增		栈、块递减	
		顶满	顶空	顶满	顶空
地址增加	先增	STMIB STMFA			LDMIB LDMED
	后增		STMIA STMEA	LDMIA LDMFD	
地址减少	先减		LDMDB LDMEA	STMDB STMFD	
	后减	LDMDA LDMFA			STMDA STMED

9．相对寻址

相对寻址是变址寻址的一种变通，由程序计数器PC提供基地址，指令中的地址码字段作为偏移量，两者相加后得到操作数的有效地址。偏移量指出的是操作数与当前指令之间的相对位置。子程序调用指令BL即是相对寻址指令。

例:

```
BL     ROUTE_A          ; 调用ROUTE_A子程序
BEQ    LOOP             ; 条件跳转到LOOP标号处
......
LOOP     MOV  R2, #2
......
ROUTE_A: ......
```

3.1.3 常用ARM指令

数据传送指令：MOV、MVN

- 语法
  - <Operation>{<cond>}{s} <Rd>, #<immed>
  - <Operation>{<cond>}{s} <Rd>, <Rm>
  - <Operation>{<cond>}{s} <Rd>, <Rm>, LSL #<immed_5>
  - <Operation>{<cond>}{s} <Rd>, <Rm>, LSL <Rs>
- 伪代码

```
if ConditionPassed(cond) then
        Rd = 第2操作数
        if s == 1 and Rd == pc then
                cpsr == spsr
        else if s == 1 then
                set NZCV flags in cpsr
```

MVN: move negative

MOV指令的功能

- 寄存器之间传送。
- 立即数传送到寄存器中（8位立即数位图）。
- 实现单纯的移位操作。MOV Rd, Rd, LSL, #3
- 实现子程序调用、从子程序中返回。当PC寄存器作为目标寄存器时可以实现程序跳转。
- 实现把当前处理器模式的SPSR寄存器内容复制到CPSR中。

  方法：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。这样可以实现从某些异常中断中返回。

  例子：MOVS PC，LR

- mvn意为“取反传送”，它把源寄存器的每一位取反，将得到的结果写入结果寄存器。

- 对于mov和mvn指令，汇编器会对其进行智能转化。如指令“mov r1, 0xffffff00”中的立即数是非法的。在汇编时，汇编器将其转化为“mvn r1, 0xff”，这样就不违背立即数的要求。所以对于mov和mvn指令，可以认为：合法的立即数反码也是合法的立即数。

- 举例
  - mov r0, r1 /* r0 = r1，不修改cpsr */
  - mov r0, #0x0 /* r0 = 0，不修改cpsr */
  - movs r0, #0x0 /* r0 = 0，同时设置cpsr的Z位 */
  - movs r0, #-10 /* r0 = 0xfffffff6，同时设置cpsr的N位 */
  - mvn r0, r2 /* r0 = NOT r2，不修改cpsr */
  - mvn r0, 0xffffffff /* r0 = 0x0，不修改cpsr */
  - mvns r0, 0xffffffff /* r0 = 0x0，同时设置cpsr的Z位 */
  - mov r0, r1, LSL #1 /* r0 = r1 << 1 */
  - mov r0, r1, LSR r2 /* r0 = r1 >> r2 */
  - movs r3, r1, LSL #2 /* r3＝r1<<2，影响标志位 */

算术指令: add、adc、sub、sbc 、 rsb 、rsc

- 语法
  - <Operation>{<cond>}{s} <Rd>, <Rn>, #<immed>
  - <Operation>{<cond>}{s} <Rd>, <Rn>, <Rm>
  - <Operation>{<cond>}{s} <Rd>, <Rn>, <Rm>, LSL #<immed_5>
  - <Operation>{<cond>}{s} <Rd>, <Rn>, <Rm>, LSL <Rs>
- 伪代码（以加法add为例）

```
if ConditionPassed(cond) then
        Rd = Rn + 第2操作数
        if s == 1 and Rd == pc then
                cpsr = spsr
        else if s == 1 then
                set NZCV flags in cpsr
```

RSB: Reverse Subtract

RSC: Reverse Subtract with Carry

- 说明：
- ADD，SUB，RSB不带进位或借位
- ADC，SBC ，RSC带进位或借位。
- 其句法是：
- op {cond} {S} Rd，Rn，Operand2
- ADD指令用于将Rn和Operand2的值相加；
- SUB指令用于从Rn的值中减去Operand2的值；
- RSB指令用于从Operand2的值中减去Rn的值；

- ADC指令用于将Rn和Operand2的值相加，再加上进位标志C的值；
- SBC指令用于从Rn的值中减去Operand2的值，若进位标志C为0，结果再减1；
- RSC指令用于从Operand2的值中减去Rn的值，若进位标志C为0，结果再减1。
- 以上指令执行的结果均存于Rd中。
- S为可选的后缀。若指定S，则根据操作结果更新条件码标志

- **注意：**
- 若在这些指令后面加上后缀S，那么这些指令将根据其运算结果更新标志N，Z，C和V。
- 若R15作为Rn使用，则使用的值是当前指令的地址加8。
- 若R15作为Rd使用，则执行完指令后，程序将转移到结果对应的地址处。若此时指令还加有后缀S，则还会将当前模式的SPSR拷贝到CPSR。可以使用这一点从异常返回。
- 在有寄存器控制移位的任何数据处理指令中，不能将R15作为Rd或任何操作数来使用。

- 例：
- ADD　R3, R7, #1020

  ；immediate为1020(0x3FC)，是0xFF循环右移30位
- SUBS　R8, R6, #240

  ；R8←R6-240，运算完成后将根据结果更新标志
- RSB　R4, R4, #1280

  ；R4←1280-R4
- RSCLES　R0, R5, R0, LSL R4

  ；有条件执行，执行完后更新标志

逻辑指令

- AND，ORR，EOR和BIC指令
- 逻辑与、或、异或等指令。其句法是：
- op {cond} {S} Rd，Rn，Operand2
- 其中所用到的符号意义与前述的相同。
- AND，ORR和EOR指令分别完成按位将Rn和Operand2的值进行“与”、“或”和“异或”操作，结果存于Rd中。BIC指令用于将Rn中的各位与Operand2的相应位的反码进行“与”操作，结果存于Rd中。

- **使用这些指令时应注意：**若加有后缀S，这些指令执行完后将根据结果更新标志N和Z，在计算Operand2时更新标志C，不影响标志V。

  如：　EORS　R0, R0, R3, ROR R6

  　　BICNES　R8, R10, R0, RRX

逻辑运算指令举例：

```
ANDS  R0，R0, #0x01        ; R0＝R0&0x01
                           ; 取出最低位数据
AND   R2, R1, R3           ; R2＝R1&R3
```

·AND指令可用于提取寄存器中某些位的值。

```
ORR   R0, R0, #0x0F        ; 将R0的低4位置1
```

·ORR指令用于将寄存器中某些位的值设置成1。

```
EOR   R1, R1, #0x0F        ; 将RI的低4位取反
EORS  R0，R5，#0x01        ; 将R0←R5异或0x01，
                           ; 并影响标志位
```

· EOR指令可用于将寄存器中某些位的值取反。将某一位与0异或，该位值不变；与1异或，该位值被求反。

```
BIC   R1，R1，#0x0F        ; 将R1的低4位清0，
                           ; 其它位不变
```

· BIC指令可用于将寄存器中某些位的值设置成0。将某一位与1做BIC操作，该位值被设置成0；将某一位与0做BIC操作，该位值不变。

比较指令

- CMP和CMN指令
- 比较和比较反值指令。其句法如下：
- op {cond} Rn，Operand2
- 其中所用到的符号意义与上述的相同。
- CMP指令从Rn的值中减去Operand2的值。除结果被舍弃外，其他与SUBS指令一样；
- CMN(Compare Negative)指令将Operand2的值加到Rn的值中，除结果被舍弃外，其他与ADDS指令一样。CMP和CMN指令执行后均根据结果更新标志N、Z、C和V。

CMN: Compare Negative

- **例：**

```
CMP     R2,R9       ; N、Z、C、V←R2-R9
CMN     R0,#6400    ; N、Z、C、V←#6400-R0
CMPGT   R13,R7,LSL #2 ;带符号大于
```

- **TST和TEQ指令**
- 测试和测试相等指令，其句法如下：
- op　{cond}　Rn，Operand2
- 其中所用到的符号意义与上述的相同。
- TST指令对Rn的值和Operand2的值进行**按位与**，除结果被舍弃外，其他与ANDS指令一样；
- TEQ指令对Rn的值和Operand2的值进行**按位异或**，除结果被舍弃外，其他与EORS指令一样。
- TST和TEQ指令执行后均根据结果更新标志N、Z、C和V。

- 例：

```
TST   R0, #0x01    ；判断R0的最低位是否为0
TST   Rl, #0x0F    ；判断R1的低4位是否为0
TEQ   R0，R1       ；比较R0与R1是否相等
                   ；(不影响V位和C位)
```

TST指令通常与EQ、NE条件码配合使用。当所有测试位均为0时，EQ有效。而只要有一个测试位不为0，则NE有效。

例：

```
TSTNE   R1，R5，ASR R1
```

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

例：

```
TEQEQ   R10，R9
```

乘法指令(*:了解)

- ARM有两类乘法指令：
  - 32位的乘法指令，即乘法操作的结果为32位；
  - 64位的乘法指令，即乘法操作的结果为64位。

MUL Rd,Rm,Rs	32位乘法指令
MUA Rd,Rm,Rs,Rn	32位乘加指令
UMULL RdL,RdH,Rm,Rs,Rn	64位无符号乘法
UMAL RdL,RdH,Rm,Rs,Rn	64位无符号乘加
SMULL RdL,RdH,Rm,Rs,Rn	64位有符号乘法
SMLAL RdL,RdH,Rm,Rs,Rn	64位有符号乘加

◆ 程序状态访问指令

- 当需要修改cpsr/spsr的内容时，首先要读取它的值到一个通用寄存器，然后修改某些位，最后将数据写回到状态寄存器(即：**修改状态寄存器一般是通过“读取－修改－写回”三个步骤的操作来实现的**)。

