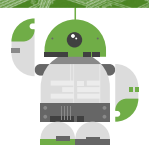
江苏师范大学 电气工程及自动化学院
JIANGSU NORMAL UNIVERSITY SCHOOL OF ELECTRICAL ENGINEERING & AUTOMATION





第3章 ARM指令系统

李灿 | 12#407A | lic@jsnu.edu.cn | https://sslic.cn/es

章节内容

- ARM指令集概述
- ARM指令寻址方式
- ARM指令
- Thumb指令

章节内容

- ARM指令集概述
- ARM指令寻址方式
- ARM指令
- Thumb指令

- 指令分类和指令格式
- ARM指令的条件码
- ARM指令集编码

3.1 ARM指令集概述

- **简介：**
  - ARM处理器是基于精简指令集计算机(RISC)原理设计的，指令集和相关译码机制较为简单
  - ARM具有32位ARM指令集和16位Thumb指令集：
    - ARM指令集效率高，但是代码密度低
    - Thumb指令集是ARM指令集的子集，具有较高代码密度，仍然保持ARM的大多数性能上的优势
  - 所有的ARM指令都是可以有条件执行的，而仅有一条Thumb指令具备条件执行功能
  - ARM程序和Thumb程序可相互调用，相互之间的状态切换开销几乎为零

Thumb 指令集小巧、灵活



ARM 指令集支持 ARM 核所有特性，且高效、快速

3.1 ARM指令集概述

- ARM指令集长度可以是以下任一种：
  - 32 bits (ARM状态)
  - 16 bits (Thumb状态)
- ARMTDMI 支持3种数据类型：
  - 字节 (8-bit)
  - 半字 (16-bit)
    - 半字必须被排列成2个字节边界对齐
  - 字 (32-bit)
    - 字必须被排成4个字节边界对齐
- ARM指令集向后兼容：
  - 新版本增加指令，同时保持指令向后兼容

3.1 ARM指令集概述

- **Load/Store结构（加载/存储结构）**
  - ARM微处理器的指令集是Load/Store型的
  - 指令集只能处理寄存器中的数据，且处理结果都要放回寄存器
  - 对系统存储器中的数据，需要通过Load/Store指令进行存取

3.1.1 指令分类和指令格式

- ARM指令分类
  - ① 跳转指令
  - ② 数据处理指令
  - ③ 程序状态寄存器（PSR）处理指令
  - ④ 加载/存储指令
  - ⑤ 协处理器指令
  - ⑥ 异常产生指令

3.1.1 指令分类和指令格式

ARM指令格式

ARM是三地址指令格式（基本格式）

<opcode> {<cond>} {S} <Rd>, <Rn>, {<operand2>}

<opcode>	操作码：指令助记符，如 MOV、ADD
<cond>	可选条件码：指令执行条件，如 EQ、GT
{S}	可选后缀：指令的操作是否影响CPSR寄存器的值
<Rd>	目标寄存器，如 R1、R2
<Rn>	存放第1个操作数的寄存器，如 R1、R2
<operand2>	第2个操作数，如 R1、R2、#2 （非常灵活）

例：R3 ← R1 + R2

指令语法	目标寄存器(Rd)	源寄存器1(Rn)	源寄存器2(Rm)
ADD R3,R1,R2	R3	R1	R2

3.1.2 ARM指令的条件码

- 每条ARM指令包含4位条件码
  - 位于指令高4位[31:28]
  - 作用：设置指令执行的条件
  - ARM状态下几乎所有的指令均可有条件执行
  - Thumb指令只有跳转指令(B)可有条件执行
  - 默认为AL

条件码	助记符后缀	标志	含义
0000	EQ	Z置位	相等
0001	NE	Z清零	不相等
0010	CS/HS	C置位	无符号数大于或等于
0011	CC/LO	C清理	无符号数小于
0100	MI	N置位	负数
0101	PL	N清零	正数或零
0110	VS	V置位	溢出
0111	VC	V清零	未溢出
1000	HI	C置位Z清零	无符号数大于
1001	LS	C清零Z置位	无符号数小于或等于
1010	GE	N等于V	带符号数大于或等于
1011	LT	N不等于V	带符号数小于
1100	GT	Z清零且N等于V	带符号数大于
1101	LE	Z置位或N不等于V	带符号数小于或等于
1110	AL	忽略	无条件执行

3.1.2 ARM指令的条件码

- 使用条件码的示例

C 代码

```
If(a > b)
        a++;
Else
        b++;
```



汇编代码

```
CMP    R0, R1          ; R0(a)与R1(b)比较
ADDHI  R0, R0, #1      ; 若R0>R1，则R0=R0+1
ADDLS  R1, R1, #1      ; 若R0≤R1，则R1=R1+1
```

使用条件码可实现高效的逻辑操作（节省跳转/条件语句），提高代码效率

3.1.3 ARM指令集编码

31:28	27	26	25	24	23	22	21	20	19:16	15:12	11:8	7	6	5	4	3:0	
cond	0	0	0	0	0	0	A	S	Rd	Rn	Rs	1	0	0	1	Rm	乘法(乘加)
cond	0	0	0	0	1	U	A	S	RdHi	RdLo	Rs	1	0	0	1	Rm	长乘(乘加)
cond	0	0	0	1	0	B	0	0	Rn	Rd	0	1	0	0	1	Rm	数据交换
cond	0	0	0	1	0	0	1	0	1	1	1	0	0	0	1	Rm	跳转交换
cond	0	0	0	P	U	0	W	L	Rn	Rd	0	1	S	H	1	Rm	半字存取寄存器偏移
cond	0	0	0	P	U	1	W	L	Rn	Rd	offset	1	S	H	1	offset	半字存取立即数偏移
cond	0	0	I	opcode				S	Rn	Rd	operand2						数据处理
cond	0	1	I	P	U	B	W	L	Rn	Rd	offset						单寄存器存取
cond	1	0	0	P	U	S	W	L	Rn	register list							多寄存器存取
cond	1	0	1	L	offset												跳转
cond	1	1	0	P	U	N	W	L	Rn	CRd	cp_num	offset					协处理器数据存取
cond	1	1	1	0	op1				CRn	CRd	cp_num	op2		0	CRm		协处理器数据操作
cond	1	1	1	0	op1			L	CRn	Rd	cp_num	op2		1	CRm		协处理器寄存器传送
cond	1	1	1	1	swi number												软中断

大部分指令编码中定义了第1操作数、第2操作数、目的操作数、条件标志影响位

章节内容

- ARM指令集概述
- **ARM指令寻址方式**
- ARM指令
- Thumb指令

3.2 ARM指令寻址方式

- **寻址方式：**
  - ✧ 根据指令中给出的**地址码字段**来实现**寻找真实操作数地址**的方式
  - ✧ ARM处理器有**9种**基本寻址方式
    1. 立即寻址
    2. 寄存器寻址
    3. 寄存器移位寻址
    4. 寄存器间接寻址
    5. 变址寻址
    6. 多寄存器寻址
    7. 堆栈寻址
    8. 块复制寻址
    9. 相对寻址

3.2.1 立即寻址

- 立即寻址也称立即数寻址，是一种特殊的寻址方式
  - ✧ 立即寻址指令中的操作码字段后面的地址码部分即是操作数本身
    - 即数据包含在指令中，取出指令也就取出了可以立即使用的操作数——称为立即数（立即数要求以“#”为前缀）

```
SUBS   R0, R0, #1     ; R0减1，结果放入R0，且影响标志位
MOV    R0, #0xFF000   ; 将立即数0xFF000放入R0
```



  - ✧ 有效的立即数（8位可直接表示，32位举例如下）
    - 0xFF，0x104，0xFF0，0xFF00
    - 0xFF000，0xFF000000，0xF000000F

3.2.2 寄存器寻址

- 寄存器寻址是利用寄存器中的数值作为操作数
  - 指令中的地址码字段给出的是寄存器编号
  - 寄存器内容是所需要的操作数

```
MOV    R1, R2          ; R1 ← R2
SUB    R0, R1, R2      ; R0 ← R1-R2
```

```
MOV    R1, R2
```



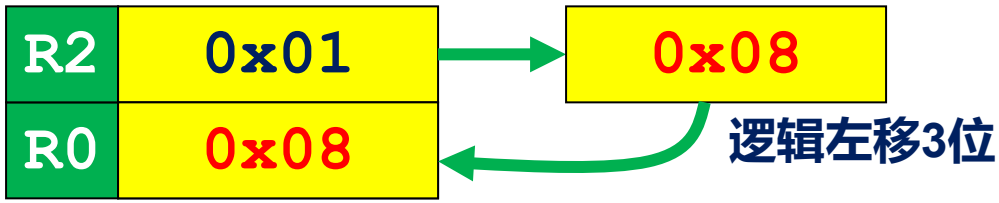
将R2的值存入R1

3.2.3 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个寄存器操作数要先进行移位操作，再与第1个操作数结合

```
MOV    R0, R2, LSL #3      ; R0 ← R2×2^3
ANDS   R1, R1, R2, LSL R3; R1 ← (R2×2^R3) & R1
```

```
MOV    R0, R2, LSL #3
```

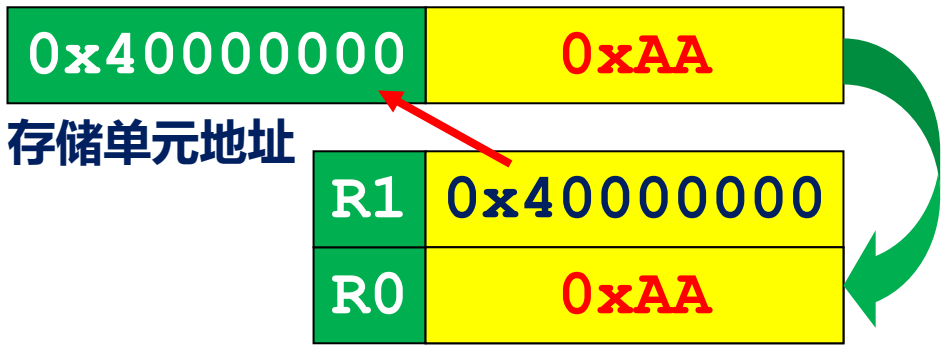


3.2.4 寄存器间接寻址

- 寄存器间接寻址是利用寄存器中指针指向的数值作为操作数
  - ✧ 指令中地址码字段给出的是一个通用寄存器编号
  - ✧ 寄存器内容是所需要的操作数的地址指针

```
LDR    R0, [R1]      ; R0 ← [R1]
STR    R0, [R1]      ; [R1] ← R0
```

```
LDR    R0, [R1]
```



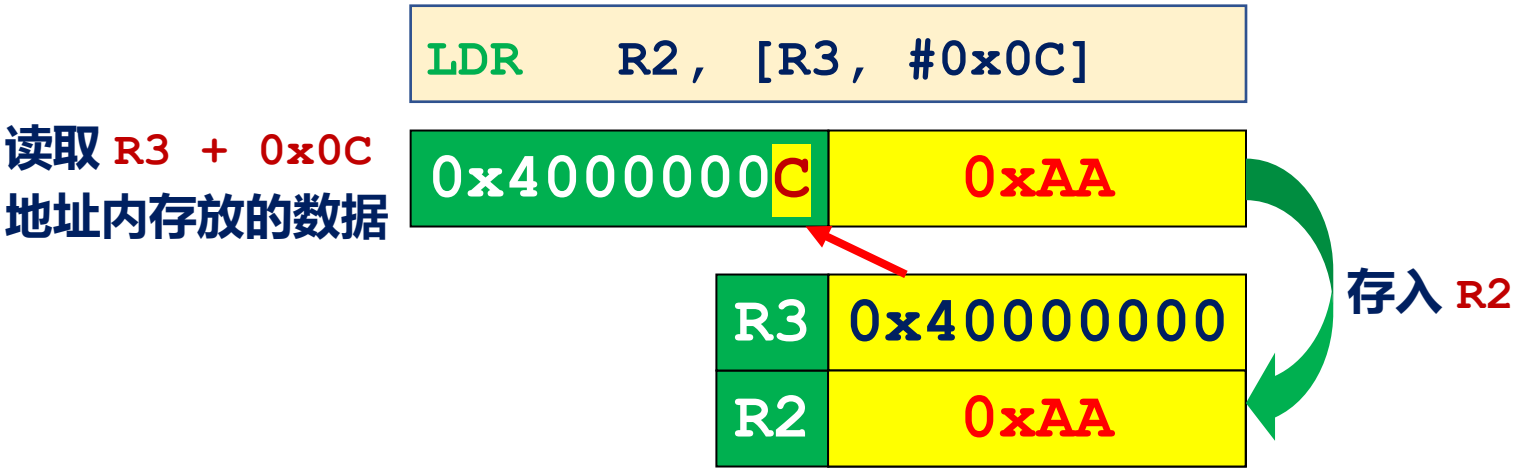
3.2.5 变址寻址

- 变址寻址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

前索引	`LDR R0, [R1, #4]`	`; R0 ← [R1+4]`
自动索引	`LDR R0, [R1, #4]!`	`; R0 ← [R1+4], R1 ← R1+4`
	`STR R0, [R1, #-4]!`	`; R0 → [R1-4], R1 ← R1-4`
后索引	`LDR R0, [R1], #4`	`; R0 ← [R1], R1 ← R1+4`
基址加索引	`LDR R0, [R1, R2]`	`; R0 ← [R1+R2]`
	`LDR R0, [R1, R2, LSL #2]`	`; R0 ← [R1 + R2×2^2]`

