



# ARM指令集

李曦

lxx@ustc.edu.cn

计算机系计算机应用研究室

2006年9月



# 内容提要

- **ARM指令集**
  - ARM指令集分类与指令格式
  - ARM指令的寻址方式
  - ARM指令集详解
  - Thumb指令及应用
- **ARM程序设计基础**

# 5.1 ARM微处理器的指令集概述



## 5.1.1 ARM微处理器的指令的分类与格式

- Load-store结构
- 指令分类
  - ◆ 数据处理指令 - 使用和改变寄存器的值
  - ◆ 数据传送指令 - 把存储器的值拷贝到寄存器中 (load) 或把寄存器中的值拷贝到存储器中 (store)
  - ◆ 控制流指令
    - 分支
    - 分支和链接, 保存返回的地址, 以恢复最先的次序
    - 陷入系统代码
  - ◆ 程序状态寄存器 (PSR) 处理指令
  - ◆ 协处理器指令
  - ◆ 异常产生指令

# ARM指令集



- 跳转指令：
  - B 跳转指令。
  - BLX 带返回和状态切换的跳转指令。
- 数据处理指令
  - MOV 数据传送指令
  - CMP 比较指令
  - TST 位测试指令
  - ADD 加法指令28
  - SUB 减法指令
  - RSB 逆向减法指令
  - AND 逻辑与指令
  - EOR 逻辑互斥指令
  - BL 带返回的跳转指令。
  - BX 带状态切换的跳转指令。
  - MVN 数据取反传送指令
  - CMN 反值比较指令
  - TEQ 相等测试指令
  - ADC 带进位加法指令
  - SBC 带借位减法指令
  - RSC 带借位的逆向减法指令
  - ORR 逻辑或指令
  - BIC 位清除指令
- 乘法指令与乘加指令
  - MUL 32 位乘法指令
  - SMULL 64 位有符号数乘法指令
  - UMULL 64 位无符号数乘法指令
  - MLA 32 位乘加指令
  - SMLAL 64 位有符号数乘加指令
  - UMLAL 64 位无符号数乘加指令
- 程序状态寄存器存取指令
  - MRS 程序状态寄存器到通用寄存器的数据传送指令。
  - MSR 通用寄存器到程序状态寄存器的数据传送指令。



# ARM指令集(续)

- 寄存器加载/存储指令：

- LDR 字数据加载指令
- LDRH 半字数据加载指令
- STRB 字节数据存储指令
- LDM 连续数据加载指令

- LDRB 字节数据加载指令
- STR 字数据存储指令
- STRH 半字数据存储指令
- STM 连续数据存储指令

- 数据交换指令：

- SWP 字数据交换指令

- SWPB 字节数据交换指令

- 移位元指令：

- LSL 逻辑左移
- LSR 逻辑右移
- ROR 循环右移

- ASL 算术左移
- ASR 算术右移
- RRX 带扩充的循环右移44

- 协处理器指令

- CDP 协处理器数操作指令
- LDC 协处理器数据加载指令
- STC 协处理器数据存储指令46
- 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



## 5.1.2 指令的条件域

- 所有的ARM指令都可以条件执行
  - 指令的执行与否取决于CPSR寄存器的N, Z, C and V标志位
  - 所有的Thumb指令都可以解压成全部条件指令
- 每一条ARM指令包含4位的条件码，位于指令的最高4位[31:28]。条件码共有16种，每种条件码可用两个字符表示，这两个字符可以添加在指令助记符的后面和指令同时使用。
  - 例如，跳转指令B可以加上后缀EQ变为BEQ表示“相等则跳转”，即当CPSR中的Z标志置位时发生跳转。



- 在16种条件标志码中，只有15种可以使用。

# 指令的条件码



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	C置位	无符号数大于或等于
0011	CC	C清零	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



# 条件执行

- 条件执行可避免使用分支指令
- Example

```
CMP r0, #5      ;  
BEQ BYPASS     ; if (r0!=5) {  
  ADD r1, r1, r0 ; r1:=r1+r0-r2  
  SUB r1, r1, r2 ;}  
BYPASS:        ...
```

注意：条件不成立时，该指令被忽略，等于执行了一条NOP。

## 使用条件执行

```
CMP r0, #5      ;  
ADDNE r1, r1, r0;  
SUBNE r1, r1, r2;  
...
```

```
; if ((a==b) && (c==d)) e++;  
  
CMP r0, r1  
CMPEQ r2, r3  
ADDEQ r4, r4, #1
```

Note: add 2 -letter condition after the 3-letter opcode





## 5.1.3 指令格式

- 指令长度
- 指令集可以是以下任一种
  - 32 bits 长 (ARM状态)
  - 16 bits 长 (Thumb 状态)
- ARM7TDMI 支持3种数据类型
  - 字节 (8-bit)
  - 半字 (16-bit)
  - 字 (32-bit)
  - 字必须被排成4个字节边界对齐, 半字必须被排列成2个字节边界对齐



## • 指令格式

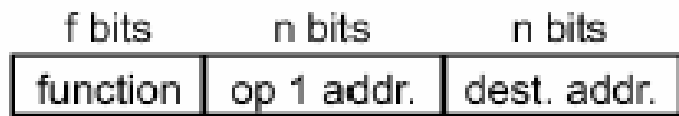
### ◆ 3地址指令格式

➤ 在ARM状态中使用



### ◆ 2地址指令格式

➤ 在ARM和 THUMB 状态下使用





# ARM7TDMI指令集编码

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	5	4	3	0		
Cond	0	0	I	Opcode			S	Rn			Rd			Operand 2						<b>Data Processing</b>		
Cond	0 0 0 0 0 0						A	S	Rd			Rn			Rs	1 0 0 1		Rm			<b>Multiply</b>	
Cond	0 0 0 1 0				B		0 0		Rn			Rd			0 0 0 0		1 0 0 1		Rm		<b>Single Data Swap</b>	
Cond	0	1	I	P	U	B	W	L	Rn			Rd			offset						<b>Single Data Transfer</b>	
Cond	0	1	1	XXXXXXXXXXXXXXXXXXXXXXX																1	XXXX	<b>Undefined</b>
Cond	1	0	0	P	U	S	W	L	Rn			Register List									<b>Block Data Transfer</b>	
Cond	1	0	1	L	offset																<b>Branch</b>	
Cond	1	1	0	P	U	N	W	L	Rn			CRd	CP#	offset						<b>Coproc Data Transfer</b>		
Cond	1	1	1	0	CP Opc			CRn			CRd	CP#	CP	0	CRm			<b>Coproc Data Operation</b>				
Cond	1	1	1	0	CP Opc		L	CRn			Rd	CP#	CP	1	CRm			<b>Coproc Register Transfer</b>				
Cond	1	1	1	1	ignored by processor																<b>Software Interrupt</b>	



## 5.2 ARM指令的寻址方式

- 立即寻址
- 寄存器寻址
- 寄存器间接寻址
- 基址变址寻址
- 多寄存器寻址
- 相对寻址
- 堆栈寻址



## 5.2.1 立即寻址

立即寻址也叫立即数寻址，这是一种特殊的寻址方式，操作数本身就在指令中给出，只要取出指令也就取到了操作数。这个操作数被称为立即数，对应的寻址方式也就叫做立即寻址。

例如以下指令：

ADD R0, R0, #1 ;  $R0 \leftarrow R0 + 1$

ADD R0, R0, #0x3f ;  $R0 \leftarrow R0 + 0x3f$

第二个源操作数即为立即数，要求以“#”为前缀，对于以十六进制表示的立即数，还要求在“#”后加上“0x”或“&”。



# 有效立即数问题

- 在**32**位指令编码中存放**32**位立即数的方法是：
  - 在**ARM**数据处理指令中，第二操作数域**12**位
  - 当参与操作的第二操作数为立即数型时，每个立即数都是采用一个**8**位的常数循环右移偶数位而间接得到，其中循环右移的位数由一个**4**位二进制的两倍表示
  - 如果立即数记作**<immediate>**，**8**位常数记作**immed\_8**，**4**位的循环右移值记作**rotate\_imm**，有效的立即数是由一个**8**位的立即数循环右移偶数位得到，可以表示成：

