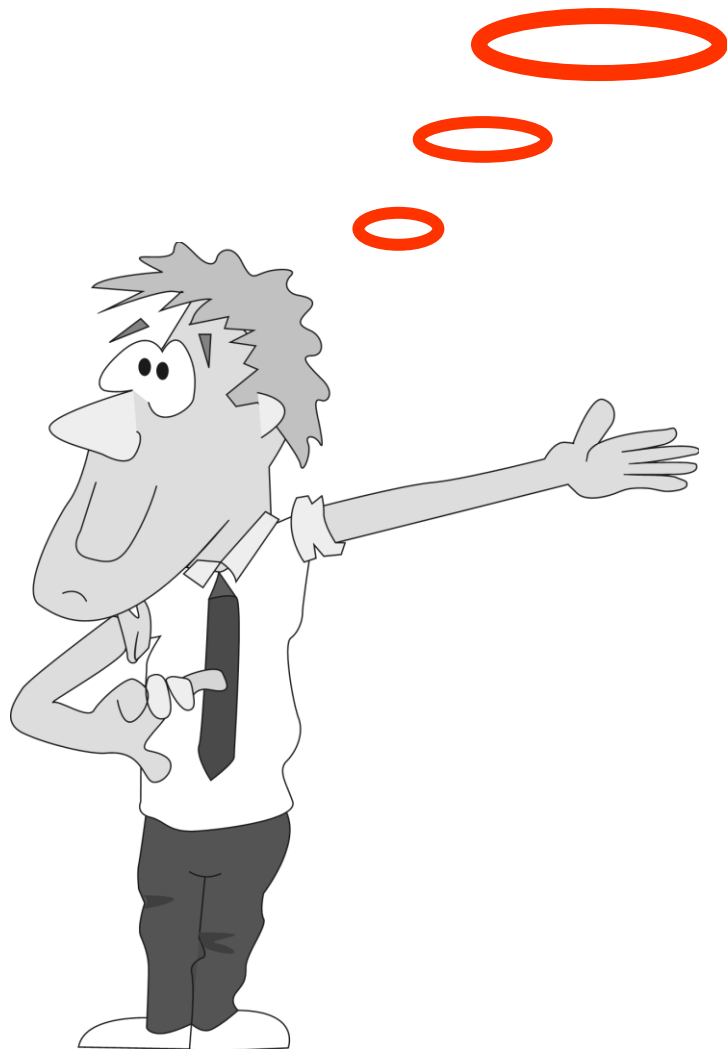


# 第3章 ARM 9指令系统

- ARM处理器的寻址方式
- ARM指令集
- Thumb指令集



# 3.1 ARM处理器的寻址方式

- **寻址方式**：就是根据指令中操作数的信息寻找操作数实际物理地址的方式。
- 依据指令中给出的操作数的不同格式，ARM指令系统具有**8种**常见的寻址方式：
  - ✓ 寄存器寻址
  - ✓ 立即寻址
  - ✓ 寄存器间接寻址
  - ✓ 变址寻址
  - ✓ 寄存器移位寻址
  - ✓ 多寄存器寻址
  - ✓ 堆栈寻址
  - ✓ 相对寻址

## 3.1.1 寄存器寻址

- 在寄存器寻址方式下，寄存器的值即为操作数。
- ARM指令普遍采用此种寻址方式。

• 例：

**ADD    R0, R1, R2            ; R0=R1+R2**

**MOV    R0, R1                ; R0=R1**

## 3.1.2 立即寻址

- 在立即数寻址中，操作数本身直接在指令中给出，取出指令也就获得了操作数，这个操作数也称为立即数。
- #后的立即数形式：
  - 0x或&表示十六进制数
  - 0b表示二进制数
  - 0d或缺省表示十进制数
- 例：
  - ADD R0, R1, #5 ; R0=R1+5**
  - MOV R0, #0x55 ; R0=0x55**
- 其中：操作数5，0x55就是立即数，立即数在指令中要以“#”为前缀，后面跟实际数值。

# 立即数的形成

- 合理常数：一个32位数用12位编码表示，符合以下规则才是合法常数。  
常数=immed\_8循环右移( $2 \times \text{rotate\_imm}$ )

例如：

- 汇编指令 `mov r0,#0x0000f200`
- 机器指令 `0xe3a0cf2`，其中`0xcf2`为立即数。
- `Immed_8=0xf2`     `rotate_imm=0x0c`
- 常数`0xf2`（循环右移动24位）  
`0x0000f200= 0xf2`循环右移 ( $2 \times 0x0c$ )

11-8	7-0
循环部分	立即数部分
Rotate_imm	Immed_8

31-28	27-25	24-21	20	19-16	15-12	11-0
cond		opcode	s	Rn	Rd	Op2

移动次数	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
每次移动2位	3130	2928	2726	2524	2322	2120	1918	1716	1514	1312	1110	98	76	54	32	10
	00	00	00	00	00	00	00	00	00	00	00	00	11	11	00	10
第1次	10	00	00	00	00	00	00	00	00	00	00	00	00	11	11	00
第2次	00	10	00	00	00	00	00	00	00	00	00	00	00	00	11	11
第3次	11	00	10	00	00	00	00	00	00	00	00	00	00	00	00	11
第12次	00	00	00	00	00	00	00	00	11	11	00	10	00	00	00	00

## 3.1.3 寄存器间接寻址

- 寄存器中的值是操作数的物理地址。
- 而实际的操作数存放在存储器中。
- 例：

**STR R0, [R1] ; [R1]=R0**

**LDR R0, [R1] ; R0=[R1]**

## 3.1.4 变址寻址

- 将寄存器（称为基址寄存器）的值与指令中给出的偏移地址量相加，**所得结果作为操作数的物理地址**。
- 变址寻址方式常用于访问某基地址附近的地址单元。
- 例：

**LDR R0, [R1, #5] ; R0=[R1+5]**

**LDR R0, [R1, R2] ; R0=[R1+R2]**

## 3.1.5 寄存器移位寻址

- 寄存器移位寻址是ARM指令集独有的寻址方式。
- 寄存器移位寻址的操作数由寄存器的数值做相应移位而得到。
- 移位的方式在指令中以助记符的形式给出，而移位的位数可用立即数或寄存器寻址方式表示。
- ARM微处理器内嵌桶型移位器，移位操作在ARM指令集中不作为单独的指令使用，它只能作为指令格式中的一个字段，在汇编语言中表示为指令中的选项。

• 例：

**ADD R0, R1, R2, ROR #5**

； R0=R1+R2循环右移5位

**MOV R0, R1, LSL R3**

； R0=R1逻辑左移R3位

- ARM指令集共有5种位移操作。



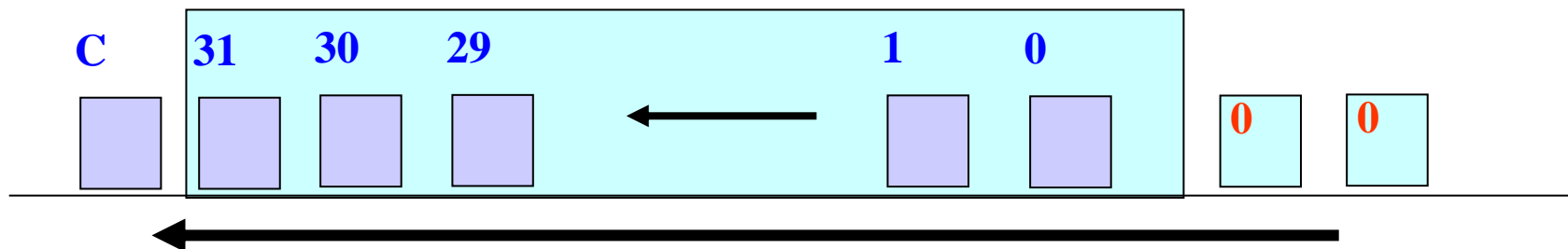
# LSL 逻辑左移

- 格式为:

通用寄存器, **LSL (或ASL)** 操作数

- 功能: LSL (或ASL) 可完成对通用寄存器中的内容进行**逻辑 (或算术) 的左移**操作, 按操作数所指定的数量**向左移位**, 低位用零来填充, 最后一个左移出的位放在状态寄存器的C位。
- 操作数: 通用寄存器, 也可以是立即数 (0~31)。
- 操作示例:

**MOV R0, R1, LSL #2**



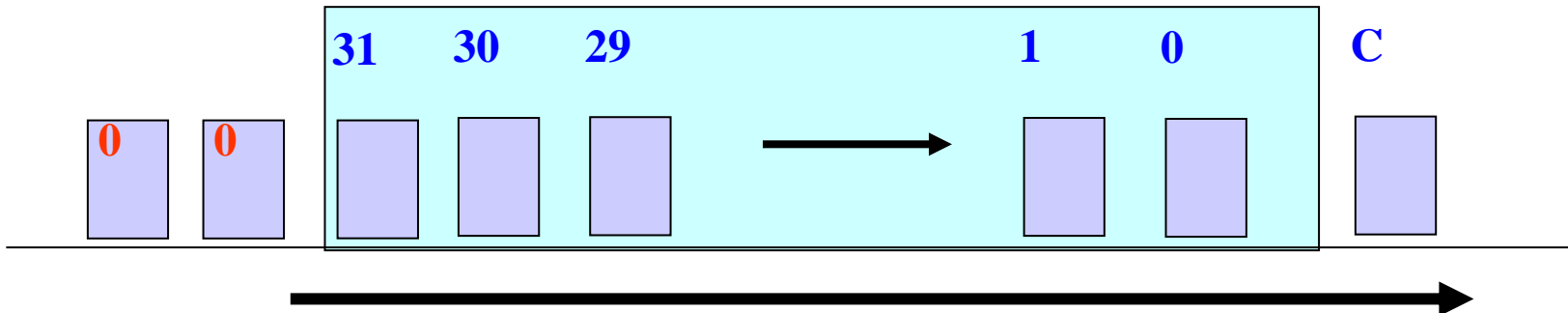
# LSR 逻辑右移

- 格式为:

通用寄存器, **LSR** 操作数

- 功能: **LSR**可完成对通用寄存器中的内容进行**逻辑右移**的操作, 按操作数所指定的数量**向右移位**, 左端用零来填充, 最后一个右移出的位放在状态寄存器的C位。
- 操作数: 通用寄存器, 也可以是立即数 (0~31)。
- 操作示例:

**MOV R0, R1, LSR #4**



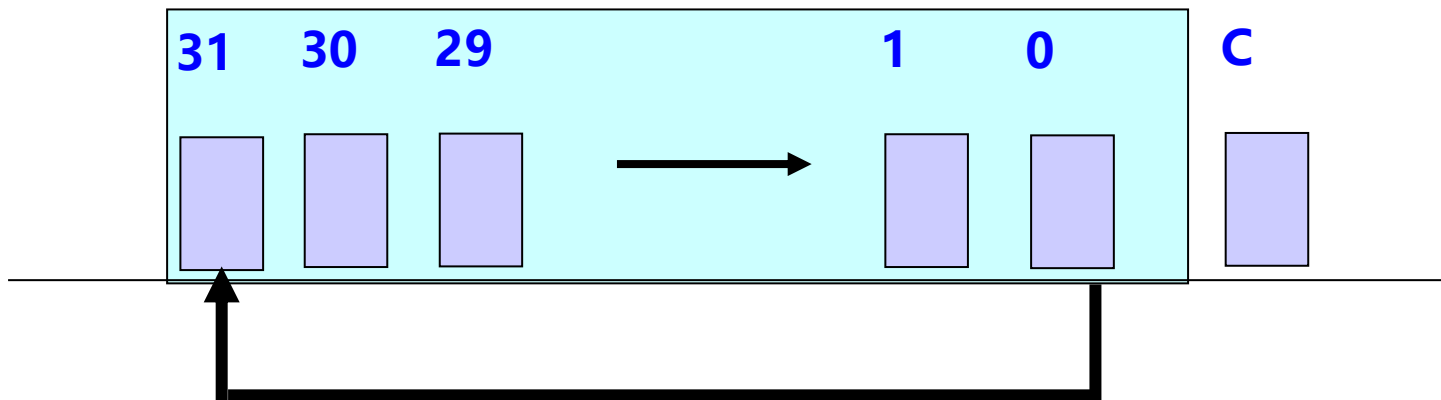
# ROR 循环右移

- 格式为:

通用寄存器, **ROR** 操作数

- 功能: **ROR**可完成对通用寄存器中的内容进行**循环右移**的操作, 按操作数所指定的数量**向右循环移位**, 右端移出的位填充在左侧的空位处, 最后一个右移出的位同时也放在状态寄存器的C位。
- 操作数: 通用寄存器, 也可以是立即数 (0~31)。
- 操作示例:

**MOV R0, R1, ROR #4**



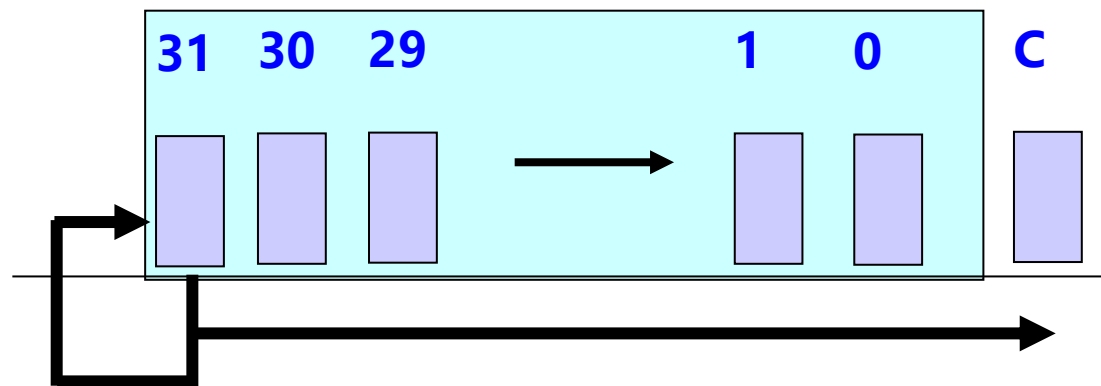
# ASR 算术右移

- 格式为:

通用寄存器, **ASR** 操作数

- 功能: **ASR**可完成对通用寄存器中的内容进行**算术右移**的操作, 按操作数所指定的数量**向右移位**, **最左端的位保持不变**, 最后一个右移出的位放在状态寄存器的**C**位。
- 操作数: 通用寄存器, 也可以是立即数 (0~31)。
- 操作示例:

**MOV R0, R1, ASR #4**



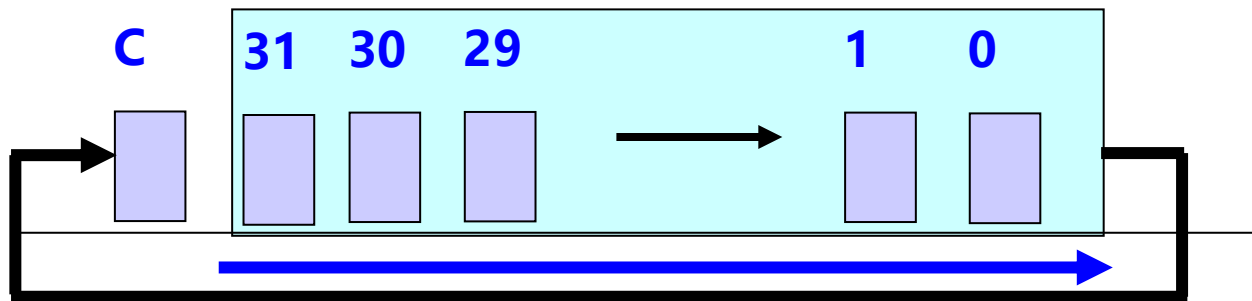
# RRX 带扩展的循环右移

- 格式为:

通用寄存器, **RRX**

- 功能: **RRX**可完成对通用寄存器中的内容进行**带扩展的循环右移**的操作, 向右循环移动1位, 左侧空位由状态寄存器C位来填充, 右侧移出的位移进状态位C中。
- 注意: 没有操作数, **每次**执行指令**只能移动一位**。
- 操作示例:

**MOV R0, R1, RRX**



## 3.1.6 多寄存器寻址

- 在多寄存器寻址方式中，一条指令可实现一组寄存器值的传送。
- 这种寻址方式可以一次对多个寄存器寻址，多个寄存器由小到大排列，最多可传送16个寄存器。
- 连续的寄存器间用“-”连接，否则用“，”分隔。
- 例：

```
LDMIA  R0, {R1-R5} ; R1=[R0]
                        ; R2=[R0+4]
                        ; R3=[R0+8]
                        ; R4=[R0+12]
                        ; R5=[R0+16]
```

- 指令中IA表示在执行完一次Load操作后，R0自增4。
- 该指令将以R0为起始地址的5个字数据分别装入R1，R2，R3，R4，R5中。

## 3.1.7 堆栈寻址

- **堆栈：**按照“先进后出”的原则组织的特定的数据存储的区域。
- 使用专门的寄存器（堆栈指针SP(R13)）指向堆栈栈顶。
- 堆栈寻址用于数据栈与寄存器组之间批量数据传输。
- 当数据写入和读出内存的顺序不同时，使用堆栈寻址可以很好的解决这个问题。
- **例：**

**STMFD R13!, {R0,R1,R2,R3,R4}**

； 将R0-R4中的数据压入堆栈， R13为堆栈指针

**LDMFD R13!, {R0,R1,R2,R3,R4}**

； 将数据出栈，恢复R0-R4原先的值

## 3.1.8 相对寻址

- 相对寻址同基址变址寻址相似，区别只是将程序计数器 **PC作为基址寄存器**，指令中的标记作为地址偏移量。
- 例：

**BEQ process1**

.....

**process1**

.....



## 3.2 ARM指令集

- **机器指令**：CPU指令，能被处理器直接执行，而伪指令宏和宏指令不能。机器指令包括ARM指令集和Thumb指令集；
- **伪指令**：汇编指令，在源程序汇编期间，由汇编编译器处理。其作用是汇编程序完成准备工作；
- **宏指令**：汇编指令，在程序中用于调用宏，宏是一段独立的程序代码；在程序汇编时，对宏调用进行展开，用宏体代替宏指令。
- ARM微处理器的指令集是**加载/存储型的**，即指令集仅能处理寄存器中的数据，处理结果仍要放回寄存器中，而对**系统存储器的访问则需要通过专门的加载/存储指令来完成**。
- ARM9指令集，包括**ARM指令集、Thumb指令集**。

## 3.2.1 指令格式

- 汇编指令语法格式为：

**ADDEQS R0, R1, R2**

- 对应的机器指令的编码格式为：

31~28	27~25	24~21	20	19~16	15~12	11 ~ 0
cond		opcode	S	Rn	Rd	op2
0000	001	0100	1	0001	0000	000000000010

**注：**该指令格式仅针对**本例**的指令，不同的指令有不同的指令格式。

# ARM指令格式

			31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
Data processing immediate shift	cond [1]	0	0	0	opcode			S	Rn			Rd			shift amount				shift	0	Rm																			
Miscellaneous instructions: See Figure 3-3	cond [1]	0	0	0	1	0	x	x	0	x x x x x x x x x x x x x x x x																0	x x x x													
Data processing register shift [2]	cond [1]	0	0	0	opcode			S	Rn			Rd			Rs			0	shift	1	Rm																			
Miscellaneous instructions: See Figure 3-3	cond [1]	0	0	0	1	0	x	x	0	x x x x x x x x x x x x x x x x																0	x	x	1	x x x x										
Multiplies, extra load/stores: See Figure 3-2	cond [1]	0	0	0	x	x	x	x	x	x x x x x x x x x x x x x x x x																1	x	x	1	x x x x										
Data processing immediate [2]	cond [1]	0	0	1	opcode			S	Rn			Rd			rotate				immediate																					
Undefined instruction [3]	cond [1]	0	0	1	1	0	x	0	0	x x																														
Move immediate to status register	cond [1]	0	0	1	1	0	R	1	0	Mask			SBO			rotate				immediate																				
Load/store immediate offset	cond [1]	0	1	0	P	U	B	W	L	Rn			Rd			immediate																								
Load/store register offset	cond [1]	0	1	1	P	U	B	W	L	Rn			Rd			shift amount				shift	0	Rm																		
Undefined instruction	cond [1]	0	1	1	x x																												1	x x x x						
Undefined instruction [4,7]	1	1	1	1	0	x x																																		
Load/store multiple	cond [1]	1	0	0	P	U	S	W	L	Rn			register list																											
Undefined instruction [4]	1	1	1	1	1	0	0	x x																																
Branch and branch with link	cond [1]	1	0	1	L	24-bit offset																																		
Branch and branch with link and change to Thumb [4]	1	1	1	1	1	0	1	H	24-bit offset																															
Coprocessor load/store and double register transfers [6]	cond [5]	1	1	0	P	U	N	W	L	Rn			CRd			cp_num				8-bit offset																				
Coprocessor data processing	cond [5]	1	1	1	0	opcode1				CRn			CRd			cp_num				opcode2	0	CRm																		
Coprocessor register transfers	cond [5]	1	1	1	0	opcode1				L	CRn			Rd			cp_num				opcode2	1	CRm																	
Software interrupt	cond [1]	1	1	1	1	swi number																																		
Undefined instruction [4]	1	1	1	1	1	1	1	1	x x																															

# ARM指令的助记符

- ARM指令在汇编程序中用助记符表示，一般ARM指令的助记符格式为：

**<opcode> {<cond>} {S} <Rd>,<Rn>,<op2>**

- 其中：

< >是必选项

{ }是可选项

<opcode> 操作码，如ADD表示算术加操作指令；

{<cond>} 决定指令执行的条件域；

{S} 决定指令执行是否影响CPSR寄存器的值；

<Rd> 目的寄存器；

<Rn> 第一个操作数，为寄存器；

<op2> 第二个操作数。

- 例如，指令 **ADDEQS R1, R2, #5**

## 第2个操作数

- ARM指令在汇编程序中用助记符表示，一般ARM指令的助记符格式为：

**<opcode> {<cond>} {S} <Rd>,<Rn>,<op2>**

- 灵活地使用第2个操作数“op2”能够提高代码效率。它有如下的形式：
  - **#immed\_8r**：常数表达式；
  - **Rm**：寄存器方式；
  - **Rm, shift**：寄存器移位方式。

## 3.2.2 条件码

- 几乎所有的ARM指令都可以根据当前程序状态寄存器CPSR中标志位的值，有条件地执行。
- ARM指令的条件域<cond>有16种类型。

	cond	编码	CPSR中标志位	含 义
相等	EQ	0000	Z置位	相等
	NE	0001	Z清零	不相等
判负	MI	0100	N置位	负数
	PL	0101	N清零	正数或零
溢出	VS	0110	V置位	溢出
	VC	0111	V清零	未溢出
无符号数	CS/HS	0010	C置位	无符号数大于或等于
	CC/LO	0011	C清零	无符号数小于
	HI	1000	C置位Z清零	无符号数大于
	LS	1001	C清零Z置位	无符号数小于或等于
有符号数	GE	1010	N等于V	带符号数大于或等于
	LT	1011	N不等于V	带符号数小于
	GT	1100	Z清零且(N等于V)	带符号数大于
	LE	1101	Z置位或(N不等于V)	带符号数小于或等于
	AL	1110	忽略	无条件执行
		1111	依版本不同，定义不同	

# 条件助记符--英文解释

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	<u>E</u> qual	Z set
0001	NE	<u>N</u> ot <u>e</u> qual	Z clear
0010	CS/HS	<u>C</u> arry <u>s</u> et/unsigned <u>h</u> igher or <u>s</u> ame	C set
0011	CC/LO	<u>C</u> arry <u>c</u> lear/unsigned <u>l</u> ower	C clear
0100	MI	<u>M</u> inus/negative	N set
0101	PL	<u>P</u> lus/positive or zero	N clear
0110	VS	Overflow	<u>V</u> <u>s</u> et
0111	VC	No overflow	<u>V</u> <u>c</u> lear
1000	HI	Unsigned <u>h</u> igher	C set and Z clear
1001	LS	Unsigned <u>l</u> ower or <u>s</u> ame	C clear or Z set
1010	GE	Signed <u>g</u> reater than or <u>e</u> qual	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed <u>l</u> ess <u>t</u> han	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed <u>g</u> reater <u>t</u> han	Z clear, and either N set and V set, or N clear and V clear (Z == 0, N == V)
1101	LE	Signed <u>l</u> ess than or <u>e</u> qual	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	<u>A</u> lways (unconditional)	-

# 条件-举例

- **CMP** R0, R1 ; R0与R1比较
- **ADDHI** R0, R0, #1 ; 若R0 > R1, 则R0=R0+1
- **ADDLS** R1, R1, #1 ; 若R0 ≤ R1, 则R1=R1+1

	cond	编码	CPSR中标志位	含 义
相等	EQ	0000	Z置位	相等
	NE	0001	Z清零	不相等
判负	MI	0100	N置位	负数
	PL	0101	N清零	正数或零
溢出	VS	0110	V置位	溢出
	VC	0111	V清零	未溢出
无符号数	CS/HS	0010	C置位	无符号数大于或等于
	CC/LO	0011	C清零	无符号数小于
	<b>HI</b>	1000	C置位Z清零	<b>无符号数大于</b>
	<b>LS</b>	1001	C清零Z置位	<b>无符号数小于或等于</b>
有符号数	GE	1010	N等于V	带符号数大于或等于
	LT	1011	N不等于V	带符号数小于
	GT	1100	Z清零且 (N等于V)	带符号数大于
	LE	1101	Z置位或 (N不等于V)	带符号数小于或等于
	AL	1110	忽略	无条件执行
		1111	依版本不同, 定义不同	