3.2.5 变址寻址

- 变址寻址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

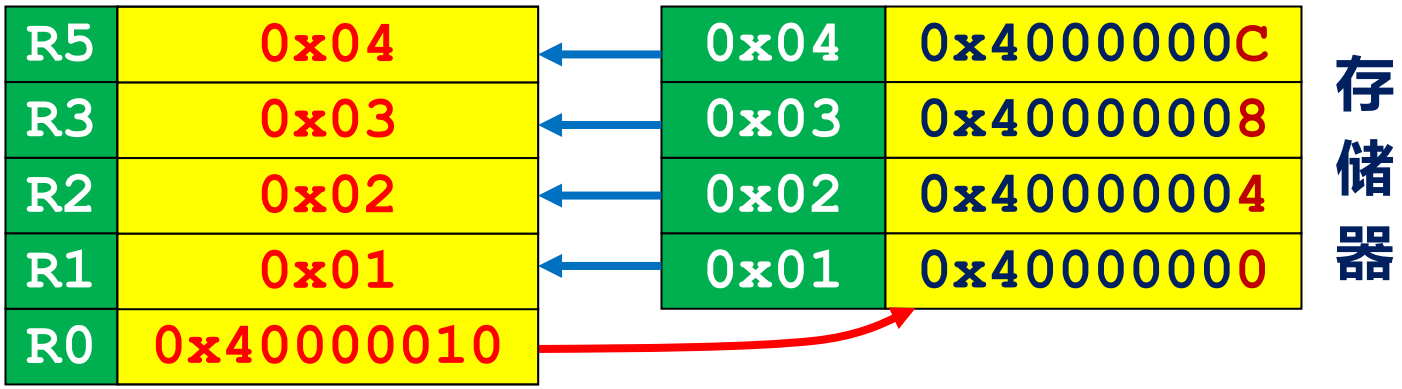


3.2.6 多寄存器寻址

- 多寄存器寻址可用一条指令传送多个寄存器值
  - ✧ 允许一条指令传送最多16个寄存器的值

```
LDMIA R0, {R1-R3, R5}; 将R0指向的连续存储单元加载到寄存器
; R1 ← [R0], R2 ← [R0+4], R3 ← [R0+8], R5 ← [R0+12]
STMIA R0, {R1-R3, R5}; 寄存器内容存到R0指向的连续存储单元
; [R0] ← R1, [R0+4] ← R2, [R0+8] ← R2, [R0+12] ← R5
```

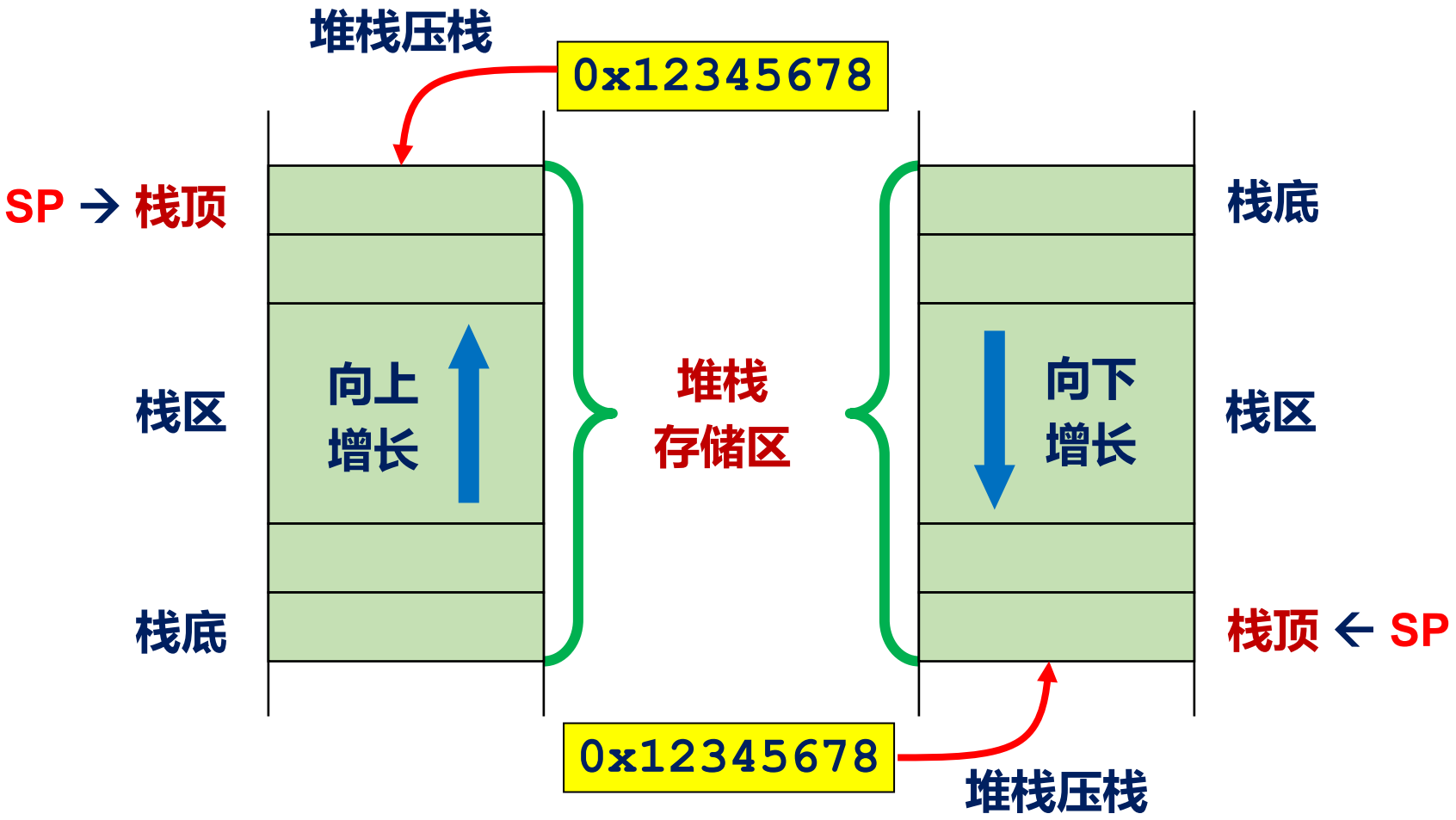
```
LDMIA R0!,{R1-R3,R5}
```
“!”表示最终 R0 更新为 R0+16 = R0+0x10



3.2.7 堆栈寻址

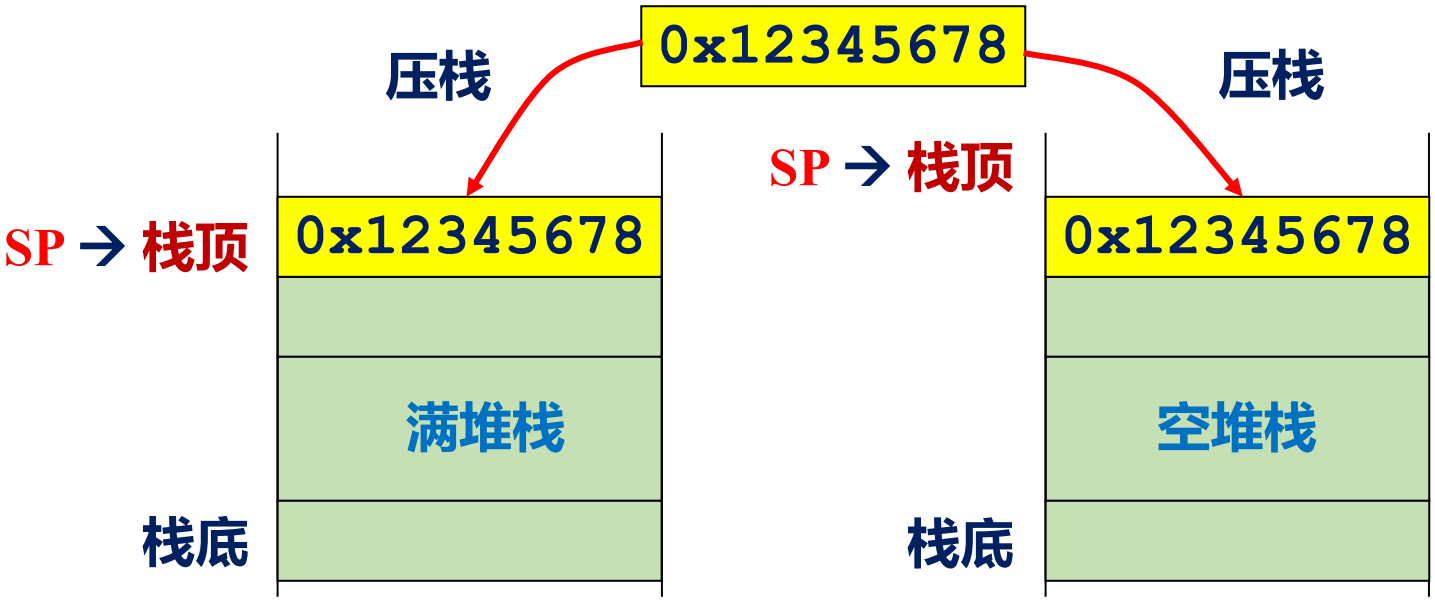
- **堆栈**是一个**按特定顺序进行存取**的**连续**存储区
  - 操作顺序为“先进后出”和“后进先出”
  - 存储器堆栈可分为两种：
    - 向上生长：向高地址方向生长，称为**递增堆栈**
    - 向下生长：向低地址方向生长，称为**递减堆栈**
- **堆栈寻址**是隐含的：
  - 使用一个**专门的寄存器**（堆栈指针SP）指示当前操作位置
  - 堆栈指针**总是指向栈顶**
  - 堆栈的操作是通过堆栈指针实现的

3.2.7 堆栈寻址


堆栈压栈
0x12345678
SP → 栈顶
栈区
栈底
向上
增长
堆栈
存储区
向下
增长
栈底
栈区
栈顶 ← SP
0x12345678
堆栈压栈

3.2.7 堆栈寻址

- ✧ 满堆栈（Full Stack）：堆栈指针指向最后压入的堆栈的有效数据项
  - SP指向堆栈中最后一个已使用的位置
- ✧ 空堆栈（Empty Stack）：堆栈指针指向下一个待压入数据的空位置
  - SP指向堆栈中第一个未使用的位置



3.2.7 堆栈寻址

- 4种堆栈方式：
  - ✧ 满递增堆栈（FA，Full Ascending）
    - 堆栈向上增长，堆栈指针指向内含有效数据项的最高地址
    - 指令如：LDMFA，STMFA
  - ✧ 空递增堆栈（EA，Full Ascending）
    - 堆栈向上增长，堆栈指针指向堆栈上的第一个空位置
    - 指令如：LDMEA，STMEA
  - ✧ 满递减堆栈（FD，Full Descending）
    - 堆栈向下增长，堆栈指针指向内含有效数据项的最低地址
    - 指令如：LDMFD，STMFD
  - ✧ 空递减堆栈（ED，Empty Descending）
    - 堆栈向下增长，堆栈指针向堆栈下的第一个空位置
    - 指令如：LDMED，STMED