- cpsr/spsr不是通用寄存器，不能使用mov指令来读写。在ARM处理器中，只有mrs指令可以读取cpsr/spsr；只有msr可以写cpsr/spsr。

31	30	29	28	27	26…8	7	6	5	4	3	2	1	0
N	Z	C	V	Q	(保留)	I	F	T	M4	M3	M2	M1	M0

图　ARM状态寄存器PSR(CPSR/SPSR)的格式

条件码标志位（保存ALU中的当前操作信息）

N：正负号/大小 标志位

0表示：正数/大于；1表示：负数/小于

Z：零标志位

0表示：结果不为零；1表示：结果为零

C：进位/借位/移出位

0表示：未进位/借位/移出0；1表示：进位/未借位/移出1

V：溢出标志位

0表示：结果未溢出；1表示：结果溢出

Q：DSP指令溢出标志位（用于v5以上E系列）

0表示：结果未溢出；1表示：结果溢出

1. MRS-读状态寄存器指令

(Move PSR status/flags to register)

指令格式如下：

MRS{cond}　Rd，psr　　；Rd ← psr

把状态寄存器psr（CPSR/SPSR）的内容传送到目标寄存器中。

其中：　Rd —— 目标寄存器。Rd不允许为R15。

psr —— CPSR或SPSR。

注意：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。

MRS指令举例：

```
MRS   R1，CPSR          ；  R1← CPSR
MRS   R2，SPSR          ；  R2 ← SPSR
```

MRS与MSR指令的应用目的：

- **获得CPSR或SPSR的状态**
  - 用MRS指令读取CPSR，可用来判断ALU的状态标志，或IRQ、FIQ中断是否允许等。
  - 用MRS指令在异常处理程序中，读SPSR可知道进行异常前的处理器状态等。
- **对CPSR或SPSR进行修改**

  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。

2. MSR——写状态寄存器指令

(Move register to PSR status/flags)

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:

MSR{cond}　psr_fields，#immed

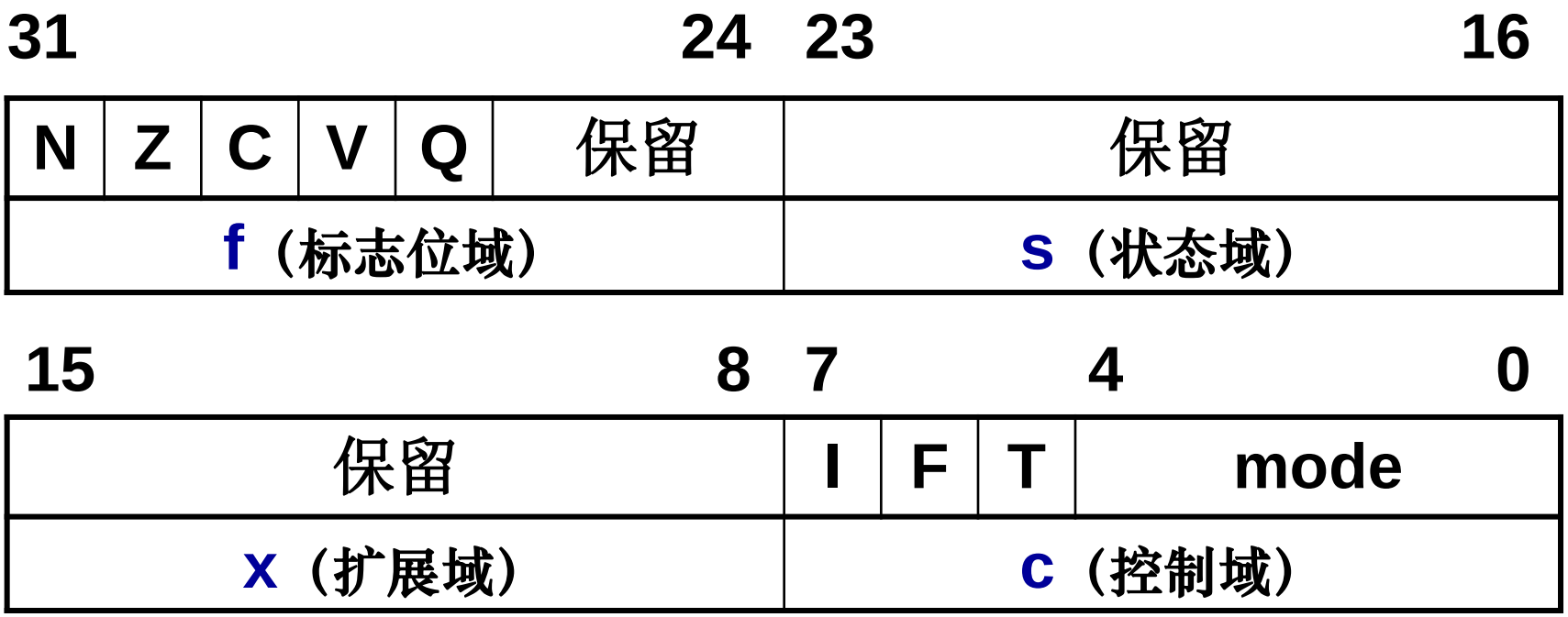
MSR{cond}　psr_fields，Rm

; psr_fields←Rd/#immed

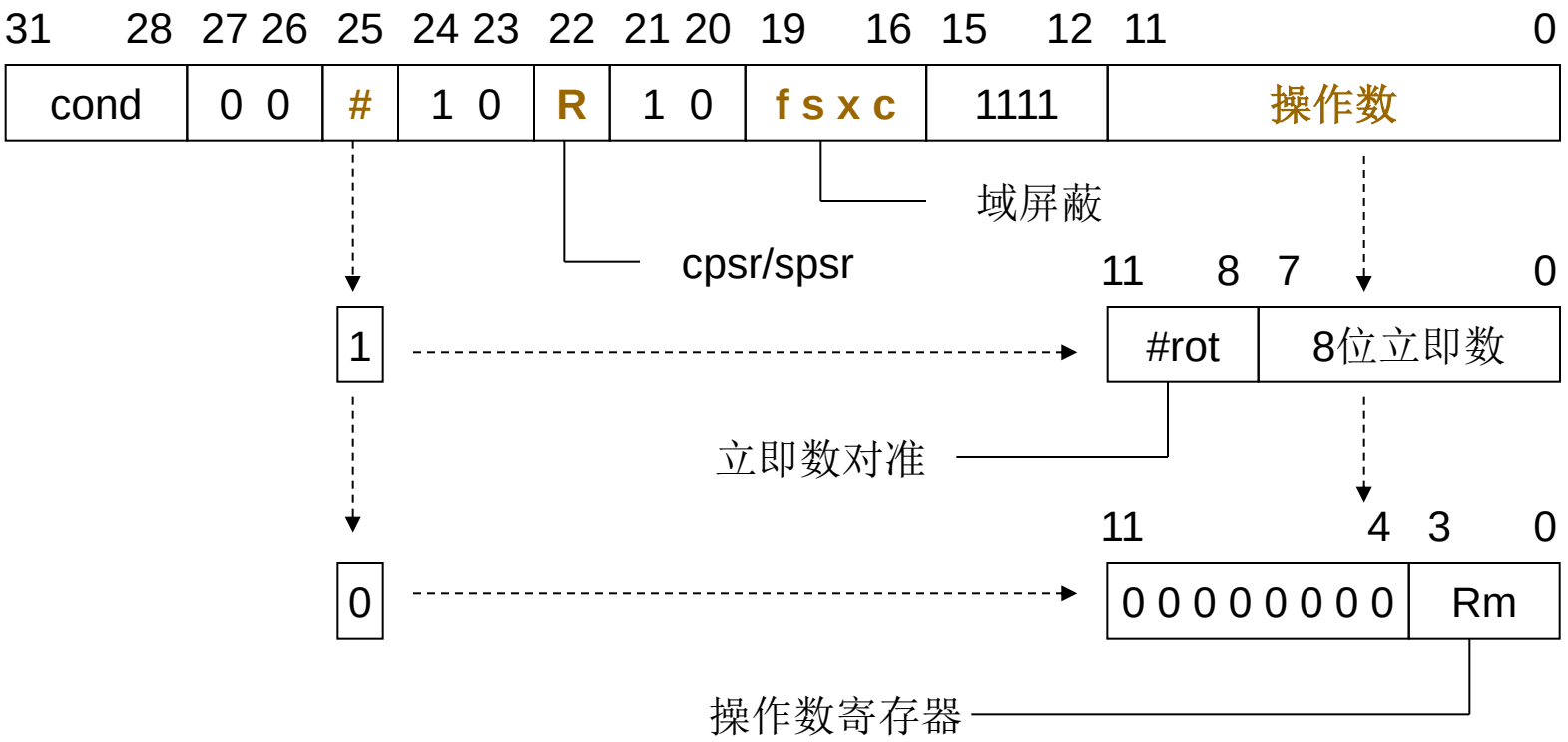
其中:

- **psr:**　CPSR/SPSR。
- **immed:**　要传送到状态寄存器指定域的8位立即数。
- **Rm:**　要传送到状态寄存器指定域的数据的源寄存器。

- **fields**　指定传送的区域。fields可以是以下的一种或多种(字母必须为小写):
  - c 控制域 (psr[7…0]);
  - x 扩展域(psr[15…8]); (暂未用)
  - s 状态域 (psr[23…16]); (暂未用)
  - f 标志位域 (psr[31…24])。



写状态寄存器指令MSR的二进制编码格式



- 指令举例：

```
MSR   CPSR_f，#0xf0        ；CPSR[31:28]＝0xf(0b1111)
                           ；即N,Z,C,V均被置1。
```

修改状态寄存器一般是通过“读取－修改－写回”三个步骤的操作来实现的。

CPSR的读—修改—写操作举例：

例1：设置进位位C

```
MRS        R0, CPSR                ;R0←CPSR
ORR        R0,R0,#0x20000000       ;置1进位位C
MSR        CPSR_f, R0              ;CPSR_f←R0[31:24]
```

例2：从管理模式切换到IRQ模式

```
MRS        R0, CPSR                ;R0←CPSR
BIC        R0,R0,#0x1f             ;低5位清零
ORR        R0,R0,#0x12             ;设置为IRQ模式(P29表2.3)
MSR        CPSR_c, R0              ;传送回CPSR
```