**<immediate>=immed\_8 循环右移 (2×rotate\_imm)**



## 5.2.2 寄存器寻址

寄存器寻址就是利用寄存器中的数值作为操作数，这种寻址方式是各类微处理器经常采用的一种方式，也是一种执行效率较高的寻址方式。以下指令：

ADD R0, R1, R2 ; R0 ← R1 + R2

## 5.2.3 寄存器间接寻址

寄存器间接寻址：以寄存器中的值作为操作数的地址，而操作数本身存放在存储器中。

ADD R0, R1, [R2] ; R0 ← R1 + [R2]

LDR R0, [R1] ; R0 ← [R1]

STR R0, [R1] ; [R1] ← R0



## • 5.2.4 基址变址寻址

将寄存器（该寄存器一般称作基址寄存器）的内容与指令中给出的地址偏移量相加，从而得到一个操作数的有效地址。变址寻址方式常用于访问某基址地址附近的地址单元。

采用变址寻址方式的指令常见有以下几种形式：

前变址、自动变址、后变址、偏移变址

LDR R0, [R1, #4] ;  $R0 \leftarrow [R1+4]$

LDR R0, [R1, #4]! ;  $R0 \leftarrow [R1+4]$ 、 $R1 \leftarrow R1+4$

LDR R0, [R1], #4 ;  $R0 \leftarrow [R1]$ 、 $R1 \leftarrow R1+4$

LDR R0, [R1, R2] ;  $R0 \leftarrow [R1+R2]$





## 5.2.5 多寄存器寻址

采用多寄存器寻址方式，一条指令可以完成多个寄存器值的传送。这种寻址方式可以用一条指令完成传送最多16个通用寄存器的值。

以下指令：

```
LDMIA R0, {R1, R2, R3, R4}      ; R1 ← [R0]
                                   ; R2 ← [R0 + 4]
                                   ; R3 ← [R0 + 8]
                                   ; R4 ← [R0 + 12]
```

该指令的后缀IA表示在每次执行完加载/存储操作后，R0按字长度增加，因此，指令可将连续存储单元的值传送到R1~R4。



## 5.2.6 相对寻址

与基址变址寻址方式相类似，相对寻址以程序计数器PC的当前值为基地址，指令中的地址标号作为偏移量，将两者相加之后得到操作数的有效地址。

以下程序段完成子程序的调用和返回，跳转指令BL采用了相对寻址方式：

```
BL    NEXT           ; 跳转到子程序NEXT处执行
```

```
.....
```

```
NEXT:
```

```
.....
```

```
MOV   PC, LR       ; 从子程序返回
```



## 5.2.7 堆栈寻址

堆栈是一种数据结构，按先进后出（First In Last Out, FILO）的方式工作，使用一个称作堆栈指针的专用寄存器指示当前的操作位置，堆栈指针总是指向栈顶。

按SP指向的位置，分：

满堆栈(Full Stack)：当堆栈指针指向最后压入堆栈的数据时；

空堆栈(Empty Stack)：当堆栈指针指向下一个将要放入数据的空位置时；

根据堆栈的生成方式，分：

递增堆栈(Ascending Stack)：当堆栈由低地址向高地址生成时；

递减堆栈(Decending Stack)：当堆栈由高地址向低地址生成时；

# ARM微处理器支持的堆栈工作方式



- 满递增堆栈FA
  - 堆栈指针指向最后压入的数据，且由低地址向高地址生成。
- 满递减堆栈FD
  - 堆栈指针指向最后压入的数据，且由高地址向低地址生成。
- 空递增堆栈EA
  - 堆栈指针指向下一个将要放入数据的空位置，且由低地址向高地址生成。
- 空递减堆栈ED
  - 堆栈指针指向下一个将要放入数据的空位置，且由高地址向低地址生成。
- 堆栈操作指令
  - LDM
  - STM

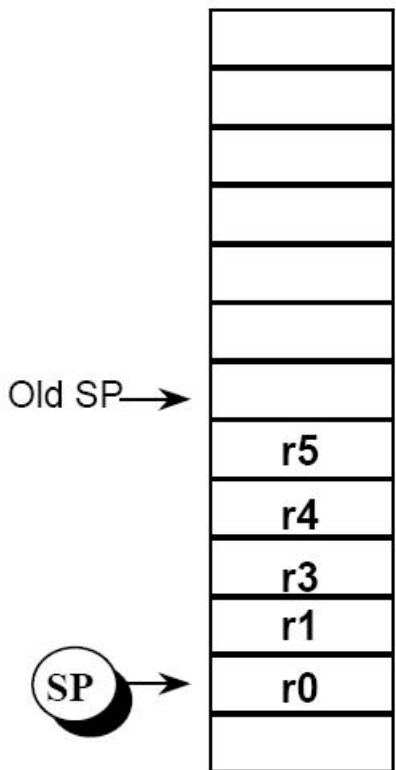
## 8086堆栈

```
Ucosii移植: #define OS_STK_GROWTH 1 /* 堆栈由高地址向低地址增长
```