## 3.2.4 ARM 数据处理类指令

- 数据处理指令只能对寄存器的内容进行操作，不允许对存储器中的数据进行操作，也不允许指令直接使用存储器的数据或在寄存器与存储器之间传送数据。
- 数据处理指令可分为3大类：数据传送指令、算术逻辑运算指令、比较指令。
- 数据传送指令：用于在寄存器和存储器之间进行数据的双向传输。
- 算术逻辑运算指令：完成常用的算术与逻辑的运算，该类指令不但将运算结果保存在目的寄存器中，同时更新CPSR中的相应条件标志位。
- 比较指令：完成对指定的两个寄存器（或1个寄存器，1个立即数）进行比较，不保存运算结果，只影响CPSR中相应的条件标志位。

# MOV 数据传送指令

- 格式: **MOV{<cond>}{S} <Rd>, <op1>**
- 功能:  **$Rd \leftarrow op1$**
- Op1: 寄存器、被移位的寄存器、立即数。

- 例如:

- **MOV R0, #5** ; **R0=5**
- **MOV R0, R1** ; **R0=R1**
- **MOV R0, R1, LSL #5** ; **R0=R1左移5位**

# MVN 数据取反传送指令

- 格式:  $\text{MVN}\{\langle\text{cond}\rangle\}\{\text{S}\} \langle\text{Rd}\rangle, \langle\text{op1}\rangle$
- 功能:  $\text{op1}$ 按位取反 $\rightarrow\text{Rd}$ , 即 $\text{Rd}=\sim\text{op1}$ 。
- Op1: 寄存器、被移位的寄存器、立即数。

- 例如:
- $\text{MVN R0, \#0} \quad ; \text{R0} = -1$

# ADD 加法指令

- 格式: **ADD{<cond>}{S} <Rd>, <Rn>, <op2>**
- 功能:  **$Rd \leftarrow Rn + op2$**
- Op2: 寄存器, 被移位的寄存器、立即数。
- 例如:
  - **ADD R0, R1, #5 ; R0=R1+5**
  - **ADD R0, R1, R2 ; R0=R1+R2**
  - **ADD R0, R1, R2, LSL #5 ; R0=R1+R2左移5位**

# ADC 带进位加法指令

- **格式:**  $\text{ADC}\{\langle\text{cond}\rangle\}\{\text{S}\} \langle\text{Rd}\rangle, \langle\text{Rn}\rangle, \langle\text{op2}\rangle$
- **功能:**  $\text{Rd} \leftarrow \text{Rn} + \text{op2} + \text{carry}$
- **Op2:** 寄存器、被移位的寄存器、立即数。
- **carry**为进位标志值。
- 该指令用于实现超过32位的数的加法。
- **例如:**
  - 第一个64位操作数存放在寄存器R3 R2中;
  - 第二个64位操作数存放在寄存器R5 R4中;
  - 64位结果存放在R1 R0中。
  - 64位的加法可由以下语句实现:
- **ADDS R0, R2, R4** ;低32位相加, S表示结果影响条件标志位的值
- **ADC R1, R3, R5** ;高32位相加

# SUB 减法指令

- 格式: SUB{<cond>}{S} <Rd>, <Rn>, <op2>
- 功能:  $Rd \leftarrow Rn - op2$
- Op2: 寄存器、被移位的寄存器、立即数。
- 例如:
  - SUB R0, R1, #5 ; R0=R1-5
  - SUB R0, R1, R2 ; R0=R1-R2
  - SUB R0, R1, R2, LSL #5 ; R0=R1-R2左移5位

# SBC 带借位减法指令

- 格式:  $\text{SBC}\{\langle\text{cond}\rangle\}\{\text{S}\} \langle\text{Rd}\rangle, \langle\text{Rn}\rangle, \langle\text{op2}\rangle$
- 功能:  $\text{Rd} \leftarrow \text{Rn} - \text{op2} - \text{!carry}$
- Op2: 寄存器、被移位的寄存器、立即数。
- SUB和SBC生成进位标志的方式不同于常规, 如果需要借位则清除进位标志, 所以指令要对进位标志进行一个非操作。
- 例如:
  - 第一个64位操作数存放在寄存器R3 R2中;
  - 第二个64位操作数存放在寄存器R5 R4中;
  - 64位结果存放在R1 R0中。
  - 64位的减法 (第一个操作数减去第二个操作数) 可由以下语句实现:
  - $\text{SUBS } \text{R0}, \text{R2}, \text{R4}$  ; 低32位相减, S表示结果影响条件标志位的值
  - $\text{SBC } \text{R1}, \text{R3}, \text{R5}$  ; 高32位相减

# RSB 反向减法指令

- 格式: **RSB{<cond>}{S} <Rd>, <Rn>, <op2>**
- 功能: 同SUB指令, 但倒换了两操作数的前后位置, 即  $Rd \leftarrow op2 - Rn$ 。
- 例如:
  - **RSB R0, R1, #5 ; R0=5-R1**
  - **RSB R0, R1, R2 ; R0=R2-R1**
  - **RSB R0, R1, R2, LSL #5 ; R0=R2左移5位-R1**



# RSC 带借位的反向减法指令

- **格式:** **RSC{<cond>}{S} <Rd>, <Rn>, <op2>**
- **功能:** 同SBC指令, 但倒换了两操作数的前后位置,  
即  $Rd \leftarrow op2 - Rn - !carry$
- **例如:**
  - 第一个64位操作数存放在寄存器R3 R2中;
  - 第二个64位操作数存放在寄存器R5 R4中;
  - 64位结果存放在R1 R0中。
  - 64位的减法（第一个操作数减去第二个操作数）可由以下语句实现:
- **SUBS R0, R4, R2** ; 低32位相减, S表示结果影响寄存器CPSR的值
- **RSC R1, R5, R3** ; 高32位相减

# AND 逻辑与指令

- 格式:  $\text{AND}\{\langle\text{cond}\rangle\}\{\text{S}\} \langle\text{Rd}\rangle, \langle\text{Rn}\rangle, \langle\text{op2}\rangle$
- 功能:  $\text{Rd} \leftarrow \text{Rn AND op2}$
- Op2: 寄存器, 被移位的寄存器、立即数。
- 一般用于清除Rn的特定几位。
- 按位操作。

- 例如: 假设  $\text{R0} = 0\text{X}12345678$
- $\text{AND R0, R0, \#5}$   
; 保持R0的第0位和第2位, 其余位清0

OP1	运算符 (与 $\wedge$ )	OP2	=	RESULT
0	$\wedge$	0	=	0
0	$\wedge$	1	=	0
1	$\wedge$	0	=	0
1	$\wedge$	1	=	1

	0001 0010 0011 0100 0101 0110 0111 1000	0X12345678
$\wedge$	0000 0000 0000 0000 0000 0000 0000 0101	5
	0000 0000 0000 0000 0000 0000 0000 0000	

# ORR 逻辑或指令

- 格式: **ORR**{<cond>}{S} <Rd>, <Rn>, <op2>
- 功能:  $Rd \leftarrow Rn \text{ OR } op2$
- Op2: 寄存器、被移位的寄存器、立即数。
- 一般用于设置Rn的特定几位。

- 例如: 假设  $R0 = 0X12345678$
- **ORR R0, R0, #5**  
; R0的第0位和第2位设置为1,  
; 其余位不变

OP1	运算符 (或V)	OP2	=	RESULT
0	V	0	=	0
0	V	1	=	1
1	V	0	=	1
1	V	1	=	1

	0001 0010 0011 0100 0101 0110 0111 1000	0X12345678
✓	0000 0000 0000 0000 0000 0000 0000 0101	5
	0001 0010 0011 0100 0101 0110 0111 1101	0x1234567D

# EOR 逻辑异或指令

- 格式: **EOR**{<cond>}{S} <Rd>, <Rn>, <op2>
- 功能:  $Rd \leftarrow Rn \text{ EOR } op2$
- Op2: 寄存器、被移位的寄存器、立即数。
- 一般用于将Rn的特定几位取反。

• 例如: 假设  $R0 = 0X12345678$

• **EOR R0, R0, #5**

; R0的第0位和第2位取反,

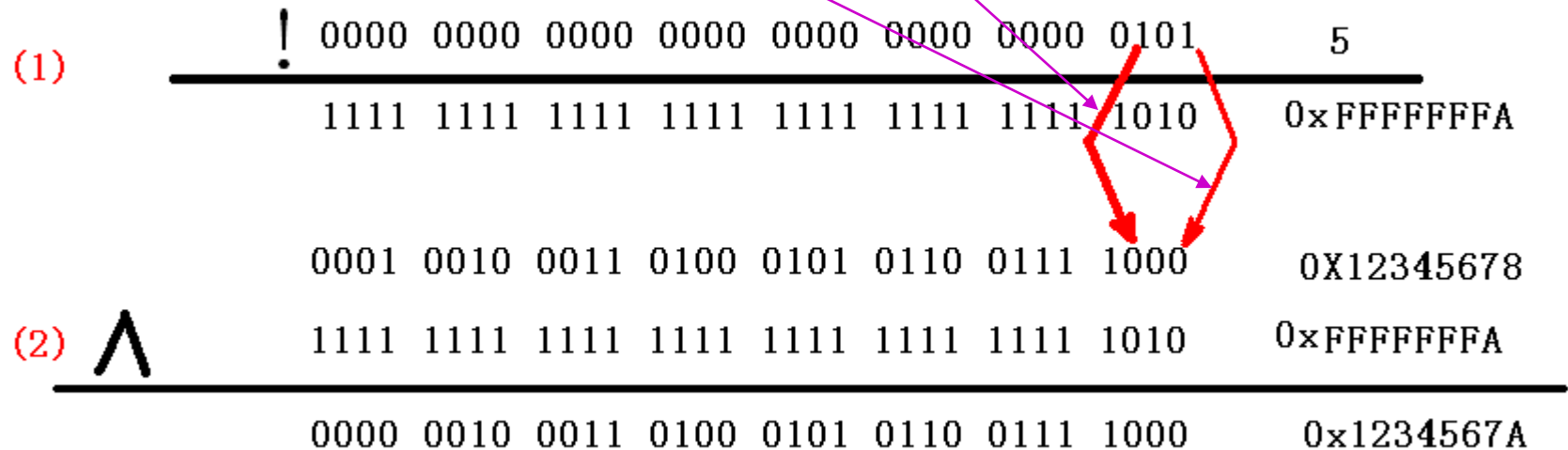
; 其余位不变

OP1	运算符(异或 $\oplus$ )	OP2	=	RESULT
0	$\oplus$	0	=	0
0	$\oplus$	1	=	1
1	$\oplus$	0	=	1
1	$\oplus$	1	=	0

	0001 0010 0011 0100 0101 0110 0111 1000	0X12345678
$\oplus$	0000 0000 0000 0000 0000 0000 0000 0101	5
	0001 0010 0011 0100 0101 0110 0111 1101	0x1234567D

# BIC 位清除指令

- 格式:  $\text{BIC}\{\langle\text{cond}\rangle\}\{\text{S}\} \langle\text{Rd}\rangle, \langle\text{Rn}\rangle, \langle\text{op2}\rangle$
- 功能:  $\text{Rd} \leftarrow \text{Rn} \text{ AND } (!\text{op2})$
- 用于清除寄存器Rn中的某些位, 并把结果存放到目的寄存器Rd中。
- 操作数op2是一个32位掩码 (mask), 如果在掩码中设置了某一位, 则清除Rn中的这一位; 未设置的掩码位指示Rn中此位保持不变。
- Op2: 寄存器、被移位的寄存器、立即数。
- 例如: 假设  $\text{R0} = 0\text{X}12345678$
- $\text{BIC R0, R0, \#5}$  ; R0中第0位和第2位清0, 其余位不变



# MUL 32位乘法指令

- 格式:  $\text{MUL}\{\langle\text{cond}\rangle\}\{\text{S}\} \langle\text{Rd}\rangle, \langle\text{Rn}\rangle, \langle\text{op2}\rangle$
- 功能:  $\text{Rd} \leftarrow \text{Rn} \times \text{op2}$
- 该指令根据S标志, 决定操作是否影响CPSR的值。
- Op2: 必须为寄存器。
- Rn和op2的值为32位的有符号数或无符号数。
- 例如:
- $\text{MULS R0, R1, R2} \quad ; \text{R0} = \text{R1} \times \text{R2}$ , 结果影响寄存器CPSR的值

# MLA 32位乘加指令

- 格式:  $\text{MLA}\{\langle\text{cond}\rangle\}\{\text{S}\} \langle\text{Rd}\rangle, \langle\text{Rn}\rangle, \langle\text{op2}\rangle, \langle\text{op3}\rangle$
- 功能:  $\text{Rd} \leftarrow \text{Rn} \times \text{op2} + \text{op3}$
- Op2、op3: 必须为寄存器。
- Rn、op2和op3的值为32位的有符号数或无符号数。
- 例如:
- $\text{MLA R0, R1, R2, R3} \quad ; \text{R0} = \text{R1} \times \text{R2} + \text{R3}$