例3：

```
MSR        CPSR_c, #0xd3           ;切换到SVC模式
MSR        CPSR_cxsf, R3           ;CPSR = r3
```

注意：

- 控制域的修改问题：只有在特权模式下才能修改状态寄存器的控制域[7:0]，以实现处理器模式转换，或设置开/关异常中断 。
- T控制位的修改问题：程序中不能通过MSR指令，直接修改CPSR中的T控制位来实现ARM状态 / Thumb状态的切换，必须使用BX指令完成处理器状态的切换。
- 用户模式下能够修改的位：在用户模式只能修改“标志位域”，不能对CPSR[23:0]做修改。
- S后缀的使用问题：在MRS/MSR指令中不可以使用S后缀。

◆ 跳转/分支指令

1. B——转移指令

指令格式：

B{cond}　label

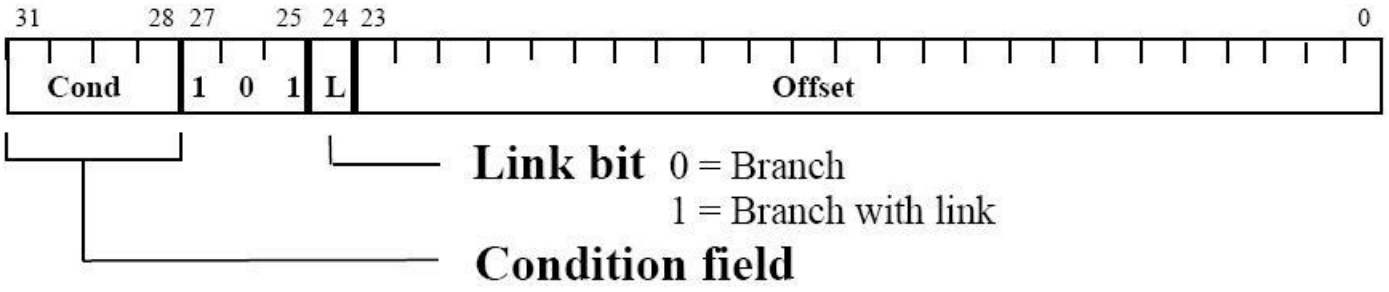
B指令跳转到指定的地址执行程序。

指令举例：

B　WAITA　；跳转到WAITA标号处

B　0x1234　；跳转到绝对地址0x1234处

转移指令B限制在当前指令的±32 MB的范围内。



◆ 跳转/分支指令

1. B——转移指令

指令格式：

B{cond}　label

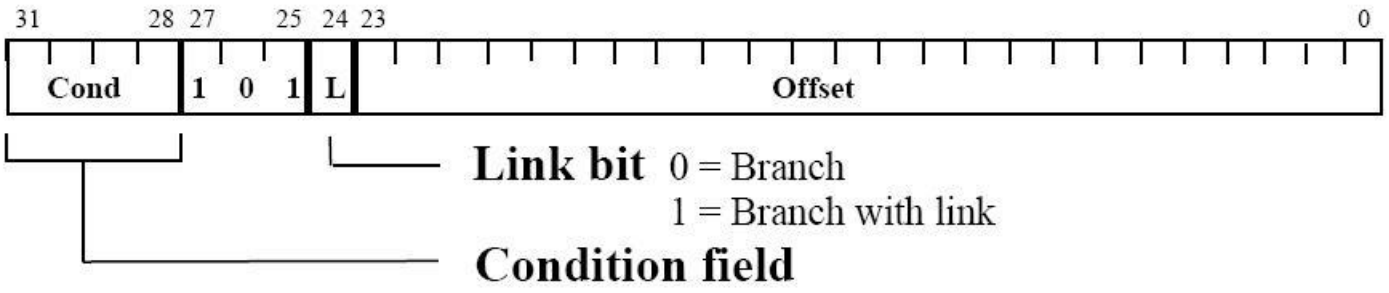
B指令跳转到指定的地址执行程序。

指令举例：

B　WAITA　；跳转到WAITA标号处

B　0x1234　；跳转到绝对地址0x1234处

转移指令B限制在当前指令的±32 MB的范围内。



举例：

无条件跳转：

```
        B     label
        ……
label   ……
```

执行10次循环：

```
        MOV     R0, #10
LOOP
        ……
        SUBS    R0，R0, #1
        BNE     LOOP        ; z=0转LOOP
```

2. BL——带链接的转移指令

指令格式：

BL{cond}　label

BL指令先将下一条指令的地址拷贝到LR 链接寄存器中，然后跳转到指定地址运行程序。

指令举例：

```
        BL    SUB1          ; LR←下条指令地址
                            ; 转至子程序SUB1处
        …
SUB1    …
        MOV   PC, LR        ; 子程序返回
```

注意：转移地址限制在当前指令的±32 MB的范围内。BL指令常用于子程序调用。

例：根据不同的条件，执行不同的子程序。

```
        CMP     R1, #5
        BLLT    ADD11           ；有符号数 <
        BLGE    SUB22           ；有符号数 ≧
          …
ADD11
          …
SUB22
          …
```

注：如果**R1<5**，只有**ADD11**不改变条件码，本例才能正常工作。

例：

```
        BL        SUB1
        ……
SUB1    STMFD     SP!, {R0-R3,R14}
        ……
        BL        SUB2
        ……
SUB2    ……
```

注意：在保存R14之前子程序不应再调用下一级的嵌套子程序。否则，新的返回地址将覆盖原来的返回地址，就无法返回到原来的调用位置。

3. BX——带状态切换的转移指令

指令格式：

BX{cond}　Rm

；PC=Rm&0xfffffffe, T=Rm[0]&1

BX指令跳转到Rm指定的地址执行程序。

若**Rm的位[0]为1**，则跳转时自动将**CPSR**中的标志**T置位**，即把目标地址的代码解释为**Thumb**代码；

若**Rm的位[0]为0**，则跳转时自动将**CPSR**中的标志**T复位**，即把目标地址的代码解释为**ARM**代码。

举例：

```
                ……
                ADR R0,THUMBCODE+1 ；将R0的bit[0]置1
                BX R0      ；   跳转,并根据R0的bit[0]实现状态切换
                CODE16                    ;16位Thumb代码
THUMBCODE  MOV R2,#2
                ……
                ADR R0,ARMCODE      ;加载ARMCODE地址到R0中
                BX R0
                CODE32                    ;32位ARM代码
ARMCODE     MOV R4,#4
                ……
```

◆ 访存指令(数据加载与存储器指令)：

1、单数据访存指令

2、多数据访存指令

3、数据交换/信号量操作指令

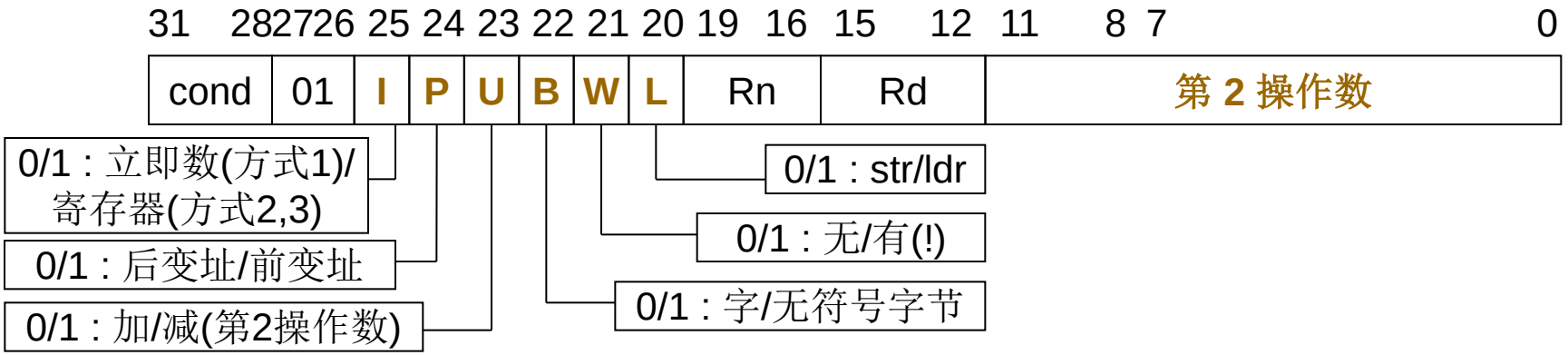
➢ 单数据访存指令

- 第一类：
  - 读写字: ldr / str
  - 读写无符号字节: ldrb / strb
- 第二类：
  - 读写无符号半字： ldrh / strh
  - 读有符号半字： ldrsh
  - 读有符号字节： ldrsb

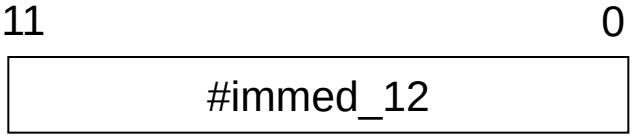
ldr: load register from memory

str: store register to memory

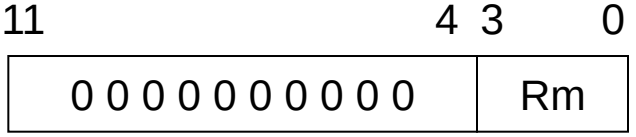
第一类指令的指令格式



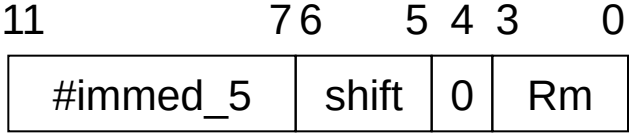
方式1：#immed_12（取值范围0 – 0xfff）
例子：ldr r0, [r1, #±0xfff]



方式2：Rm
例子：ldr r0, [r1, ±r2]



方式3：Rm LSL #immed_5（取值范围0-31）
例子：ldr r0, [r1, r2 LSL #31]



第一类指令的语法

- 语法
  - ldr|str{<cond>}{b} <Rd>, [<Rn>, #±<immed_12]{!}
  - ldr|str{<cond>}{b} <Rd>, [<Rn>, ±<Rm>]{!}
  - ldr|str{<cond>}{b} <Rd>, [<Rn>, ±<Rm>, <shift> #<immed_5>]{!}
  - ldr|str{<cond>}{b} <Rd>, [<Rn>], #±<immed_12
  - ldr|str{<cond>}{b} <Rd>, [<Rn>], ±<Rm>
  - ldr|str{<cond>}{b} <Rd>, [<Rn>], ±<Rm>, <shift> #<immed_5>
- 举例
  - ldrb r0, [r1, #±0xfff]　　；　把r1±0xfff地址的字节读入r0
  - ldr r0, [r1, ±r2]! ；把r1±r2地址的32比特数读入r0，然后r1= r1±r2
  - str r0, [r1, ±r2, LSL #31] ；把r0 (32bit)写到地址r1±(r2<<31)
  - ldr r0, [r1], #±0xfff　　；把r1地址的数读入r0，然后r1=r1±0xfff
  - ldr r0, [r1], ±r2　　；　把r1地址的数读入r0，然后r1=r1±r2
  - ldr r0, [r1], ±r2, LSL #31 ；　把r1地址的数读入r0，然后r1=r1±(r2<<31)

- 说明
  - ldr/str 读/写一个32bit字到/从一个32bit的寄存器，要求读/写地址字对齐。
  - ldrb：读一个8bit字节到一个32bit的寄存器，不要求地址对齐，寄存器的高24位清零。
  - strb：将寄存器的低8位，写入内存的某个地址。不要求地址对齐。

第二类指令

读写无符号半字：　ldrh / str
读有符号半字：　ldrsh
读有符号字节：　ldrsb

- 指令汇总：

LDR	字数据读取指令
LDRH	字节数据读取指令
LDRBT	用户模式的字节数据读取指令
LDRH	半字数据读取指令
LDRSB	有符号的字节数据读取指令
LDRSH	有符号的半字数据读取指令
LDRT	用户模式的字数据读取指令
STR	字数据写入指令
STRB	字节数据写入指令
STRBT	用户模式字节数据写入指令
STRH	半字数据写入指令
STRT	用户模式字数据写入指令

T后缀

- T为可选后缀，若指令有T，那么即使处理器是在特权模式下，存储系统也将访问看成是处理器在用户模式下。
- 用于存储器保护。
- 不能与前变址寻址、自动变址寻址一起使用（即不能改变基址寄存器值）。
- T在用户模式下无效。

操作数的寻址方式：

LDR/STR指令为变址寻址，由两部分组成：

——基地址部分：为一个基址寄存器，可以为任一个通用寄存器；

——偏移地址部分：这一部分非常灵活，实际就是第2个操作数，可以有以下3种形式：

立即数

寄存器

寄存器移位及移位常数

① 立即数

——12位立即数可以是一个无符号的数值。这个数据可以加到基址寄存器，也可以从基址寄存器中减去这个数值。

指令举例：

- LDR　R1，[R0，#0x12]

；将R0+0x12地址处的数据读出，保存到R1中(R0的值不变)

- LDR　R1，[R0，# -0x12]

；将R0-0x12地址处的数据读出，保存到R1中(R0的值不变)

② 寄存器

——寄存器中的数值可以加到基址寄存器，也可以从基址寄存器中减去这个数值。

指令举例如下：

- LDR　R1，[R0，R2]

  ；将R0+R2地址处的数据读出，保存到R1中

- LDR　R1，[R0，-R2]

  ；将R0-R2地址处的数据读出，保存到R1中

③ 寄存器移位及移位常数

——寄存器移位后的值可以加到基址寄存器，也可以从基址寄存器中减去这个数值。

指令举例：

- 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　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)

注意：移位位数最多是5比特位的立即数，不能使用寄存器指定移位位数。

④ PC（即R15）使用的几个问题

- 使用PC作为基址时，使用的数值是指令的地址加8个字节（取指与执行相差8个字节）。
- PC不能用做偏移寄存器，也不能用于任何变址寻址模式。
- 把一个字加载到PC，将使程序转移到所加载的地址，这是一个公认的实现跳转的方法。但是应当避免将一个字节加载到PC。
- 把PC存到存储器的操作在不同体系结构的处理器中产生不同的结果，应尽可能避免。

半字和有符号字节的加载/存储指令说明：

这类LDR/STR指令可实现半字（有符号和无符号）、有符号字节数据的传送。

(1) 特点:

偏移量格式、寻址方式与加载/存储字和无符号字节指令基本相同。

立即数偏移量限定在8位，寄存器偏移量不可经过移位得到。

（2）两点说明：

- **符 号 位**——有符号字节或有符号半字的加载，用“符号位”扩展到32位；无符号半字传送是用0扩展到32位。

- **地址对齐**——**对半字传送的地址必须为偶数**。非半字对齐的半字加载将使Rd内容不可靠；非半字对齐的半字存储将使指定地址的2字节存储内容不可靠。

（3）指令举例

LDRSB R1，[R0，R3]

；将R0+R3地址上的字节数据读到R1,高24位用符号位扩展。

LDRSH R1，[R9]

；将R9地址上的半字数据读出到R1，高16位用符号位扩展。

LDRH R6，[R2]，#2

；将R2地址上的半字数据读出到R6，高16位用零扩展，然后修改R2=R2+2。

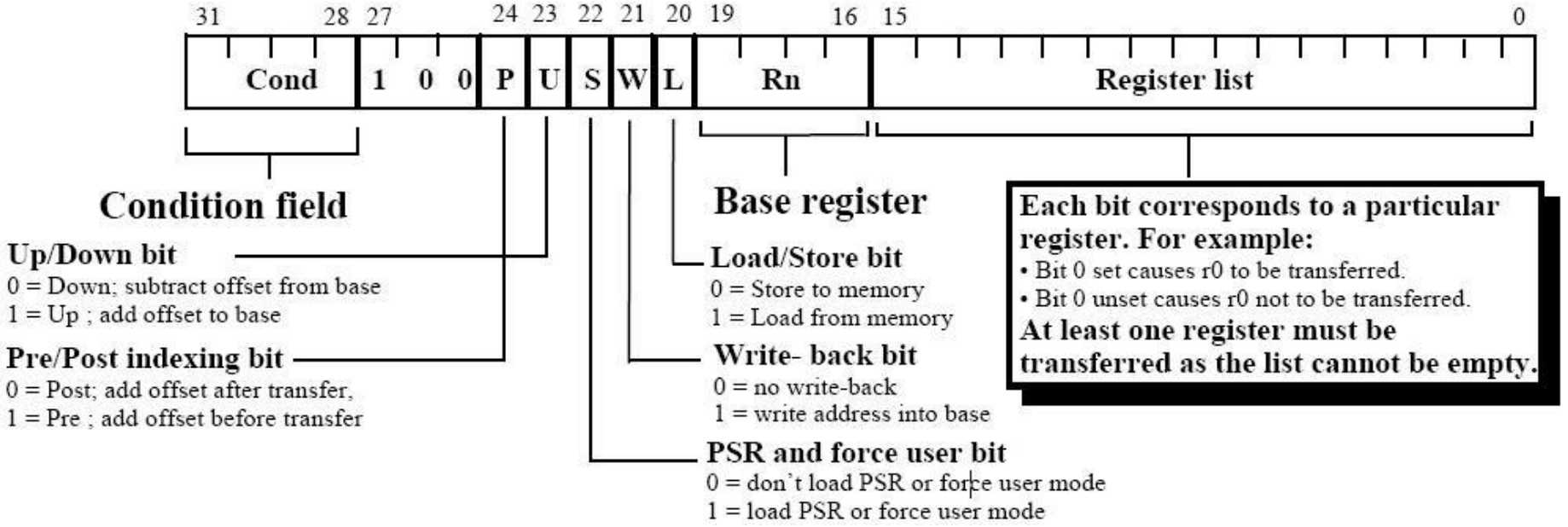
STRH R1，[R0，#2]!

；将R1的数据保存到R0+2地址中，只存储低2字节数据，并且修改R0=R0+2。

➢多数据访存指令

- 批量访存指令可以实现一组（1-16）寄存器和一块（4-64字节）连续内存单元之间的数据传输。

指令格式


31
28
27
24
23
22
21
20
19
16
15
0
Cond
1 0 0
P
U
S
W
L
Rn
Register list
Condition field
Base register
Each bit corresponds to a particular register. For example:
• Bit 0 set causes r0 to be transferred.
• Bit 0 unset causes r0 not to be transferred.
At least one register must be transferred as the list cannot be empty.
Up/Down bit
0 = Down; subtract offset from base
1 = Up ; add offset to base
Pre/Post indexing bit
0 = Post; add offset after transfer,
1 = Pre ; add offset before transfer
Load/Store bit
0 = Store to memory
1 = Load from memory
Write- back bit
0 = no write-back
1 = write address into base
PSR and force user bit
0 = don't load PSR or force user mode
1 = load PSR or force user mode

LDM/STM指令的语法

- 语法
  - ldm|stm{<cond>}<addressing_mode> <Rn>{!}, <registers>{^}
  - <addressing_mode>——地址变化方式有4种：
    - IA (Increment After)　　事后递增
    - IB (Increment Before)　　事先递增
    - DA (Decrement After)　　事后递减
    - DB (Decrement Before)　　事先递减
- 例子
  - ldmia r0, {r5-r8}

  /* 将内存中(r0)到(r0+12)4个字读取到r5~r8的4个寄存器中 */
  - ldmib r0, {r5-r8}

  /* 将内存中(r0+4)到(r0+16)4个字读取到r5~r8的4个寄存器中 */
  - ldmda r0, {r5-r8}

  /* 将内存中(r0-12)到(r0)4个字读取到r5~r8的4个寄存器中 */
  - ldmdb r0, {r5-r8}

  /* 将内存中(r0-16)到(r0-4)4个字读取到r5~r8的4个寄存器中 */

--指令格式说明

(1)Rn：表示基址寄存器，装有传送数据的初始地址，Rn不允许为R15（即PC）。

(2)Rn后缀“!”：表示最后的地址写回到Rn中。

(3)Registers：表示寄存器列表，可包含多个序号连续的或者分离的寄存器，用“，”分开。

- 格式例子：{R1, R2, R6-R9}
- 列表寄存器和存储器地址的关系规则：

  编号低的寄存器对应于存储器中低地址单元，编号高的寄存器对应于存储器中高地址单元。

（4）后缀“^”说明

- 寄存器列表不包含PC：使用后缀“^”进行数据传送时，加载/存储的是用户模式的寄存器，而不是当前模式的寄存器。
- 寄存器列表包含有PC：除了正常的多寄存器传送外，还要将SPSR拷贝到CPSR中。

  该用法可用于异常处理返回。
- 禁用情况：后缀“^”不允许在用户模式或系统模式下使用，因为它们没有SPSR。

（5）当Rn在寄存器列表中且使用后缀“!”