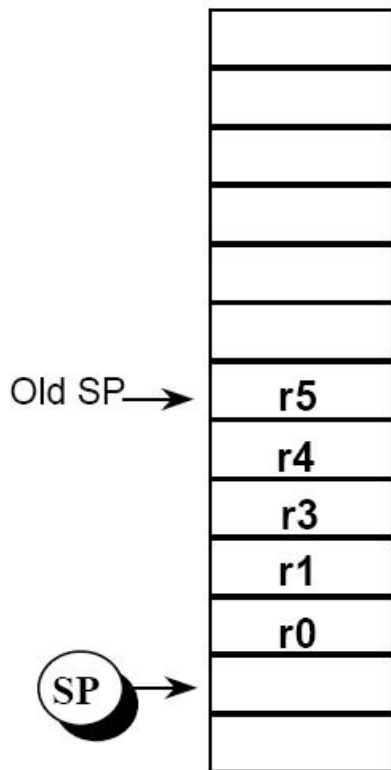


# 栈操作图示

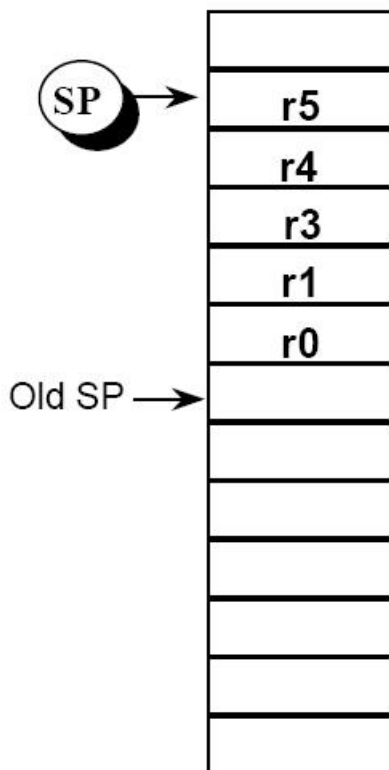
STMFD sp!,  
{r0,r1,r3-r5}



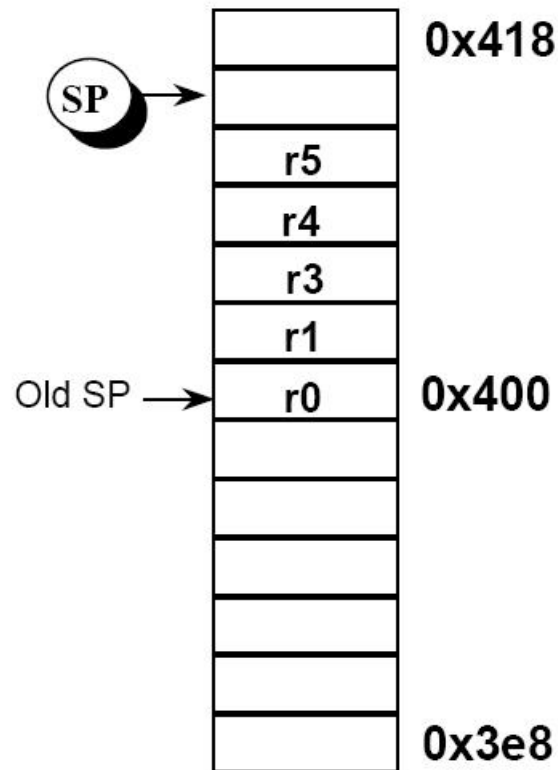
STMED sp!,  
{r0,r1,r3-r5}



STMFA sp!,  
{r0,r1,r3-r5}



STMEA sp!,  
{r0,r1,r3-r5}



# ARM指令——第2个操作数



ARM指令的基本格式如下：

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

灵活的使用第2个操作数“**operand2**”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；



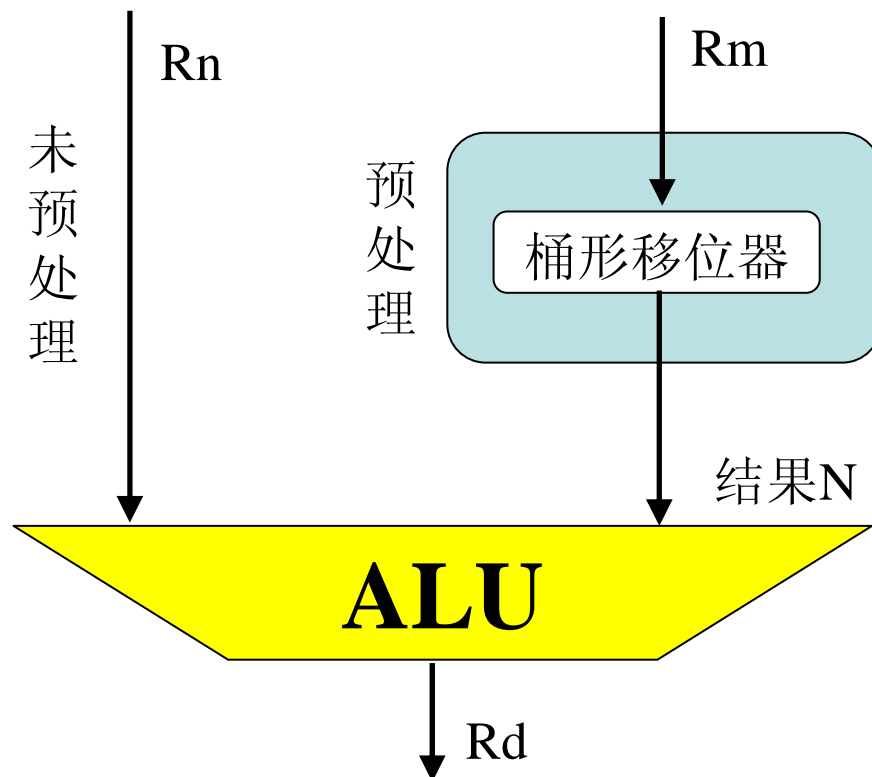
# ARM指令——第2个操作数

## ■ Rm, shift——寄存器移位方式

将寄存器的移位结果作为操作数（移位操作不消耗额外的时间），但Rm值保持不变。

**ADD R1, R1, R1, LSL #3 ;R1=R1+R1<<3**

移位方法如下：



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	Type为移位的一种类型，Rs为偏移量寄存器，低8位有效。



# 桶形移位器操作

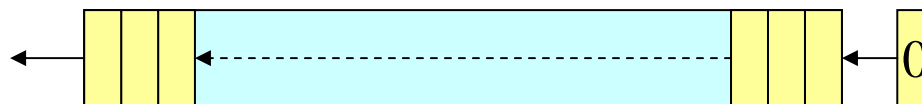
助记符	说明	移位操作	结果	Y值
LSL	逻辑左移	$x \text{ LSL } y$	$x \ll y$	#0-31 or Rs
LSR	逻辑右移	$x \text{ LSR } y$	$(\text{unsigned})x \gg y$	#1-32 or Rs
ASR	算术右移	$x \text{ ASR } y$	$(\text{signed})x \gg Y$	#1-32 or Rs
ROR	算术左移	$x \text{ ROR } y$	$((\text{unsigned})x \gg y   (x \ll 32 - y))$	#1-32 or Rs
RRX	扩展的循环右移	$x \text{ RRX } y$	$(c \text{ flag} \ll 31)   ((\text{unsigned})x \gg 1)$	none