# SMULL 64位有符号数乘法指令

- 格式:

**SMULL{<cond>}{S} <Rdl>, <Rdh>, <Rn>, <op2>**

- 功能: **Rdh Rdl** ← **Rn** × **op2**

- Rdh**、**Rdl**、**op2**: 均为寄存器。

- Rn**和**op2**的值为32位的有符号数。

- 例如:

- SMULL R0, R1, R2, R3** ; **R0** = **R2** × **R3** 的低32位  
; **R1** = **R2** × **R3** 的高32位



# SMLAL 64位有符号数乘加指令

- 格式:

**SMLAL{<cond>}{S} <Rdl>, <Rdh>, <Rn>, <op2>**

- 功能:  **$Rdh\ Rdl \leftarrow Rn \times op2 + Rdh\ Rdl$**

- Rdh、Rdl、op2:** 均为寄存器。
- Rn和op2**的值为32位的有符号数。
- Rdh Rdl**的值为64位的加数。

- 例如:

- SMLAL R0, R1, R2, R3**                   ;  **$R0 = R2 \times R3$ 的低32位+R0**  
   ;  **$R1 = R2 \times R3$ 的高32位+R1**

# UMULL 64位无符号数乘法指令

- 格式:

**UMULL{<cond>}{S} <Rdl>, <Rdh>, <Rn>, <op2>**

- 功能: 同SMULL指令:

**Rdh Rdl ← Rn × op2**

- 但Rn和op2的值为32位的无符号数。

- 例如:

- **UMULL R0, R1, R2, R3** ; R0=R2×R3的低32位  
; R1=R2×R3的高32位
- 其中R2、R3的值为无符号数

# UMLAL 64位无符号数乘加指令

- 格式:

**UMLAL {<cond>}{S} <Rdl>, <Rdh>, <Rn>, <op2>;**

- 功能: 同SMLAL指令:

**$Rdh\ Rdl \leftarrow Rn \times op2 + Rdh\ Rdl$**

- 但Rn, op2的值为32位的无符号数, Rdh Rdl的值为64位无符号数。

- 例如:

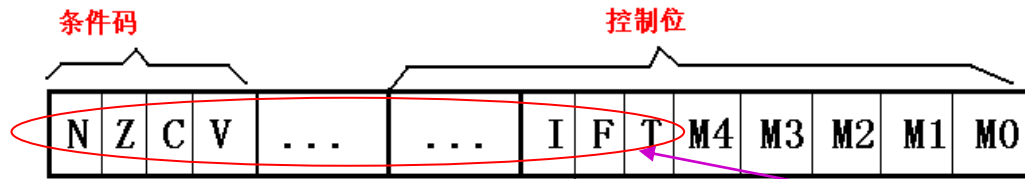
- UMLAL R0, R1, R2, R3 ; R0=R2×R3的低32位+R0**  
**; R1=R2×R3的高32位+R1**

- 其中R2、R3的值为32位无符号数
- R1、R0的值为64位无符号数

# CMP 比较指令

- 格式: `CMP{<cond>} <Rn>, <op1>`
- 功能: `Rn-op1`
- 该指令进行一次减法运算, 但不存储结果, 根据结果更新CPSR中条件标志位的值。
- 该指令不需要显式地指定S后缀来更改状态标志。
- Op1: 寄存器、立即数。
- 例如:
- `CMP R0, #5` ; 计算R0-5, 根据结果设置条件标志位
- `ADDGT R0, R0, #5` ; 如果R0>5, 则执行ADDGT指令

# 如何影响标志位



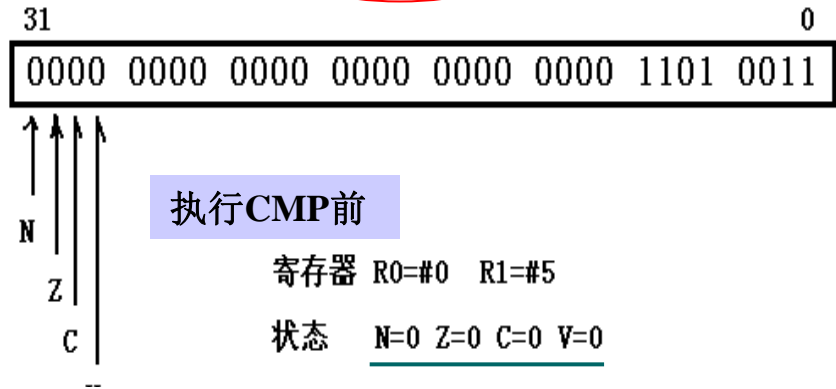
N-->Negative  
Z-->Zero  
C-->Carry  
V -->Overflow

M[4:0]	模式
10000	用户
10001	FIQ
10010	IRQ
10011	管理
10111	中止
11011	未定义
11111	系统



```
AND R0, R0, #0
MOV R1, #5
CMP R0, R1
```

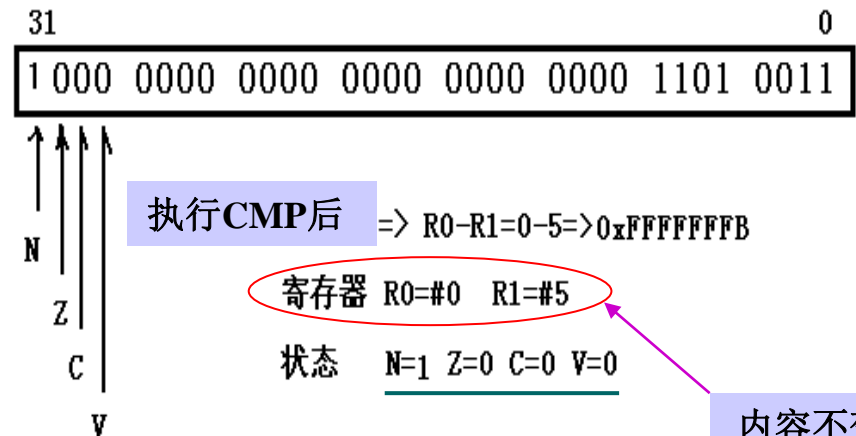
CPSR=0X000000D3=nZcvqift\_



CPSR=0x800000D3 = NzCvQifT\_Svc



```
AND R0, R0, #0
MOV R1, #5
CMP R0, R1
```



# CMN 反值比较指令

- 格式:  $\text{CMN}\{\langle\text{cond}\rangle\} \langle\text{Rn}\rangle, \langle\text{op1}\rangle$
- 功能: 同CMP指令, 但寄存器Rn的值是和op1取负的值进行比较:  
 $\text{Rn} - (-\text{op1})$
- 例如:
- $\text{CMN R0, \#5}$  ; 把R0与-5进行比较

# TST 位测试指令

- 格式:  $\text{TST}\{\langle\text{cond}\rangle\} \quad \langle\text{Rn}\rangle, \langle\text{op1}\rangle$
- 功能:  $\text{Rn AND op1}$
- 根据结果更新CPSR中条件标志位的值, 但不存储结果。
- 用于检查寄存器Rn是否设置了op1中相应的位。
- 例如:
- $\text{TST} \quad \text{R0}, \#5$  ; 测试R0中第0位和第2位是否为1

# TEQ 相等测试指令

- **格式:** TEQ {<cond>} <Rn>, <op1>
- **功能:** Rn EOR op1
- 将寄存器Rn的值和操作数op1所表示的值按位作逻辑异或操作，根据结果更新CPSR中条件标志位的值，但不存储结果。
- 用于检查寄存器Rn的值是否和op1所表示的值相等。
- **例如:**
- TEQ R0, #5 ; 判断R0的值是否和5相等



## 3.2.5 ARM 分支指令

- 分支指令作用：程序的跳转、程序状态的切换。
- ARM程序设计中，实现程序跳转有**两种方式**：
  - (1) **跳转指令**：跳转空间依据指令确定。
  - (2) **直接向程序寄存器PC（R15）中写入目标地址值**：在4GB空间任意跳转。
- **程序状态切换**：ARM状态、Thumb状态。

# B 跳转指令 (Branch)

- 格式:  $B\{\langle cond \rangle\} \langle addr \rangle$
- 功能:  $PC \leftarrow PC + \langle addr \rangle$  左移两位
- 说明:  $\langle addr \rangle$  的值是相对当前PC (即寄存器R15) 的值的一个偏移量, 而不是一个绝对地址, 它是24位有符号数。实际地址的值由汇编器来计算。
- 目标地址计算方法:

跳转的目的地址 =  $\langle addr \rangle$  的值有符号扩展为32位后左移两位 (即乘4) + PC值

- 跳转的范围为-32MB~+32MB。
- 例如: (类似8086 JMP)

```
B      exit      ; 程序跳转到标号exit处
```

```
...
```

```
exit ...
```

- 例如: 执行10次循环。

```
MOV    R0, #10    ; 初始化循环计数器
```

```
loop ...          ; 循环体
```

```
SUBS   R0, #1     ; 计数器减1, 设置条件码
```

```
BNE    loop       ; 如果计数器R0≠0, 重复循环
```

# BL 带返回的跳转指令 (Branch with Link)

- 格式: **BL**{<cond>} <addr>
- 功能: 子程序调用。
- 同B指令, 但BL指令执行跳转操作的同时, 还将子程序的返回地址 (注意: 不是PC内容) 保存到LR寄存器 (寄存器R14) 中。
- 该指令用于实现子程序调用, 程序的返回可通过把LR寄存器的值复制到PC寄存器中来实现。

• 例如:

```
BL    func           ; 调用子程序func
...
func
...
MOV   R15, R14       ; 子程序返回
```

• 例如: 条件子程序调用。

```
...
CMP   R0, #5
BLLT  SUB1           ; 若R0<5, 调用SUB1子程序
BLGE  SUB2           ; 否则调用SUB2子程序
```

# BX带状态切换的跳转指令

- 格式: **BX** <Rn>
- 功能: 状态切换。

处理器跳转到目标地址处，从那里继续执行。

目标地址为寄存器**Rn**的值和**0xFFFFFFFF**作与操作的结果。

目标地址处的指令可以是ARM指令，也可以是Thumb 指令。

- 例如:

**ADR R0, exit** ; 标号**exit**处的地址装入**R0**中

**BX R0** ; 跳转到**exit**处

# BLX 带返回和状态切换的跳转指令

- 格式: BLX <addr>

BLX <Rn>

- 功能: 调用、状态切换。
- 处理器跳转到目标地址处, 并将返回地址保存到LR寄存器中。
- 如果目标地址处为Thumb指令, 则程序状态从ARM状态切换为Thumb状态。

- 例如:

BLX T16 ; 跳转到T16处执行, T16后面的指令为Thumb指令

...

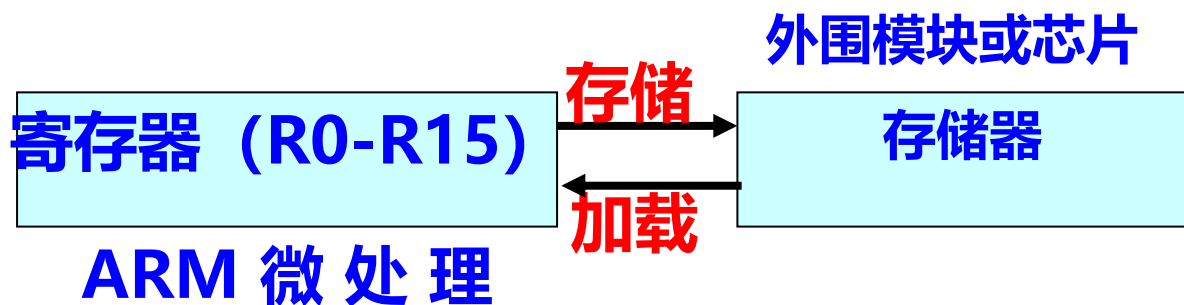
CODE16

T16 ; 后面指令为Thumb指令

...

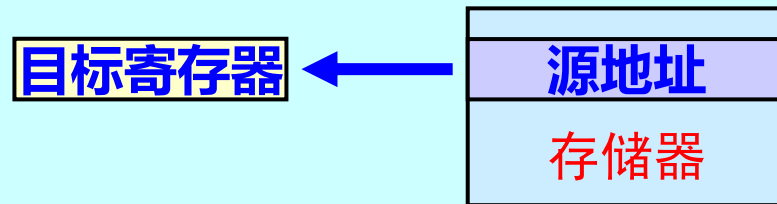
## 3.2.3 ARM 存储器访问指令

- **存储器访问指令功能**：寄存器和内存之间数据的传送。
- **Load（加载）**：寄存器←内存
- **Store（存储）**：内存←寄存器
- 该组指令使用频繁，在指令集中最为重要，因为其他指令只能操作寄存器，当数据存放在内存中时，必须先把数据从内存装载到寄存器，执行完后再把寄存器中的数据存储到内存中。
- **Load/Store指令分为3类**：
  - （1）单一数据传送指令（LDR和STR等）
  - （2）多数据传送指令（LDM和STM）
  - （3）数据交换指令（SWP和SWPB）

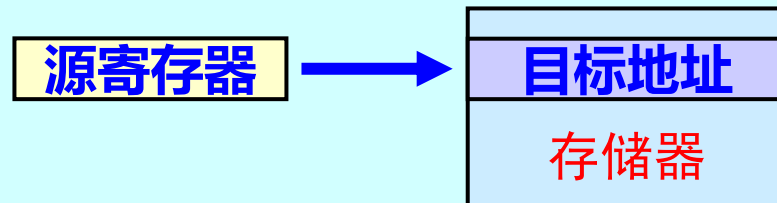


# 加载指令与存储指令

加载指令: **LDR**      目标寄存器, 源地址



存储指令: **STR**      源寄存器, 目标地址



# LDR 字数据加载指令

- 格式:  $\text{LDR}\{\text{<cond>}\} \text{ <Rd>, <addr>}$
- 功能:
  - (1)  $\text{addr}$ 所表示的内存地址中的字数据 $\rightarrow$ 目标寄存器Rd
  - (2) 合成的有效地址 $\rightarrow$ 基址寄存器。
- 地址 $\text{addr}$ : 简单的值、偏移量、被移位的偏移量。
- 地址 $\text{addr}$ 的寻址方式涉及的寄存器意义:
  - Rn: 基址寄存器。
  - Rm: 变址寄存器。
  - Index: 偏移量, 12位的无符号数。



# Load/Store指令一般形式

- **LDR Rd, [Rn]** ; 内存地址为Rn的字数据→Rd
- **LDR Rd, [Rn, Rm]** ; 内存地址为Rn+Rm的字数据→Rd
- **LDR Rd, [Rn, #index]** ; 内存地址为Rn+index的字数据→Rd
- **LDR Rd, [Rn, Rm, LSL #k]** ; 内存地址为Rn+Rm×2的k次方的字数据→Rd
- **LDR Rd, [Rn, Rm] !** ; 内存地址为Rn+Rm的字数据→Rd  
; 并将新地址Rn+Rm→Rn
- **LDR Rd, [Rn, #index] !** ; 内存地址为Rn+index的字数据→Rd  
; 并将新地址Rn+index→Rn
- **LDR Rd, [Rn, Rm, LSL #k] !** ; 内存地址为Rn+Rm×2的k次方的字数据→Rd  
; 并将新地址Rn+Rm×2的k次方→Rn
- **LDR Rd, [Rn], Rm** ; 内存地址为Rn的字数据→Rd  
; 并将新地址Rn+Rm→Rn
- **LDR Rd, [Rn], #index** ; 内存地址为Rn的字数据→Rd  
; 并将新地址Rn+index→Rn
- **LDR Rd, [Rn], Rm, LSL #5** ; 内存地址为Rn的字数据→Rd  
; 并将新地址Rn+Rm×32→Rn

# Store指令一般形式

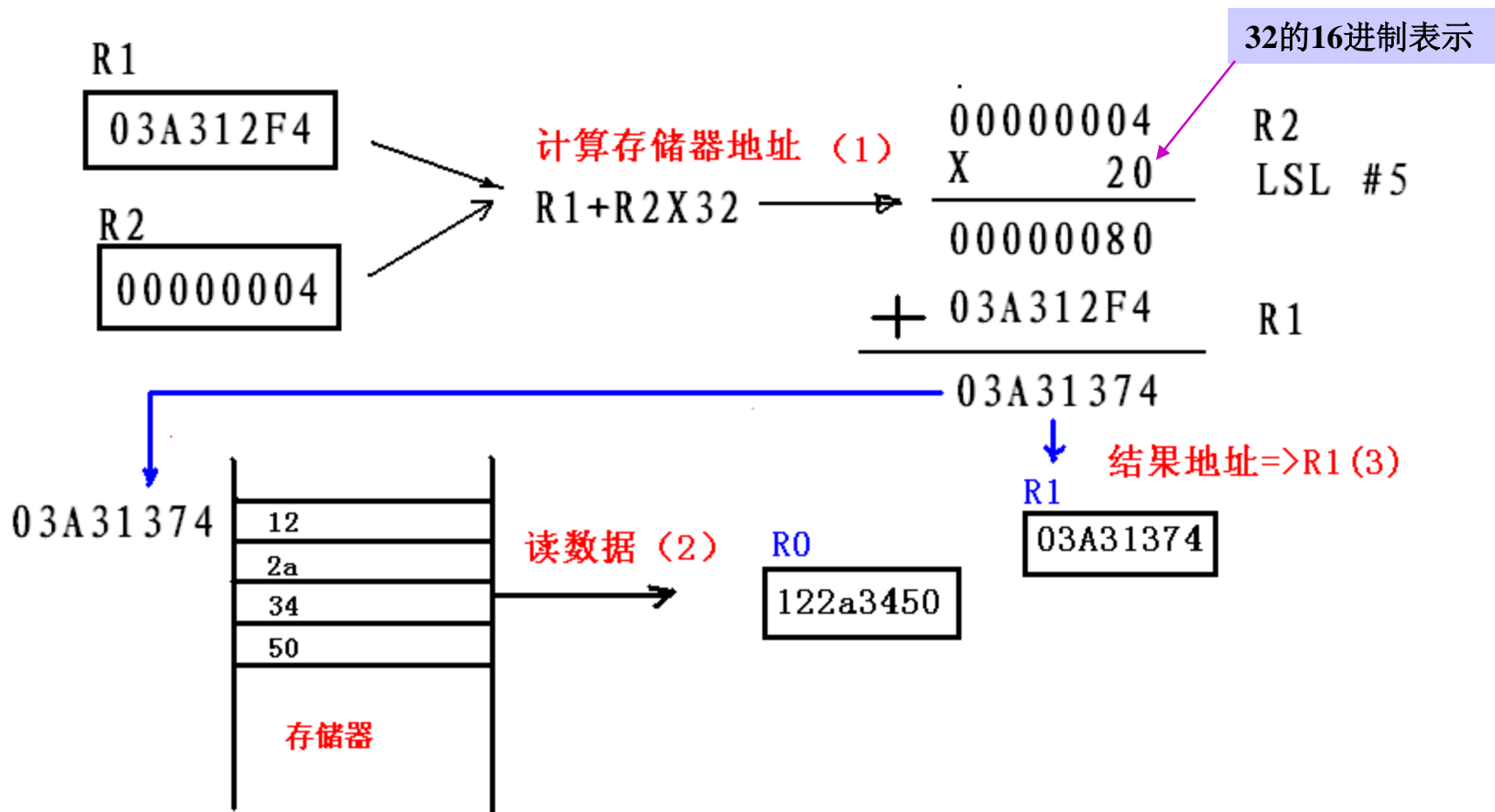
- **STR Rd, [Rn]** ; 内存地址Rn的字存储单元 $\leq$ Rd
- **STR Rd, [Rn, Rm]** ; 内存地址Rn+Rm的字存储单元 $\leq$ Rd
- **STR Rd, [Rn, #index]** ; 内存地址Rn+index的字存储单元 $\leq$ Rd
- **STR Rd, [Rn, Rm, LSL #k]** ; 内存地址Rn+Rm $\times 2^k$ 的字存储单元 $\leq$ Rd
- **STR Rd, [Rn, Rm] !** ; 内存地址为Rn+Rm的字存储单元 $\leq$ Rd  
; 并将新地址Rn+Rm $\rightarrow$ Rn
- **STR Rd, [Rn, #index] !** ; 内存地址为Rn+index的字存储单元 $\leq$ Rd  
; 并将新地址Rn+index $\rightarrow$ Rn
- **STR Rd, [Rn, Rm, LSL #k] !** ; 内存地址为Rn+Rm $\times 2^k$ 的字存储单元 $\leq$ Rd  
; 并将新地址Rn+Rm $\times 2^k \rightarrow$ Rn
- **STR Rd, [Rn], Rm** ; 内存地址为Rn的字存储单元 $\leq$ Rd  
; 并将新地址Rn+Rm $\rightarrow$ Rn
- **STR Rd, [Rn], #index** ; 内存地址为Rn的字存储单元 $\leq$ Rd  
; 并将新地址Rn+index $\rightarrow$ Rn
- **STR Rd, [Rn], Rm, LSL #k** ; 内存地址为Rn的字存储单元 $\leq$ Rd  
; 并将新地址Rn+Rm $\times 2^k \rightarrow$ Rn

# Load/Store指令过程

• 例如:

• **LDR R0, [R1, R2, LSL #5]!**

; 将内存中地址为 $R1+R2 \times 32$ 的字数据装入  
; 寄存器R0, 并将新地址 $R1+R2 \times 32$ 写入R1



# Load/Store指令伪代码

- LDR {<cond>}<Rd>, <addressing\_mode>

- 指令操作的伪代码:

if ConditionPassed(cond) then

if adderss[1:0] == 0b00 then

Value = Memory[adderss, 4]

else if adderss[1:0] == 0b01 then

Value = Memory[adderss, 4] Rotate\_Right 8

else if adderss[1:0] == 0b10 then

Value = Memory[adderss, 4] Rotate\_Right 16

else if adderss[1:0] == 0b11 then

Value = Memory[adderss, 4] Rotate\_Right 24

if (Rd is R15) then

if (architecture version 5 or above) then

PC = value AND 0xFFFFFFF0

T Bit = value[0]

else

PC = value AND 0xFFFFFFF0

else

Rd = value

# Load/Store指令举例

- **LDR R1, [R0, #0x12]** ; 读出R0+0x12地址中的数据, 保存到R1  
; 中 (R0的值不变)
  - **LDR R1, [R0, -R2]** ; 将R0-R2地址处的数据读出, 保存到R1  
; 中 (R0的值不变)
  - **LDR R1, [R0]** ; 将R0地址处的数据读出, 保存到R1中
  - **LDR R1, localdata** ; 2的幂次方字, 该字位于变量localdata所  
; 在地址
  - **LDR R0, [R1], R2, LSL #2** ; 将地址为R1的内存单元数据读取到R0  
; 中, 然后 $R1 = R1 + R2 \times 4$
- 
- 在编程中常使用该指令将外设端口数据读到R0中:
  - **MOV R1, #UARTADD** ; 将外设端口地址UART ADD装入R1中
  - **LDR R0, [R1]** ; 将外设端口数据存到R0中

# LDRB 字节数据加载指令

- 格式: **LDR{<cond>}B <Rd>, <addr>**
- 功能: 同LDR指令。但
  - (1) **addr**地址内8位的字节数据→ **Rd**
  - (2) **Rd**的高24位清0
- 例如:
- **LDRB R0, [R1]** ; 将内存中起始地址为**R1**的一个字节数据装入**R0**中

# LDRH 半字数据加载指令

- 格式: **LDR{<cond>}H <Rd>, <addr>**
- 功能: 同LDR指令。但
  - (1) **addr**地址内16位的半字数据→ **Rd**
  - (2) **Rd**的高16位清0
- 例如:
- **LDRH R0, [R1]** ; 将内存中起始地址为**R1**的半字数据  
; 装入**R0**中

# LDRT 用户模式的字数据加载指令

- 格式: **LDR**{<cond>}T <Rd>, <addr>
- 功能: 同**LDR**指令。
- 说明: 无论处理器处于何种模式, 都将该指令当作一般用户模式下的内存操作。
- **addr**所表示的有效地址必须是字对齐的, 否则从内存中读出的数值需进行循环右移操作。



# LDRBT 用户模式的字节数据加载指令

- 格式: LDR{<cond>}BT <Rd>, <addr>
- 功能: 同LDRB指令。
- 说明: 无论处理器处于何种模式, 都将该指令当作一般用户模式下的内存操作。

# LDRSB 有符号的字节数据加载指令

- 格式: **LDR{<cond>}SB <Rd>, <addr>**
- 功能: 同LDRB指令, 但
  - (1) **addr**地址内8位的字节数据→ **Rd**
  - (2) **Rd**高24位按字节符号位扩展
- 例如:

**R1=0x10000000**

**[0x10000000]=0x93**

**LDRSB R0, [R1]**

结果 **R0=0xFFFFFFFF93**

# LDRSH 有符号的半字数据加载指令

- 格式: **LDR{<cond>}SH <Rd>, <addr>**
- 功能: 同LDRH指令, 但
  - (1) **addr**地址内16位的半字数据→ **Rd**
  - (2) **Rd**高16位按半字符号位扩展
- 例如:

**R1=0x10000000**

**[0x10000000]=0x933D**

**LDRSH R0, [R1]**

**结果 R0=0xFFFF933D**

# STR 字数据存储指令

- 格式: **STR**{<cond>} <Rd>, <addr>
- 功能:
  - (1) 寄存器Rd字数据 (32位) → addr地址中
  - (2) 还可以: 合成的有效地址 → 基址寄存器
- 地址addr: 简单的值、一个偏移量、被移位的偏移量。
- 寻址方式同LDR指令。
- 例如:

**R1=0x40003000 R0=0x1A27E4F3**

**STR R0, [R1, #4] !**

- (1) 存储器地址 = **R1+#4=0x40003000+#4 =0x40003004**
- (2) **R0→[0x40003004]= 0x1A27E4F3**
- (3) 修改R1      **R1+#4→ R1= 0x40003004**

# STR 字数据存储指令举例

- **STR R2, [R1, #16]** ; 将R2的内容保存到R1+16为地址的内存中
- **STR R0, [R7], # -8** ; 将R0的内容保存到R7中地址对应的内存  
; 中,  $R7 \leftarrow R7 - 8$
- **STR R2, [R9, #consta-struct]** ; consta-struct是一个常量表达式,  
; 范围为-4095~4095
- 在编程中常使用该指令将R0中的一个字存到外设端口寄存器中:
- **MOV R1, #UARTADD** ; 将外设端口地址UARTADD装入R1中
- **STR R0, [R1]** ; 将数据保存到外设端口寄存器中

# STR 指令应用举例-1

- (1) 用ARM指令实现 $x=(a+b)-c$

```
LDR    R4, =a           ; 变量a处的地址装入R4中
LDR    R0, [R4]          ; a→R0
LDR    R5, =b           ; 变量b处的地址装入R5中
LDR    R1, [R5]          ; b→R1
ADD    R3, R0, R1        ; R0+R1 →R3, 即a+b →R3
LDR    R4, =c           ; 变量c处的地址装入R4中
LDR    R2, [R4]          ; c→R2
SUB    R3, R3, R2        ; R3-R2 →R3, 即(a+b)-c →R3
LDR    R4, =x           ; 变量x处的地址装入R4中
STR    R3, [R4]          ; 存x
```

# STR 指令应用举例-2

- (2) 用ARM指令实现  $x = a \times (b+c)$

ADR	R4, b	; 变量b处的地址装入R4中
LDR	R0, [R4]	; 取b
ADR	R4, c	; 变量c处的地址装入R4中
LDR	R1, [R4]	; 取c
ADD	R2, R0, R1	; $R0+R1 \rightarrow R2$ , 即 $b+c \rightarrow R2$
ADR	R4, a	; 变量a处的地址装入R4中
LDR	R0, [R4]	; 取a
MUL	R2, R2, R0	; $R2 \times R0 \rightarrow R2$ , 即 $(b+c) \times a \rightarrow R2$
ADR	R4, x	; 变量x处的地址装入R4中
STR	R2, [R4]	; 存x

# STRB 字节数据存储指令

- 格式: STR{<cond>}B <Rd>, <addr>
- 功能: Rd低8位字节数据  $\rightarrow$  addr内存地址
- 其他用法同STR指令。
- 例如:

假设R1=0x40003000 R0=0x1A27E4F3

**STRB R0, [R1, #4] !**

- (1) 存储器地址=R1+#4=0x40003000+#4=0x40003004
- (2) R0低8位字节数据 $\rightarrow$ [0x40003004]= 0xF3
- (3) 修改R1 R1+#4  $\rightarrow$  R1= 0x40003004



# STRH 半字数据存储指令

- 格式: **STR{<cond>}H <Rd>, <addr>**
- 功能: **Rd**低16位半字数据→**addr**内存地址
- 说明: **addr**所表示的地址必须是半字对齐的。
- 其他用法同**STR**指令。
- 例如:

假设**R1=0x40003000**   **R0=0x1A27E4F3**

**STRH R0, [R1, #4] !**

(1) 存储器地址 = **R1+#4=0x40003000+#4=0x40003004**

(2) **R0 → [0x40003004] = 0xE4F3**

(3) 修改**R1**     **R1+#4 → R1 = 0x40003004**

# STRT 用户模式的字数据存储器指令

- 格式: STR{<cond>}T <Rd>, <addr>
- 功能: 同STR指令。
- 说明: 无论处理器处于何种模式, 该指令都将被当作一般用户模式下的内存操作。

# STRBT 用户模式的字节数据存储指令

- 格式: STR{<cond>}BT <Rd>, <addr>
- 功能: 同STRB指令。
- 说明: 无论处理器处于何种模式, 该指令都将被当作一般用户模式下的内存操作。

# LDM 批量数据加载指令

- 格式:

LDM{<cond>}{<type>} <Rn> {!}, <regs> {^}

- 功能: 连续存储单元→寄存器 (多个)

- 该指令一般用于多个寄存器数据的出栈。

- 格式说明:

- type字段种类: 8种。

- Rn: 基址寄存器, 其值是内存单元的起始地址。不允许为R15。

- Regs: 寄存器列表, 从最低序号寄存器开始, 与书写顺序无关。

- !后缀: 指令执行完毕后, 将最后的地址写入基址寄存器。

- ^后缀: 当regs中不包含PC时, 该后缀用于指示指令所用的寄存器为用户模式下的寄存器。

否则, 指示指令执行时, SPSR →CPSR。

# type字段种类

## • type字段种类:

### (1) 拷贝

IA: 每次传送后地址加	LDMIA (内存块读)	STMIA (内存块写)
IB: 每次传送前地址加	LDMIB (内存块读)	STMIB (内存块写)
DA: 每次传送后地址减	LDMDA (内存块读)	STMDA (内存块写)
DB: 每次传送前地址减	LDMDB (内存块读)	STMDB (内存块写)

### (2) 堆栈

FD: 满递减堆栈	LDMFD (出栈)	STMFD (进栈)
ED: 空递减堆栈	LDMED (出栈)	STMED (进栈)
FA: 满递增堆栈	LDMFA (出栈)	STMFA (进栈)
EA: 空递增堆栈	LDMEA (出栈)	STMEA (进栈)

# IA/IB/DA/DB举例

- 例如 **LDMIA R13!, {R0-R1, R2}**

**LDMIA R13!, {R0, R1, R2}**

高地址	0x12345694	36338832
	0x12345690	14543862
	0x1234568C	15548545
	0x12345688	54693645
R13	0x12345684	66663333
	0x12345680	00008888
	0x1234567C	59595959
低地址	0x12345678	26262626

LDMIA执行前，  
内存情况

高地址	0x12345694	36338832	
	0x12345690	14543862	
	0x1234568C	15548545	R2
	0x12345688	54693645	R1
	0x12345684	66663333	R0
	0x12345680	00008888	
	0x1234567C	59595959	
低地址	0x12345678	26262626	

LDMIA执行后  
内存情况

# IA/IB/DA/DB举例

- 例如 **LDMIB R13!, {R0-R1, R2}**

LDMIB R13!, {R0, R1, R2}

高地址	0x12345694	36338832
	0x12345690	14543862
	0x1234568C	15548545
	0x12345688	54693645
R13	0x12345684	66663333
	0x12345680	00008888
	0x1234567C	59595959
低地址	0x12345678	26262626

LDMIB执行前，  
内存情况

高地址	0x12345694	36338832	
R13	0x12345690	14543862	R2
	0x1234568C	15548545	R1
	0x12345688	54693645	R0
	0x12345684	66663333	
	0x12345680	00008888	
	0x1234567C	59595959	
低地址	0x12345678	26262626	

LDMIB执行后  
内存情况

# IA/IB/DA/DB举例

- 例如 **LDMDA R13!, {R0-R1, R2}**

**LDM DA R13!, {R0, R1, R2}**

高地址	0x12345694	36338832
	0x12345690	14543862
	0x1234568C	15548545
	0x12345688	54693645
R13	0x12345684	66663333
	0x12345680	00008888
	0x1234567C	59595959
低地址	0x12345678	26262626

**LDMDA执行前，  
内存情况**

高地址	0x12345694	36338832	
	0x12345690	14543862	
	0x1234568C	15548545	
	0x12345688	54693645	
	0x12345684	66663333	R2
	0x12345680	00008888	R1
	0x1234567C	59595959	R0
R13	0x12345678	26262626	
低地址	0x12345674		

**LDMDA执行后  
内存情况**



# IA/IB/DA/DB举例

- 例如 **LDMDB R13!, {R0-R1, R2}**

**LDMDB R13!, {R0, R1, R2}**

高地址	0x12345694	36338832
	0x12345690	14543862
	0x1234568C	15548545
	0x12345688	54693645
R13 →	0x12345684	66663333
	0x12345680	00008888
	0x1234567C	59595959
低地址	0x12345678	26262626

**LDMDB执行前，  
内存情况**

高地址	0x12345694	36338832	
	0x12345690	14543862	
	0x1234568C	15548545	
	0x12345688	54693645	
	0x12345684	66663333	
	0x12345680	00008888	→ R2
	0x1234567C	59595959	→ R1
低地址	0x12345678	26262626	→ R0

**LDMDB执行后  
内存情况**

# FD/ED/FA/EA

- **FD、ED、FA和EA**指定是满栈还是空栈，是升序栈还是降序栈，用于堆栈寻址。
- **满堆栈**：栈指针指向上次写的最后一个数据单元。
- **空堆栈**：栈指针指向第一个空闲单元。
- 降序栈在内存中反向增长；升序栈在内存中正向增长。
- **ARM微处理器支持四种类型的堆栈工作方式**，即：
  - (1) 满递增方式FA (Full Ascending)**：堆栈指针指向最后入栈的数据位置，且由低地址向高地址生成。
  - (2) 满递减方式FD (Full Decending)**：堆栈指针指向最后入栈的数据位置，且由高地址向低地址生成。
  - (3) 空递增方式EA (Empty Ascending)**：堆栈指针指向下一个入栈数据的空位置，且由低地址向高地址生成。
  - (4) 空递减方式ED (Empty Decending)**：堆栈指针指向下一个入栈数据的空位置，且由高地址向低地址生成。

# FD/ED/FA/EA举例

0X12345694	36338832
0X12345690	14543862
0X1234568c	15548545
0X12345688	54693645
0X12345684	66663333
0X12345680	
0X1234567c	
0X12345678	

栈指针

FD

0X12345694	
0X12345690	
0X1234568c	
0X12345688	54693645
0X12345684	66663333
0X12345680	00008888
0X1234567c	59595959
0X12345678	26262626

栈指针

FA

0X12345694	36338832
0X12345690	14543862
0X1234568c	15548545
0X12345688	54693645
0X12345684	66663333
0X12345680	
0X1234567c	
0X12345678	

栈指针

ED

0X12345694	
0X12345690	
0X1234568c	
0X12345688	54693645
0X12345684	66663333
0X12345680	00008888
0X1234567c	59595959
0X12345678	26262626

栈指针

EA

# LDMFD实验例 (出栈)

ARM7TDMI - Registers

Register	Value
Current	{...}
r0	0x00000000
r1	0x00000000
r2	0x00000000
r3	0x00000000
r4	0x00000000
r5	0x00000000

ARM7TDMI - D:\ARM\Exp01\Exp01.s

```
1 AREA Word, CODE, READONLY
2 T1 EQU 0x40001000
3 ENTRY
4 start
5 LDR R0, =T1
6 MOV R13, R0
7 LDMFD R13!, {R0, R1, R2}
8 END
```

执行LDMFD R13!,{R0,R1,R2}后

ARM7TDMI - Registers

Register	Value
Current	{...}
r0	0x11223344
r1	0x55667788
r2	0xAABBCCDD
r3	0x00000000
r4	0x00000000
r5	0x00000000

ARM7TDMI - Memory Start address 0x40001000

Tab1 - Hex - No prefix

Tab2 - Hex - No prefix

Tab3 - Hex - No prefix

Address	0	4	8	c
0x40001000	11223344	55667788	AABBCCDD	E800E800
0x40001010	E7FF0010	E800E800	E7FF0010	E800E800
0x40001020	E7FF0010	E800E800	E7FF0010	E800E800
0x40001030	E7FF0010	E800E800	E7FF0010	E800E800

0x40001000的值

0x40001004的值

0x40001008的值

# LDMFD实验例-调整寄存器列表顺序

执行LDMFD R13!,{R2,R1,R0}后

ARM7TDMI - Registers

Register	Value
Current	{...}
r0	0x11223344
r1	0x55667788
r2	0xAABBCCDD
r3	0x00000000

注意：寄存器列表顺序改变

```
ARM7TDMI - D:\ARM\Exp01\Exp01.s
1      AREA Word, CODE, READONLY
2      T1 EQU 0x40001000
3      ENTRY
4      start
5      LDR R0, =T1
6      MOV R13, R0
7      LDMFD R13!, {R0, R1, R2}
8      END
```

执行LDMFD R13!,{R0,R1,R2}后

ARM7TDMI - Registers

Register	Value
Current	{...}
r0	0x11223344
r1	0x55667788
r2	0xAABBCCDD
r3	0x00000000
r4	0x00000000
r5	0x00000000