- 对于STM指令，若Rn为寄存器列表中的最低数字的寄存器，则会将Rn的初值保存；
- 其它情况下Rn的加载值和存储值不可预知。

（6）地址字对齐

——这些指令寻址是字对齐的，即忽略地址位[1:0]。

(7) 关于模式项

LDM/STM的主要用途是现场保护、数据复制和参数传送等。其模式有如下8种{前面4种用于数据块的传送(为访存操作), 后面4种是堆栈操作}:

- IA： 先传、后地址加4；
- IB： 先地址加4 、后传；
- DA：先传、后地址减4；
- DB：先地址减4 、后传；
- FD： 满递减堆栈，先地址减4、后传，与DB对应；
- ED： 空递减堆栈，先传、后地址减4，与DA对应；
- FA： 满递增堆栈，先地址加4、后传，与IB对应；
- EA： 空递增堆栈，先传、后地址加4，与IA对应。

LDM指令的三种用法

编号	!	^	pc	ldm指令不同的功能	用法
1	是	是	是	同(5)，但是更新<Rn>	3
2	是	是	否	编译警告(ADS1.2)：Warning : A1329W: Unsafe instruction (forced user mode xfer with write-back to base)	2
3	是	否	是	同(7)，但是更新<Rn>	1
4	是	否	否	同(8)，但是更新<Rn>，且<Rn>不能出现在<registers>中	1
5	否	是	是	数据读取的同时，拷贝spsr到cpsr	3
6	否	是	否	<registers>采用user模式下的寄存器	2
7	否	否	是	同(8)，可用于程序跳转。	1
8	否	否	否	正常读取	1

STM指令的两种用法

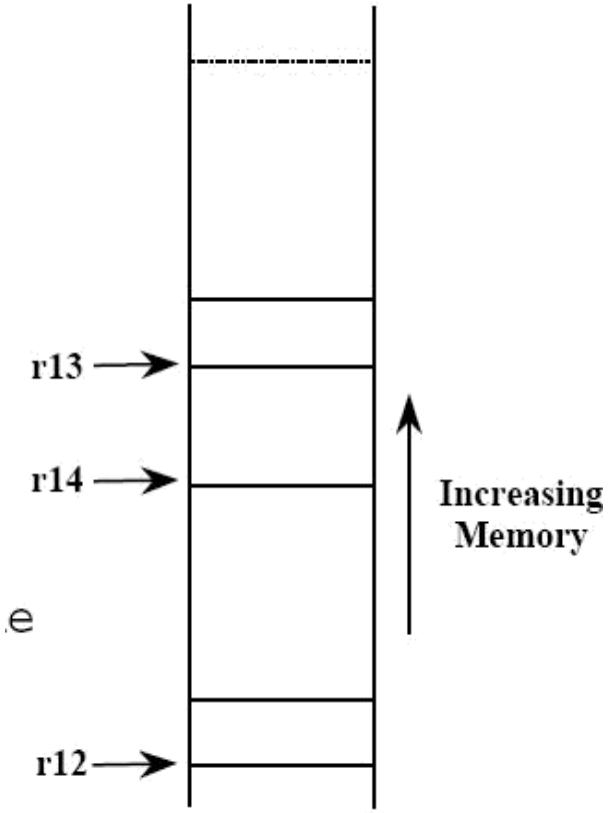
编号	!	^	ldm指令不同的功能	用法
1	是	是	同(3)，但是更新<Rn>	2
2	是	否	同(4)，但是更新<Rn>。当<Rn>是<registers>中编号最小的寄存器时，指令将<Rn>的初值保存；否则结果不可预测。	1
3	否	是	<registers>采用user模式下的寄存器	2
4	否	否	正常写入	1

多数据访存程序举例：

例：

```
/* r12 指向源数据起点 */
/* r14 指向源数据终点 */
/* r13 指向目标地址 */

loop:
    ldmia r12!, {r0-r11} /* 读48字节 */
    stmia r13!, {r0-r11} /* 写48字节 */
    cmp r12, r14         /* 是否到末尾 */
    bne loop             /* 重复循环 */
```



- 模拟栈操作：
- ARM指令集中没有用于栈的操作指令，但是可以用多数据访存指令来模拟。
- 栈的分类（按指针）：
  - 栈指针指向最后一个被占用的地址 (Full Stack)
    - 在push前需要先将栈指针减小
  - 栈指针指向下一个被占用的地址 (Empty Stack)
    - 在push之后需要将栈指针减小
- 栈的分类（按方向）：
  - 上升的栈（Ascending Stack）
    - 向高地址扩展
  - 下降的栈（Descending Stack）
    - 向低地址扩展

FD　满递减堆栈
ED　空递减堆栈
FA　满递增堆栈
EA　空递增堆栈

模拟压栈操作

块数据访问指令	栈访问指令	说明
stmda	stm**ed**	下降型空栈
stmia	stm**ea**	上升型空栈
stmdb	stm**fd**	下降型满栈
stmib	stm**fa**	上升型满栈

模拟退栈操作

块数据访问指令	栈访问指令	说明
ldmda	ldm**fa**	上升型满栈
ldmia	ldm**fd**	下降型满栈
ldmdb	ldm**ea**	上升型空栈
ldmib	ldm**ed**	下降型空栈

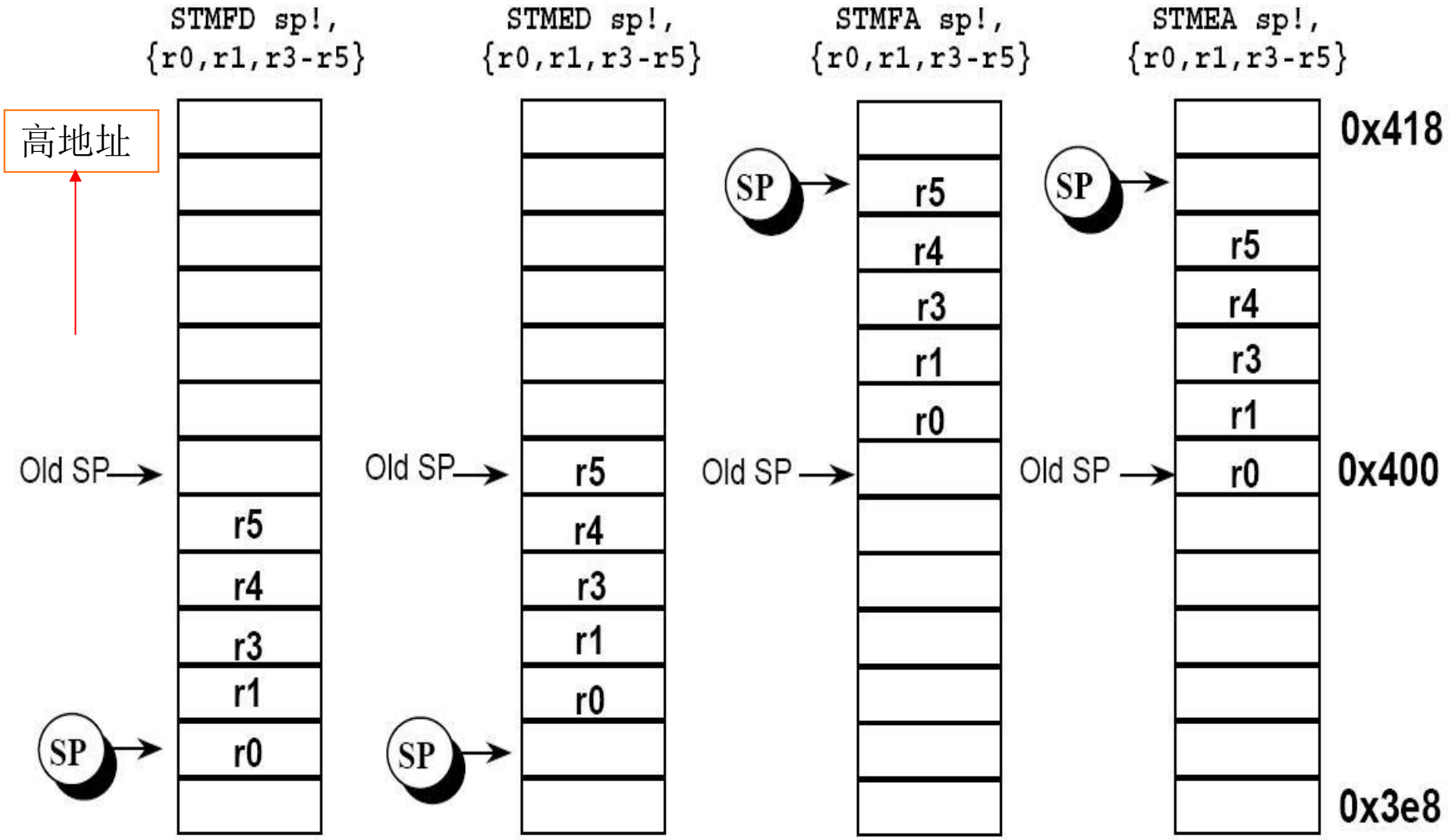
- 常见的情况
  - stmfd sp!, {r0-r12, lr}

// 保存所有寄存器（包括返回地址）

.....

  - ldmfd sp!, {r0-r12, pc}

// 恢复所有寄存器（包括pc）


STMFD sp!, {r0,r1,r3-r5}
STMED sp!, {r0,r1,r3-r5}
STMFA sp!, {r0,r1,r3-r5}
STMEA sp!, {r0,r1,r3-r5}
高地址
Old SP
SP
r5
r4
r3
r1
r0
0x418
0x400
0x3e8

举例：多数据访问

- **LDMIA　R0!, {R3-R9}**

  ；加载R0指向地址上的多字数据，保存到R3~R9中，R0值更新。
- **STMIA　R1!, {R3-R9}**

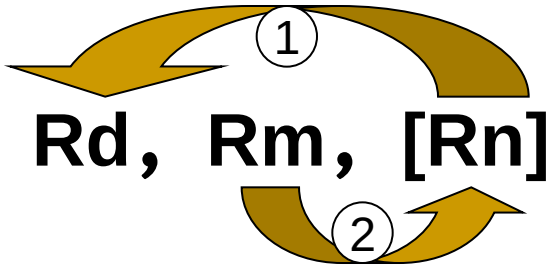
  ；将R3～R9的数据存储到R1指向的地址上，R1值更新 。
- **STMFD　SP!, {R0-R7, LR}**

  ；保护现场，将R0～R7、LR入栈，SP值更新 。
- **LDMFD　SP!, {R0-R7, PC}**

  ；恢复现场，包括R0～R7 和PC(异常处理返回)，SP值更新。

数据交换/信号量操作指令swp/swpb

- 用于进程间的同步互斥，提供对信号量的原子操作（在一个指令周期中完成信号量的读取和修改操作）。
- **指令格式**



SWP{cond}{B}　Rd，Rm，[Rn]

SWP指令用于将一个存储单元(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的内容写入到该存储单元中。

B为可选后缀，若有B，则交换字节，否则交换32位字

- **Rd为被加载的寄存器**
- **Rm的数据用于存储到Rn所指的地址中**

  若Rm与Rd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。**

- 举例
  - swp r1, r2, [r3]

    ; 将内存单元(r3)中的字读取到r1，同时将r2中的数据写入内存单元(r3)中
  - swp r1, r1, [r2]

    ; 将r1寄存器内容和内存单元(r2)的内容互换
  - swpb r1, r2, [r0]

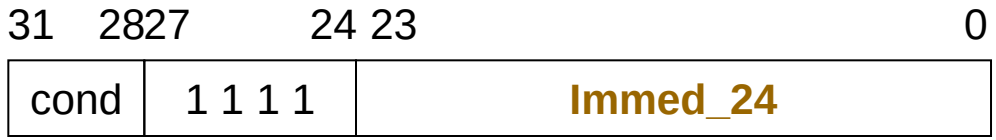
    ; 将R0指向的存储单元的内容读取1字节数据到R1中(高24位清零)，并将R2的内容写入到该内存单元中(最低字节有效)
- 说明
  - **swpb读取8bit数据到寄存器后会对高24位清零，写到内存的8bit数来自寄存器低8位**

◆异常中断产生指令：SWI、BKPT、CLZ

异常中断指令可以分为以下几种：

- **SWI**：　软件中断指令
- **BKPT(*)**：断点指令(v5T及以上体系)
- **CLZ(*)**：　前导0计数(v5T及以上体系)

1.SWI——软件中断指令



指令格式:

语法: SWI {<cond>} <24位立即数>

软件中断指令SWI产生软件异常中断，用来实现用户模式到特权模式的切换。用于用户模式下对操作系统中特权模式的程序的调用；它将处理器置于管理（svc）模式，中断向量地址为0x08。

说明:

- 主要用于用户程序调用操作系统的API。
- 参数传递通常有两种方法:
  - 指令中的24bit立即数指定API号，其它参数通过寄存器传递。
  - 忽略指令中的24bit立即数，r0指定API号，其它参数通过其它寄存器传递。

SWI指令举例：

·软中断号在指令中，不传递其它参数

```
SWI     10          ；中断号为10
SWI     0x123456    ；中断号为0x123456
```

.软中断号在指令中，其它参数在寄存器中传递

```
MOV     R0，#34     ；准备参数
SWI     12          ；调用12号软中断
```

.不用指令中的立即数，软中断号和其它参数都在寄存器中传递

```
MOV     R0，#12     ；准备中断号
MOV     R1，#34     ；准备参数
SWI     0           ；进入软中断
```

SWI程序举例：

```
T_bit    EQU    0x20              ；用于测试Thumb标志位(第5位)
SWI_handler
        STMFD    SP!, {R0-R3, R12, LR}
        MRS      R0, SPSR          ；保存中断前的CPSR值
        STMFD    SP!, {R0}         ；到堆栈中
        TST      R0, #T_bit        ；测试T位标志
        LDRNEH   R0, [LR，# -2]    ；z=0(即T=1)则读取16位的SWI指令码
        BICNE    R0, R0，#0xFF00   ；获取SWI中的中断号
        LDREQ    R0, [LR，# -4]    ；读取32的SWI指令码
        BICEQ    R0, R0，#0xFF000000；获取SWI中的中断号
                 ……                ；转去处理相应的软中断
        LDMFD    SP!, {R0-R3, R12, PC}^
                                   ；中断返回，包括恢复原CPSR值
```

2. 断点指令BKPT(*)

- 语法
  - BKPT <immed_16>
- 说明

  immed_16：16位立即数。该立即数被调试软件用来保存额外的断点信息。

  断点指令用于软件调试；它使处理器停止执行正常指令而进入相应的调试程序。V5T及以上体系使用。

- 例：
  - BKPT　　0xF02C

3.CLZ——前导0计数指令(*)

指令格式:

CLZ{<cond>} Rd, Rm

前导0计数指令CLZ 对Rm中的前导0的个数进行计数，结果放到Rd中。 V5T及以上体系使用。

举例:

```
MOV   R2, #0x17C00
; R2=0b0000 0000 0000 0001 0111 1100 0000 0000
CLZ    R3, R2                    ; R3=15
```

◆协处理器指令(*:了解)

ARM协处理器： ARM支持16个协处理器，用于各种协处理器操作，最常使用的协处理器是用于控制片上功能的系统控制协处理器CP15，例如控制高速缓存(Cache)和存储器管理单元(MMU)，浮点ARM协处理器等，还可以开发专用的协处理器。

ARM协处理器指令根据其用途主要分为以下三类：

- 协处理器数据处理指令。
- ARM寄存器与协处理器寄存器的数据传送指令。
- 协处理器寄存器和内存单元之间数据存/取指令。

ARM协处理器指令包括以下5条：

— CDP 协处理器数据处理指令
— LDC 协处理器数据加载指令
— STC 协处理器数据存储指令
— MCR ARM处理器寄存器到协处理器寄存器的数据传送指令
— MRC 协处理器寄存器到ARM处理器寄存器的数据传送指令

3.2 ARM汇编伪指令和伪操作

主要内容

概念

ARM汇编伪指令

ARM汇编伪操作

一.符号定义伪操作

二. 数据定义伪操作

三.汇编控制伪操作

四.其他伪操作

ARM汇编宏指令(略)

伪指令、伪操作和宏指令概念

. 伪指令——是汇编语言程序里的特殊指令助记符，在汇编时被合适的机器指令替代。

. 伪操作——为汇编程序所用，在源程序进行汇编时由汇编程序处理，只在汇编过程起作用，不参与程序运行。

. 宏指令——通过伪操作定义的一段独立的代码。在调用它时将宏体插入到源程序中。也就是常说的宏。

说明：所有的伪指令、伪操作和宏指令，均与具体的开发工具中的编译器有关，当前主要采用ARM公司的“ADS IDE”开发工具，因此后面的讨论中，均是基于ARM公司的开发工具。

ARM汇编伪指令

ARM伪指令不属于ARM指令集中的指令，是为了编程方便而定义的。伪指令可以像其它ARM指令一样使用，但在编译时这些指令将被等效的ARM指令代替。ARM伪指令有四条，分别是：

- ADR：小范围的地址读取伪指令。
- ADRL：中等范围的地址读取伪指令。
- LDR：大范围的地址读取伪指令。
- NOP：空操作伪指令。

一. ADR——小范围的地址读取

ADR伪指令功能：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读取到寄存器中。

ADR伪指令功能的实现方法：在汇编编译器编译源程序时，ADR伪指令被编译器替换成一条合适的指令。通常，编译器用一条ADD指令或SUB指令来实现此ADR伪指令的功能，若不能用一条指令实现，则产生错误，编译失败。

语法格式：

ADR{cond}　register，expr

其中：

register：加载的目标寄存器。

expr：地址表达式。当地址值是非字对齐时，取值范围在-255～255字节之间；当地址值是字对齐时，取值范围在-1020～1020字节之间。

例：

```
        ……
        ADR  R1,Delay
        ……
Delay
        MOV  R0,R14
        ……
```

使用ADR将程序标号Delay所表示的地址存入R1。

例：查表

```
        ADR    R0,D_TAB     ;加载转换表地址
        LDRB   R1,[R0,R2]   ;使用R2作为参数，进行查表
          ……
D_TAB
        DCB   0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92
```

二. ADRL——中等范围的地址读取

ADRL伪指令功能：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读取到寄存器中，比ADR伪指令可以读取更大范围的地址。

ADRL伪指令功能实现方法：在汇编编译器编译源程序时，ADRL被编译器替换成两条合适的指令。若不能用两条指令实现，则产生错误，编译失败。

语法格式：

ADRL{cond}　　register,expr

其中：

- register：加载的目标寄存器。
- expr：地址表达式。当地址是非字对准时，取值范围在-64~64KB之间；当地址是字对准时，取值范围在-256~256KB之间。

例3：

```
        ……
        ADRL  R1,Delay
        ……
Delay
        MOV  R0,R14
        ……
```

使用ADRL将程序标号Delay所表示的地址存入R1。

三. LDR ——大范围的地址读取

LDR伪指令功能：用于加载32位立即数或一个地址值到指定的寄存器。

LDR伪指令功能实现方法：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。

- 若加载的常数未超过MOV或MVN的范围，则使用MOV或MVN指令代替该LDR伪指令；
- 否则汇编器将常量放入文字池(literal pool)/数据缓冲池，并使用一条程序相对偏移的LDR指令从文字池读出常量。

语法格式：

LDR{cond}　register,=expr

其中：

- Register：加载的目标寄存器。
- expr：32位常量或地址表达式。

例4：

```
        ……
        LDR  R1,=Delay
        ……
Delay
        MOV  R0,R14
        ……
```

使用**LDR**将程序标号**Delay**所表示的地址存入**R1**。

注意：

- **从指令位置到文字池的偏移量必须小于4KB。**
- **与ARM指令的LDR的区别：伪指令LDR的参数有“=”号。**

四.NOP——空操作伪指令

NOP伪指令功能实现方法：在汇编时将被替代成ARM中的空操作，比如可能是“MOV R0,R0”指令等。

用途：NOP可用于延时操作。

语法格式： NOP

例：延时子程序

```
;R1-入口参数
Delay
    NOP      ;空操作
    NOP
    NOP
    SUBS  R1,R1,#1   ;循环次数减1
    BNE    Delay
    MOV    PC,LR
```

ARM汇编伪操作

ADS编译环境下的伪操作可分为以下几类:

- 符号定义（Symbol Definition）伪操作
- 数据定义（Data Definition）伪操作
- 汇编控制（Assembly Control）伪操作
- 其他（Miscellaneous）伪操作

一.符号定义伪操作

- **GBLA，GBLL，GBLS：声明全局变量。**
- **LCLA，LCLL，LCLS：声明局部变量。**
- **SETA，SETL，SETS：给变量赋值。**
- **RLIST：为通用寄存器列表定义名称。**

```
GBLA  A1                ;定义全局数值变量A1
A1    SETA 0xaa         ；将该变量赋值为0xaa
GBLL  A2                ;定义全局逻辑变量A2
A2    SETL {TRUE}       ；将该变量赋值为真
GBLS  A3                ;定义全局字符串变量A3
A3    SETS “Tesiting”   ；将该变量赋值为“Testing”
```

二. 数据定义伪操作

.LTORG：声明一个文字池的开始

.SPACE：分配一块字节内存单元，并用0初始化

.DCB：分配一段字节内存单元，并初始化

.DCD、DCDU：分配一段字内存单元，并初始化

.MAP：定义一个结构化的内存表的首地址

.FIELD/#：定义结构化内存表中的一个数据域

```
MAP 0x100, R0  ; 定义结构化内存表首地址的值为0x100＋R0
A FIELD 16     ; 定义A的长度为16字节，位置为0x110+R0
LTORG          ;定义数据缓冲池
Data  SPACE  4200    ;从当前位置开始分配4200字节的内存单元，
                     ;并初始化为0。
```

1. LTORG

用于声明一个数据缓冲池（文字池）的开始。

语法格式：LTORG

```
例：start   BL   func          ;
        ……
func   LDR   R1,=0x8000   ;子程序
        ……
       MOV  PC,LR          ;子程序返回
       LTORG               ;定义数据缓冲池
Data   SPACE    4200       ;从当前位置开始分配4200字节的内
                            存单元，并初始化为0。
       END
```

默认数据缓冲池为空

说明：

- ARM汇编编译器一般把文字池放在代码段的最后，即下一个代码段开始之前，或END伪操作之前。
- LTORG伪操作通常放在无条件分支指令之后，或者子程序返回指令之后，这样处理器就不会错误地将文字池中的数据当作指令来执行。

3.DCB——也可以用符号”=”表示

用于定义并且初始化一个或者多个字节的内存区域。

语法格式：

{label}　DCB　expr{,expr}……

或　{label}　=　expr{,expr}

其中expr表示：

- 0到255之间的一个数值常量或者表达式。
- 一个字符串。

注意：当DCB后面紧跟一个指令时，可能需要使用ALIGN确保指令是字对准的。

例：

```
short   DCB  1                    ;为short分配了一个
                                   ;字节,并初始化为1。
string  DCB  “string”,0            ;构造一个以0
                                   ;结尾的字符串
```

4. DCD、DCDU分配一段字内存单元

（1）DCD ——分配一段字对齐的内存单元

用于分配一段字对齐的内存单元，并初始化。 DCD也可以用符号”&”表示

语法格式:

{label} DCD expr{,expr}……

或 {label} & expr{,expr}…… expression

其中:

expr ： 数字表达式或程序中的标号。

注意: DCD伪操作可能在分配的第一个内存单元前插入填补字节以保证分配的内存是字对齐的。

例:

```
data1   DCD   2,4,6
              ;为data1分配三个字,内容初始化为2,4,6
data2   DCD   label+4
                ;初始化data2为label+4对应的地址
```

（2）DCDU ——分配一段字非严格对齐的内存单元

DCDU与DCD的不同之处在于DCDU分配的内存单元并不严格字对齐。

汇编控制伪操作

- IF,ELSE及ENDIF：有条件选择汇编
- WHILE及WEND：有条件循环(重复)汇编
- MACRO,MEND及MEXIT：宏定义汇编

其他伪操作

1. AREA：定义一个代码段或数据段
2. CODE16、CODE32：告诉编译器后面的指令序列的指令代码位数
3. ENTRY：指定程序的入口点
4. ALIGN：将当前的位置以某种形式对齐(ALIGN或ALIGN n：以字或n字节对齐)
5. END：源程序结尾
6. EQU/*：为数字常量、基于寄存器的值和程序中的标号定义一个字符名称。
7. EXPORT、GLOBAL：声明源文件中的符号可以被其他源文件引用
8. IMPORT、EXTERN：声明某符号是在其他源文件中定义的
9. GET、INCLUDE：将一个源文件包含到当前源文件中，并将被包含的文件在其当前位置进行汇编处理。
10. INCBIN：将一个文件包含到当前源文件中，而被包含的文件不进行汇编处理

1. AREA

用于定义一个代码段或是数据段。

语法格式：

AREA sectionname{,attr} {,attr}…attribute

其中：

- sectionname：为所定义的段的名称。
- attr：该段的属性。具有的属性为：
  - CODE：定义代码段。
  - DATA：定义数据段。
  - READONLY：指定本段为只读，代码段的默认属性。
  - READWRITE：指定本段为可读可写，数据段的默认属性。

- **ALIGN**：指定段的对齐方式为$2^{expression}$。expression的取值为0~31。
- **COMMON**：指定一个通用段。该段不包含任何用户代码和数据。
- **NOINIT**：指定此数据段仅仅保留了内存单元，而没有将各初始值写入内存单元，或者将各个内存单元值初始化为0。

注意：一个大的程序可包含多个代码段和数据段。一个汇编程序至少包含一个代码段。

2. CODE16和CODE32

CODE16告诉汇编器后面的指令序列为16位的Thumb指令。

CODE32告诉汇编器后面的指令序列为32位的ARM指令。

语法格式：

CODE16

CODE32

注意：CODE16和CODE32只是告诉编译器后面指令的类型，该伪操作本身不进行程序状态的切换。

例：

```
AREA    ChangeState, CODE, READONLY
        ENTRY
        CODE32                  ;下面为32位ARM指令
        LDR   R0,=start+1
        BX     R0
          ……
        CODE16                  ;下面为16位Thumb指令
start  MOV   R1,#10
          …….
        END
```

3. ENTRY

指定程序的入口点。

语法格式：

ENTRY

注意：

一个程序（可包含多个源文件）中至少要有一个ENTRY（可以有多个ENTRY），但一个源文件中最多只能有一个ENTRY（可以没有ENTRY）

4. ALIGN

ALIGN伪操作通过填充0将当前的位置以某种形式对齐。

语法格式：

ALIGN {expr{,offset}}

其中：

- expr：一个数字，表示对齐的单位。这个数字是2的整数次幂，范围在2^0~2^{31}之间。

  如果没有指定expr，则当前位置对齐到下一个字边界处。
- Offset：偏移量，可以为常数或数值表达式。不指定offset表示将当前位置对齐到以expr为单位的起始位置。

例1：

```
chara DCB   1       ;本操作使字对齐被破坏
ALIGN               ;重新使其为字对齐
MOV   R0,1
```

例2：

```
ALIGN    8          ;当前位置以2个字的方式对齐
```

5. END

END伪操作告诉编译器已经到了源程序结尾。

语法格式:

END

注意:

每一个汇编源程序都必须包含END伪操作，以表明本源程序的结束。

6. EQU ——也可以用符号“*”表示

EQU伪操作为数字常量、基于寄存器的值和程序中的标号定义一个字符名称。

语法格式：

name EQU expr{，type}

其中：

- name：为expr定义的字符名称。
- expr：基于寄存器的地址值、程序中的标号、32位的地址常量或者32位的常量。表达式，为常量。
- type：当expr为32位常量时，可以使用type指示expr的数据的类型。取值为：
  - CODE32
  - CODE16
  - DATA

例：

```
abcd    EQU   2          ;定义abcd符号的值为2
abcd    EQU   label+16
                         ;定义abcd符号的值为(label+16)
abcd    EQU   0x1c,CODE32
                         ;定义abcd符号的值为绝对地址
                         ;值0x1c，而且此处为ARM指令
```

7. EXPORT及GLOBAL

声明一个源文件中的符号，使此符号可以被其他源文件引用。

语法格式：

EXPORT/GLOBAL symbol {[,weak]}

其中：

- symbol：声明的符号的名称。（区分大小写）
- [weak]：声明其他同名符号优先于本符号被引用。

例：
```
AREA   example，CODE，READONLY
EXPORT   DoAdd
DoAdd    ADD   R0，R0，R1
```

8. IMPORT及EXTERN

声明一个符号是在其他源文件中定义的。

语法格式：

IMPORT symbol{[,weak]}

EXTERN symbol{[,weak]}

其中：

- symbol：声明的符号的名称。
- [weak]：
  - 当没有指定此项时，如果symbol在所有的源文件中都没有被定义，则连接器会报告错误。
  - 当指定此项时，如果symbol在所有的源文件中都没有被定义，则连接器不会报告错误，而是进行下面的操作。
    - 如果该符号被B或者BL指令引用，则该符号被设置成下一条指令的地址，该B或BL指令相当于一条NOP指令。
    - 其他情况下此符号被设置成0。

例：

```
AREA Init, CODE, READONLY
IMPORT Main   ;通知编译器当前文件要引用符号
              ;Main，但Main在其他文件中定义
 ……
END
```

9. GET及INCLUDE

将一个源文件包含到当前源文件中，并将被包含的文件在其当前位置进行汇编处理。

指令格式:

GET　　filename

INCLUDE　filename

其中:

- filename:包含的源文件名，可以使用路径信息(可包含空格)。

例：GET　d:\arm\file.s

10. INCBIN

将一个文件包含到当前源文件中，而被包含的文件不进行汇编处理。

指令格式：

INCBIN　filename

其中：

- filename：被包含的文件名称，可使用路径信息(不能有空格)。

适用情况：通常使用此伪操作将一个可执行文件或者任意数据包含到当前文件中。

例：INCBIN　d:\arm\file.txt

ARM汇编中的文件格式

源程序文件	文件名	说　明
汇编程序文件	*.s	用ARM汇编语言编写的ARM程序或Thumb程序。
C程序文件	*.c	用C语言编写的程序代码。
头文件	*.h	为了简化源程序，把程序中常用到的常量命名、宏定义、数据结构定义等等单独放在一个文件中，一般称为头文件。

基于ARM的编程

- 在只关心系统所具有功能的设计中，采用高级编程语言编写程序更合适，由于其隐藏了CPU执行指令的许多细节。但是，CPU执行指令的细节差异会反应在系统的非功能特性上，例如系统程序的规模和运行速度。因此，掌握汇编语言程序设计对于嵌入式系统的设计者来说是非常必要的。

- ARM汇编程序中每一行的通用格式为：
  {标号}{指令|指示符|伪指令}{；注解}。
- 在ARM汇编语言源程序中，除了标号和注释外，指令、伪指令和指示符都必须有前导空格，而不能顶格书写。
- 如果每一行的代码太长，可以使用字符“\”将其分行书写，并允许有空行。
- 指令助记符、指示符和寄存器名既可以用大写字母，也可以用小写字母，但不能混用。
- 注释从“；”开始，到该行结束为止。

预定义变量(ATPCS规则)

- ARM汇编器对ARM的寄存器进行了预定义，所有的寄存器和协处理器名都是大小写敏感的。预定义的寄存器如下：
- R0~R15和r0~r15；
- a1~a4（参数、结果或临时寄存器，与r0~r3同义）；
- v1~v8（变量寄存器，与r4~r11同义）；
- sb和SB（静态基址寄存器，与r9同义）；
- sl和SL（堆栈限制寄存器，与r10同义）；
- fp和FP（帧指针，与r11同义）；

(ATPCS: ARM-THUMB Procedure Call Standard)

- ip和IP（过程调用中间临时寄存器，与r12同义）；
- sp和SP（堆栈指针，与r13同义）；
- lr和LR（链接寄存器，与r14同义）；
- pc和PC（程序计数器，与r15同义）；
- cpsr和CPSR（程序状态寄存器）；
- spsr和SPSR（程序状态寄存器）；
- f0~f7和F0~F7（FPA寄存器）；
- p0~p15（协处理器0~15）；
- c0~c15（协处理器寄存器0~15）。

（FPA: Floating Point Accelerator）

子程序调用规则

- ARM9处理器的子程序调用指令有别于Intel X86的子程序调用指令CALL
- 程序设计时，通常会把完成某个特定功能的一段程序代码编写成子程序，在需要的地方进行调用。ARM9汇编程序中，使用下面语句调用子程序。
- BL　next
- 其中，next为子程序中的第一条指令代码的标号。

- 任何一个子程序进入前，处理器需要保存主程序中的现场，即需要保存当前工作寄存器（注意：当采用了子程序嵌套调用时，应该保存LR寄存器）。
- 汇编指令BL的功能是将BL指令的下一条指令地址放到LR寄存器中，作为返回地址。并将子程序的第一条指令地址赋予PC寄存器，实现程序转移，即进入子程序执行。子程序执行完后，通过把LR寄存器值赋予PC寄存器，实现返回。

C与汇编混合编程

- ARM汇编工具支持在C、C++语言程序中嵌入汇编编写的程序段，其语法格式如下：

  _asm（“指令[；指令]”） 或 asm（“指令[；指令]”）

- 内嵌汇编的语法
  - 操作数：作为操作数的寄存器和常量可以是char\short\int(8位\16位\32位)型的C表达式。
  - 物理寄存器：不要使用复杂的C表达式；一般不要使用R0~R3,R12(IP),R14(LR)
  - 不要使用寄存器代替变量

- 交互规则(ATPCS规则)：
  - 寄存器规则：v1-v8（R4-R11）用来保存局部变量
  - 堆栈规则：FD类型（满递减堆栈)
  - 参数传递规则：如果参数个数小于等于4，用R0-R3保存参数；参数个数多于4的情况下，剩余的参数传入堆栈
  - 子程序结果返回规则：
    - 结果为一个32位整数，通过R0返回；
    - 结果为一个64位整数，通过R0、R1返回；
    - 对于位数更多的结果，通过内存传递。

C调用汇编

C代码

```
//Filename: test.c
…
extern void scopy(char *d, char *s);
…
scopy(dststr, srcstr);
```

汇编代码

```
; Filename: strcpy.s
AREA strcpy,  CODE, READONLY
EXPORT scopy
scopy
    LDRB   R2, [R1], #1
    STRB   R2, [R0], #1
    CMP    R2, #0
    BNE    scopy
    MOV    PC, LR    ;R0=Result
END
```

//R0存放第一个参数d，R1存放第二个参数s

汇编调用C

汇编代码: example.s

```
AREA  example, CODE, READONLY
IMPORT sum5 ;使用IMPORT伪操作声明C函数
callsum5
   STMFD  SP!, {LR}          ;LR入栈
   ADD R1, R0, R0            ;R0=a,R1=b=2*a
   ADD R2, R1, R0            ;R2=c=3*a
   ADD R3, R1, R2            ;R3=e=5*a
   STR  R3, [SP,#-4]!        ;e in stack
   ADD R3, R1, R1            ;R3=d=4*a
   BL  sum5                  ;call sum5,
                             ;R0=result
   ADD SP,SP,#4
   LDMFD  SP!, {PC}
   END
```

C代码:

```
#include<stdio.h>
int sum5(int a, int b ,int c, int d, int e)
{
  return (a+b+c+d+e);
}
```

本例的汇编程序是调用 sum5(a,2*a,3*a, 4*a,5*a);

例子

```
#include   <stdio.h>
void  test_example1(char * s1, const char *s2);
int  main(void)
{
        const  char  *string1 = “test example”；
        char   s[20]；
        _asm
        {
        MOV  R0, string1
        MOV  R1, s
        BL   test_example1, {R0, R1}
        }
return  0；
}
```

```
void   test_example1(char * s1, const char *s2) {
        int  a1；
        _asm
        {
loop：
#ifndef    _thumb
        LDRB  a1, [s1], #1
        STRB  a1, [s2], #1
#else
        LDRB  a1, [s1]
        ADD   s1, #1
        STRB  a1, [s2]
        ADD   s2, #1
#endif
        CMP   a1, #0
        BNE   loop
        }
}
```

- 上面一段用C语言编写的程序中内嵌了汇编程序。例子中，test_example1是一个子程序（函数），主函数是main()。main函数和test_example1函数内部各嵌入了一段汇编语言编写的程序，该程序完成的是一个字符串的拷贝。
- 仍然这个例子，采用子程序形式而不是内部嵌入。

```
#include   <stdio.h>
extern void  test_example2(char * s1，const char *s2);
int  main(void)
{
        const  char  *string1 = “test example”；
        char   *string2=“xxxxxxxx”；
        printf(“first string:\n”);
        printf(“%s\n %s\n”,string1,string2);
        test_example2(string2,string1);
        printf(“second string:\n”);
        printf(“%s\n %s\n”,string1,string2);
        return  0；
}
```

```
  AREA example2,CODE,READONLY
  EXPORT test_example2
test_example2
        LDRB  R2,[R1],#1  ;字节加载并更新地址
        STRB  R2,[R0],#1  ;字节存储并更新地址
        CMP R2,#0         ;判断R2是否为0
        BNE test_example2 ;若条件不成立则继续执行
        MOV PC,LR         ;从子程序返回
        END
```

一些基本的ARM指令功能段

- 位操作

```
MOV R0,R2,LSR #24
ORR R3,R0,R3,LSL #8
```

；将R2的高8位数据传到R0中，R0的高24位设置成0

；将R3中的数据逻辑左移8位，这时R3的低8位为0，ORR操作将R0（高24位为0）中低8位数据传送到R3中

- 跳转指令的应用
  - 子程序调用

……

BL function

……

……

function

…

…

MOV PC,LR

- 条件执行

```
int gcd(int a,int b)
 { while (a!=b)
        if (a>b)
                a=a-b;
        else
                b=b-a;
        return a;
  }
```

对应的ARM汇编代码段。代码执行前R0中存放a，R1中存放b；代码执行后R0中存放返回结果。

```
gcd
CMP     R0,R1       ;比较a和b的大小
SUBGT   R0,R0,R1 ;if(a>b)  a=a-b
SUBLT   R1,R1,R0 ;if(b>a)  b=b-a
BNE     gcd     ;if(a!=b)跳转到gcd继续执行
MOV     PC,LR       ;子程序结束，返回
```

- 条件判断语句

```
if (a==0 || b==1)
    c=d+e;

CMP R0,#0
CMPNE R1,#1
ADDEQ R2,R3,R4
```

- 循环语句

```
MOV R0,#loop_count
loop
……
SUBS R0,R0,#1
BNE loop
```

链表操作中的条件码应用

在链表中搜索与某一数据相等的元素。

链表的每个元素包括两个字，第1个字中包含一个字节数据；第2个字中包含指向下一个链表元素的指针，当这个指针为0时表示链表结束。

代码执行前R0指向链表的头元素，R1中存放将要搜索的数据；代码执行后R0指向第1个匹配的元素，或者当没有匹配元素时，R0为0。

```
SEARCH
    CMP       R0,#0          ;R0指针是否为空
    LDRNEB    R2,[R0]        ;读取当前元素中的字节数据
    CMPNE     R1,R2          ;判断数据是否为搜索的数据
    LDRNE     R0,[R0,#4]     ;如果不是,指针R0指向下一个元素
    BNE       SEARCH         ;跳转到search执行
    MOV       PC,LR          ;搜索完成，程序返回
```

End of Chapter 3