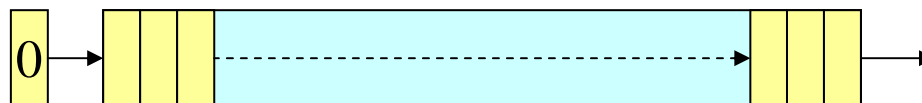


# 桶形移位器操作（续）

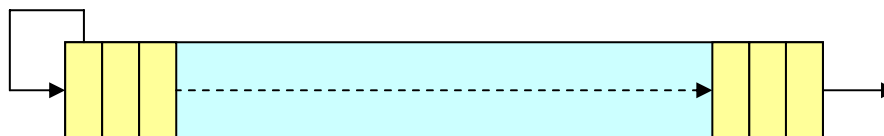
LSL 移位操作：



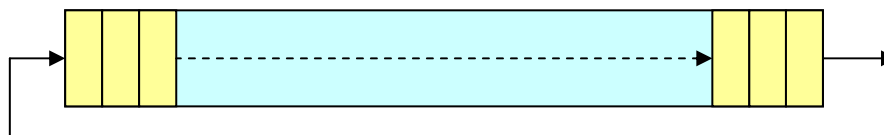
LSR 移位操作：



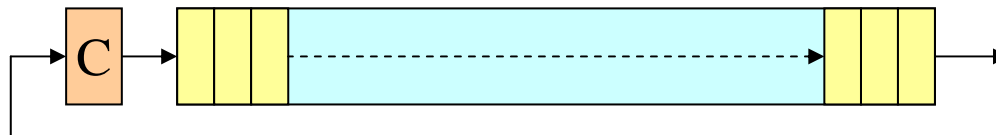
ASR 移位操作：



ROR 移位操作：



RRX 移位操作：

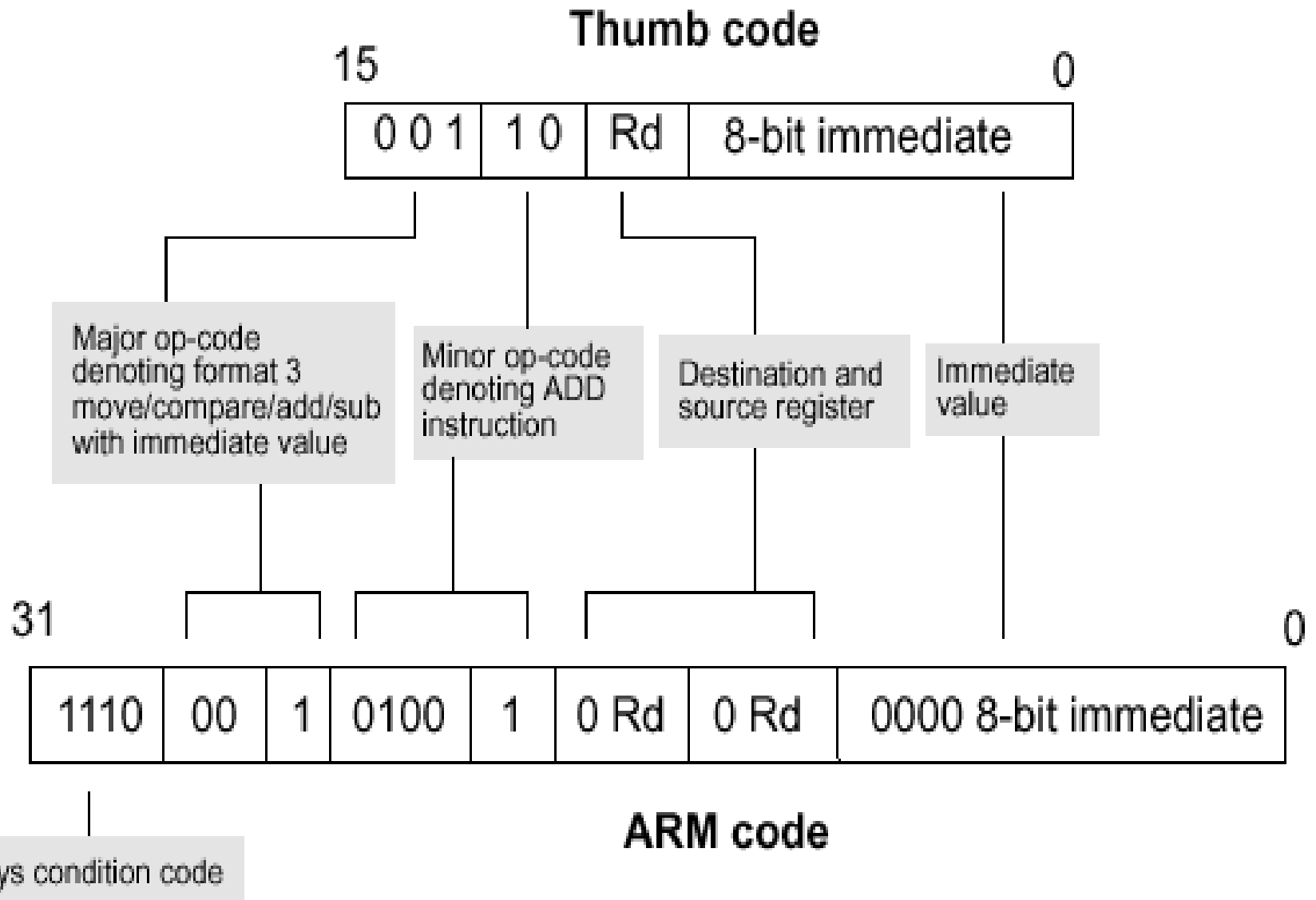




## 5.4 Thumb指令及应用

- 为兼容数据总线宽度为**16**位的应用系统。
- **Thumb**指令集是**ARM**指令集的一个子集
  - 所有的**Thumb**指令都有对应的**ARM**指令。
  - 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。
- **Thumb**指令集在保留**32**代码优势的同时，大大的节省了系统的存储空间。
  - **Thumb**指令集中操作数仍然是**32**位，指令地址也为**32**位，指令编码**16**位
  - 由于**Thumb**指令的长度为**16**位，即只用**ARM**指令一半的位数来实现同样的功能，所以，要实现特定的程序功能，所需的**Thumb**指令的条数较**ARM**指令多。

# Example: ADD Rd, #Constant





# Thumb指令优势

- 在一般的情况下，**Thumb**指令与**ARM**指令的时间效率和空间效率关系为：
  - **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**位的存储器，**ARM**代码比**Thumb**代码快约**40%**
  - 若使用**16**位的存储器，**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%**

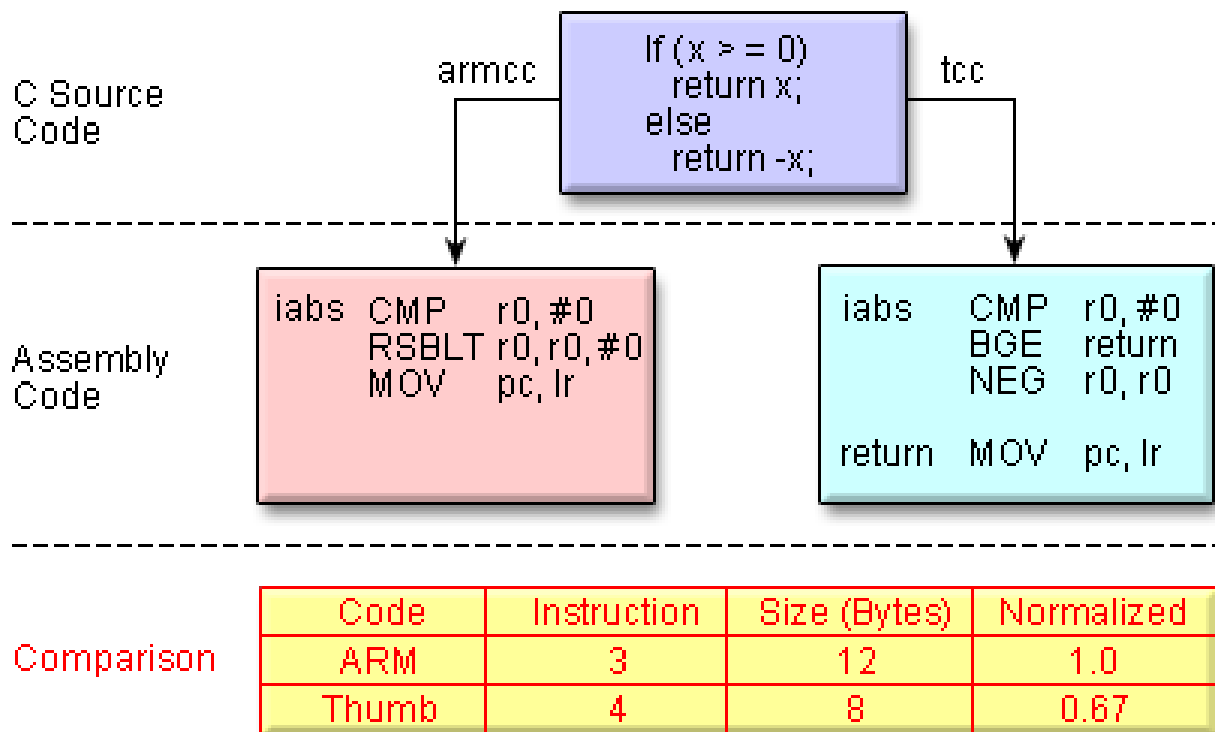


# Thumb指令集的局限性

- 条件跳转限制在**256byte** 偏移范围内，无条件跳转限制为**4K**偏移范围内
  - 而ARM为**32 Mbytes**偏移
- 不支持乘法和累加指令
- 没有协处理器指令
- 没有信号量指令
- 没有访问**CPSR**指令

# 指令集选择

- 若对系统的性能有较高要求，应使用**32位**的存储系统和**ARM**指令
- 若对系统的成本及功耗有较高要求，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。





# 内容提要

- ARM指令集详解
  - 数据处理指令
  - 数据传输指令
  - 控制流指令
  - 状态寄存器操作指令
  - 协处理器指令
  - 异常产生指令
  - ARM伪汇编指令



# 数据处理指令

- 所有操作数是32位宽，来自寄存器，或指令中的立即数
- 数据计算结果是32位，并放在寄存器里
- 一般数据处理指令包括两个操作数和一个目标寄存器





# 数据处理指令

## 算术操作

ADD r0, r1, r2	$r0 := r1 + r2$
ADC r0, r1, r2	$r0 := r1 + r2 + C$
SUB r0, r1, r2	$r0 := r1 - r2$
SBC r0, r1, r2	$r0 := r1 - r2 + C - 1$
RSB r0, r1, r2	$r0 := r2 - r1$
RSC r0, r1, r2	$r0 := r2 - r1 + C - 1$

## 寄存器移位

MOV r0, r2	$r0 := r2$
MVN r0, r2	$r0 := \text{not } r2$

## 按位逻辑操作

AND r0, r1, r2	$r0 := r1 \text{ and } r2$
ORR r0, r1, r2	$r0 := r1 \text{ or } r2$
EOR r0, r1, r2	$r0 := r1 \text{ xor } r2$
BIC r0, r1, r2	$r0 := r1 \text{ and } (\text{not}) r2$

## 比较操作

CMP r1, r2	set cc on $r1 - r2$
CMN r1, r2	set cc on $r1 + r2$
TST r1, r2	set cc on $r1 \text{ and } r2$
TEQ r1, r2	set cc on $r1 \text{ xor } r2$



# 算术指令

**ADD**  $r0, r1, r2$  ;  $r0 := r1 + r2$   
**ADC**  $r0, r1, r2$  ;  $r0 := r1 + r2 + C$   
**SUB**  $r0, r1, r2$  ;  $r0 := r1 - r2$   
**SBC**  $r0, r1, r2$  ;  $r0 := r1 - r2 + C - 1$   
**RSB**  $r0, r1, r2$  ;  $r0 := r2 - r1$   
**RSC**  $r0, r1, r2$  ;  $r0 := r2 - r1 + C - 1$