ARM7TDMI - Memory Start address 0x40001000

Tab1 - Hex - No prefix		Tab2 - Hex - No prefix		Tab3 - Hex - 1	
Address	0	4	8	c	
0x40001000	11223344	55667788	AABBCCDD	E800E800	
0x40001010	E7FF0010	E800E800	E7FF0010	E800E800	
0x40001020	E7FF0010	E800E800	E7FF0010	E800E800	
0x40001030	E7FF0010	E800E800	E7FF0010	E800E800	

0x40001000的值

0x40001004的值

0x40001008的值

# LDM指令操作的伪代码

- LDM {<cond>} <addressing\_mode> <Rd> {!}, <registers>

- 指令操作的伪代码:

if ConditionPassed(cond) then

    adderss = start\_address

    for i = 0 to 14

        if registers\_list[i] == 1 then

            Ri = Memory[adderss,4]

            adderss = address + 4

if registers\_list[15] == 1 then

    value = Memory[adderss,4]

    if (architecture version 5 or above) then

        PC = value AND 0xFFFFFFF0

        T Bit = value[0]

    else

        PC = value AND 0xFFFFFFF0

        adderss = address + 4

assert end\_adderss = address - 4

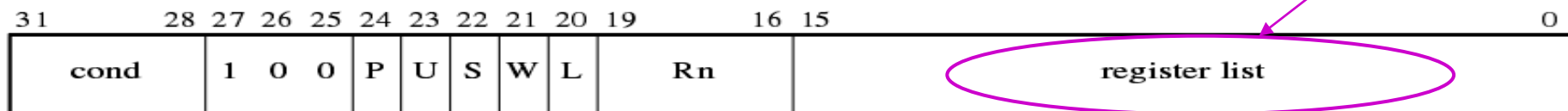
即8种type字段种类

; 从R0开始检查并设置

; 广义修正

; R15是PC

R15-R0的序号



# STM 批量数据存储指令

- **格式:** STM {<cond>} {<type>} <Rn> {!}, <regs> {^}
- **功能:** 寄存器列表的值→连续的内存单元
- **Rn:** 基址寄存器, 其值为内存单元的起始地址为。
- **Regs:** 寄存器列表。
- 该指令一般用于多个寄存器数据的**入栈**。
- **注意:** **最高序号的寄存器数据最先进栈**。
- 其他参数用法同LDM指令。
- **例如:**
  - STMEA R13!, {R0-R12, PC} ; 将寄存器R0~R12以及程序计数器  
; PC的值保存到R13指示的堆栈中

# STM 入栈举例

ARM7TDMI - Registers

Register	Value
Current	{...}
r0	0x11223344
r1	0x55667788
r2	0xAABBCCDD
r3	0x00000000
r4	0x00000000
r5	0x00000000
r6	0x00000000
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000
r11	0x00000000
r12	0x00000000
r13	0x40001000
r14	0x00000000
pc	0x40000014

ARM7TDMI - D:\ARM\Exp01\Exp01.s

1	AREA Word, CODE, READONLY
2	T1 EQU 0x40001000
3	T2 EQU 0x11223344
4	T3 EQU 0x55667788
5	T4 EQU 0xAABBCCDD
6	
7	ENTRY
8	start
9	LDR R0, =T1
10	MOV R13, R0
11	LDR R0, =T2
12	LDR R1, =T3
13	LDR R2, =T4
14	<u>STMFD R13!, {R0, R1, R2}</u>
15	END

最高序号的寄存器数据最先进栈

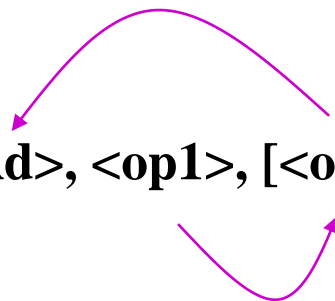
ARM7TDMI - Memory Start addr: 0x40000ff0

Tab1 - Hex - No prefix		Tab2 - Hex - No prefix		Tab3 - Hex - No	
Address	0	4	8	c	
0x40000FF0	00000000	11223344	55667788	AABBCCDD	
0x40001000	00000000	00000000	00000000	00000000	
0x40001010	E7FF0010	E800E800	E7FF0010	E800E800	



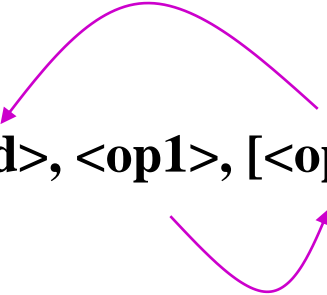
# SWP 字数据交换指令

- 格式: SWP {<cond>} <Rd>, <op1>, [<op2>]



- 功能: Rd=[op2]
- [op2]=op1
- 从op2所表示的内存装载一个字并把这个字放置到目的寄存器Rd中, 然后把寄存器op1的内容存储到同一内存地址中。
- op1、op2: 均为寄存器。
- 例如:
- SWP R0, R1, [R2] ; [R2]字数据→R0, R1字数据→[R2]

# SWPB 字节数据交换指令

- 格式: SWP {<cond>} B <Rd>, <op1>, [<op2>]
- 功能: [op2]一个字节→Rd低8位, Rd的高24位清0  
op1低8位数据→[op2]
- 例如:
- SWPB R0, R1, [R2] ; [R2]一个字节数据→R0低8位  
; R1低8位字节数据→[R2]

## 3.2.6 ARM 协处理器指令

- ARM处理器最多可支持16个协处理器，用于辅助ARM完成各种协处理操作。
- 在程序执行过程中，各协处理器只执行自身的协处理指令，而忽略属于ARM处理器和其他协处理器的指令。
- **ARM协处理器指令可分为3类：**
  - (1) ARM处理器用于初始化协处理器的数据操作指令（**CDP**）。
  - (2) 协处理器寄存器和内存单元之间的数据传送指令（**LDC, STC**）。
  - (3) ARM处理器寄存器和协处理器寄存器之间的数据传送指令（**MCR, MRC**）。

# CDP 协处理器数据操作指令

- 格式:
- CDP {条件} 协处理器编码, 协处理器操作码1, 目的寄存器, 源寄存器1, 源寄存器2, 协处理器操作码2
- 功能: 用于ARM处理器通知ARM协处理器执行特定的操作, 若协处理器不能成功完成特定的操作, 则产生未定义指令异常。其中协处理器操作码1和协处理器操作码2为协处理器将要执行的操作, 目的寄存器和源寄存器均为协处理器的寄存器, 指令不涉及ARM处理器的寄存器和存储器。

- 指令示例:

CDP P1, 2, C1, C2, C3

; 命令P1 (1号) 协处理器, 把自己的寄存器C2和寄存器C3作为操作数, 进行第“2”方式的操作, 结果放在寄存器C1中。

; 即, P1: C2 操作2 C3 → C1

# LDC 协处理器加载指令

- 格式:

LDC {条件} {L} 协处理器编码, 目的寄存器, [源寄存器]

- 功能: 协处理器的目的寄存器  $\leftarrow$  [ARM的源寄存器]间址存储器字单元。
- 用于将源寄存器所指向的存储器中的字数据传送到目的寄存器中, 若协处理器不能成功完成传送操作, 则产生未定义指令异常。
- {L}选项: 表示指令为长读取操作, 如用于双精度数据的传输。
- 指令示例:

LDC P3, C4, [R5]

; 将ARM处理器的寄存器R5所指向的存储器中的字数据传送到协处理器P3的寄存器C4中。

# STC 协处理器存储指令

- 格式:

STC {条件} {L} 协处理器编码, 源寄存器, [目的寄存器]

- 功能: 协处理器源寄存器字数据 → [ARM目的寄存器]间址存储器字单元。
- 用于将源寄存器中的字数据传送到目的寄存器所指向的存储器中, 若协处理器不能成功完成传送操作, 则产生未定义指令异常。
- {L}选项: 表示指令为长读取操作, 如用于双精度数据的传输。
- 指令示例:

STCEQ P2, C4, [R5]

; 当Z=1时, 执行将协处理器P2的寄存器C4中的字数据传送到ARM处理器的寄存器R5所指向的存储器中。

# MCR和MRC指令

- 格式:

**MCR/MRC {条件} 协处理器编码, 协处理器操作码1, 目的寄存器,  
源寄存器1, 源寄存器2, 协处理器操作码**

- **MCR功能:** 协处理器寄存器 $\leftarrow$ ARM处理器寄存器。
- **MRC功能:** ARM处理器寄存器 $\leftarrow$ 协处理器寄存器。
- ARM处理器指定一个寄存器作为传送数据的源或接收数据的目标。
- 协处理器指定两个寄存器, 同时可以像CDP指令一样指定操作要求。
- 指令示例:

**MCR P2, 3, R2, C4, C5, 6**

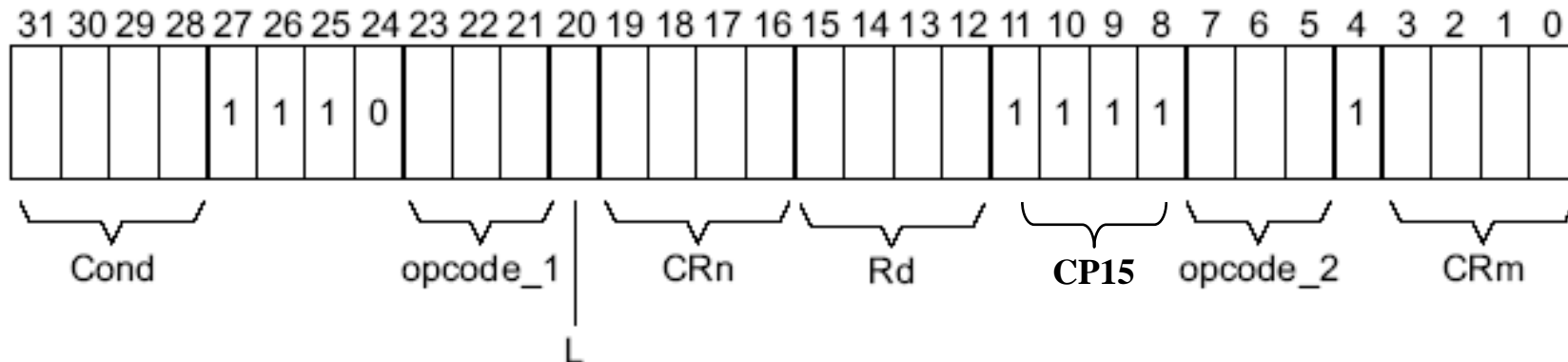
; 指定协处理器P2执行第3种操作, 类型6, 操作数是 R2, 结果放在C4中。

**MRC P0, 3, R2, C4, C5, 6**

; 指定协处理器P0执行第3种操作, 类型6, 操作数C4和C5, 结果送给R2 。

# ARM 920T的协处理器

- ARM 920T 只有两个协处理器： CP14、 CP15
- CP14: 调试通信通道协处理器DCC，提供了两个32bits寄存器用于传送数据。
- CP15: 系统控制协处理器，配置和控制caches、MMU、保护系统、时钟模式。有16个寄存器： CR0-CR15。
- CP15寄存器只能被MRC和MCR指令访问。
- MRC/MCR {cond} coproc, opcode1, Rd, CRn, CRm {,opcode2}  
L用于区分MRC(L=1)和MCR(L=0)





# CP15寄存器CR0

- 寄存器CR0有2个，具体被访问的寄存器由Opcode\_2字段的值决定。

## ID编码寄存器

该寄存器为只读寄存器，返回32位的设备ID编码。  
通过读CP15寄存器0并将opcode\_2字段设置0可以访问ID编码寄存器。例如：

MRC p15, 0, Rd, c0, c0, 0 ;返回ID寄存器

## Cache类型寄存器

该寄存器为只读寄存器，它包含ICache和DCache的大小以及体系结构的相关信息。

通过读CP15寄存器0并将opcode\_2字段设置为1可以访问Cache类型寄存器，例如：

MRC p15, 0, Rd, c0, c0, 1 ;返回Cache的详细资料

## 3.2.7 ARM 软件中断指令

- ARM处理器支持两条异常中断指令：  
    软件中断指令SWI  
    断点中断指令BKPT
- ARM指令集中的软件中断指令是唯一一条不使用寄存器的ARM指令，也是一条可以条件执行的指令。
- 因为ARM指令在用户模式中受到很大的局限，有一些资源不能够访问。所以，在需要访问这些资源时，使用软件控制的唯一方法就是使用软件中断指令。

# SWI 软件中断指令

- 格式: SWI {<cond>} 24位的立即数
- 功能: 产生软件中断, 调用系统例程。
- 24位的立即数: 指定用户程序调用系统例程的类型, 其参数通过通用寄存器传递。
- 当24位的立即数被忽略时, 系统例程类型由寄存器R0指定, 其参数通过其他通用寄存器传递。
- 例如:
- SWI 0X05 ; 调用编号为05的系统例程。

# SWI指令操作的伪代码

- SWI {<cond>}<immed\_24>

- 指令操作的伪代码:

if ConditionPassed(cond) then

    R14\_svc = address of next instruction after the SWI instruction

    SPSR\_svc = CPSR

    CPSR[4:0] = 0b10011 /\*进入管理模式\*/

    CPSR[5] = 0 /\* Execute in ARM state, T Bit = 0\*/

        /\* CPSR[6] is unchanged, fiq不变\*/

    CPSR[7] = 1 /\* Disable normal interrupt, I Bit = 1, 禁止irq \*/

if high vectors configured then

    PC = 0xFFFF0008

else

    PC = 0x00000008

异常类型	工作模式	特定地址（低端）	特定地址（高端）	优先级
复位	管理模式	0x00000000	0xFFFF0000	1
未定义指令	未定义指令中止模式	0x00000004	0xFFFF0004	6
软件中断（SWI）	管理模式	0x00000008	0xFFFF0008	6

# SWI的参数传递

- 使用SWI指令时，通常使用以下两种方法进行参数传递：
  - (1) 指令中使用24位的立即数（非0），参数通过通用寄存器传递。此时，24位的立即数指定了用户请求的服务类型。
    - 例：  

MOV R0, #34	； 设置子功能号为34，作为入口参数
SWI 12	； 调用12号软中断
    - (2) 指令中的24位立即数被忽略（为0），用户请求的服务类型由寄存器R0的值决定，参数通过其他的通用寄存器传递。
      - 例：  

MOV R0, #12	； 调用12号软中断
MOV R1, #34	； 设置子功能号为34
SWI 0	； 软中断，中断立即数为0

# SWI的操作如何获取立即数

- SWI中断处理程序要通过读取SWI指令，来获取24位立即数。
- 步骤：
  - (1) 确定引起中断的SWI指令是ARM指令还是Thumb指令，可由SPSR得到。
  - (2) 取SWI指令的地址，可由LR得到。
  - (3) 读出指令，分解出立即数。

- 例：

T\_bit EQU 0x20

## SWI\_Handler

STMFD SP!, {R0-R3, R12, LR}	; 现场保护
MRS R0, SPSR	; 读取SPSR
TST R0, #T_bit	; 测试T标志位
LDRNEH R0, [LR, # -2]	; 若是Thumb指令，读取16位指令码
BICNE R0, R0, #0xFF00	; 对应位清0，取Thumb指令的8位立即数
LDREQ R0, [LR, # -4]	; 若是ARM指令，读取32位指令码
BICEQ R0, R0, #0xFF000000	; 对应位清0，取ARM指令的24位立即数

...

LDMFD SP!, {R0-R3, R12, PC}^	; 恢复现场，同时SPSR→CPSR
------------------------------	--------------------

# BKPT 断点中断指令

- 格式: BKPT 16位的立即数
- 功能: 产生软件断点中断, 用于软件调试。
- 16位的立即数用于保存软件调试中额外的断点信息。
- 例:

**BKPT**

**BKPT 0xF02C**

## 3.2.8 程序状态寄存器指令

- 作用：状态寄存器和通用寄存器间传送数据。
- 总共有两条指令：MRS和MSR。
- 两者结合可用来修改程序状态寄存器的值。



# MRS 程序状态寄存器到通用寄存器的数据传送指令

- 格式: **MRS{<cond>} <Rd>, CPSR/SPSR**
- 功能: **CPSR/SPSR → Rd**
- 用于以下场合保存状态:
  - 进入中断服务程序
  - 进程切换
- 例如:
- **MRS R0, CPSR** ; 状态寄存器CPSR→R0

# MSR 通用寄存器到程序状态寄存器的数据传送指令

- 格式:  $\text{MSR}\{\text{<cond>}\} \text{CPSR/SPSR\_}<\text{field}>, <\text{op1}>$
- 功能:  $\text{CPSR/SPSR\_field} \leftarrow \text{op1}$
- **op1**: 通用寄存器、立即数
- 用于以下场合恢复状态:

退出中断服务程序  
进程切换

- 例如:

**MSR CPRS\_f, R0** ; R0 → CPRS (只修改条件域)

**MRS CPSR\_fsxc, #5** ; #5 → CPRS

状态寄存器位	Field域	标识
位[31:24]	条件标志域	f
位[23:16]	状态位域	s
位[15:8]	扩展位域	x
位[7:0]	控制位域	c

## 3.3 Thumb指令集

- Thumb指令集可以认为是ARM指令集的子集。
- Thumb指令长度为**16位**，但其数据处理指令的操作数仍然是32位的，指令寻址地址也是32位的。
- 由于是压缩的指令，因此，在ARM指令流水线中实现Thumb指令时，**先动态解压缩**，然后作为标准的ARM指令来执行。
- 如何区分指令取决于CPSR中的第五位：**T**（T=0 为ARM状态，T=1 为Thumb状态）。
- **Thumb指令集中没有**：乘加指令、64位乘法指令、协处理器指令、数据交换指令、程序状态寄存器指令，而且指令的第二操作数受到限制，除了**跳转指令B有条件执行**功能外，其他指令均为无条件执行。
- Thumb指令系统不是完整的指令体系，必须与ARM配合使用。
- **Thumb指令集有4大类**：
  - 数据处理指令
  - 跳转指令
  - Load/Store指令
  - 软件中断指令

# Thumb指令集与ARM指令集区别

- Thumb指令集与ARM指令集在以下几个方面有区别：
- 跳转指令。条件跳转在范围上有更多的限制，转向子程序只具有无条件转移B。
- 数据处理指令。对通用寄存器进行操作，操作结果需放入其中一个操作数寄存器，没有第三个寄存器。
- 单寄存器加载和存储指令。Thumb状态下，单寄存器加载和存储指令只能访问寄存器R0~R7。
- 批量寄存器加载和存储指令。LDM和STM指令可以将任何范围为R0~R7的寄存器子集加载或存储，PUSH和POP指令使用堆栈指针R13作为基址实现满递减堆栈，除R0~R7外，PUSH指令还可以存储链接寄存器R14，并且POP指令可以加载程序指令PC。

# 数据处理指令1

格 式	功 能
MOV Rd,imm_8	Rd=imm_8; <b>Rd为R0~R7</b> , imm_8为8位立即数
MOV Rd,Rn	Rd=Rn; Rd、Rn为R0~R15
MVN Rd,Rn	Rd= $\sim$ Rn; Rd、Rn为R0~R7
NEG Rd,Rn	Rd=-Rn; Rd、Rn为R0~R7
ADD Rd,Rn,imm	Rd=Rn+imm; Rd为R0~R7, Rn为R0~R7或PC或SP。Rn为PC或SP时, imm为10位立即数; 否则, imm为3位立即数
ADD Rd,Rn,Rm	Rd=Rn+Rm; Rd、Rn、Rm为R0~R7
ADD Rd,imm	Rd=Rd+imm; Rd为R0~R7或SP Rd为SP时, imm为-508~+508间的4整数倍的数; 否则, imm为8位立即数
ADD Rd,Rn	Rd=Rd+Rn; Rd、Rn为R0~R15
ADC Rd,Rn	Rd=Rd+Rn+carry; Rd、Rn为R0~R7, carry为进位标志值
SUB Rd,Rn,imm_3	Rd=Rn-imm_3; Rd、Rn为R0~R7, imm_3为3位立即数
SUB Rd,Rn,Rm	Rd=Rn-Rm; Rd、Rn、Rm为R0~R7,
SUB Rd,imm	Rd=Rd-imm; Rd为R0~R7或SP Rd为SP时, imm为-508~+508间的4整数倍的数; 否则, imm为8位立即数
SBC Rd,Rn	Rd=Rd-Rn-!carry; Rd、Rn为R0~R7, carry为进位标志值
MUL Rd,Rn	Rd=Rd $\times$ Rn; Rd、Rn为R0~R7

# 数据处理指令2

格 式	功 能
AND Rd,Rn	$Rd = Rd \& Rn$ ; Rd、Rn为R0~R7
ORR Rd,Rn	$Rd = Rd   Rn$ ; Rd、Rn为R0~R7
EOR Rd,Rn	$Rd = Rd \wedge Rn$ ; Rd、Rn为R0~R7
BIC Rd,Rn	$Rd = Rd \& (\sim Rn)$ ; Rd、Rn为R0~R7
ASR Rd,Rn	$Rd = Rd$ 算术右移Rn位; Rd、Rn为R0~R7
ASR Rd,Rn,imm_5	$Rd = Rn$ 算术右移imm_5位; Rd、Rn为R0~R7, imm_5为1~32之间的数值
LSL Rd,Rn	$Rd = Rd$ 逻辑左移Rn位; Rd、Rn为R0~R7
LSL Rd,Rn,imm_5	$Rd = Rn$ 逻辑左移imm_5位; Rd、Rn为R0~R7
LSR Rd,Rn	$Rd = Rd$ 逻辑右移Rn位; Rd、Rn为R0~R7
LSR Rd,Rn,imm_5	$Rd = Rn$ 逻辑右移imm_5位; Rd、Rn为R0~R7
ROR Rd,Rn	$Rd = Rd$ 循环右移Rn位; Rd、Rn为R0~R7
CMP Rn,Rm	根据Rn-Rm的值, 修改CPSR的状态标志位; Rn、Rm为R0~R7
CMP Rn,imm_8	根据Rn-imm_8的值, 修改CPSR的状态标志位; Rn为R0~R7
CMN Rn,Rm	根据Rn+Rm的值, 修改CPSR的状态标志位; Rn、Rm为R0~R7
TST Rn,Rm	根据Rn&Rm的值, 修改CPSR的状态标志位; Rn、Rm为R0~R7

# 跳转指令

格 式	功 能
<b>B{cond} label</b>	<b>PC=label</b> 若有cond，则label必须在当前指令的-256~+256字节范围内； 否则，label必须在当前指令的-2KB~+2KB范围内
<b>BL label</b>	<b>R14=PC+4, PC=label</b> label必须在当前指令的-4MB~+4MB范围内
<b>BX Rn</b>	<b>PC=Rn</b> ，且切换处理器状态

# Load/Store指令

格 式	功 能
LDR Rd,[Rn,imm]	Rd=地址 (Rn+imm) 中的字数据; Rd为R0~R7, Rn为R0~R7或SP或PC; 若Rn为PC或SP, imm为5位立即数, 否则imm为8位立即数
LDR Rd,[Rn,Rm]	Rd=地址 (Rn+Rm) 中的字数据; Rd、Rn、Rm为R0~R7
LDRH Rd,[Rn,imm_5]	Rd=地址 (Rn+imm_5) 中的无符号半字数据; Rd、Rn为R0~R7, imm_5为5位立即数
LDRH Rd,[Rn,Rm]	Rd=地址 (Rn+Rm) 中的无符号半字数据; Rd、Rn、Rm为R0~R7
LDRB Rd,[Rn,imm_5]	Rd=地址 (Rn+imm_5) 中的无符号字节数据; Rd、Rn为R0~R7
LDRB Rd,[Rn,Rm]	Rd=地址 (Rn+Rm) 中的无符号字节数据; Rd、Rn、Rm为R0~R7
LDRSH Rd,[Rn,Rm]	Rd=地址 (Rn+Rm) 中的有符号半字数据; Rd、Rn、Rm为R0~R7
LDRSB Rd,[Rn,Rm]	Rd=地址 (Rn+Rm) 中的有符号字节数据; Rd、Rn、Rm为R0~R7
LDR Rd,label	Rd=地址 (label) 中的字数据; Rd为R0~R7
STR Rd,[Rn,imm]	地址 (Rn+imm) 处的字数据=Rd; Rd为R0~R7, Rn为R0~R7或SP或PC; 若Rn为PC或SP, imm为5位立即数, 否则imm为8位立即数
STR Rd,[Rn,Rm]	地址 (Rn+Rm) 处的字数据=Rd; Rd、Rn、Rm为R0~R7
STRH Rd,[Rn,imm_5]	地址 (Rn+imm_5) 处的无符号半字数据=Rd; Rd、Rn为R0~R7
STRH Rd,[Rn,Rm]	地址 (Rn+Rm) 处的无符号半字数据=Rd; Rd、Rn、Rm为R0~R7
STRB Rd,[Rn,imm_5]	地址 (Rn+imm_5) 处的无符号字节数据=Rd; Rd、Rn为R0~R7
STRB Rd,[Rn,Rm]	地址 (Rn+Rm) 处的无符号字节数据=Rd; Rd、Rn、Rm为R0~R7
LDMIA Rd{!},regs	regs=以Rd为起始地址的连续字数据; regs为寄存器列表
STMIA Rd{!},regs	以Rd为起始地址的连续字数据=regs; regs为寄存器列表
PUSH regs{,LR}	[SP...]=regs{, LR}; LR即R14, SP即R13
POP regs{,PC}	regs{, PC}=[SP...]; PC即R15, SP即R13



# 软件中断指令

格 式	功 能
SWI 8位立即数	8位立即数为中断号

# 第3章 结 束