3.2.8 块复制寻址

- 块复制寻址是多寄存器指令 LDM/STM 的寻址方式
  - ✧ 作用：将存储器中的一个数据块加载到多个寄存器中
  - ✧ 使用堆栈寻址的方式也能实现块复制功能

```
STMIA R0!, {R1-R3}; [R0] ← R1-R3
                    ; 注意：基地址R0在加载R1后，向上增长(+4)
                    ; 执行完毕后，R0的地址变为最开始地址+0x0C

STMIB R0!, {R1-R3}; [R0] ← R1-R3
                    ; 注意：基地址R0在加载R1前，向上增长(+4)
                    ; 执行完毕后，R0的地址变为最开始地址+0x0C
```

3.2.9 相对寻址

- 相对寻址是变址寻址的一种变通
  - ① 以程序计数器PC的当前值为基地址
  - ② 指令中的地址码字段作为偏移量
  - ③ 两者相加后得到操作数的有效地址

```
LDR     PC, [PC, #+0xFF0]  ; PC ← [当前指令地址 + 8 + 0xFF0]
```

章节内容

- ARM指令集概述
- ARM指令寻址方式
- **ARM指令**
- Thumb指令

```
; 文件名：TEST.S
; 功能：实现两个寄存器相加
; 说明：使用 Keil4 软件仿真调试
        AREA    Example1, CODE, READONLY        ; 代码段Example1开始，只读
        ENTRY                                   ; 程序入口
        CODE32                                  ; 声明下面为32位ARM指令
START   MOV     R0, #0                          ; 设置参数
        MOV     R1, #10
LOOP    BL      ADD_SUB                         ; 调用子程序ADD_SUB
        B       LOOP                            ; 跳转到LOOP
ADD_SUB
        ADDS    R0, R0, R1                      ; R0 ← R0 + R1
        MOV     PC, LR                          ; 子程序返回
        END                                     ; 程序结束
```

3.3 ARM指令

- ARM指令集可分为：
  - ① 跳转指令（分支指令）
  - ② 数据处理指令
  - ③ 程序状态寄存器指令
  - ④ 加载和存储指令（存储器访问指令）
  - ⑤ 协处理指令
  - ⑥ 异常产生指令
  - ⑦ 伪指令*(4.3节介绍)

3.3.1 跳转指令

- 跳转指令用于实现程序流程的跳转，指令编码：

31:28	27	26	25	24	23	22	21	20	19:16	15:12	11:8	7	6 5 4	3:0
cond	1	0	1	L	Offset (24 bits)									

- 在ARM中有两种方式：

① 使用专门的跳转指令

指令	含义
B	跳转
BL	带返回的跳转
BX	带状态切换的跳转
BLX	带返回和状态切换的跳转

② 直接向程序计数器PC写入跳转地址值

3.3.1 跳转指令

助记符	操作	条件码位置
B Label	PC ← Label	B{<cond>}
BL Label	LR ← PC-4, PC ← Label	BL{<cond>}
BX <Rm>	PC ← Rm，切换状态	BX{<cond>}

3.3.1 跳转指令

助记符	操作	条件码位置
B Label	PC ← Label	B{<cond>}
BL Label	LR ← PC-4, PC ← Label	BL{<cond>}
BX <Rm>	PC ← Rm，切换状态	BX{<cond>}

- ✧ **B指令**：程序计数器PC立即跳转到给定的目标地址，从那里执行
  - ARM指令字对齐，传送到PC中的目标地址值的低2位被忽略
- ✧ **相对跳转地址**：PC+offset的地址
  - 跳转范围offset限制在当前指令的±32MB地址内
- ✧ **绝对跳转地址**：直接跳转到某固定地址

```
B       WAITA,          ; 跳转到WAITA标号处
B       0x1234,         ; 跳转到绝对地址0x1234处
```

3.3.1 跳转指令

助记符	操作	条件码位置
B Label	PC ← Label	B{<cond>}
BL Label	LR ← PC-4, PC ← Label	BL{<cond>}
BX <Rm>	PC ← Rm，切换状态	BX{<cond>}

✧ **BL指令：适用子程序调用，是实现子程序调用的基本但常用手段**

① 将下一条指令的地址复制到R14（LR）链接寄存器中

② 再（同时）跳转到指定地址运行程序

- 跳转范围offset限制在当前指令的±32MB地址内
- 返回地址存在LR中，一般不能嵌套下一级子程序，可以用堆栈存LR

```
BL      Label   ; 跳转到Label处执行，同时下一条指令地址存入LR
```

3.3.1 跳转指令

① 当程序执行到BL跳转指令Addr1处时，硬件将下一条指令的地址Addr2装入LR寄存器，并把跳转地址Label装入PC

② 程序跳转到目标地址Label继续执行，当子程序执行结束后，将LR寄存器内容存入PC，返回调用函数继续执行

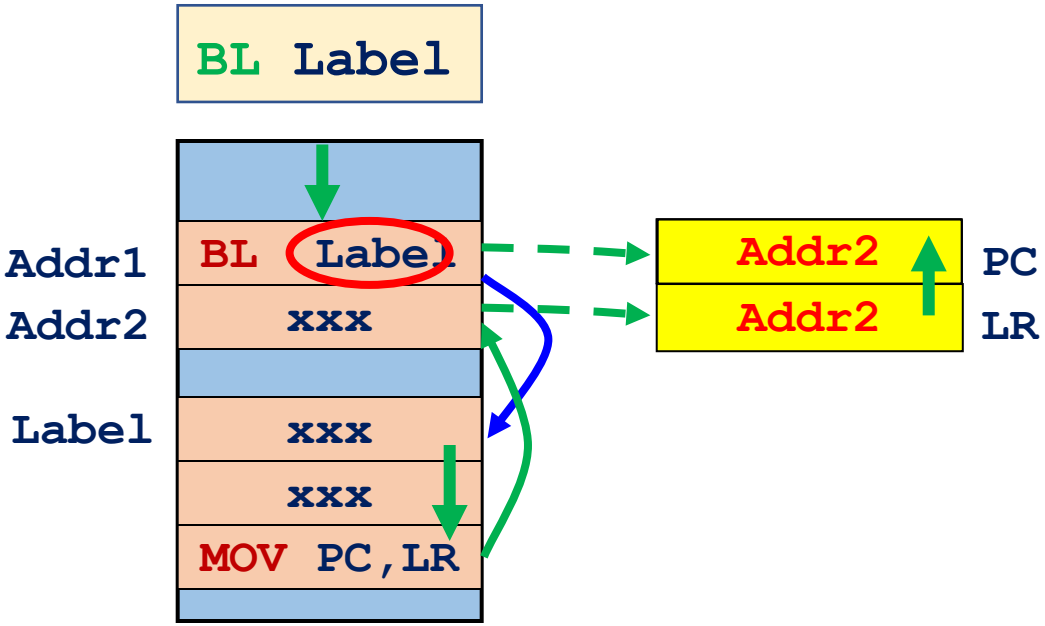


图. 利用BL指令调用子程序过程

3.3.1 跳转指令

助记符	操作	条件码位置
B Label	PC ← Label	B{<cond>}
BL Label	LR ← PC-4, PC ← Label	BL{<cond>}
BX <Rm>	PC ← Rm，切换状态	BX{<cond>}

- **BX指令**：跳转到指定的目标地址
  目标地址处的指令可以是ARM指令或者Thumb指令
  - ✧ 目标地址 = <Rm> & 0xFFFFFFFE，即Rm中高31位存入PC
  - ✧ 根据跳转地址<Rm>的最低位Rm[0]来切换处理器状态
    Rm[0]复制给CPSR的T标志位：
    - 若Rm[0]=1，目标地址处为Thumb指令
    - 若Rm[0]=0，目标地址处为ARM指令
      此时Rm[1]不能为1（因ARM需字对齐）

Rm[0]	跳转后	
	CPSR标志T位	处理器状态
1	1	Thumb
0	0	ARM

3.3.1 跳转指令

助记符	操作	条件码位置
B Label	PC ← Label	B{<cond>}
BL Label	LR ← PC-4, PC ← Label	BL{<cond>}
BX <Rm>	PC ← Rm，切换状态	BX{<cond>}

- **BX指令：跳转到指定的目标地址**
  目标地址处的指令可以是ARM指令或者Thumb指令
  - ✧ 目标地址 = <Rm> & 0xFFFFFFFE，即Rm中高31位存入PC
  - ✧ 根据跳转地址<Rm>的最低位Rm[0]来切换处理器状态

```
BX      R0      ; 跳转到R0指定地址，并根据R0[0]切换处理器状态
                ; 若R0[0] = 1，进入Thumb状态
BLX     R0      ; 功能为BL和BX之并
```

3.3.2 数据处理指令

- 功能：主要完成寄存器中数据的各种运算操作
- 特点：

① 所有操作数都是32位的，可以是寄存器，也可以是立即数
不能对内存中的数据进行操作

② 所有ARM数据处理指令均可使用S后缀，使指令影响状态标志

③ 如果数据操作有结果，结果为32位，放在一个目的寄存器中
例外：长乘指令产生64位结果

④ ARM指令中使用3地址模式
  - 操作数寄存器在指令中分别指定（1-2个）
  - 结果寄存器指令中分别指定（1个）

3.3.2 数据处理指令

- 数据处理指令格式

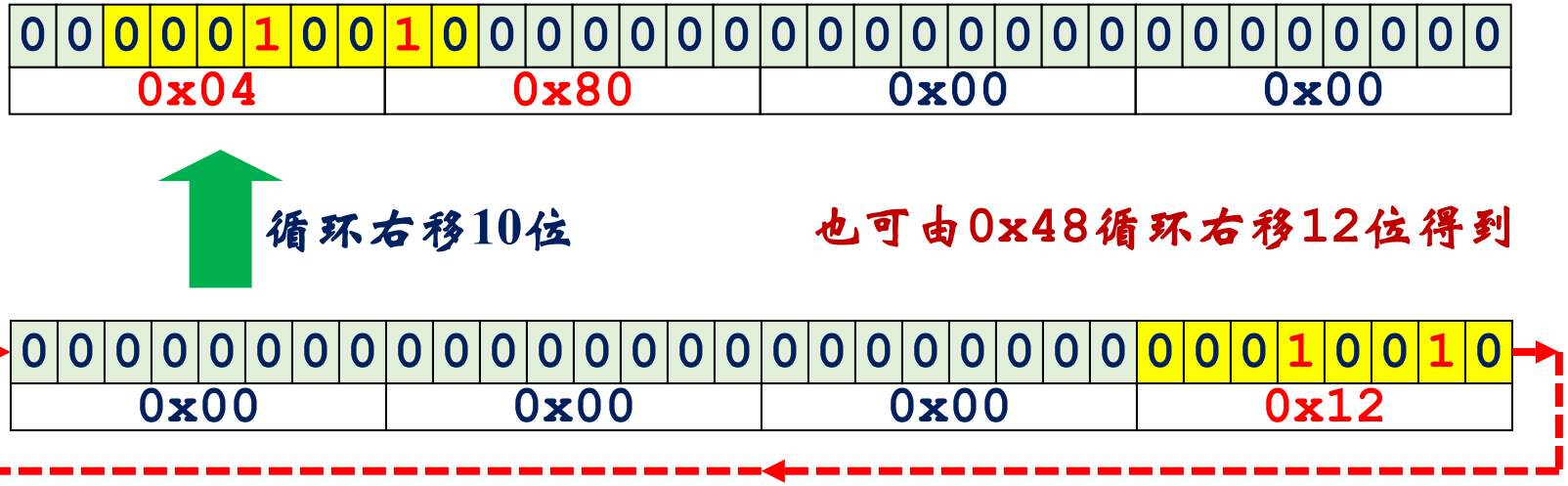
<opcode> {<cond>} {S} <Rd>, <Rn>, {<operand2>}	
<opcode>	操作码：指令助记符，如 MOV、ADD
<cond>	可选条件码：指令执行条件，如 EQ、GT
{S}	可选后缀：指令的操作是否影响CPSR寄存器的值
<Rd>	目标寄存器，如 R1、R2
<Rn>	存放第1个操作数的寄存器，如 R1、R2
<operand2>	第2个操作数，如 R1、R2、#2 （非常灵活）

- 灵活的使用第2个操作数“operand2”能够提高代码效率
  - ✧ 常数表达式：#immed_8r
  - ✧ 寄存器方式：Rm
  - ✧ 寄存器移位方式：Rm,shift

3.3.2 数据处理指令

- **常数表达式：#immed_8r**
  - ✧ 该常数必须对应8位位图（与图3.1数据处理指令编码有关）
    即该常数是由一个8位常数通过循环右移偶数位得到
    - 当常数本身为8位时，不需要右移（移0位）直接表示为低8位
    - 当常数为8~32位时，偶次循环右移能将所有的1移到低8位即有效

```
AND     R1, R2, #0x04800000;  该常数可由0x12循环右移10位得到
```



3.3.2 数据处理指令

- **常数表达式：#immed_8r**
  - ✧ 该常数必须对应8位位图
    即该常数是由一个8位常数通过循环右移偶数位得到

```
AND     R1, R2, #0x04800000; 该常数可由0x12循环右移10位得到
```

0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x04										0x80										0x00								0x00			

本质上：有效常数循环右（左）移
偶数次位一定能完全进入低8位

右移22位

右移20位

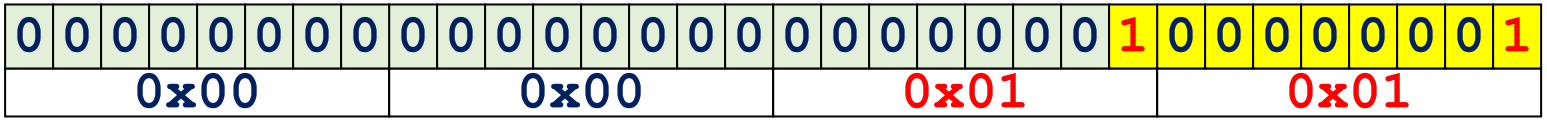
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0
0x00								0x00								0x00								0x12							

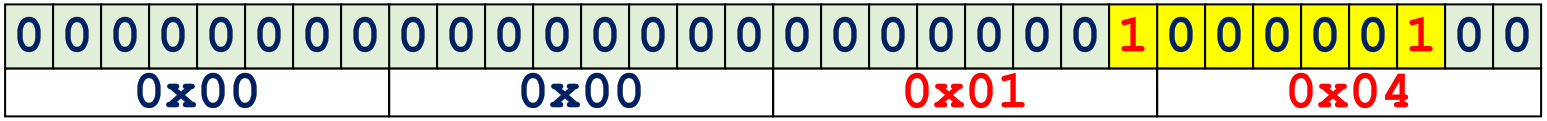
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0
0x00								0x00								0x00								0x48							

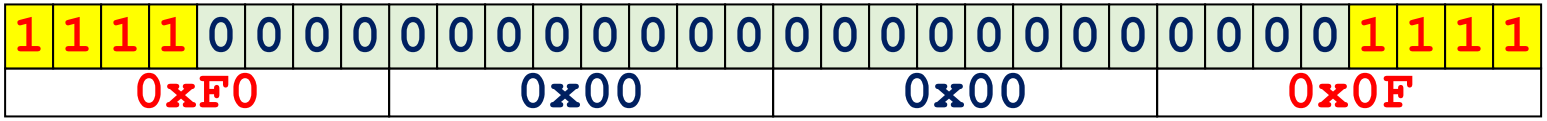
3.3.2 数据处理指令

- **常数表达式：#immed_8r**
  - ✧ 该常数必须对应8位位图
    即该常数是由一个8位常数通过循环右移偶数位得到
    - 当常数本身为8位时，不需要右移（移0位）直接表示为低8位
    - 当常数为8~32位时，偶次循环右移能将所有的1移到低8位即有效
  - ✧ 以下常数表达式是否有效？

```
#0x0F,   #0x101,   #0x104,   #0xFF1,   #0xF000000F
```







3.3.2 数据处理指令

- **寄存器方式：Rm**
  - ✧ 在寄存器方式下，操作数即为寄存器的数值

```
SUB    R1, R2, R2   ; R1 ← R1-R2
```

3.3.2 数据处理指令

- **寄存器移位方式：Rm,shift**
  - ✧ 将寄存器移位结果作为操作数，但Rm值保持不变
    - 移位操作不消耗额外的时间

操作码	说明	操作码	说明
ASR #n	算术右移n位	ROR #n	循环右移n位
LSL #n	逻辑左移n位	RRX	带扩展的循环右移1位
LSR #n	逻辑右移n位	Type Rs	Rs为偏移量寄存器，低8位有效

Type是ASR，LSL，LSR，ROR中的一种，寄存器Rs的低8位表示移位位数

```
ADD   R1, R1, R1, LSL #3      ; R1 ← R1+R1×2^3 = 9R1
SUB   R1, R1, R2, LSR R3      ; R1 ← R1-R2/2^R3
```

```
; R1 ← R1 + R1<<3
; R1 ← R1 - R2>>R3
```

3.3.2 数据处理指令

- 寄存器移位方式：Rm,shift

LSL移位操作



LSR移位操作



ASR移位操作



ROR移位操作



RRX移位操作



ASR：右移，左边用符号位填充

RRX：右移1位，标志位C移入bit[31]，bit[0]移入C

3.3.2 数据处理指令

- 指令编码

31:28	27	26	25	24	23	22	21	20	19:16	15:12	11:8	7	6 5 4	3:0
cond	0	0	I	opcode				S	Rn	Rd	operand2			

- bit[25]
  - I=1，立即数移位
  - I=0，寄存器移位
- bit[24:21]
  - 数据处理指令操作码
- Bit[20]
  - 后缀S设置的条件码

opcode	指令助记符	说明
0000	AND	逻辑与
0001	EOR	逻辑异或
0010	SUB	减法
0011	RSB	逆向减法
0100	ADD	加法
0101	ADC	带进位加法
0110	SBC	带进位减法
0111	RSC	带进位逆向减法
1000	TST	位测试
1001	TEQ	相等测试
1010	CMP	比较
1011	CMN	负数比较
1100	ORR	逻辑或
1101	MOV	数据传送
1110	BIC	位清除
1111	MVN	数据非传送

3.3.2 数据处理指令

- 数据处理指令根据指令实现处理功能有以下6类
  - 数据传送指令
  - 算术运算指令
  - 逻辑运算指令
  - 比较指令
  - 测试指令
  - 乘法指令
    - 乘法有独立的2条编码
    - 后面将重点介绍

opcode	指令助记符	说明
0000	AND	逻辑与
0001	EOR	逻辑异或
0010	SUB	减法
0011	RSB	逆向减法
0100	ADD	加法
0101	ADC	带进位加法
0110	SBC	带借位减法
0111	RSC	带借位逆向减法
1000	TST	位测试
1001	TEQ	相等测试
1010	CMP	比较
1011	CMN	负数比较
1100	ORR	逻辑或
1101	MOV	数据传送
1110	BIC	位清除
1111	MVN	数据取反传送

3.3.2 数据处理指令：数据传送

- 数据传送指令用于在寄存器之间进行数据的双向传输
  - ✧ 有2条数据传送指令

指令格式	说明	操作
MOV{cond}{S} Rd, operand2	数据传送	Rd ← operand2
MVN{cond}{S} Rd, operand2	数据取反传送	Rd ← ~operand2

3.3.2 数据处理指令：数据传送

指令格式	说明	操作
MOV{cond}{S} Rd, operand2	数据传送	Rd ← operand2
MVN{cond}{S} Rd, operand2	数据取反传送	Rd ← ~operand2

MOV指令

- 将8位图立即数、寄存器、移位寄存器传送到目标寄存器（Rd）
  - MOVS根据结果更新标志N和Z，计算operand2时更新标志C，不影响V

```
MOV    R1, #0xF000000B     ; R1 ← 0xF000000B
MOVEQ  R0, R1              ; R0 ← R1(有条件地)
MOVS   R0, R1              ; R0 ← R1, 影响CPSR标志位
MOV    PC, LR              ; PC ← LR, 常用于子程序返回
MOV    R1, R0, LSL #3      ; R1 ← R0<<3
```

3.3.2 数据处理指令：数据传送

指令格式	说明	操作
MOV{cond}{S} Rd, operand2	数据传送	Rd ← operand2
MVN{cond}{S} Rd, operand2	数据取反传送	Rd ← ~operand2

- **MVN指令**
  - ✧ 将8位图立即数、寄存器、移位寄存器按位取反后传送到Rd
    - MVNS根据结果更新标志N和Z，计算operand2时更新标志C，不影响V
    - 因为其具有取反功能，所以可以装载范围更广的立即数

```
MNV     R0, #0          ; R0 ← 立即数0按位取反，结果为R0=-1
MNV     R1, #0xFF       ; R1 ← 0xFF按位取反
MNV     R1, R2          ; R1 ← R2按位取反
```

3.3.2 数据处理指令：算术运算

- 算术运算指令用于完成常用的算术运算：
  - ✧ 运算结果保存在目标寄存器
  - ✧ 可以更新CPSR中的条件标志位
  - ✧ 有6条数算术运算指令（这些指令影响N，Z，C和V标志位）

指令格式	说明	操作
ADD{cond}{S} Rd, Rn, opd2	加法	Rd ← Rn+opd2
ADC{cond}{S} Rd, Rn, opd2	带进位加法	Rd ← Rn+opd2+Carry
SUB{cond}{S} Rd, Rn, opd2	减法	Rd ← Rn-opd2
RSB{cond}{S} Rd, Rn, opd2	逆向减法	Rd ← opd2-Rn
SBC{cond}{S} Rd, Rn, opd2	带借位减法	Rd ← Rn-opd2-NCarry
RSC{cond}{S} Rd, Rn, opd2	带借位逆向减法	Rd ← opd2-Rn-NCarry

opd2：operand2　　Carry：进位（CPSR中的C条件标志位）　　Ncarry：借位

3.3.2 数据处理指令：算术运算

指令格式	说明	操作
ADD{cond}{S} Rd, Rn, opd2	加法	Rd ← Rn+opd2
ADC{cond}{S} Rd, Rn, opd2	带进位加法	Rd ← Rn+opd2+Carry
SUB{cond}{S} Rd, Rn, opd2	减法	Rd ← Rn-opd2
RSB{cond}{S} Rd, Rn, opd2	逆向减法	Rd ← opd2-Rn
SBC{cond}{S} Rd, Rn, opd2	带借位减法	Rd ← Rn-opd2-NCarry
RSC{cond}{S} Rd, Rn, opd2	带借位逆向减法	Rd ← opd2-Rn-NCarry

- ADD指令：Rd ← Rn + operand2

```
ADD    R0, R1, R2          ; R0 ← R1+R2
ADDS   R0, R1, #256        ; R0 ← R1+256, 并影响标志位
ADD    R1, R2, R3, LSL #3  ; R1 ← R2+(R3<<3)
```

3.3.2 数据处理指令：算术运算

指令格式	说明	操作
ADD{cond}{S} Rd, Rn, opd2	加法	Rd ← Rn+opd2
ADC{cond}{S} Rd, Rn, opd2	带进位加法	Rd ← Rn+opd2+Carry
SUB{cond}{S} Rd, Rn, opd2	减法	Rd ← Rn-opd2
RSB{cond}{S} Rd, Rn, opd2	逆向减法	Rd ← opd2-Rn
SBC{cond}{S} Rd, Rn, opd2	带借位减法	Rd ← Rn-opd2-NCarry
RSC{cond}{S} Rd, Rn, opd2	带借位逆向减法	Rd ← opd2-Rn-NCarry

- ADC指令：Rd ← Rn + operand2 + Carry
  - Carry：CPSR中的条件标志位(进位标志)C的值
  - 注意：做大于32位的加法时，应先设置S后缀来更新进位标志C

```
ADDS   R4, R0, R2   ; 加低位的字(低32位相加)，并更新标志位
ADC    R5, R1, R3   ; 加高位的字(高32位相加)，带进位
       ; 实现了两个64位整数相加结果按高到低存放在R5和R4中
```

3.3.2 数据处理指令：算术运算

指令格式	说明	操作
ADD{cond}{S} Rd, Rn, opd2	加法	Rd ← Rn+opd2
ADC{cond}{S} Rd, Rn, opd2	带进位加法	Rd ← Rn+opd2+Carry
SUB{cond}{S} Rd, Rn, opd2	减法	Rd ← Rn-opd2
RSB{cond}{S} Rd, Rn, opd2	逆向减法	Rd ← opd2-Rn
SBC{cond}{S} Rd, Rn, opd2	带借位减法	Rd ← Rn-opd2-NCarry
RSC{cond}{S} Rd, Rn, opd2	带借位逆向减法	Rd ← opd2-Rn-NCarry

- SUB指令：Rd ← Rn - operand2

```
SUB    R0, R1, R2          ; R0 ← R1-R2
SUBS   R0, R1, #256        ; R0 ← R1-256, 并影响标志位
SUB    R1, R2, R3, LSL #3  ; R1 ← R2-(R3<<3)
```

3.3.2 数据处理指令：算术运算

指令格式	说明	操作
ADD{cond}{S} Rd, Rn, opd2	加法	Rd ← Rn+opd2
ADC{cond}{S} Rd, Rn, opd2	带进位加法	Rd ← Rn+opd2+Carry
SUB{cond}{S} Rd, Rn, opd2	减法	Rd ← Rn-opd2
RSB{cond}{S} Rd, Rn, opd2	逆向减法	Rd ← opd2-Rn
SBC{cond}{S} Rd, Rn, opd2	带借位减法	Rd ← Rn-opd2-NCarry
RSC{cond}{S} Rd, Rn, opd2	带借位逆向减法	Rd ← opd2-Rn-NCarry

- RSB指令：Rd ← operand2 - Rn

```
RSB    R0, R1, R2          ; R0 ← R2-R1
RSBS   R0, R1, #256        ; R0 ← 256-R1, 并影响标志位
RSB    R1, R2, R3, LSL #3  ; R1 ← (R3<<3)-R2
```

3.3.2 数据处理指令：算术运算

指令格式	说明	操作
ADD{cond}{S} Rd, Rn, opd2	加法	Rd ← Rn+opd2
ADC{cond}{S} Rd, Rn, opd2	带进位加法	Rd ← Rn+opd2+Carry
SUB{cond}{S} Rd, Rn, opd2	减法	Rd ← Rn-opd2
RSB{cond}{S} Rd, Rn, opd2	逆向减法	Rd ← opd2-Rn
SBC{cond}{S} Rd, Rn, opd2	带借位减法	Rd ← Rn-opd2-NCarry
RSC{cond}{S} Rd, Rn, opd2	带借位逆向减法	Rd ← opd2-Rn-NCarry

- SBC指令：Rd ← Rn - operand2 - Ncarry
  - ✧ NCarry：CPSR中的条件标志位（进位标志）C的值的反码（借位）
  - ✧ 注意：做大于32位的加法时，应先设置S后缀来更新进位标志C

```
SUBS   R4, R0, R2   ; 减低位的字（低32位相减），并更新标志位
SBC    R5, R1, R3   ; 减高位的字（高32位相减），带借位
       ; 实现了两个64位整数相减结果按高到低存放在R5和R4中
```

3.3.2 数据处理指令：算术运算

指令格式	说明	操作
ADD{cond}{S} Rd, Rn, opd2	加法	Rd ← Rn+opd2
ADC{cond}{S} Rd, Rn, opd2	带进位加法	Rd ← Rn+opd2+Carry
SUB{cond}{S} Rd, Rn, opd2	减法	Rd ← Rn-opd2
RSB{cond}{S} Rd, Rn, opd2	逆向减法	Rd ← opd2-Rn
SBC{cond}{S} Rd, Rn, opd2	带借位减法	Rd ← Rn-opd2-NCarry
RSC{cond}{S} Rd, Rn, opd2	带借位逆向减法	Rd ← opd2-Rn-NCarry

- RSC指令：Rd ← operand2 - Rn - Ncarry
  - ✧ NCarry：CPSR中的条件标志位（进位标志）C的值的反码（借位）
  - ✧ 注意：使用RSC指令，应先设置S后缀来更新进位标志C

```
RSBS   R2, R0, #0   ; R2 ← -R0
RSC    R3, R1, #0   ; R3 ← -R1 - NCarry
       ; 实现R1和R0中一个64位数值取负结果按高到低存放在R3和R2中
```

3.3.2 数据处理指令：逻辑运算

- 算术运算指令用于完成常用的逻辑运算
  - ✧ 运算结果保存在目标寄存器
  - ✧ 可以更新CPSR中的条件标志位
  - ✧ 有4条数逻辑运算指令
    这些指令影响N，Z，在计算operand2时更新C，不影响V标志位

指令格式	说明	操作
AND{cond}{S} Rd, Rn, opd2	逻辑与	Rd ← Rn & opd2
ORR{cond}{S} Rd, Rn, opd2	逻辑或	Rd ← Rn | opd2
EOR{cond}{S} Rd, Rn, opd2	逻辑异或	Rd ← Rn ^ opd2
BIC{cond}{S} Rd, Rn, opd2	位清除	Rd ← Rn & (~opd2)

opd2：operand2

3.3.2 数据处理指令：逻辑运算

指令格式	说明	操作
AND{cond}{S} Rd, Rn, opd2	逻辑与	Rd ← Rn & opd2
ORR{cond}{S} Rd, Rn, opd2	逻辑或	Rd ← Rn | opd2
EOR{cond}{S} Rd, Rn, opd2	逻辑异或	Rd ← Rn ^ opd2
BIC{cond}{S} Rd, Rn, opd2	位清除	Rd ← Rn & (~opd2)

- AND指令：Rd ← Rn & operand2

```
AND   R0, R0, #0x01  ; R0 ← R0 & 0x01, 即取出R0最低位数据
ANDS  R0, R0, #3     ; 取出最低2位数据, 并置标志位
ANDEQ R0, R1, R2     ; R0 ← R1 & R2, 标志位N=1时执行
```

3.3.2 数据处理指令：逻辑运算

指令格式	说明	操作
AND{cond}{S} Rd, Rn, opd2	逻辑与	Rd ← Rn & opd2
ORR{cond}{S} Rd, Rn, opd2	逻辑或	Rd ← Rn | opd2
EOR{cond}{S} Rd, Rn, opd2	逻辑异或	Rd ← Rn ^ opd2
BIC{cond}{S} Rd, Rn, opd2	位清除	Rd ← Rn & (~opd2)

- ORR指令：Rd ← Rn | operand2

```
ORR    R0, R0, #0x0F  ; R0低4位置1，其余位不变
ORR    R0, R0, #3     ; R0低2位置1，其余位不变
```

```
MOV    R1, R2, LSR #24      ; R1 ← R2>>24
ORR    R3, R1, R3, LSL #8   ; R3 ← R1 | (R3<<8)
       ; 两条语句实现R2的高8位数据移入R3的低8位中
```

3.3.2 数据处理指令：逻辑运算

指令格式	说明	操作
AND{cond}{S} Rd, Rn, opd2	逻辑与	Rd ← Rn & opd2
ORR{cond}{S} Rd, Rn, opd2	逻辑或	Rd ← Rn | opd2
EOR{cond}{S} Rd, Rn, opd2	逻辑异或	Rd ← Rn ^ opd2
BIC{cond}{S} Rd, Rn, opd2	位清除	Rd ← Rn & (~opd2)

- EOR指令：Rd ← Rn ^ operand2
  - ✧ 常用于反转第1个操作数Rn中的某些位

```
EOR    R1, R1, #0x0F        ; 将R1的低4位取反
EOR    R0, R0, #3           ; 将R0的低2位取反
EOR    R0, R1, R2           ; R0 ← R1 ^ R2
EORS   R0, R5, #0x01        ; R0 ← R5 ^ 0x01, 并影响标志位
```

3.3.2 数据处理指令：逻辑运算

指令格式	说明	操作
AND{cond}{S} Rd, Rn, opd2	逻辑与	Rd ← Rn & opd2
ORR{cond}{S} Rd, Rn, opd2	逻辑或	Rd ← Rn | opd2
EOR{cond}{S} Rd, Rn, opd2	逻辑异或	Rd ← Rn ^ opd2
BIC{cond}{S} Rd, Rn, opd2	位清除	Rd ← Rn & (~opd2)

- BIC指令：Rd ← Rn & (~operand2)
  - ✧ 常用于寄存器中某些位置零

```
BIC    R1, R1, #0x0F        ; 将R1的低4位清零，其他位不变
BIC    R0, R0, #0b1011      ; 将R0的第0,1,3位清零，其他位不变
BIC    R1, R2, R3           ; R1 ← R2 & (~R3)
```

3.3.2 数据处理指令：比较（含测试）

- 比较指令不保存运算结果，只更新CPSR中相应的条件标志位
  - ✧ 只有第1、2操作数，没有目标寄存器Rd
  - ✧ 比较指令影响标志位N、Z、C、V，不需要加后缀S
  - ✧ 有4条比较指令

指令格式	说明	操作
CMP{cond} Rn, opd2	比较	标志N/Z/C/V ← Rn - opd2
CMN{cond} Rn, opd2	负数比较	标志N/Z/C/V ← Rn + opd2
TST{cond} Rn, opd2	位测试	标志N/Z/C ← Rn & opd2
TEQ{cond} Rn, opd2	相等测试	标志N/Z/C ← Rn ^ opd2

opd2：operand2

CPSR中条件码标志位N/Z/C/V的设置，参考表2.3

3.3.2 数据处理指令：比较（含测试）

指令格式	说明	操作
CMP{cond} Rn, opd2	比较	标志N/Z/C/V ← Rn - opd2
CMN{cond} Rn, opd2	负数比较	标志N/Z/C/V ← Rn + opd2
TST{cond} Rn, opd2	位测试	标志N/Z/C ← Rn & opd2
TEQ{cond} Rn, opd2	相等测试	标志N/Z/C ← Rn ^ opd2

- CMP指令：N/Z/C/V ← Rn - operand2
  - ✧ 更新N/Z/C/V，以便后面的指令根据相应的条件标志判断执行
  - ✧ 例如：Rn>operand2时，后面有GT后缀的指令可以执行
  - ✧ 注意：SUBS也可用于此比较，但CMP不需要用Rd保存运算结果

```
CMP   R1, R0        ; 计算R1-R0，根据结果设置CPSR标志位
CMP   R1, #100      ; 计算R1-100，根据结果设置CPSR标志位
CMPGT R1, R2        ; 如果R1>100，则执行：CMP R1, R2
```

3.3.2 数据处理指令：比较（含测试）

指令格式	说明	操作
CMP{cond} Rn, opd2	比较	标志N/Z/C/V ← Rn - opd2
CMN{cond} Rn, opd2	负数比较	标志N/Z/C/V ← Rn + opd2
TST{cond} Rn, opd2	位测试	标志N/Z/C ← Rn & opd2
TEQ{cond} Rn, opd2	相等测试	标志N/Z/C ← Rn ^ opd2

- CMN指令：N/Z/C/V ← Rn + operand2
  - ✧ 更新N/Z/C/V，以便后面的指令根据相应的条件标志判断执行
  - ✧ 注意：ADDS也可用于此比较，但CMN不需要用Rd保存运算结果

```
CMN    R1, R0      ; 计算R1+R0，根据结果设置CPSR标志位
CMN    R1, #100    ; 计算R1+100，根据结果设置CPSR标志位
CMN    R0, #1      ; 计算R0+1，可判断R0是否为1的补码
                   ; 若是，设置Z标志位(即R0+1=0)
```

3.3.2 数据处理指令：比较（含测试）

指令格式	说明	操作
CMP{cond} Rn, opd2	比较	标志N/Z/C/V ← Rn - opd2
CMN{cond} Rn, opd2	负数比较	标志N/Z/C/V ← Rn + opd2
TST{cond} Rn, opd2	位测试	标志N/Z/C ← Rn & opd2
TEQ{cond} Rn, opd2	相等测试	标志N/Z/C ← Rn ^ opd2

- TST指令：N/Z/C ← Rn & operand2
  - ✧ 更新N/Z/C，以便后面的指令根据相应的条件标志判断执行
  - ✧ 注意：ANDS也可用于测试，但TST不需要用Rd保存运算结果
  - ✧ 常与EQ/NE条件码配合使用
    - 测试所有位均为零，则EQ有效（Z=1），否则NE有效（Z=0）

```
TST    R1, #1          ; 测试R1中的最低位是否为1，由结果设置Z标志
TST    R1, #0xFFE      ; N/Z ← R1 & 0xFFE
```

3.3.2 数据处理指令：比较（含测试）

指令格式	说明	操作
CMP{cond} Rn, opd2	比较	标志N/Z/C/V ← Rn - opd2
CMN{cond} Rn, opd2	负数比较	标志N/Z/C/V ← Rn + opd2
TST{cond} Rn, opd2	位测试	标志N/Z/C ← Rn & opd2
TEQ{cond} Rn, opd2	相等测试	标志N/Z/C ← Rn ^ opd2

- TEQ指令：N/Z/C ← Rn ^ operand2
  - ✧ 更新N/Z/C，以便后面的指令根据相应的条件标志判断执行
  - ✧ 注意：EORS也可用于测试，但TEQ不需要用Rd保存运算结果
  - ✧ 常与EQ/NE条件码配合使用
    - 两个数据相等时，EQ有效（Z=1），否则NE有效(Z=0)

```
TEQ    R0, R1          ; 比较R0与R1是否相等(不影响V位和C位)
```

3.3.2 数据处理指令：乘法

31:28	27	26	25	24	23	22	21	20	19:16	15:12	11:8	7	6	5	4	3:0	
cond	0	0	0	0	0	0	A	S	Rd	Rn	Rs	1	0	0	1	Rm	乘法(乘加)
cond	0	0	0	0	1	U	A	S	RdHi	RdLo	Rs	1	0	0	1	Rm	长乘(乘加)

- bit[23:21]：乘法指令操作码
  - A表示是否有“加”
  - U表示是否有“符号”
- 结果寄存器：Rd/RdHi/RdLo
- 操作数寄存器：Rn/Rs/Rm
- 注意：长乘表示结果为64位乘法
  - 有高/低位结果寄存器
  - 仅两个操作数

opcode	指令助记符	说明
000	MUL	32位乘法
001	MLA	32位乘加
100	UMULL	64位无符号乘法
101	UMLAL	64位无符号乘加
110	SMULL	64位有符号乘法
111	SMLAL	64位有符号乘加

3.3.2 数据处理指令：乘法

- 乘法指令用于完成乘法和乘加运算
  - ✧ 可以更新CPSR中的条件标志位
  - ✧ 与前面数据处理指令不同，乘法指令中所有操作数、目标寄存器必须为通用寄存器，不能对操作数使用立即数或者被移位寄存器
  - ✧ 目标寄存器不能同时作为第1个操作数寄存器
    - Rd、RdLo、RdHi不能与Rm同寄存器，RdLo和RdHi也不能同寄存器
  - ✧ 有6条数乘法指令（2+4）

指令格式	说明	操作
MUL{cond}{S} Rd,Rm,Rs	32位乘法	Rd ← Rm×Rs
MLA{cond}{S} Rd,Rm,Rs,Rn	32位乘加	Rd ← Rm×Rs+Rn
UMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘法	(RdLo,RdHi) ← Rm×Rs
UMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘加	(RdLo,RdHi) += Rm×Rs
SMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘法	(RdLo,RdHi) ← Rm×Rs
SMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘加	(RdLo,RdHi) += Rm×Rs

3.3.2 数据处理指令：乘法

指令格式	说明	操作
MUL{cond}{S} Rd,Rm,Rs	32位乘法	Rd ← Rm×Rs
MLA{cond}{S} Rd,Rm,Rs,Rn	32位乘加	Rd ← Rm×Rs+Rn
UMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘法	(RdLo,RdHi)← Rm×Rs
UMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘加	(RdLo,RdHi)+= Rm×Rs
SMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘法	(RdLo,RdHi)← Rm×Rs
SMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘加	(RdLo,RdHi)+= Rm×Rs

- MUL指令：Rd ← Rm × Rs
  - Rm和Rs均为32位（有符号/无符号均可），结果的低32位存入Rd
  - Rd,Rm,Rs不能为R15
  - 结果只影响CPSR标志N,Z，不影响V,C

```
MUL    R1, R2, R3   ; R1 ← R2×R3
MULS   R1, R2, R3   ; R1 ← R2×R3, 影响CPSR中的N位和Z位
```

3.3.2 数据处理指令：乘法

指令格式	说明	操作
MUL{cond}{S} Rd,Rm,Rs	32位乘法	Rd ← Rm×Rs
MLA{cond}{S} Rd,Rm,Rs,Rn	32位乘加	Rd ← Rm×Rs+Rn
UMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘法	(RdLo,RdHi) ← Rm×Rs
UMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘加	(RdLo,RdHi) += Rm×Rs
SMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘法	(RdLo,RdHi) ← Rm×Rs
SMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘加	(RdLo,RdHi) += Rm×Rs

```
MUL    R1, R2, R3   ; R1 ← R2×R3
MULS   R1, R2, R3   ; R1 ← R2×R3, 影响CPSR中的N位和Z位
```

- 若：R2=0x00FFFFFF，R3=0x00123456
- 则：R2×R3=0x123455EDCBAA，其中低32位存入R1
- 即：R1=0x55EDCBAA

3.3.2 数据处理指令：乘法

指令格式	说明	操作
MUL{cond}{S} Rd,Rm,Rs	32位乘法	Rd ← Rm×Rs
MLA{cond}{S} Rd,Rm,Rs,Rn	32位乘加	Rd ← Rm×Rs+Rn
UMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘法	(RdLo,RdHi)← Rm×Rs
UMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘加	(RdLo,RdHi)+= Rm×Rs
SMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘法	(RdLo,RdHi)← Rm×Rs
SMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘加	(RdLo,RdHi)+= Rm×Rs

- MLA指令：Rd ← Rm × Rs + Rn
  - Rm和Rs均为32位（有符号/无符号均可），结果的低32位存入Rd
  - Rd,Rm,Rs,Rn不能为R15
  - 结果只影响CPSR标志N,Z，不影响V，而C不确定

```
MLA    R0, R1, R2, R3   ; R0 ← R1×R2+R3
MLAS   R0, R1, R2, R3   ; R0 ← R1×R2+R3, 影响CPSR中标志位
```

3.3.2 数据处理指令：乘法

指令格式	说明	操作
MUL{cond}{S} Rd,Rm,Rs	32位乘法	Rd ← Rm×Rs
MLA{cond}{S} Rd,Rm,Rs,Rn	32位乘加	Rd ← Rm×Rs+Rn
UMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘法	(RdLo,RdHi)← Rm×Rs
UMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘加	(RdLo,RdHi)+= Rm×Rs
SMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘法	(RdLo,RdHi)← Rm×Rs
SMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘加	(RdLo,RdHi)+= Rm×Rs

- UMULL指令：(RdLo,RdHi) ← Rm × Rs
  - Rm和Rs均为32位（无符号），结果低32位存RdLo，高32位存RdHi
  - RdLo/RdHi,Rm,Rs不能为R15
  - 结果只影响CPSR标志N,Z，标志V,C不确定

```
UMULL R0, R1, R2, R3      ; R0 ← (R2×R3)的低32位
                          ; R1 ← (R2×R3)的高32位
```

3.3.2 数据处理指令：乘法

指令格式	说明	操作
MUL{cond}{S} Rd,Rm,Rs	32位乘法	Rd ← Rm×Rs
MLA{cond}{S} Rd,Rm,Rs,Rn	32位乘加	Rd ← Rm×Rs+Rn
UMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘法	(RdLo,RdHi) ← Rm×Rs
UMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘加	(RdLo,RdHi) += Rm×Rs
SMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘法	(RdLo,RdHi) ← Rm×Rs
SMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘加	(RdLo,RdHi) += Rm×Rs

```
UMULL R0, R1, R2, R3      ; R0 ← (R2×R3)的低32位
                          ; R1 ← (R2×R3)的高32位
```

- 若：R2=0x00FFFFFF，R3=0x00123456
- 则：R2×R3=0x123455EDCBAA
- 即：R0=0x55EDCBAA，R1=0x00001234

3.3.2 数据处理指令：乘法

指令格式	说明	操作
MUL{cond}{S} Rd,Rm,Rs	32位乘法	Rd ← Rm×Rs
MLA{cond}{S} Rd,Rm,Rs,Rn	32位乘加	Rd ← Rm×Rs+Rn
UMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘法	(RdLo,RdHi)← Rm×Rs
UMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘加	(RdLo,RdHi)+= Rm×Rs
SMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘法	(RdLo,RdHi)← Rm×Rs
SMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘加	(RdLo,RdHi)+= Rm×Rs

- UMLAL指令：(RdLo,RdHi) += Rm × Rs
  - Rm和Rs均为32位（无符号），结果低32位存RdLo，高32位存RdHi
  - RdLo/RdHi,Rm,Rs不能为R15
  - 结果只影响CPSR标志N,Z，标志V,C不确定

```
UMLAL R0, R1, R2, R3      ; R0 ← (R2×R3)的低32位+R0
                          ; R1 ← (R2×R3)的高32位+R1
```

3.3.2 数据处理指令：乘法

指令格式	说明	操作
MUL{cond}{S} Rd,Rm,Rs	32位乘法	Rd ← Rm×Rs
MLA{cond}{S} Rd,Rm,Rs,Rn	32位乘加	Rd ← Rm×Rs+Rn
UMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘法	(RdLo,RdHi) ← Rm×Rs
UMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘加	(RdLo,RdHi) += Rm×Rs
SMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘法	(RdLo,RdHi) ← Rm×Rs
SMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘加	(RdLo,RdHi) += Rm×Rs

- SMULL指令：(RdLo,RdHi) ← Rm × Rs
  - Rm和Rs均为32位（有符号），结果低32位存RdLo，高32位存RdHi
  - RdLo/RdHi,Rm,Rs不能为R15
  - 结果只影响CPSR标志N,Z，标志V,C不确定

```
SMULL R0, R1, R2, R3      ; R0 ← (R2×R3)的低32位
                          ; R1 ← (R2×R3)的高32位
```

3.3.2 数据处理指令：乘法

指令格式	说明	操作
MUL{cond}{S} Rd,Rm,Rs	32位乘法	Rd ← Rm×Rs
MLA{cond}{S} Rd,Rm,Rs,Rn	32位乘加	Rd ← Rm×Rs+Rn
UMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘法	(RdLo,RdHi)← Rm×Rs
UMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位无符号乘加	(RdLo,RdHi)+= Rm×Rs
SMULL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘法	(RdLo,RdHi)← Rm×Rs
SMLAL{cond}{S} RdLo,RdHi,Rm,Rs	64位有符号乘加	(RdLo,RdHi)+= Rm×Rs

- SMLAL指令：(RdLo,RdHi) += Rm × Rs
  - Rm和Rs均为32位（有符号），结果低32位存RdLo，高32位存RdHi
  - RdLo/RdHi,Rm,Rs不能为R15
  - 结果只影响CPSR标志N,Z，标志V,C不确定

```
SMLAL R0, R1, R2, R3      ; R0 ← (R2×R3)的低32位+R0
                          ; R1 ← (R2×R3)的高32位+R1
```

3.3.3 程序状态寄存器指令

- 用于在程序状态寄存器和通用寄存器之间传送数据
  - ✧ MRS：程序状态寄存器 → 通用寄存器（读PSR）
    - MRS指令可以对状态寄存器CPSR和SPSR进行读操作
    - 读CPSR寄存器可以了解当前处理器的工作状态
    - 读SPSR寄存器可以了解进入异常前的处理器状态
    - 该指令不影响条件码
    - 注意：用户/系统模式下，不能操作SPSR
  - ✧ MSR：程序状态寄存器 ← 通用寄存器（写PSR）
    - MSR指令可以对状态寄存器CPSR和SPSR进行写操作
    - 与MRS配合使用，可实现对CPSR/SPSR寄存器的读-修改-写操作
    - 可以切换处理器模式、或者允许/禁止IRQ/FIQ中断等
    - 注意：用户模式不能修改CPSR；用户/系统模式下，不能操作SPSR
      不能用MSR指令直接修改CPSR中的T控制位切换到Thumb状态

3.3.3 程序状态寄存器指令

- 用于在程序状态寄存器和通用寄存器之间传送数据
  - ✧ MRS：程序状态寄存器 → 通用寄存器（读PSR）
  - ✧ MSR：程序状态寄存器 ← 通用寄存器（写PSR）

31:28	27:24	23	22	21	20	19:16	15:12	11:0
cond	0	0	R	0	0	1	Rd	0

条件码

R=1，SPSR
R=0，CPSR

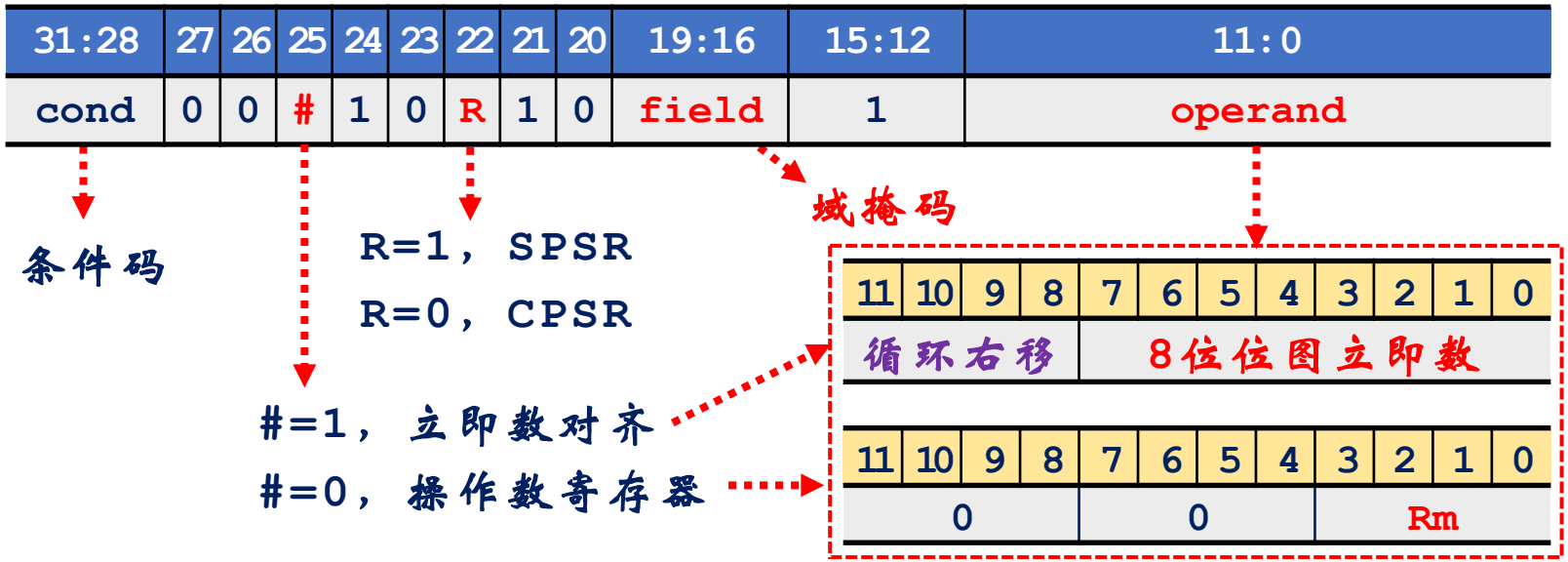
目标寄存器
不能为R15

指令格式	说明	操作
MRS{cond}　Rd，PSR	读PSR寄存器	Rd ← CPSR/SPSR

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

3.3.3 程序状态寄存器指令

- 用于在程序状态寄存器和通用寄存器之间传送数据
  - ✧ MRS：程序状态寄存器 → 通用寄存器（读PSR）
  - ✧ MSR：程序状态寄存器 ← 通用寄存器（写PSR）



指令格式	说明
MSR{cond}　PSR_field, #immed_8r	用有效立即数修改PSR的指定域
MSR{cond}　PSR_field, Rm	用源寄存器值修改PSR的指定域

3.3.3 程序状态寄存器指令

- 用于在程序状态寄存器和通用寄存器之间传送数据
  - MRS：程序状态寄存器 → 通用寄存器（读PSR）
  - MSR：程序状态寄存器 ← 通用寄存器（写PSR）

31:28	27	26	25	24	23	22	21	20	19:16	15:12	11:0
cond	0	0	#	1	0	R	1	0	field	1	operand

指令格式	说明
MSR{cond}　PSR_field, #immed_8r	用有效立即数修改PSR的指定域
MSR{cond}　PSR_field, Rm	用源寄存器值修改PSR的指定域

指定传送的区域field，可为以下小写字母的一个或者组合		
c	PSR[7:0]	控制域屏蔽字节
x	PSR[15:8]	扩展域屏蔽字节
s	PSR[23:16]	状态域屏蔽字节
f	PSR[31:24]	标志域屏蔽字节

3.3.3 程序状态寄存器指令

指令格式	说明
MSR{cond}　PSR_field, #immed_8r	用有效立即数修改PSR的指定域
MSR{cond}　PSR_field, Rm	用源寄存器值修改PSR的指定域

指定传送的区域field，可为以下小写字母的一个或者组合

c	PSR[7:0]	控制域屏蔽字节
x	PSR[15:8]	扩展域屏蔽字节
s	PSR[23:16]	状态域屏蔽字节
f	PSR[31:24]	标志域屏蔽字节

```
MRS    R0, CPSR          ; 读取CPSR到R0
BIC    R0, R0, #0x1F     ; 低5位清零,清除当前模式
ORR    R0, R0, #0x13     ; 设置位SVC模式(对照表2.4)
MRS    CPSR_c, R0        ; 将修改值写回CPSR
                         ; 上述过程可实现运行模式切换
```

3.3.4 加载和存储指令（存储器访问指令）

- ARM处理器是典型的RISC处理器，对存储器的访问只能使用加载/存储指令实现
  - ✧ 数据操作过程：
    - 存储器 → 片内寄存器（加载后处理，Load）
    - 寄存器 → 存储器（处理后回存，Store）
- ARM系统中，程序空间、RAM空间、I/O映射空间统一编址
  - ✧ 除对RAM操作以外，对外围IO、程序数据、外设（含外设寄存器）的访问均要通过加载/存储指令进行
- 三种基本的存储器访问指令
  - ✧ 单寄存器存取指令（LDR，STR）
  - ✧ 多寄存器存取指令（LDM，STM）
  - ✧ 单寄存器交换指令（SWP，SWPB）

3.3.4 加载和存储指令：单寄存器存取

31:28	27	26	25	24	23	22	21	20	19:16	15:12	11:0
cond	0	1	I	P	U	B	W	L	Rn	Rd	Offset (12位)

- 单寄存器存取指令是ARM在寄存器和存储器之间传送单个字和字节的最灵活方式
  - ✧ 支持立即数、寄存器(偏移)、自动变址、相对PC的寻址
  - ✧ LDR/STR指令用于对内存变量的访问、内存缓冲区数据的访问、查表、外围部件的控制操作等
  - ✧ 若使用LDR指令加载数据到PC寄存器，则实现程序跳转功能
  - ✧ 所有单寄存器加载/存储指令可分为
    - 字和无符号字节加载/存储指令
    - 半字和有符号字节加载/存储指令
  - ✧ 存储指令STR中，“无符号”和“有符号”无区别

3.3.4 加载和存储指令：单寄存器存取

■ 单寄存器存取指令（LDR，STR）

指令格式	说明	操作
LDR{cond} Rd, [<地址>]	加载字数据	Rd ← [<地址>] 从存储器加载数据至寄存器
LDR{cond}B Rd, [<地址>]	加载无符号字节数据	
LDR{cond}T Rd, [<地址>]	以用户模式加载字数据	
LDR{cond}BT Rd, [<地址>]	以用户模式加载无符号字节数据	
LDR{cond}H Rd, [<地址>]	加载无符号半字数据	
LDR{cond}SB Rd, [<地址>]	加载有符号字节数据	
LDR{cond}SH Rd, [<地址>]	加载有符号半字数据	
STR{cond} Rd, [<地址>]	存储字数据	[<地址>] ← Rd 从寄存器存储数据到存储器
STR{cond}B Rd, [<地址>]	存储字节数据	
STR{cond}T Rd, [<地址>]	以用户模式存储字数据	
STR{cond}BT Rd, [<地址>]	以用户模式存储字节数据	
STR{cond}H Rd, [<地址>]	存储半字数据	

3.3.4 加载和存储指令：单寄存器存取

- **字和无符号字节加载/存储指令**
  - ✧ LDR：内存中单个字/字节 → 寄存器
  - ✧ STR：寄存器中单个字/字节 → 内存

```
LDR{cond}{T}     Rd, [<地址>]  ; 将指定地址上的字数据读入Rd
STR{cond}{T}     Rd, [<地址>]  ; 将Rd中的字数据存入指定地址
LDR{cond}B{T}    Rd, [<地址>]  ; 将指定地址上的字节数据读入Rd
STR{cond}B{T}    Rd, [<地址>]  ; 将Rd中的字节数据存入指定地址
```

Rd　目标寄存器　　[<地址>]　4种格式

  - ✧ “T”为可选后缀
    - 若指令有T，存储系统总是将访问看成是在用户模式下进行的
    - T在用户模式下无效，不能与前索引偏移一起使用T
  - ✧ 字节传送时，用0将8位操作数扩充到32位

3.3.4 加载和存储指令：单寄存器存取

- **字和无符号字节加载/存储指令**
  - ✧ LDR/STR指令有3种寻址格式：
    - ① 立即数（可以是12位无符号数）
    - ② 寄存器
    - ③ 寄存器+移位常数（移位常数只能是5位立即数）

```
LDR    R1, [R0, #0x12]          ; 格式1: R1 ← [R0+0x12]
LDR    R1, [R0, R2]             ; 格式2: R1 ← [R0+R2]
LDR    R1, [R0, -R2]            ; 格式2: R1 ← [R0-R2]
LDR    R1, [R0, R2, LSL #2]     ; 格式3: R1 ← [R0+R2<<2]
```

3.3.4 加载和存储指令：单寄存器存取

- **字和无符号字节加载/存储指令**
  - ✧ LDR/STR指令从寻址方式的地址计算有以下4种格式：
    1. 零偏移
    2. 前索引偏移
    3. 后索引偏移
    4. 程序相对偏移

```
LDR    Rd, [Rn]              ; 格式1: Rd ← [Rn]
LDR    Rd, [Rn, #0x04]!      ; 格式2: Rd ← [R0+0x04], 更新Rn
LDR    Rd, [Rn], #-0x04      ; 格式4: Rd ← [Rn], 更新Rn
LDR    Rd, LABEL             ; 格式3: Rd ← LABEL
```

注意：必须保证字数据操作的地址32位对齐

3.3.4 加载和存储指令：单寄存器存取

- **字和无符号字节加载/存储指令**
  - ✧ **指令举例：**

```
LDR    R0, [R1]    ; 寄存器R0 ← 存储器地址为R1的"字数据"
LDRB   R0, [R1]    ; 寄存器R0 ← 存储器地址为R1的"字节数据"
                   ; R0的高24位清零
STRB   R0, [R1]    ; 寄存器R0中的"字节数据" → 地址为R1的存储器
LDR    R0, [R1,R2]! ; R0 ← [R1+R2], R1 ← R1+R2
LDR    R0, [R1], R2 ; R0 ← [R1], R1 ← R1+R2
LDR    R0, LocalData; R0 ← 位于标号LocalData所在地址的"字"
```

注意：避免把PC值存到存储器，也要避免将1个字节读取到PC（字可以）

3.3.4 加载和存储指令：单寄存器存取

- **半字和有符号字节加载/存储指令**
  - ✧ 偏移量格式、寻址方式与加载/存储字和无符号字节指令相同

```
LDR{cond}SB    Rd, [Rn,Rm]  ; 将指定地址上的有符号字节读入Rd
LDR{cond}SH    Rd, [Rn,Rm]  ; 将指定地址上的有符号半字读入Rd
LDR{cond}H     Rd, [Rn,Rm]  ; 将指定地址上的半字数据读入Rd
STR{cond}H     Rd, [Rn,Rm]  ; 将Rd中的半字数据存入指定地址
```

注意：

① 有符号位半字/字节加载是指用符号位加载扩展到32位
无符号半字加载是指用0扩展到32位（有区别）

② 半字读写的指令地址必须为偶数，否则将产生不可靠的结果

③ 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（如Rm<<）

3.3.4 加载和存储指令：单寄存器存取

- **半字和有符号字节加载/存储指令**
  - ✧ **指令举例：**

```
LDRSH R0, [R1]      ; 寄存器R0 ← 存储器地址为R1的"有符号半字"
                    ; R0的高16位用符号位扩充
STRH  R0, [R1]      ; 寄存器R0中的"半字节" → 地址为R1的存储器
LDRH  R0, [R1,#8]   ; R0 ← [R1+8]的半字，R0的高16位清零
LDRH  R0, [R1],#2   ; R0 ← [R1]的半字
                    ; R0的高16位清零，R1 ← R1+2
LDRSB R0, Const     ; R0 ← 位于标号Const所在地址"有符号字节"
                    ; R0的高24位用符号位扩充
LDRSB R1, [R6],R3,LSL #4; 错误格式，不能经寄存器移位！
```

3.3.4 加载和存储指令：单寄存器存取

- **双字加载/存储指令**
  - ✧ 两个相邻的寄存器，64位双字
  - ✧ **4种格式：**

```
LDR|STR{cond}D    Rd, [Rn]                ; 零偏移
LDR|STR{cond}D    Rd, [Rn, offset]{!}     ; 前索引偏移
LDR|STR{cond}D    Rd, [Rn], offset        ; 后索引偏移
LDR|STR{cond}D    Rd, LABEL               ; 程序相对偏移
```

注意：

① Rd,R(d+1)为相邻两个加载/存储寄存器，Rd须为偶数寄存器，且非R14

② Rn不能与Rd和R(d+1)相同，除非指令为零偏移，或无"!"的前索引偏移

③ LABEL必须在当前指令的±252字节范围内

3.3.4 加载和存储指令：单寄存器存取

- **双字加载/存储指令**
  - ✧ **指令举例：**

```
LDRD   R6, [R11]     ; R6 ← [R11], R7 ← [R11+4]
STRD   R4, [R9,#24]  ; R4 → [R9+24], R5 → [R9+28]
LDRD   R8, abc4      ; R8 ← 位于标号abc4所在地址的"字"
                     ; R8 ← 位于标号abc4所在地址+4的"字"

LDRD   R1, [R6]      ; 错误使用，Rd必须为偶数寄存器!
LDRD   R14, [R9]     ; 错误使用，Rd不能为R14!
LDRD   R2, [R3],R6   ; 错误使用，Rn不能为Rd或R(d+1)!
```

3.3.4 加载和存储指令：多寄存器存取

31:28	27	26	25	24	23	22	21	20	19:16	15:0
cond	1	0	0	P	U	S	W	L	Rn	register list (16位)

- 多寄存器加载/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
  - ✧ 允许一条指令传送最多16个寄存器
  - ✧ 主要用于现场保护、数据复制、常数传递等

指令格式	说明
LDM{cond} <add_mode> Rn{!}, <reg_list>	多寄存器加载
STM{cond} <addr_mode> Rn{!}, <reg_list>	多寄存器存储

3.3.4 加载和存储指令：多寄存器存取

指令格式	说明
LDM{cond} <addr_mode> Rn{!}, <reg_list>	多寄存器加载
STM{cond} <addr_mode> Rn{!}, <reg_list>	多寄存器存储

- cond　　条件码
- add_mode　　控制地址增长方式，共8种模式（4种*2重）
- Rn　　基址寄存器，不能是R15
- !　　操作结束后，将最后地址写回Rn中
- reg_list　　寄存器列表，多个寄存器用”,”隔开

  如{R1,R2,R6-R9}，寄存器由小到大排列（约定）
- ^　　可选后缀，不允许在用户模式/系统模式下使用
  在LDM指令且寄存器列表包含PC时
  还将SPSR复制到CPSR

3.3.4 加载和存储指令：多寄存器存取

指令格式	说明
LDM{cond} <addr_mode> Rn{!}, <reg_list>	多寄存器加载
STM{cond} <addr_mode> Rn{!}, <reg_list>	多寄存器存储

Addr_mode	说明	Addr_mode	说明
IA	每次传送后地址加4	EA	空递增堆栈
IB	每次传送前地址加4	FA	满递增堆栈
DA	每次传送后地址减4	ED	空递减堆栈
DB	每次传送前地址减4	FD	满递减堆栈
数据块传送操作		堆栈操作	

① 进行数据复制时，先设置源数据指针和目标指针，再使用块复制寻址指令LDMIA/IB/DA/DB、STMIA/IB/DA/DB进行读取和存储

② 进行堆栈操作操作时，先设置堆栈指针（SP），再使用堆栈寻址指令LDMEA/FA/ED/FD、STMEA/FA/ED/FD实现堆栈操作

3.3.4 加载和存储指令：单寄存器交换

- 交换指令把字或无符号字节的读和写组合在一条指令中
  - ✧ 成为一个不能被外部存储器访问分隔开的基本存储操作
  - ✧ 一般用于共享的信号量、数据结构之间进行互斥的访问

指令格式	说明	操作
SWP{cond} Rd, Rm, [Rn]	寄存器和存储器字数据交换	Rd ← [Rn] Rm → [Rn] (Rn ≠ Rd或Rm)
SWPB{cond} Rd, Rm, [Rn]	寄存器和存储器字节数据交换	

- Rd　用于保存从存储器中读入的数据
- Rm　Rm的数据用于存储到存储器中
  若Rm与Rd相同，则为寄存器与存储器内容进行互换
- Rn　要进行数据交换的存储器地址
  Rn不能与Rd和Rm相同

3.3.4 加载和存储指令：单寄存器交换

- 交换指令把字或无符号字节的读和写组合在一条指令中
  - ✧ 指令举例：

```
SWP    R0, R1, [R2]     ; R0 ← [R2]字, R1 → [R2]
SWPB   R0, R0, [R2]     ; R2指向存储单元中的字节数据与R0中的
                        ; 低8位数据交换
SWPB   R0, R1, [R2]     ; R0 ← [R2]字节, R0高24位清零
                        ; R1中低8位数据传送到R2指向的存储单元
```

3.3.5* 协处理指令

- ARM处理器最常使用的协处理器是用于控制片上功能的系统协处理器
  - ✧ 例如：控制ARM720上的高速缓存和存储管理单元等
  - ✧ ARM处理器可支持多达16个协处理器，用于各种协处理操作
    - 每个协处理器只执行针对自身的协处理指令，忽略其他来源的指令
- ARM协处理指令主要作用：
  - ① ARM处理器初始化、
  - ② ARM协处理器的数据处理操作
  - ③ ARM处理器的寄存器和协处理器的寄存器之间传送数据
  - ④ ARM协处理器的寄存器和存储器之间传送数据
- 5条协处理指令

3.3.5* 协处理指令

- 5条协处理指令
  - CDP　　协处理器数据操作
  - LDC　　协处理器数据加载
  - STC　　协处理器数据存储
  - MCR　　ARM处理器寄存器到协处理器寄存器的数据传送
  - MRC　　协处理器寄存器到ARM处理器寄存器的数据传送

3.3.6 异常产生指令

- 异常产生指令主要指软件中断指令SWI
  - 用于产生SWI异常，使CPU模式变换到管理模式，并且将CPSR保存到管理模式的SPSR中，然后程序跳转到SWI异常入口
    - 不影响条件码标志
  - 用于用户程序调用操作系统的系统服务，操作系统在SWI异常处理程序中进行相应的系统服务

31:28	27	26	25	24	23:0
cond	1	1	1	1	swi number (immed_24)

指令格式	说明
SWI{cond} <immed_24>	产生软件中断，处理器进入管理模式 指令传递参数为24位立即数，范围0~$2^{24}-1$

3.3.6 异常产生指令

- 根据SWI指令传递的参数，SWI异常处理程序作出相应处理
- SWI指令传递参数的2种方法：
  - 指令中的immed_24指定用户请求的服务类型，参数通过通用寄存器传递

```
MOV    R0, #34        ; 设置子功能号为34，传递的参数
SWI    12             ; 调用12号软中断
```

  - 指令中的immed_24被忽略，用户请求的服务类型由寄存器R0的值决定，参数通过其他通用寄存器传递

```
MOV    R0, #12        ; 调用12号软中断
MOV    R1, #34        ; 设置子功能号为34，传递的参数
SWI    0              ; 输出传递的参数/SWI_WriteC
```

- 在SWI异常中断处理程序中，取出SWI指令中立即数的步骤：
  - ① 确定引起软中断的SWI指令是ARM指令还是Thumb指令
    通过访问SPSR得到
  - ② 获取该SWI指令的地址，通过访问LR寄存器得到
  - ③ 读出该SWI指令，分解出立即数

NE：Z=0
EQ：Z=1

```
SWI_Handler
    STMFD   SP!, {R0-R3, R12, LR}       ; 现场保护
    MRS     R0, SPSR                    ; 读取SPSR
①   STMFD   SP!, {R0}                   ; 保存SPSR
    TST     R0, #0x20                   ; 测试T标志位
②   LDRNEH  R0, [LR, #-2]               ; 若是Thumb指令，读取指令码(16位)
③   BICNE   R0, R0, #0xFF00             ; 取得Thumb指令的8位立即数（低8位）
    LDREQ   R0, [LR, #-4]               ; 若是ARM指令，读取指令码(32位)
    BICEQ   R0, R0, #0xFF000000         ; 取得ARM指令的24位立即数（低23位）
        ...
    LDMFD   SP!, {R0-R3, R12, PC}^      ; SWI异常中断返回
```

- 在SWI异常中断处理程序中，取出SWI指令中立即数的步骤：
  - ① 确定引起软中断的SWI指令是ARM指令还是Thumb指令
    通过访问SPSR得到
  - ② 获取该SWI指令的地址，通过访问LR寄存器得到
  - ③ 读出该SWI指令，分解出立即数

NE：Z=0
EQ：Z=1

```
SWI_Handler
    STMFD   SP!, {R0-R3, R12, LR}       ; 现场保护
    MRS     R0, SPSR                    ; 读取SPSR
①   STMFD   SP!, {R0}                   ; 保存SPSR
    TST     R0, #0x20                   ; 测试T标志位
    LDRNEH  R0, [LR, #-2]               ; 若是Thumb指令，读取指令码(16位)
    BICNE   R0, R0, #0xFF00             ; 取得Thumb指令的8位立即数（低8位）
②   LDREQ   R0, [LR, #-4]               ; 若是ARM指令，读取指令码(32位)
③   BICEQ   R0, R0, #0xFF000000         ; 取得ARM指令的24位立即数（低23位）
        ...
    LDMFD   SP!, {R0-R3, R12, PC}^      ; SWI异常中断返回
```

章节内容

- ARM指令集概述
- ARM指令寻址方式
- ARM指令
- Thumb指令*

3.4 *Thumb指令

- ARM体系结构除了支持执行效率很高的32位ARM指令集以外，同时支持16位的Thumb指令集。Thumb指令集是ARM指令集的一个子集，允许指令编码为16位的长度。与等价的32位代码相比较，Thumb指令集在保留32代码优势的同时，大大节省了系统的存储空间。
- 所有的Thumb指令都有对应的ARM指令，而且Thumb的编程模型也对应于ARM的编程模型，在应用程序的编写过程中，只要遵循一定调用的规则，Thumb子程序和ARM子程序就可以互相调用。当处理器在执行ARM程序段时，称ARM处理器处于ARM工作状态，当处理器在执行Thumb程序段时，称ARM处理器处于Thumb工作状态。
- 与ARM指令集相比较，Thumb指令集中的数据处理指令的操作数仍然是32位，指令地址也为32位，但Thumb指令集为实现16位的指令长度，舍弃了ARM指令集的一些特性，如大多数的Thumb指令是无条件执行的，而几乎所有的ARM指令都是有条件执行的；大多数的Thumb数据处理指令的目的寄存器与其中一个源寄存器相同。

3.4 *Thumb指令

- Thumb指令与ARM指令的时间效率和空间效率关系：
  - Thumb代码所需的存储空间约为ARM代码的60%~70%
  - Thumb代码使用的指令数比ARM代码多约30%~40%
  - 若使用32位数据宽度的存储器，ARM代码比Thumb代码快约40%
  - 若使用16位数据宽度的存储器，Thumb代码比ARM代码快约40%~50%
  - 与ARM代码相比较，使用Thumb代码，存储器的功耗会降低约30%
- ARM指令集和Thumb指令集各有其优点：
  - 若对系统性能要求较高，应使用32位的存储系统和ARM指令集。
  - 若对系统的成本及功耗要求较高，则应使用16位的存储系统和Thumb指令集。
  - 若两者结合使用，充分发挥其各自的优点，会取得更好的效果。

本章小结

ARM指令的分类、格式、条件码、编码

ARM指令的9种寻址方式

✧ 立即数、寄存器、变址、堆栈、块复制、相对

ARM指令的6类指令

✧ 跳转、数据处理、状态寄存器
加载/存储、协处理器、异常产生