注意：如果不再指令中明确指出的话，数据运算不会改变标志位（上面的指令不改变标志位，而下面的例子改变标志位）

**ADCS**  $R1, R5, R9$  ; 带进位加，该运算改变标志位

# 逻辑运算



**AND**  $r0, r1, r2$  ;  $r0 := r1 \text{ and } r2$

**ORR**  $r0, r1, r2$  ;  $r0 := r1 \text{ or } r2$

**EOR**  $r0, r1, r2$  ;  $r0 := r1 \text{ xor } r2$

**BIC**  $r0, r1, r2$  ;  $r0 := r1 \text{ and not } r2$

注意：**BIC**从r1中挑出了由r2指定的比特，r2可以看成屏蔽寄存器，r2设置为“1”的比特表示对应的r1的比特要被清除，r2设置为“0”的比特表示对应的r1的比特被保留



# 寄存器传输

```
MOV r0, r2           ; r0 := r2
MVN r0, r2           ; r0 := not r2
MOV r0, r0           ; r0 := r0 (NOP 指令)
MOV r0, r0, LSL #3   ; r0 := r0×8
```

下面的 MOV 命令对地址寄存器操作，相当于程序跳转指令

```
MOVS pc, r14        ; 返回到r14保存的地址，
                    ; 同时恢复CPSR
```



# 比较运算

<b>CMP</b> r1, r2	i set c on r1 - r2
<b>CMN</b> r1, r2	i set c on r1 + r2
<b>TST</b> r1, r2	i set c on r1 and r2
<b>TEQ</b> r1, r2	i set c on r1 xor r2

比较运算改变标志位N, Z, C和V.



# 立即数运算

**ADD r3, r3, #1** ; r3 := r3 + 1  
**AND r8, r7, #&fff** ; r8 := r7[7:0]

注意：**#**表示立即数，**&**表示**16**进制数，由于指令格式编码原因，可以表示的立即数范围是

$$(0 \sim 255) \times 2^{2n}$$

$$0 \leq n \leq 12$$



# 寄存器移位操作

**ADD r3, r2, r1, LSL #3** ;  $r3 := r2 + 2^3 \times r1$

**ADD r5, r5, r3, LSL r2** ;  $r5 += 2^{r2} \times r3$

## 设置标志位

**ADDS r2, r2, r0** ; 32-bit carry-out  $\rightarrow$  C

只有当算术运算指令中有**S**时，运算结果才会影响标志位

注意：对于地址寄存器的操作，指令中的**S**表示“特殊指令”，用于异常处理返回

如 **SUBS pc, r14, #8,**



# 乘法指令和乘加命令

**MUL** *r4*, *r3*, *r2* ; *r4* := (*r3* × *r2*) [31:0]

**MLA** *r4*, *r3*, *r2*, *r1* ; *r4* := (*r3* × *r2* + *r1*) [31:0]

注意：结果保留低32位，另外只能用寄存器做操作数





# 数据传送指令

- 单寄存器Load和Store指令
- 多寄存器Load和Store指令
- 单寄存器交换指令

常用的加载存储指令如下：

LDR 字数据加载指令

LDRB 字节数据加载指令

LDRH 半字数据加载指令

STR 字数据存储指令

STRB 字节数据存储指令

STRH 半字数据存储指令

**LDM (或STM) {条件}{类型} 基址寄存器{!}, 寄存器列表{^}**

{!}为可选后缀，若选用该后缀，则当数据传送完毕之后，将最后的地址写入基址寄存器，否则基址寄存器的内容不改变；

基址寄存器不允许为R15，寄存器列表可以为R0~R15的任意组合；

{^}为可选后缀，当指令为LDM且寄存器列表中包含R15，选用该后缀时表示：除了正常的的数据传送之外，**还将SPSR复制到CPSR**。同时，该后缀还表示传入或传出的是用户模式下的寄存器，而不是当前模式下的寄存器。



# 单寄存器存储

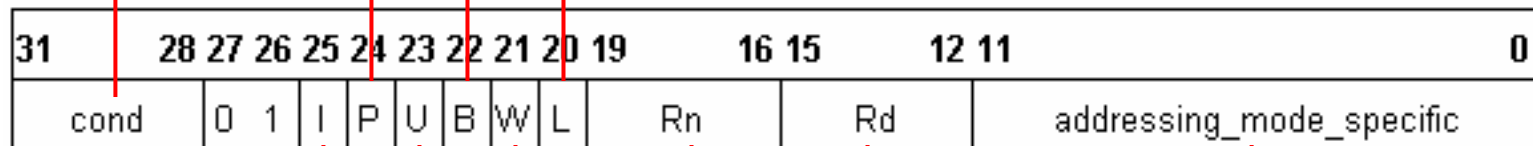
## • LDR和STR——字和无符号字节加载/存储指令编码

**B** 为1表示字节访问，为0表示字访问

**P**表示前/后变址

指令执行的条件码

**L**用于区别加载 (L为1) 或存储 (L为0)



**I**为0时，偏移量为12位立即数，为1时，偏移量为寄存器移位

**U**表示加/减

为指令的寻址方式

**Rd**为源/目标寄存器

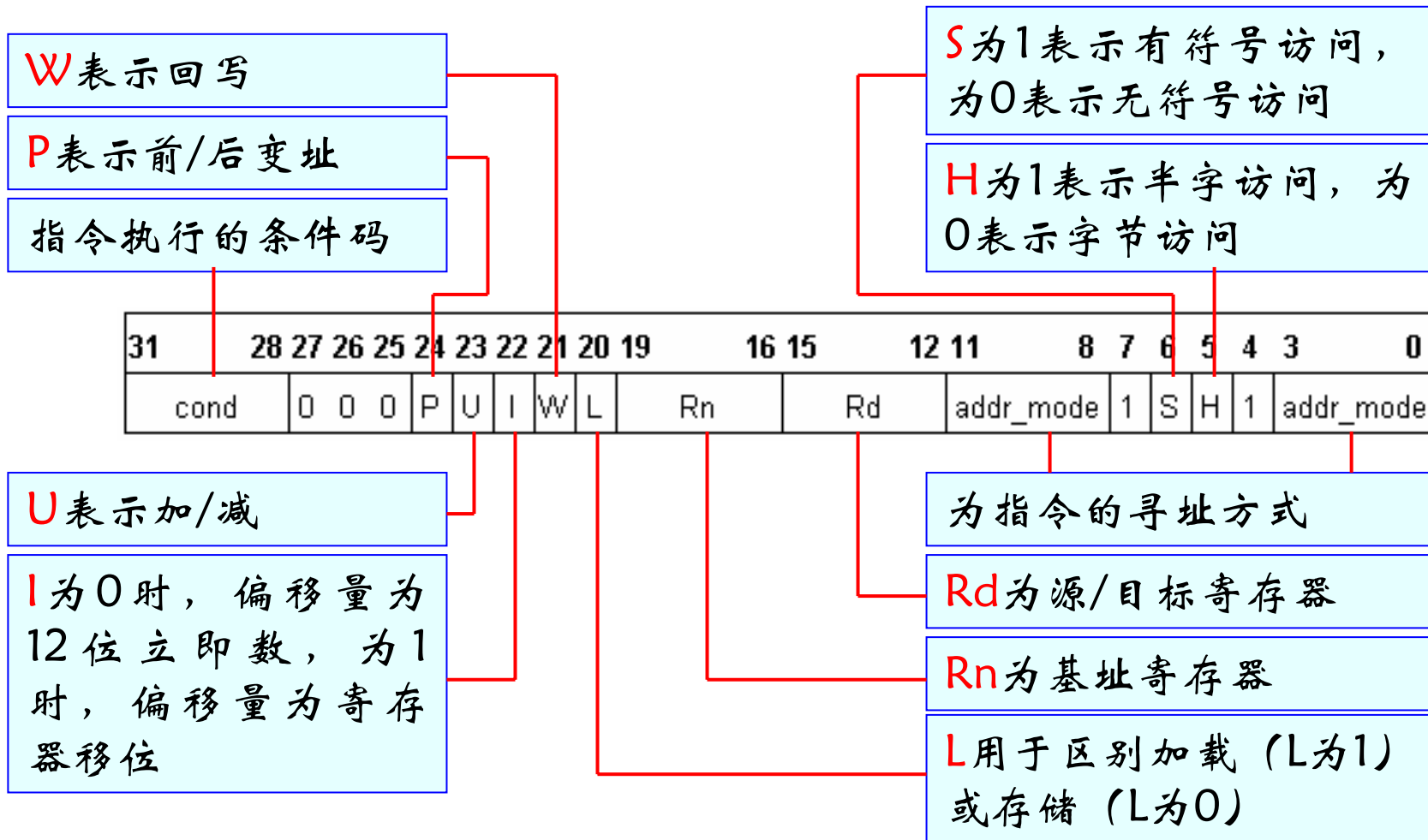
**Rn**为基址寄存器

**W**表示回写



# 单寄存器存储

## • LDR和STR——半字和有符号字节加载/存储指令编码





# 单寄存器Load和Store指令

**LDR r0, [r1, #4]** ; r0 := mem[r1+4]

注意：偏移量在+/- 4 Kbytes之间

**LDR r0, [r1, #4]!** ; r0 := mem[r1+4]  
; r1 := r1 + 4

注意：r1会自动加4——前变址

**LDR r0, [r1], #4** ; r0 := mem[r1]  
; r1 := r1 + 4

注意：r1会在运算后自动加4——后变址

**LDRB r0, [r1]** ; r0 := mem8[r1]

注意：是Byte操作，STR指令和Load指令类似



# 存储器 and 寄存器数据互换

同时完成数据的加载和存储（中间不会被中断）  
用于操作系统多个进程访问共享资源时的加锁操作

```
SWP r0, r1, [r2]      ; r0 := mem[r2]  
                        ; mem8[r2] := r1
```

```
SWPB r1, r1, [r0]    ; r1 := mem[r0]  
                        ; mem8[r0] := r1
```

# 多寄存器数据传输—数据块操作



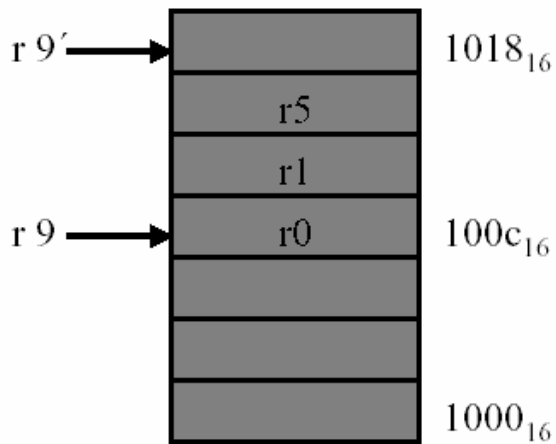
```
LDMIA r1, {r0, r2, r5}; r0 := mem [r1]
                        ; r2 := mem [r1+4]
                        ; r5 := mem [r1+8]
```

注意：{..}里可以包括多个寄存器（r0 - r15）

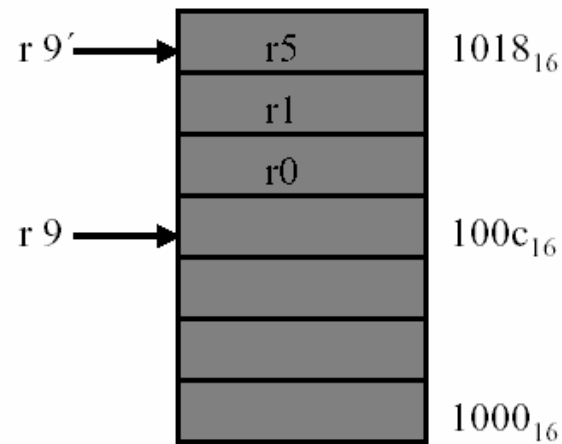
由于r15是pc，对它操作可能会造成程序跳转。

号码最小的寄存器总是占据最低的地址，不考虑{...}里的次序

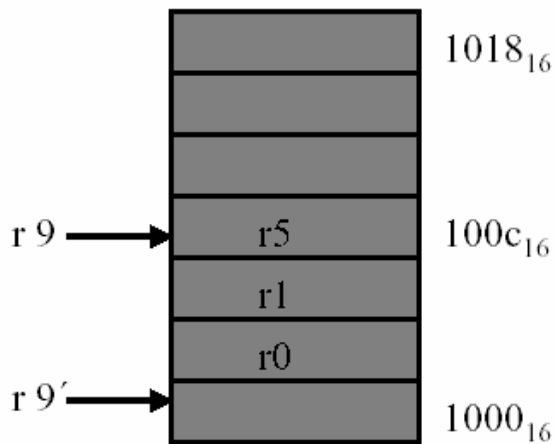
# 多寄存器数据传输—数据块操作



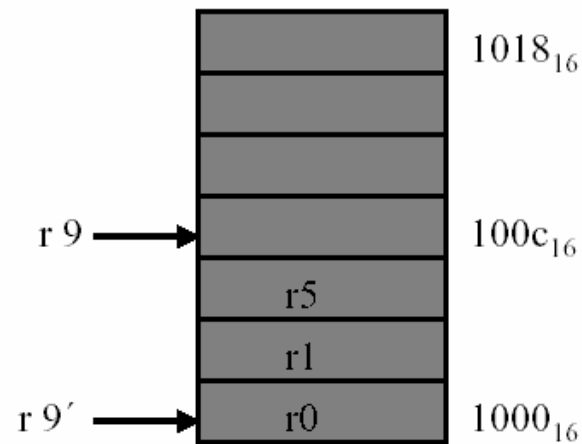
STMIA  $r9!$ , { $r0, r1, r5$ }



STMIB  $r9!$ , { $r0, r1, r5$ }



STMDA  $r9!$ , { $r0, r1, r5$ }



STMDB  $r9!$ , { $r0, r1, r5$ }

# 多寄存器数据传输——堆栈



- IA → Increment After
- IB → Increment Before
- DA → Decrement After
- DB → Decrement Before

- FA → Full Ascending (r13存放地址)
- FD → Full Descending (r13存放地址)
- EA → Empty Ascending (r13存放地址)
- ED → Empty Descending (r13存放地址)

		Ascending		Descending	
		Full	Empty	Full	Empty
Increment	Before	STMIB STMFA			LDMIB LDMED
	After		STMIA STMEA	LDMIA LDMFD	
Decrement	Before		LDMDB LDMEA	STMDB STMFD	
	After	LDMDA LDMFA			STMDA STMED





# 数据块操作 vs. 堆栈操作

堆栈操作和数据块传送指令类似，都有4种模式，其关系如下表所示：

数据块传送 存储	堆栈操作 压栈	说明
STMDA	STMED	空递减
STMIA	STMEA	空递增
STMDB	STMFD	满递减
STMIB	STMFA	满递增

数据块传送 加载	堆栈操作 出栈	说明
LDMDA	LDMFA	满递减
LDMIA	LDMFD	满递增
LDMDB	LDMEA	空递减
LDMIB	LDMED	空递增

；使用数据块传送指令进行堆栈操作

STMDA R0!, {R5-R6}

...

LDMIB R0!, {R5-R6}



；使用堆栈指令进行堆栈操作

STMED R13!, {R5-R6}

...

LDMED R13!, {R5-R6}

两段代码的执行结果是一样的，但是使用堆栈指令的压栈和出栈操作编程很简单（只要前后一致即可），而使用数据块指令进行压栈和出栈操作则需要考虑空与满、加与减对应的问题。

# 数据传送指令



## 单寄存器 load and store

## 寄存器间接寻址

LDR r0, [r1]	$r0 := \text{mem}_{32}[r1]$
--------------	-----------------------------

STR r0, [r1]	$\text{mem}_{32}[r1] := r0$
--------------	-----------------------------

Note: r1 keeps a word address (2 LSBs are 0)

## 基址+偏移量寻址

(offset of up to 4Kbytes)

LDR r0, [r1, #4]	$r0 := \text{mem}_{32}[r1 + 4]$
------------------	---------------------------------

LDRB r0, [r1]	$r0 := \text{mem}_8[r1]$
---------------	--------------------------

Note: no restrictions for r1

## 自动变址寻址

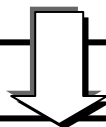
LDR r0, [r1, #4]!	$r0 := \text{mem}_{32}[r1 + 4]$ $r1 := r1 + 4$
-------------------	---

## 后变址寻址

LDR r0, [r1], #4	$r0 := \text{mem}_{32}[r1]$ $r1 := r1 + 4$
------------------	---

# 数据传送指令

```
COPY:  ADR r1, TABLE1 ; r1 points to TABLE1
        ADR r2, TABLE2; r2 points to TABLE2
LOOP:  LDR r0, [r1]
        STR r0, [r2]
        ADD r1, r1, #4
        ADD r2, r2, #4
        ...
TABLE1:...
TABLE2:...
```



```
COPY:  ADR r1, TABLE1 ; r1 points to TABLE1
        ADR r2, TABLE2; r2 points to TABLE2
LOOP:  LDR r0, [r1], #4
        STR r0, [r2], #4
        ...
TABLE1:...
TABLE2:...
```



# 控制流指令

- 无条件转移
- 有条件转移

助记符	说明	操作	条件码位置
B      label	分支指令	$PC \leftarrow \text{label}$	B {cond}
BL     label	带链接的分支指令	$LR \leftarrow PC - 4, PC \leftarrow \text{label}$	BL {cond}
BX     Rm	带状态切换的分支指令	$PC \leftarrow Rm$ , 切换处理器状态	BX {cond}



# 无条件转移

**B LABEL**

...

... ; these instructions are **skipped**

...

**LABEL**

...

...

## 有条件转移（例子）

```
MOV r0, #0 ; initialize counter
```

**LOOP**

```
ADD r0, r0, #1 ; increment counter
```

```
CMP r0, #10 ; compare with limit
```

```
BNE LOOP ; repeat if not equal
```

```
... ; else continue
```

...

# 有/无条件转移



B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BNE	Not equal	Comparison not equal or zero result
BMI	Minus	Result minus or negative
BEQ	Equal	Comparison equal or zero result
BHS	Higher or same	Unsigned comparison gave higher or same
BCC	Carry clear	Arithmetic operation did not give carry-out
BLT	Less than	Signed integer comparison gave less than
BPL	Plus	Result positive or zero
BLO	Lower	Unsigned comparison gave lower
BGE	Greater or equal	Signed integer comparison gave greater or equal
BGT	Greater than	Signed integer comparison gave greater than
BVS	Overflow set	Signed integer operation ; overflow occurred
BVC	Overflow clear	Signed integer operation ; no overflow occurred
BCS	Carry set	Arithmetic operation gave carry-out
BLS	Lower or same	Unsigned comparison gave lower or same
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher

# 子程序调用



```
BL SUBR           ; branch to SUBR
...              ; return to here
...
SUBR
...              ; subroutine entry point
MOV pc, r14      ; return
```

转跳时返回地址存放在**r14**中，子程序返回时只要把原先保存在**r14**的地址装入**pc**即可（没有类似**RET**的语句）

注意：如果子程序要使用**r14**的话，需要存在里面的保存返回地址。



# 程序嵌套调用

子程序里再调用子程序时会改变r14，因此遇到多层程序调用时得保存r14

```
BL SUB1                ; branch to SUB!  
...  
...  
  
SUB1  
STMFA r13!, {r0-r2, r14} ; save regs  
BL SUB2  
...  
...  
LDMFA r13!, {r0-r2, pc} ; return  
  
SUB2  
...  
MOV pc, r14           ; return
```



# 软中断



```
SWI SWI_WriteC      ; output character in r0
SWI SWI_Exit        ; return to monitor
```

- **SWI**将处理器置于监控（**SVC**）模式，从地址**0x08**开始执行指令。这些中断调用的作用与实现的系统有关。
- 立即数“**SWI\_WriteC**”和“**SWI\_Exit**”出现在**SWI**指令的0~23比特，它被操作系统用来判断用户程序调用系统例程的类型，由中断服务程序决定是否使用。
- 当指令中**24**位的立即数被忽略时，用户程序调用系统例程的类型由通用寄存器**R0**的内容决定，同时，参数通过其他通用寄存器传递。

## BKPT

- 断点中断指令，用于产生软件断点，供调试程序用。



# 软中断服务程序

在SWI异常中断处理程序中，取出SWI指令中立即数的步骤为：

- 首先确定引起软中断的SWI指令是ARM指令还是Thumb指令，这可通过对SPSR访问得到；
- 然后取得该SWI指令的地址，这可通过访问LR寄存器得到；
- 接着读出该SWI指令，分解出立即数。

SWI\_Handler

STMFD SP!, {R0-R3, R12, LR} ; 现场保护

MRS R0, SPSR ; 读取SPSR

STMFD SP!, {R0} ; 保存SPSR

TST R0, #0x20 ; 测试T标志位

NE: Z=0

EQ: Z=1

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)

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



# 状态寄存器操作指令

下面是从特殊寄存器到普通寄存器的传输指令

```
MRS r0, CPSR ; r0 := CPSR
```

```
MRS r3, SPSR ; r3 := SPSR
```

下面是把数据传输到特殊寄存器的指令

```
MSR CPSR, R0 ; 复制 R0 到 CPSR 中
```

```
MSR SPSR, R0 ; 复制 R0 到 SPSR 中
```

```
MSR CPSR_flg, R0 ; 复制 R0 的标志位到 CPSR 中
```

```
MSR CPSR_flg, #0xF0000000 ; NVCZ="1111"
```

```
MSR SPSR_all, Rm ;
```

```
MSRNE CPSR_ctl, Rm ;
```

```
MRS Rd, CPSR
```

- 在 **user** 模式中，不能改变 **CPSR** 的控制位，只能改变条件标志。在其他模式中，可获得整个 **CPSR**。
- 在 **user** 模式中，不能尝试访问 **SPSR**，因为它不存在！

# 使用特殊寄存器指令的例子



设置 V 标志:

```
MSR CPSR_flg, #&10000000
```

; 设置V标志但  
; 不影响控制位

要改变模式:

```
MRS R0, CPSR_all
```

; 复制PSR

```
BIC R0, R0, #&1F
```

; 清除模式位

```
ORR R0, R0, #new_mode
```

; 把模式位设置为新模

式

```
MSR CPSR_all, R0
```

; 写回PSR, 变更模式



# 协处理器指令

- **ARM**微处理器可支持多达**16**个协处理器
  - 每个协处理器只执行针对自身的协处理指令
- **ARM**处理器初始化**ARM**协处理器的数据处理操作
- **ARM**处理器寄存器和协处理器的寄存器之间传送数据
  - **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
- 在**ARM**协处理器的寄存器和存储器之间传送数据。
  - **CDP**: 协处理器数据操作指令
  - **LDC**: 协处理器数据加载指令
  - **STC**: 协处理器数据存储指令
- 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



# ARM伪汇编指令ADR

**ADR**伪指令将基于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。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指令来实现该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。

应用示例（源程序）：

```
...  
ADR    R0,Delay  
...  
Delay  
MOV    R0,r14  
...
```

使用伪指令将程序标号  
Delay的地址存入R0

编译后的反汇编代码：

```
...  
0x20  ADD    r1,pc,#0x3c  
...  
...  
0x64  MOV    r0,r14  
...
```

ADR伪指令被汇编成一条指令

# ARM伪汇编指令LDR



向寄存器加载立即数。当立即数的大小超出MOV或MVN指令能操作的范围时，生成额外的代码（利用pc得到立即数）。

注意：这样使用的话，代码的运行地址必须在Link时固定，不能移动。

```
LDR r1,=0xffff      ; loads 0xffff into r1
                       ; => LDR r1,[pc,offset_to_litpool]
                       ; ...
                       ; litpool DCD 0xffff

LDR r2,=place       ; loads the address of
                       ; place into r2
                       ; => LDR r2,[pc,offset_to_litpool]
                       ; ...
                       ; litpool DCD place
```

下面的代码应该用MOV，写成LDR时被汇编器自动转回MOV语句

```
LDR r3,=0xff0      ; loads 0xff0 into r3
                       ; => MOV r3,#0xff0
```

# ARM伪汇编指令NOP



**NOP**伪指令在汇编时将会被代替成ARM中的空操作，比如可能是“MOV R0,R0”指令等。

NOP可用于延时操作。

```
mov      R1,#0x1234
```

Delay

```
NOP      ;空操作
```

```
NOP
```

```
NOP
```

```
SUBS    R1,R1,#1      ;循环次数减一
```

```
BNE     Delay        ;如果循环没有结束，跳转Delay继续
```

```
MOV     PC,LR        ;子程序返回
```





# 简单的ARM程序

```
;文件名: TEST1.S
;功
;说 使用“;”进行注释 仿真调试

        AREA      Example1, CODE, READONLY      ;声明代码段Example1
        ENTRY                               ;标识程序入口
        CODE32                                ;声明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     声明文件结束                          ;文件结束
```

;功  
;说

使用“;”进行注释

仿真调试

START  
LOOP  
ADD\_SUB

标号顶格写

实际代码段

END

声明文件结束

# Example: Hello ARM World!



```
                AREA  HelloW,CODE,READONLY ;声明代码区
SWI_WriteC  EQU  &0                ; 输出r0中的字符
SWI_Exit    EQU  &11                ; 程序结束
                ENTRY                ; 代码入口
START      ADR    r1,TEXT            ; r1---“Hello World”
LOOP      LDRB   r0,[r1],#1          ;读取下一字节
                CMP    r0,#0          ; 检查文本终点
                SWINE  SWI_WriteC     ; 若非终点，则打印
                BNE   LOOP            ; 并返回LOOP
                SWI   SWI_Exit        ; 执行结束
TEXT      =      “Hello World”,&0a,&0d,0
                END                    ; 程序结束
```



# ARM ADS汇编伪指令

- ALIGN  
地址对齐 ( $2^n$ 边界)
- AREA  
指定Section
- CODE16  
指示下面是Thumb (16-bit) 代码
- CODE32  
指示下面是ARM (32-bit) 代码
- DATA  
指示当前代码段里的标号是存放数据的
- DCB, DCD  
定义初始化的数据
- ENTRY  
程序入口
- EQU  
为常量起名
- EXPORT  
定义一个可以在其他模块访问的名称
- EXTERN, IMPORT  
申明一个存在于其他模块的名称
- GET  
相当于C中的#include
- OPT  
打印控制
- SETS, SETA  
设置字符串变量和数据变量
- SPACE  
申请空白数据空间

# APCS(ARM Procedure Call Standard)



- 对寄存器使用的限制
- 使用栈的惯例
  - **APCS**规定数据栈为**FD**（满递减）类型
- 在函数调用之间传递/返回参数
  - 当参数不超过**4**个时，可以使用寄存器**R0~R3**来传递参数；
- 可以被‘回溯’的基于栈的结构格式，用来提供从失败点到程序入口的函数(和给予的参数)的列表
- 开发一个系统，不要求一定要符合APCS。

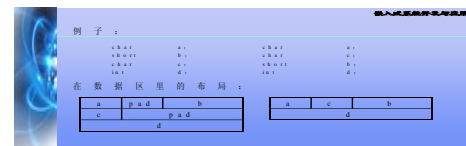
# 寄存器命名



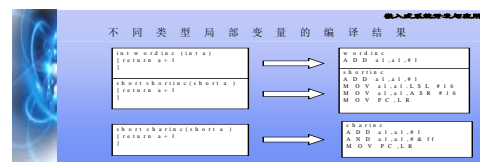
寄存器号	APCS名字	意义
R0	a1	可以改变的输入变量
R1	a2	可以改变的输入变量
R2	a3	可以改变的输入变量
R3	a4	可以改变的输入变量
R4	v1	不可改变的输入变量
R5	v2	不可改变的输入变量
R6	v3	不可改变的输入变量
R7	v4	不可改变的输入变量
R8	v5	不可改变的输入变量
R9	v6	不可改变的输入变量
R10	s1	栈限制
R11	fp	帧指针
R12	ip	内部过程调用寄存器
R13	sp	栈指针
R14	lr	连接寄存器
R15	pc	程序计数器

# 一、变量定义

在变量声明的时候，最好把所有相同类型的变量放在一起定义，这样可以优化存储器布局。由下例可以看出：



对于局部变量类型的定义，使用short或char来定义变量并不是总能节省存储空间。有时使用32位int或unsigned int局部变量更有效率一些，如下图所示：



变量定义中，为了精简程序，程序员总是竭力避免使用冗余变量。但有时使用冗余变量可以减少存储器访问的次数这可以提高系统性能。

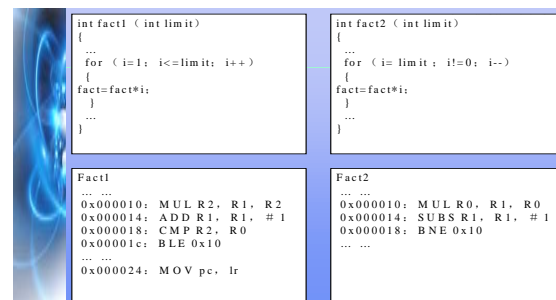
## 二、循环条件

➤计数循环是程序中十分常用的流程控制结构，一般有以下两种形式：

➤for (loop=1; loop<=limit; loop++)

➤for (loop=limit; loop!=0; loop--)

➤这两种循环形式在逻辑上并没有效率差异，但是映射到具体的体系结构中时，就产生了很大的不同，如下图所示。





# 三、以空间换时间

➤ 计算机程序中最大的矛盾是空间和时间之间的矛盾，从这个角度出发逆向思维来考虑程序的效率问题，比如若系统的实时性要求很高，内存还有剩余，则我们就有可以用以空间换时间的方法来提高程序执行的效率。

嵌入式系统开发与应用

例1: 字符串的赋值。

方法A:	方法B:
<pre>#define LEN 32 char string1 [LEN]; memset (string1, 0, LEN); strcpy (string1, "This is an example!")</pre>	<pre>const char string2[LEN]="This is an example!" char *cp; cp=string2; (使用的时候可以直接用指针 来操作。)</pre>

可以看出，B的效率要比A的高许多。在同样的存储空间下，B直接使用指针就可以操作了，而A需要调用两个字符函数才能完成。

嵌入式系统开发与应用

例2: 使用宏函数代替函数

**函数和宏函数的区别:**  
宏函数占用了大量的空间，而函数占用了时间。

**函数调用需要一些CPU时间:**  
要使用系统的栈来保存数据的，CPU要在函数调用时保存和恢复当前的现场。

**宏函数占用了空间:**  
作为预先写好的代码嵌入到当前程序，不会产生函数调用，频繁调用同一个宏函数的时候，该现象尤其突出。





# 四、数学方法解决问题

➤数学是计算机之母，计算机的发展是以数学为依据和基础的，所以在编写程序的时候，适当地采用一些数学方法会对程序的执行效率有数量级的提高，如下例所示：

<pre>方法1: int i,,j; For (i=1;i&lt;=100;i=++){ j+=i; }</pre>	<pre>方法2: int i; i=(100*(1+100))/2</pre>
---	--



# 五、使用位操作

➤一般的位操作是用来控制硬件的，或者做数据变换使用，但是，灵活的位操作可以减少除法和取模的运算有效地提高程序运行的效率，如下例所示：

方法1: int I,J; I=257/8; J=456%32;	方法2: int I,J; I=257>>3; J=456-(456>>5<<5);
---	---



Thank You

# 寄存器命名



寄存器类别	寄存器在汇编中的名称	各模式下实际访问的寄存器						
		用户	系统	管理	中止	未定义	中断	快中断
通用寄存器和程序计数器	R0(a1)	R0						
	R1(a2)	R1						
	R2(a3)	R2						
	R3(a4)	R3						
	R4(v1)	R4						
	R5(v2)	R5						
	R6(v3)	R6						
	R7(v4)	R7						
	R8(v5)	R8						R8_fiq
	R9(SB,v6)	R9						R9_fiq
	R10(SL,v7)	R10						R10_fiq
	R11(FP,v8)	R11						R11_fiq
	R12(IP)	R12						R12_fiq
	R13(SP)	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq	
	R14(LR)	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq	
R15(PC)	R15							
状态寄存器	CPSR	CPSR						
	SPSR	无	SPSR_abt	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq	