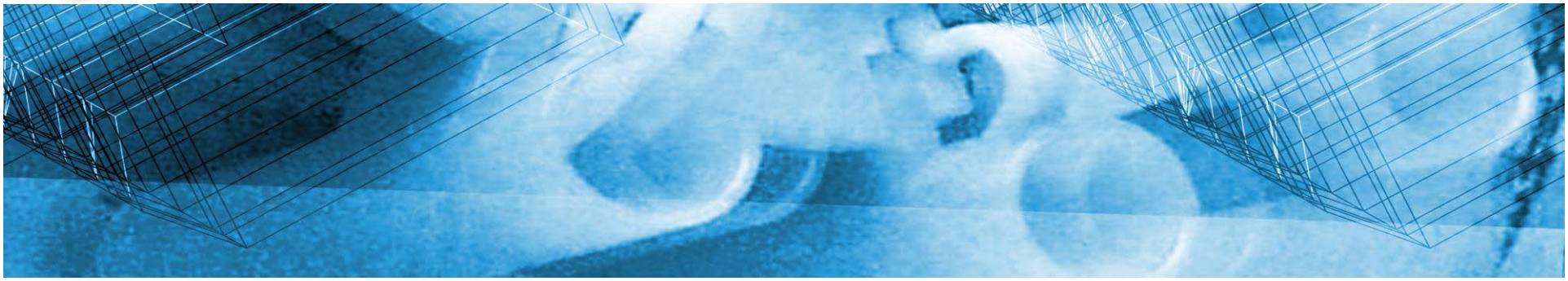


第三章 ARM处理器指令系统

❖孙永奎 讲师/博士

❖E-mail:yksun@swjtu.edu.cn

课程内容

- ARM处理器寻址方式
- **ARM处理器指令集**

背景

RISC体系结构特点：

- 在进行指令系统设计时，只选择使用频率很高的指令，在此基础上增加少量能有效支持操作系统和高级语言实现以及其他功能的指令，使指令条数大大减少
- 采用固定长度的指令格式，指令归整、简单、基本寻址方式有2～3种
- 使用单周期指令，便于流水线操作执行
- 大量使用寄存器，数据处理指令只对寄存器进行操作，只有加载/存储指令可以访问存储器，以提高指令的执行效率
- 为提高指令执行速度，大部分指令直接采用硬件电路实现，少量采用微码实现

背景

ARM体系结构还采用了一些特别的技术，在保证高性能的前提下尽量缩小芯片的面积，并降低功耗。

- 大多数的指令都可根据前面的执行结果决定是否被执行，从而提高指令的执行效率
- 可用加载/存储指令批量传输数据，以提高数据的传输效率
- 可在一条数据处理指令中同时完成逻辑处理和移位处理
- 在循环处理中使用地址的自动增减来提高运行效率

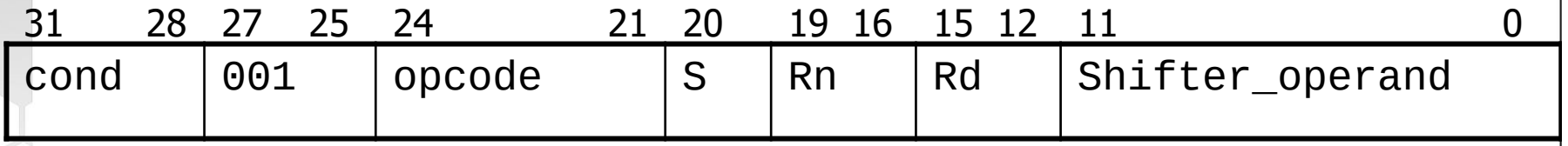
3.1 ARM指令概述

- ARM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

3.1 概述

- 向后兼容：新版本增加指令，并保持指令向后兼容；
- Load-store 结构*
  - load/store –从存储器中读某个值,操作完后再将其放回存储器中
  - 只对存放在寄存器的数据进行处理
  - 对于存储器中的数据，只能使用load/store指令进行存取

ARM指令的基本格式




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

其中<>号内的项是必须的，{}号内的项是可选的。

各项的说明如下：

opcode：指令助记符；　**cond**：执行条件；

S：是否影响CPSR寄存器的值；

Rd：目标寄存器；　**Rn**：第1个操作数的寄存器；

operand2：第2个操作数；

例：

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

2.2 ARM处理器寻址方式

寻址方式是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处理器具有9种基本寻址方式。

(1).寄存器寻址；　　(2).立即寻址；

(3).寄存器偏移寻址；　　(4).寄存器间接寻址；

(5).基址寻址；　　(6).多寄存器寻址；

(7).堆栈寻址；　　(8).块拷贝寻址；

(9).相对寻址。

(1) 寄存器寻址

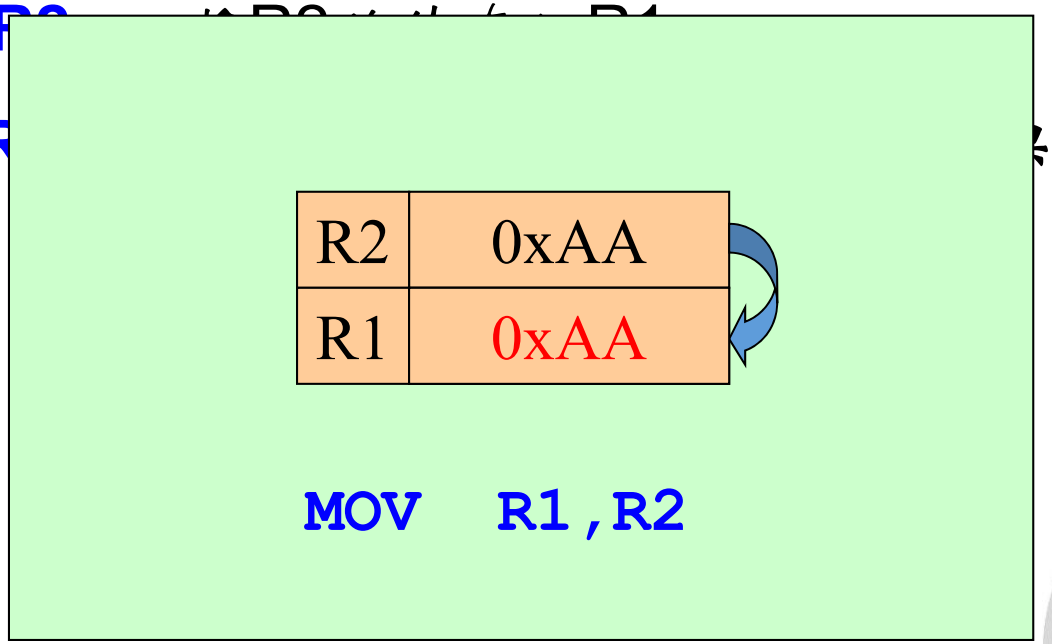
操作数的值在寄存器中，指令中的地址码字段指出的是寄存器编号，指令执行时直接取出寄存器值来操作。

寄存器寻址指令举例如下：

MOV R1,R2

SUB R0,R

保存到R0



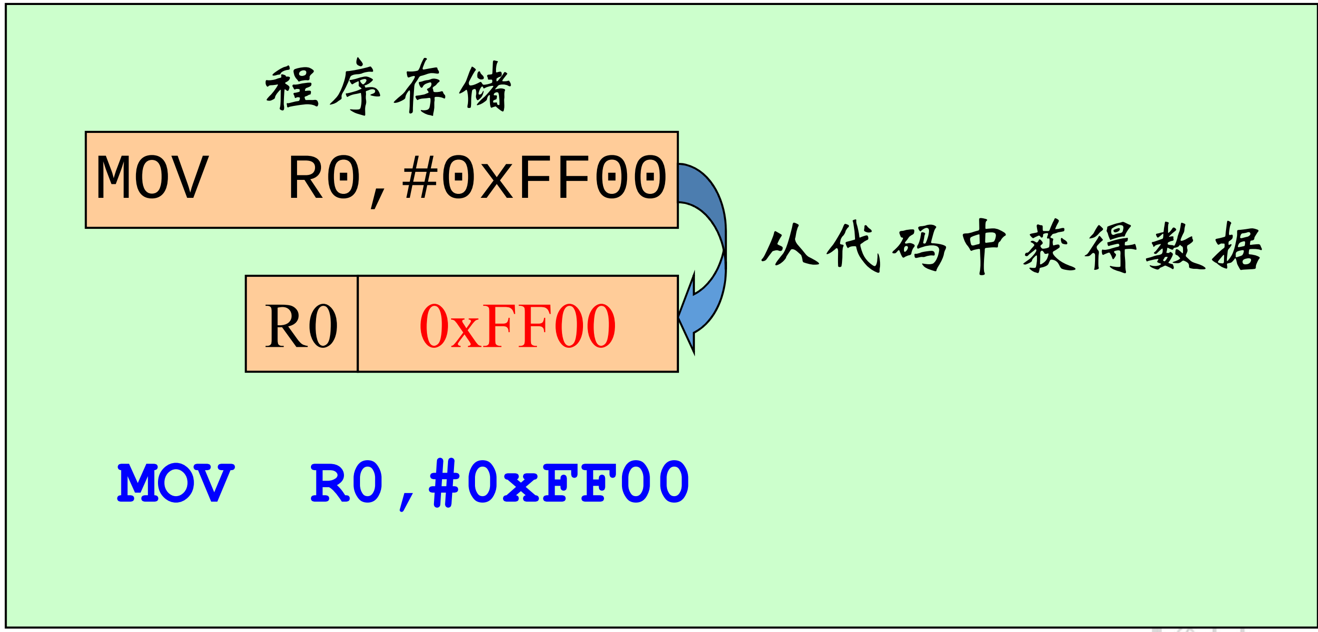
(2) 立即寻址

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，也就是说，数据就包含在指令当中，取出指令也就取出了可以立即使用的操作数(这样的数称为立即数)。

立即寻址指令举例如下：

SUBS … 标志位

MOV … 器

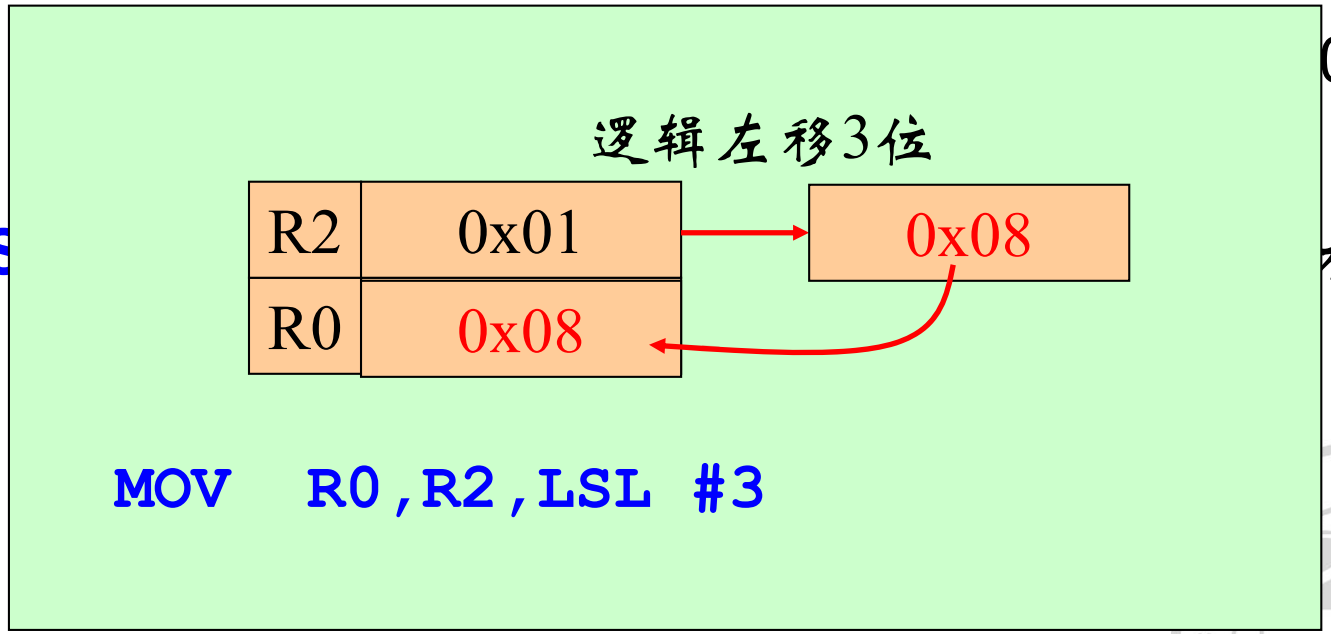


(3) 寄存器移位寻址

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个寄存器操作数在与第1个操作数结合之前，选择进行移位操作。**寄存器移位寻址指令举例如下：**

MOV 0,

ANDS 相

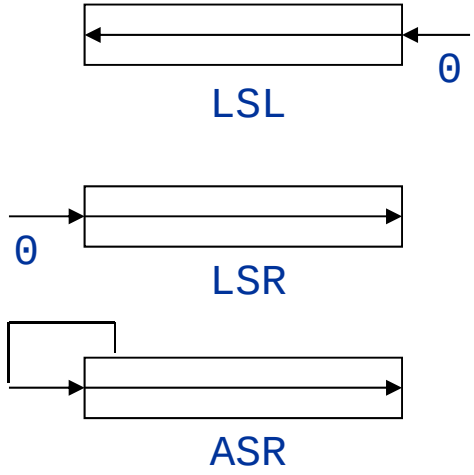


(3) 寄存器移位寻址

- 寄存器移位寻址

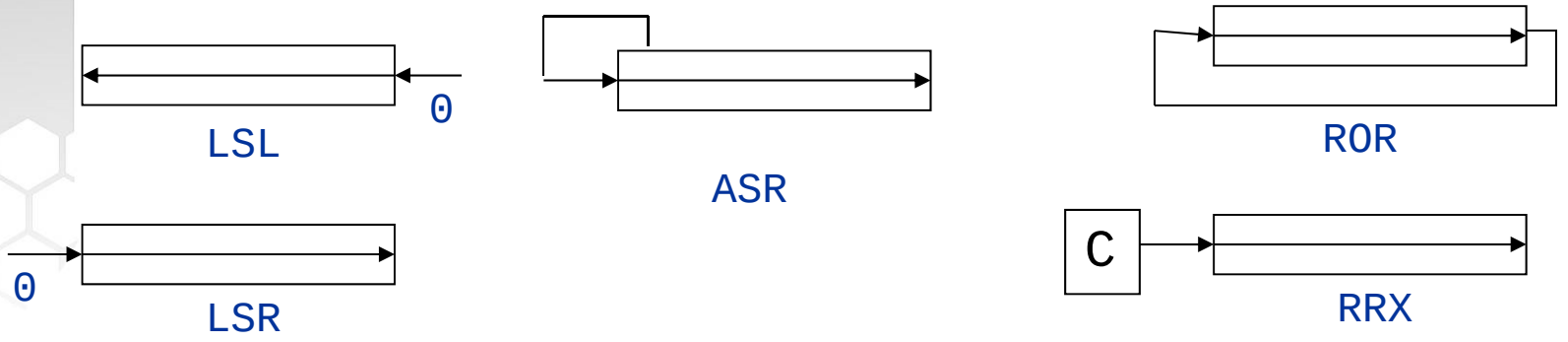
ARM集特有。第二个操作数先进行移位操作。

```
ADD   R3, R2, R1, LSL #3
; R3←R2＋8*R1
```



LSL: 逻辑左移（Logical Shift Left）
LSR: 逻辑右移（Logical Shift Right）
ASR: 算术右移（Arithmetic Shift Right）
ROR: 循环右移（Rotate Right）
RRX: 带扩展的循环右移（Rotate Right eXtended by 1 place ）

(3) 寄存器移位寻址



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

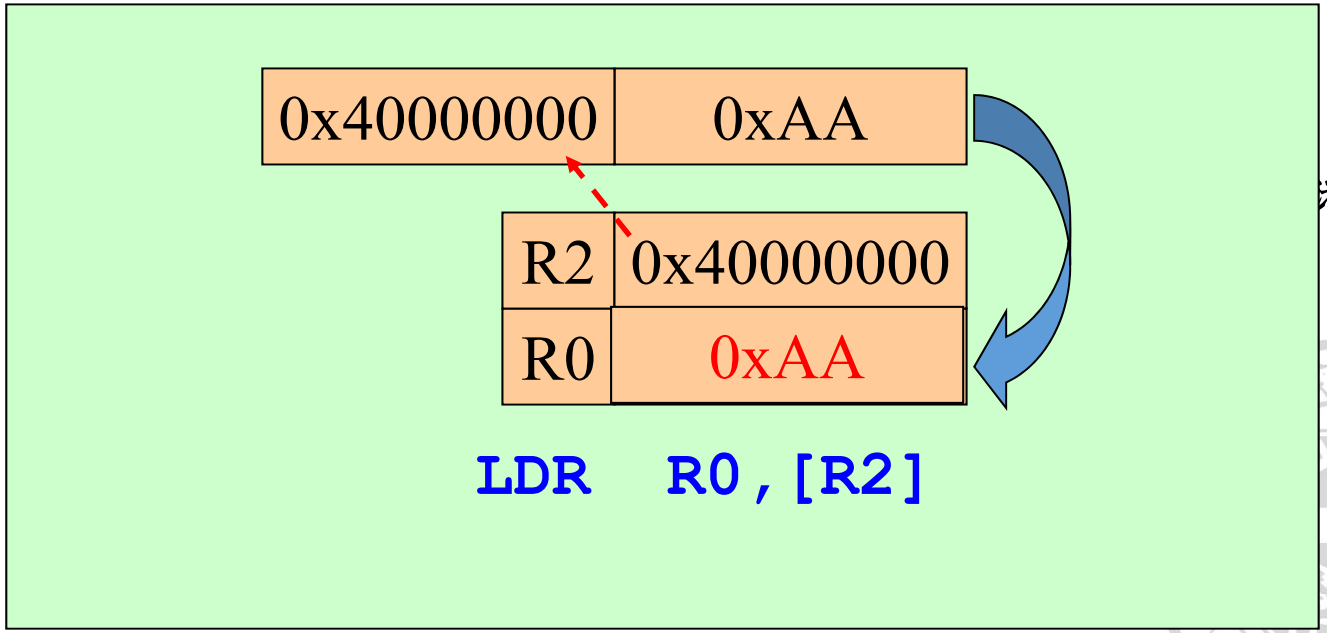
带扩展的循环右移操作

- 格式：通用寄存器，RRX 操作数
- 用途：对通用寄存器中的内容进行带扩展的循环右移操作，按操作数所指定 的数量向右循环移位，左端用进位标志位来填充。其中，操作数可以是通用寄存器中的数值，也可以是立即数(立即数的取值范围是0~31之间的整数)。
- 例如：MOV R0，R1，RRX#2 ；将R1中的内容进行带扩展的循环右移2位后传送到R0中

(4).寄存器间接寻址

寄存器间接寻址指令中的地址码给出的是一个通用寄存器的编号，所需的操作数保存在寄存器指定地址的存储单元中，即寄存器为操作数的地址指针。寄存器间接寻址指令举例如下：

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

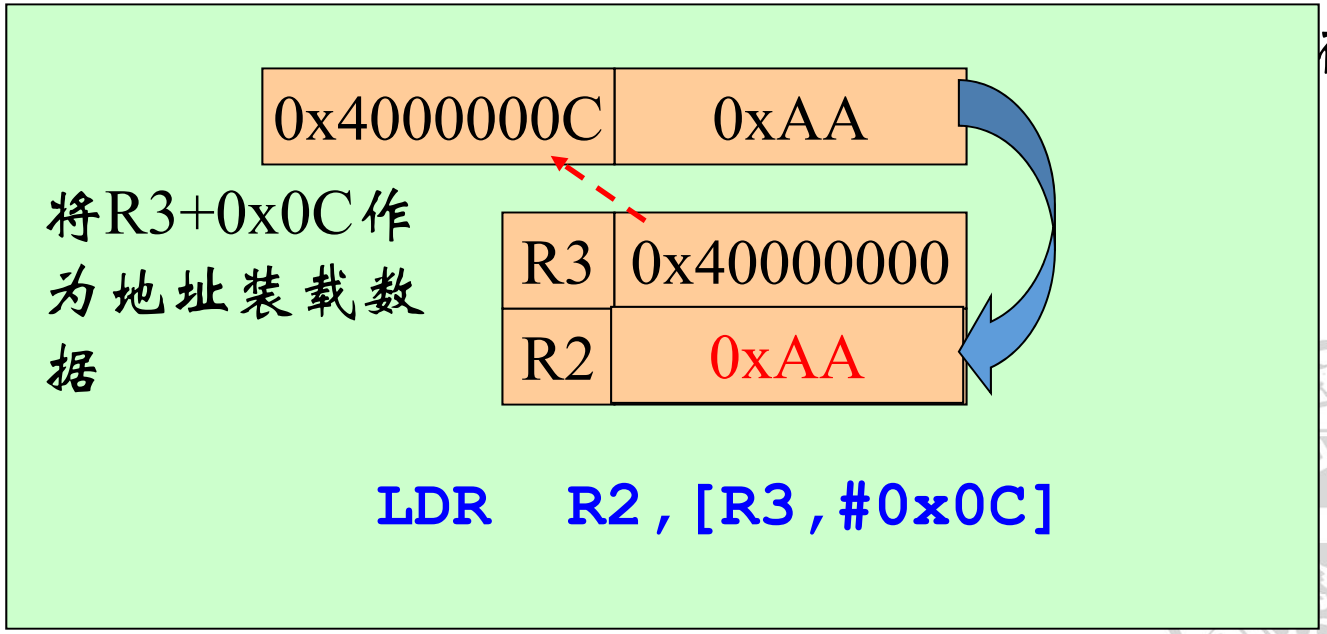


(5)基址寻址

基址寻址就是将基址寄存器的内容与指令中给出的偏移量相加，形成操作数的有效地址。基址寻址指令举例如下：

LDR R2,[R3,#0x0C] ;读取R3+0x0C地址上的存储单元
;的内容，放入R2

STR 存



(5)基址寻址

◆基址寻址

将基址寄存器的内容与指令中给出的地址偏移量相加，形成操作数的有效地址。基址寻址常用于访问基地址附近的存储单元。包括基址加偏移和基址加索引寻址。

基址加偏移—前索引寻址

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

基址加偏移—带自动索引的前索引寻址

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

基址加偏移—后索引寻址

```
LDR R0, [R1], #4          ; R0←[R1]、R1←R1+4
```

基址加索引寻址

```
LDR R0, [R1, R2]          ; R0←[R1+R2]
```

(6).多寄存器寻址

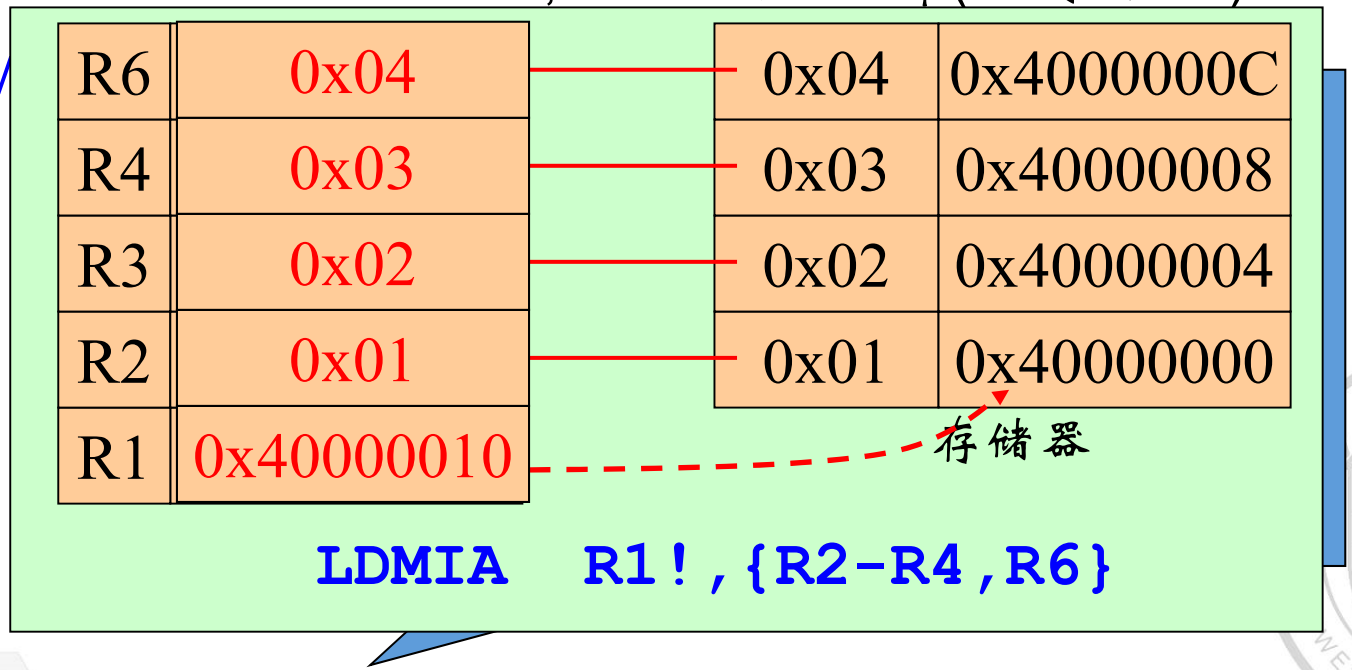
多寄存器寻址一次可传送几个寄存器值，允许一条指令传送16个寄存器的任何子集或所有寄存器。

多寄存器寻址指令举例如下：

LDMIA R1!,{R2-R7,R12} ;将R1指向的单元中的数据读出到

;R2～R7、R12中(R1自动加1)

STM

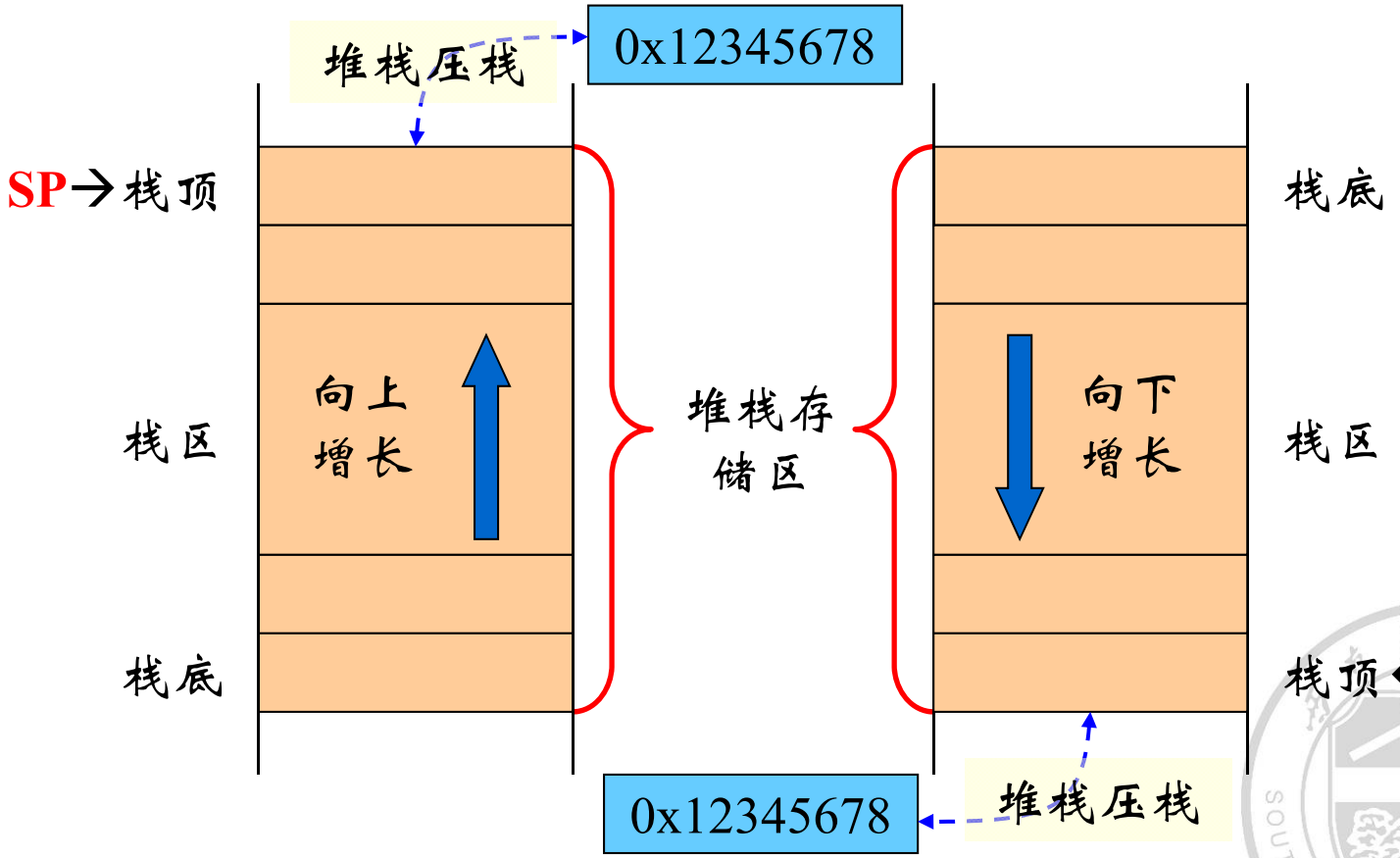


(7).堆栈寻址

堆栈是一个按特定顺序进行存取的存储区，操作顺序为“后进先出”。堆栈寻址是隐含的，它使用一个专门的寄存器(堆栈指针)指向一块存储区域(堆栈)，指针所指向的存储单元即是堆栈的栈顶。**存储器堆栈可分为两种：**

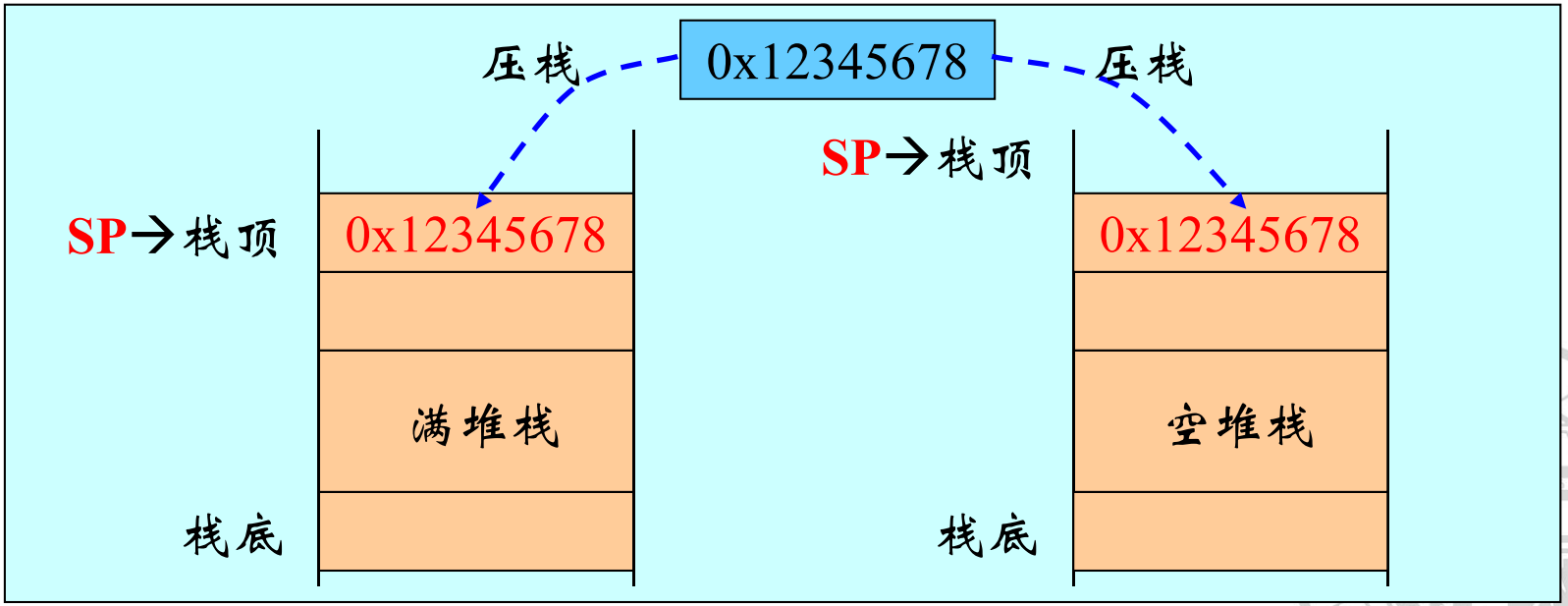
- **向上生长**：向高地址方向生长，称为递增堆栈
- **向下生长**：向低地址方向生长，称为递减堆栈

(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

(7).堆栈寻址

堆栈指针指向最后压入的堆栈的有效数据项，称为**满堆栈**；堆栈指针指向下一个待压入数据的空位置，称为**空堆栈**。



(7).堆栈寻址

所以可以组合出四种类型的堆栈方式：

- **满递增**：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等；
- **空递增**：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等；
- **满递减**：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等；
- **空递减**：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等。

(7) 堆栈寻址

```
STMFA r13!, {r0-r3} ; Push onto a Full Ascending Stack
LDMFA r13!, {r0-r3} ; Pop from a Full Ascending Stack
STMFD r13!, {r0-r3} ; Push onto a Full Descending Stack
LDMFD r13!, {r0-r3} ; Pop from a Full Descending Stack
STMEA r13!, {r0-r3} ; Push onto an Empty Ascending Stack
LDMEA r13!, {r0-r3} ; Pop from an Empty Ascending Stack
STMED r13!, {r0-r3} ; Push onto Empty Descending Stack
LDMED r13!, {r0-r3} ; Pop from an Empty Descending Stack
```

注意:

1.无论压栈过程还是出栈过程,存储器中的高地址的数据对应高编号寄存器.并且与大括号中寄存器的排放顺序无关

(8) 块拷贝寻址

多寄存器传送指令用于将一块数据从存储器的某一位置拷贝到另一位置。 如：

```
STMIA  R0!,{R1-R7}     ;将R1～R7的数据保存到存储器中。
                       ;存储指针在保存第一个值之后增加，
                       ;增长方向为向上增长。
STMIB  R0!,{R1-R7}     ;将R1～R7的数据保存到存储器中。
                       ;存储指针在保存第一个值之前增加，
                       ;增长方向为向上增长。
```

(8) 块拷贝寻址

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

地址增加在先（IB）：STMIB，LDMIB
地址增加在后（IA）：STMIA，LDMIA
地址减少在先（DB）：STMDB，LDMDB
地址减少在后（DA）：STMDA，LDMDA

I: Increment
D: Decrement
B: Before
A: After

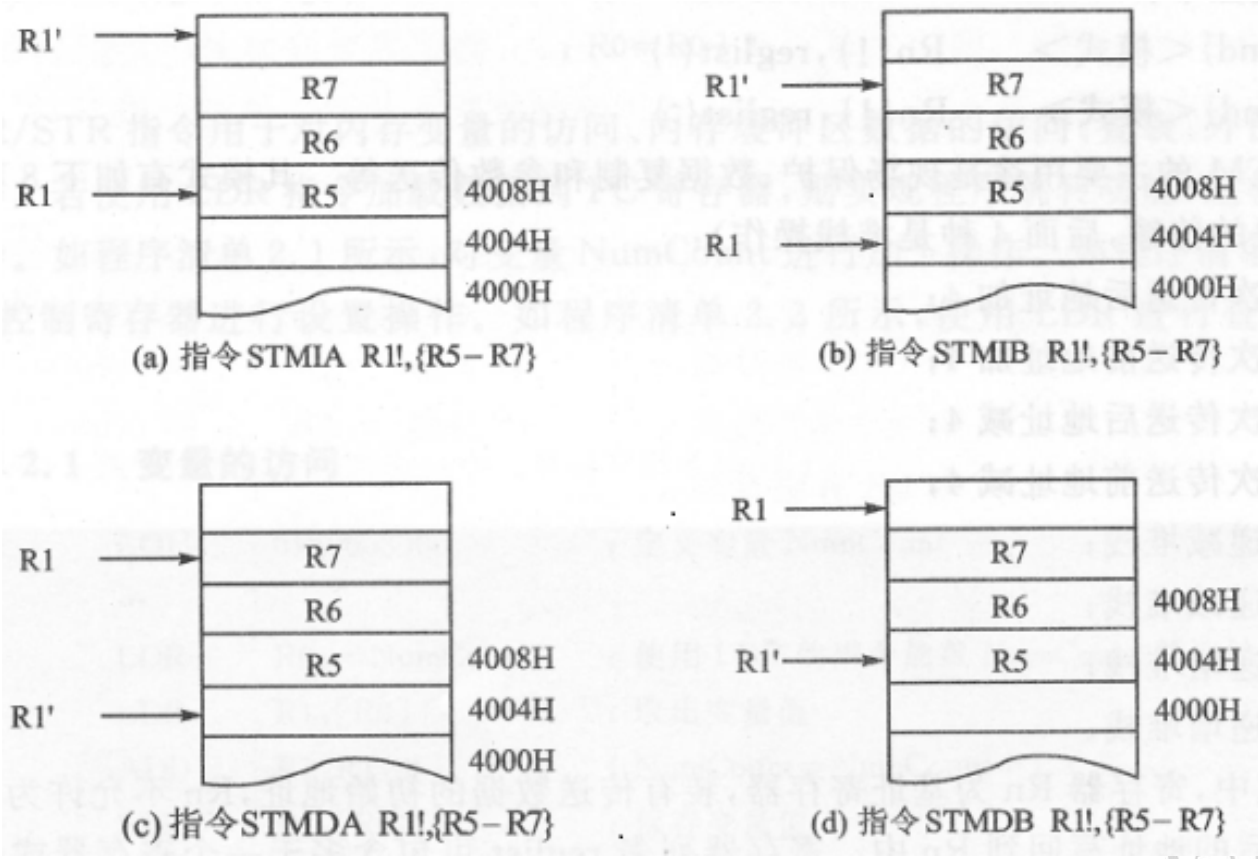
(8) 块拷贝寻址

多寄存器传送指令映射

增长的方向 / 增长的先后		向上生长		向下生长	
		满	空	满	空
增加	之前	STMIB STMFA			LDMIB LDMED
	之后		STMIA STMEA	LDMIA LDMFD	
减少	之前		LDMDB LDMEA	STMDB STMFD	
	之后	LDMDA LDMFA			STMDA STMED

(8) 块拷贝寻址

❖块复制指令操作



(9). 相对寻址

相对寻址是基址寻址的一种变通。由程序计数器PC提供基准地址，指令中的地址码字段作为偏移量，两者相加后得到的地址即为操作数的有效地址。

相对寻址指令举例如下：

```
    BL   SUBRI   ；调用到SUBRI子程序
    . . .
SUBR1
    …
    MOV PC,R14 ；返回
```

❖ 简单的ARM程序

```
;文件名：TEST1.S
;功[illegible]
;说[illegible]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                               ;文件结束
```

使用“；”进行注释

标号顶格写

实际代码段

声明文件结束

❖ 简单的ARM程序

```
;文件名：TEST1.S
;功能：实现两个寄存器相加
;说明：使用ARMulate软件仿真调试
        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                              ;文件结束
```

C语言调用汇编程序

```
;文件名：function.s
;功能：实现两个寄存器相加
;说明：使用ARMulate软件仿真调试
      AREA function, CODE, READONLY ;声明代码段function
      ENTRY                         ;标识程序入口
      CODE32                        ;声明32位ARM指令
      EXPORT   myadd                ;导出函数申明，可被别的文件调用
Myadd:
      ADD r0,r1,r0                  ;r0=r1+r0
      MOV pc,lr                     ;子程序返回
      END                           ;文件结束
```

```
//文件名：main.c
int myadd(int a,int b);        //申明在汇编中定义的函数，函数调用时，
                               //参数a放入r0,参数b放入r1中

int main(void){
   while(1){
       printf("myadd 2 and 3 is :%d\n",myadd(2,3);//函数调用
   }
}
```

第三节 ARM指令集

(1) 指令格式
(2) 条件码
(3) ARM存储器访问指令
(4) ARM数据处理指令
(5) 乘法指令
(6) ARM分支指令
(7) 协处理器指令
(8) 杂项指令
(9) 伪指令

一、ARM指令概述

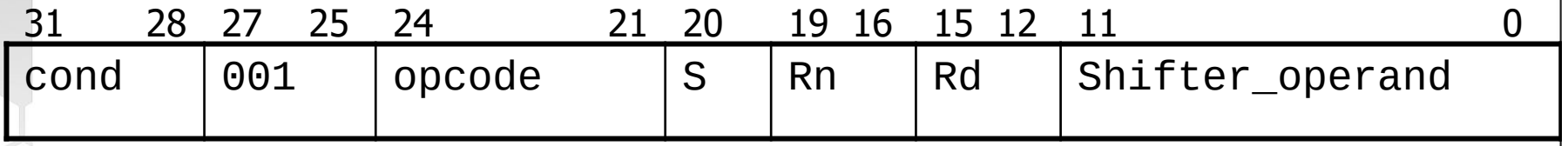
- ARM7TDMI(-S)的指令集，包括

  ARM指令集

  Thumb指令集

- 重点介绍ARM指令的基本格式及灵活的操作数，然后介绍条件码，再对ARM指令集按类分别说明。

(1)ARM指令的基本格式




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

其中<>号内的项是必须的，{}号内的项是可选的。

各项的说明如下：

opcode：指令助记符；　**cond**：执行条件；

S：是否影响CPSR寄存器的值；

Rd：目标寄存器；　**Rn**：第1个操作数的寄存器；

operand2：第2个操作数；

例：

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

(2)指令格式举例

- LDR　R0，[R1]；读取R1地址上的存储器单元内容，执行条件AL
- BEQ　D1　　；分支指令，执行条件EQ，即相等则跳转到D1
- ADDS　R1，R1，#1 ；加法指令，R1+1＝>R1，影响CPSR
  ；寄存器(S)
- SUBNES R1，R1，#0x10；条件执行减法运算(NE)，R1 -
  ; 0x10=>R1，影响CPSR寄存器(S)

（3）第2个操作数

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：

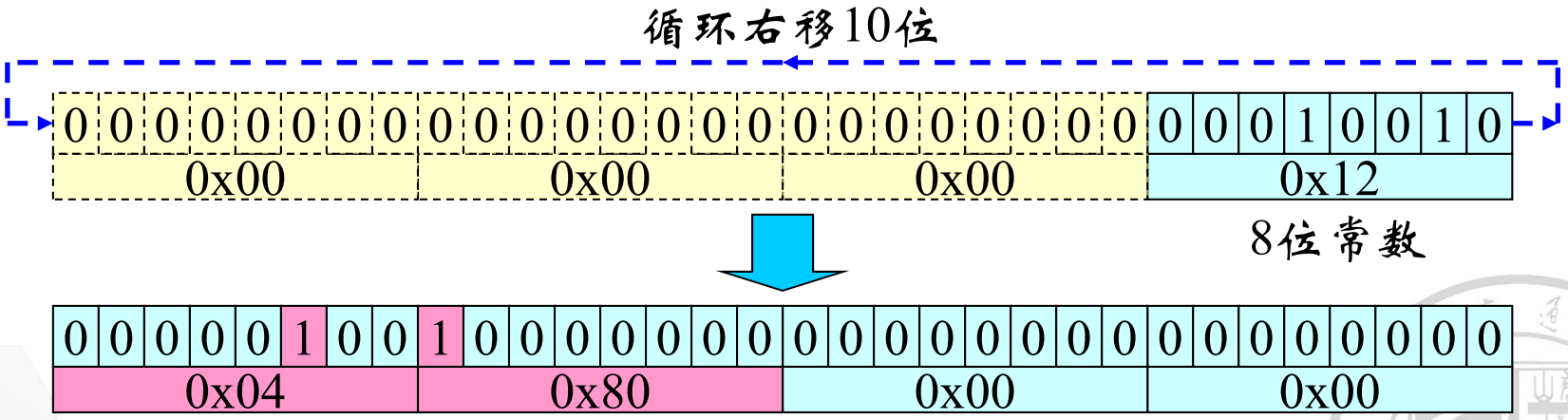
- **#immed_8r**——常数表达式；
- **Rm**——寄存器方式；
- **Rm,shift**——寄存器移位方式；

（3）第2个操作数

31 28	27 25	24 21	20	19 16	15 12	11 0
cond	001	opcode	S	Rn	Rd	Shifter_operand

- #immed_8r——常数表达式

该常数必须对应8位位图，即一个**8位的常数通过循环右移偶数位**得到。



（3）第2个操作数

- #immed_8r——常数表达式

合法常量：0x3FC、0、0xF0000000、0xC8。

非法常量：0xlFE、0x1FF、0xFFFF、0x1010、0xF0000001 。

常数表达式应用举例：

```
MOV   R0，#1                    ；R0＝1
AND   R1，R2，#0x0F   ；R2与0x0F，结果保存在RI
LDR   R0，[R1]，#-4     ；读取R1地址上的存储器单元内容，且 R1＝R1- 4
```

（3）第2个操作数

- Rm——寄存器方式

在寄存器方式下，操作数即为寄存器的数值。寄存器方式应用举例：

SUB　R1，R1，R2 ；R1-R2→R1

MOV　PC，R0　　；PC=R0，程序跳转到指定地址

LDR　R0，[R1]，-R2 ；RI所指存储器单元内容存人R0，且R1 = R1 - R2

（3）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，但Rm值保持不变，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。

（3）第2个操作数

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：

寄存器偏移方式应用举例：

```
ADD   R1，R1，R1，LSL #3    ；R1=R1×8
SUB   R1，R1，R2，LSR #2    ；R1=R1-R2/4
```

二、条件码

ARM指令的基本格式

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

使用条件码“cond”可以实现高效的逻辑操作，提高代码效率。

所有的ARM指令都可以条件执行，而Thumb指令只有B（跳转）指令具有条件执行 功能。如果指令不标明条件代码，将默认为无条件（AL）执行。

• 指令条件码表

操作码	条件助记符	标志	含义
0000	EQ	Z=1	相等
0001	NE	Z=0	不相等
0010	CS/HS	C=1	无符号数大于或等于
0011	CC/LO	C=0	无符号数小于
0100	MI	N=1	负数
0101	PL	N=0	正数或零
0110	VS	V=1	溢出
0111	VC	V=0	没有溢出
1000	HI	C=1,Z=0	无符号数大于
1001	LS	C=0,Z=1	无符号数小于或等于
1010	GE	N=V	有符号数大于或等于
1011	LT	N!=V	有符号数小于
1100	GT	Z=0,N=V	有符号数大于
1101	LE	Z=1,N!=V	有符号数小于或等于
1110	AL	任何	无条件执行 (指令默认条件)
1111	NV	任何	从不执行(不要使用)

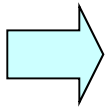
二、条件码

❖ ARM指令集——条件码

示例：

C代码：

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



对应的汇编代码：

```
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
```

三、ARM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和存储指令实现。ARM处理器是冯•诺依曼存储结构，程序空间、RAM空间及I/O映射空间统一编址，除对RAM操作以外，对外围IO、程序数据的访问均要通过加载/存储指令进行。

存储器访问指令分为**单寄存器操作指令**和**多寄存器操作指令**。

1、单寄存器加载

助记符	说明	操作	条件码位置
LDR Rd,addressing	加载字数据	Rd←[addressing]，addressing索引	LDR{cond}
LDRB Rd,addressing	加载无符号字节数据	Rd←[addressing]，addressing索引	LDR{cond}B
LDRT Rd,addressing	以用户模式加载字数据	Rd←[addressing]，addressing索引	LDR{cond}T
LDRBT Rd, addressing	以用户模式加载无符号字节数据	Rd←[addressing]，addressing索引	LDR{cond}BT
LDRH Rd, addressing	加载无符号半字数据	Rd←[addressing]，addressing索引	LDR{cond}H
LDRSB Rd, addressing	加载有符号字节数据	Rd←[addressing]，addressing索引	LDR{cond}SB
LDRSH Rd, addressing	加载有符号半字数据	Rd←[addressing]，addressing索引	LDR{cond}SH

2、单寄存器存储

助记符	说明	操作	条件码位置
STR Rd, addressing	存储字数据	[addressing]←Rd，addressing索引	STR{cond}
STRB Rd,addressing	存储字节数据	[addressing]←Rd，addressing索引	STR{cond}B
STRT Rd,addressing	以用户模式存储字数据	[addressing]←Rd，addressing索引	STR{cond}T
STRBT Rd,addressing			STR{cond}BT
STRH Rd,addressing			STR{cond}H

说明：

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。

LDR/STR指令用于对[illegible]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，这样也就实现了程序散转。

2、单寄存器存储

•LDR和STR——字和无符号字节加载/存储指令

LDR指令用于从内存中读取单[illegible]存入寄存器中，STR指[illegible]字节数据保存到内存[illegible]

注意：T为可选后缀。若指令有T，那么即使处理器是在特权模式下，存储系统也将访问看成是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。

LDR{cond}{T} Rd,<地址> ;[illegible]

STR{cond}{T} Rd,<地址> ;[illegible]

LDR{cond}B{T} Rd,<地址> ;将指[illegible]读入Rd

STR{cond}B{T} Rd,<地址> ;将[illegible]的字节数据存入指定地址

2、单寄存器存储

- LDR和STR——字和无符号字节加载/存储指令

LDR/STR指令寻址非常灵活，它由两部分组成，其中一部分为一个基址寄存
分为一个地址偏移

- **立即数：** 立即
到基址寄存器，也

- **寄存器：** 寄存器
址寄存器中减去这

指令举例如下：
LDR R1，[R0，R2，LSL #2] ；将**R0+R2×4**地址处的数据读出，保存到**R1**中**(R0、R2**的值不变)
LDR R1，[R0，-R2，LSL #2]；将**R0-R2×4**地址处的数据读出，保存到**R1**中**(R0、R2**的值不变)

- **寄存器及移位常数：** 寄存器移位后的值可以加到基址寄存器，也可以从基址寄存器中减去这个数值。

2、单寄存器存储

- LDR和STR——字和无符号字节加载/存储指令

从寻址方式的地址计算方法分，加载/存储指令有以下4种格式：

- 零偏移：　　如：LDR　Rd,[Rn]
- 前索引偏移：　　如：LDR　Rd,[Rn,#0x04]!

　　LDR　Rd，[Rn，# -0x04]

- 程序相对偏移：　　如：LDR　Rd,labe1
- 后索引偏移：　　如：LDR

注意：

大多数情况下，必须保证字数据操作的地址是32位对齐的。

2、单寄存器存储

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零扩展到32位；
2.地址对齐——半字读写的指定地址必须为偶数，否则将产生不可靠的结果。

LDR{cond}SB Rd,<地址>;将指定地址上的有符号字节读入Rd

LDR{cond}SH Rd,<地址>;将指定地址上的有符号半字读入Rd

LDR{cond}H Rd,<地址>;将指定地址上的半字数据读入Rd

STR{cond}H Rd,<地址>;将Rd中的半字数据存入指定地址

2、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

举例如下：

```
LDRSB  R1，[R0，R3]  ；将R0+R3地址上的字节数据读到R1，高24位
                     ；用符号位扩展
LDRSH  R1，[R9]      ；将R9地址上的半字数据读出到R1，高16位用符号
                     ；位扩展
LDRH  R6,[R2]，#2；将R2地址                                    扩
                     ；展，R2=
STRH   R1，[Ro，#2]!  ；将R                                     2
                     ；字节
```

说明：

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，这样也就实现了程序散转。

3、多寄存器存取

LDM和STM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。

LDM为加载多个寄存器；

STM为存储多个寄存器。

允许一条指令传送16个寄存器的任何子集或所有寄存器。

指令格式如下：

LDM{cond}<模式> Rn{!},reglist{^}

STM{cond}<模式> Rn{!},reglist{^}

LDM和STM的**主要用途**是现场保护、数据复制、常数传递等

3、多寄存器存取

多寄存器加载/存储指令的8种模式如下表所示，右边四种为堆栈操作、左边四种为数据传送操作。

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

进行数据复制时，先设置好源数据指针和目标指针，然后使用块拷贝寻址指令LDMIA/STMIA、LDMIB/STMIB、LDMDA/STMDA、LDMDB/STMDB进行读取和存储。

进行堆栈操作时，要先设置堆栈指针（SP），然后使用堆栈寻址指令STMFD/LDMFD、STMED/LDMED、STMFA/LDMFA和STMEA/LDMEA实现堆栈操作。

3、多寄存器存取

- 指令格式中，寄存器Rn为基址寄存器，装有传送数据的初始地址，Rn不允许为R15。
- 后缀“!”表示最后的地址写回到Rn中。
- 寄存器列表reglist可包含多于一个寄存器或包含寄存器范围，使用“，”分开，如{R1，R2，R6-R9}，寄存器按由小到大排列。
- 后缀“^”不允许在用户模式或系统模式下使用。若在LDM指令且寄存器列表中包含有PC时使用，那么除了正常的多寄存器传送外，将SPSR也拷贝到CPSR中，这可用于异常处理返回。
- 使用后缀“^”进行数据传送且寄存器列表不包含PC时，加载／存储的是用户模式的寄存器，而不是当前模式的寄存器。

3、多寄存器存取

- 当Rn在寄存器列表中且使用后缀“!”时,对于STM指令,若Rn为寄存器列表中的最低数字的寄存器,则会将Rn的初值保存;其它情况下Rn的加载值和存储值不可预知。

- 地址对齐—这些指令忽略地址位[1:0]。

举例如下：

```
LDMIA   R0!，{R3 - R9}；加载R0指向地址上的多字数据，保存
                       ；到R3~R9中，R0值更新
STMIA   R1!，{R3 - R9}；将R3～R9的数据存储到R1指向的地址
                       ；上，R1值更新
STMFD    SP!，{R0 - R7，LR}   ；现场保存，将R0～R7、LR入栈
LDMFD    SP!，{R0 - R7，PC}   ；恢复现场，异常处理返回
```

4、寄存器和存储器交换指令

SWP指令用于将一个内存单元(该单元地址放在寄存器Rn中)的内容读取到一个寄存器Rd中，同时将另一个寄存器Rm的内容写入到该内存单元中。使用SWP可实现信号量操作。

指令格式如下：

SWP指令应用示例：

```
SWP     R1,R1,[R0]      ;将R1的内容与R0指向的存储单元的内容进行交换
SWPB    R1,R2,[R0]      ;将R0指向的存储单元内的容读取一字节数据到R1中
                        ;(高24位清零)，并将R2的内容写入到该内存单元中
                        ;(最低字节有效)
```

例题

- 内存0X30000000开始有80个字数据，编写程序，将其拷贝到0x40000000的位置，

R0=0x30000000,R1=0X40000000,R2=80

(可以使用LDMIA和STMIA指令，一次传送8个字数据)

方法1

- LOOP
- LDMIA R0!,{R4-R11}
- STMIA R1!,{R4-R11}
- SUBS R2,R2,#8
- BNE LOOP

方法2

- LOOP

  LDR R3，[R0，R2，LSL #2]

  STR R3，[R1，R2，LSL #2]

  SUBS R2，R2，#1

  BNE　LOOP

ARM指令集

(1).指令格式
(2).条件码
(3).ARM存储器访问指令
(4). ARM数据处理指令
(5).乘法指令
(6). ARM分支指令
(7).协处理器指令
(8).杂项指令
(9).伪指令

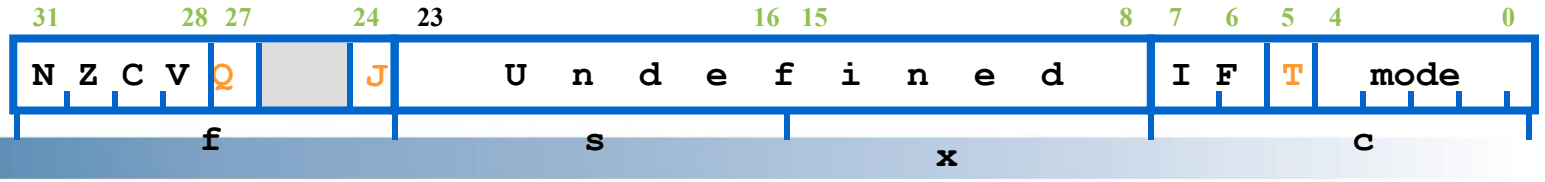
四、ARM数据处理指令

数据处理指令大致可分为3类：

- 数据传送指令；
- 算术逻辑运算指令；
- 比较指令。

数据处理指令只能对寄存器的内容进行操作，而不能对内存中的数据进行操作。所有ARM数据处理指令均可选择使用S后缀，并影响状态标志。

比较指令**CMP**、**CMN**、**TST**和**TEQ**不需要后缀**S**，它们会直接影响状态标志。



❖ 后缀s

- ❖ 数据处理指令可以选择s后缀，以影响状态标志。但是比较指令（cmp、cmn、tst和teq）不需要后缀s，它们总会直接影响cpsr中的状态标志。
- ❖ 在数据处理指令中，除了比较指令以外，其它的指令如果带有s后缀，同时又以pc为目标寄存器进行操作，则操作的同时从spsr恢复cpsr。比如：
  - movs pc, #0xff /* cpsr = spsr; pc = 0xff */
  - adds pc, r1, #0xffffff00 /* cpsr = spsr; pc = r1 + 0xffffff00 */
  - ands pc, r1, r2 /* cpsr = spsr; pc = r1 & r2; */

• 指令条件码表

操作码	条件助记符	标志	含义
0000	EQ	Z=1	相等
0001	NE	Z=0	不相等
0010	CS/HS	C=1	无符号数大于或等于
0011	CC/LO	C=0	无符号数小于
0100	MI	N=1	负数
0101	PL	N=0	正数或零
0110	VS	V=1	溢出
0111	VC	V=0	没有溢出
1000	HI	C=1,Z=0	无符号数大于
1001	LS	C=0,Z=1	无符号数小于或等于
1010	GE	N=V	有符号数大于或等于
1011	LT	N!=V	有符号数小于
1100	GT	Z=0,N=V	有符号数大于
1101	LE	Z=1,N!=V	有符号数小于或等于
1110	AL	任何	无条件执行 (指令默认条件)
1111	NV	任何	从不执行(不要使用)

条件码

❖ ARM指令集——条件码

示例：

C代码：

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

对应的汇编代码：

```
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
```

C代码：

```
for(i=100;i!=0;i--)
```

对应的汇编代码：

```
        MOV    R0,#100
LOOP
        SUBS   R0,R0,#1;R0减1,影响标志位
        BNE    LOOP;不等于0,运行下一循环
```

1、数据传送

助记符	说明	操作	条件码位置
MOV Rd, operand2	数据传送	Rd←operand2	MOV{cond}{S}
MVN Rd, operand2	数据非传送	Rd←(~operand2)	MVN{cond}{S}

MOV指令举例如下：

```
MOVS R3, R1, LSL #2      ; R3=R1<<2，并影响标志位
MOV  PC, LR              ; PC=LR，子程序返回
```

1、数据传送

助记符	说明	操作	条件码位置
MOV Rd,operand2	数据传送	Rd←operand2	MOV{cond}{S}
MVN Rd,operand2	数据非传送	Rd←(~operand2)	MVN{cond}{S}

MVN指令举例如下：

```
MVN     R1,#0xFF    ; R1=0xFFFFFF00
MVN     R1,R2       ; 将R2取反，结果存到R1
```

2、算术运算

助记符	说明	操作	条件码位置
ADD Rd, Rn, operand2	加法运算指令	Rd←Rn+operand2	ADD{cond}{S}
SUB Rd, Rn, operand2	减法运算指令	Rd←Rn-operand2	SUB{cond}{S}
RSB Rd, Rn, operand2	逆向减法指令	Rd←operand2-Rn	RSB{cond}{S}
ADC Rd, Rn, operand2	带进位加法	Rd←Rn+operand2+Carry	ADC{cond}{S}
SBC Rd, Rn, operand2	带进位减法指令	Rd←Rn-operand2-(NOT)Carry	SBC{cond}{S}
RSC Rd, Rn, operand2	带进位逆向减法指令	Rd←operand2-Rn-(NOT)Carry	RSC{cond}{S}

2、算术运算

助记符	说明	操作	条件码位置
ADD Rd, Rn, operand2	加法运算指令	Rd←Rn+operand2	ADD{cond}{S}
SUB Rd, Rn, operand2	减法运算指令	Rd←Rn-operand2	SUB{cond}{S}
RSB Rd, Rn, operand2	逆向减法指令	Rd←operand2-Rn	RSB{cond}{S}
ADC Rd, Rn, operand2	带进位加法	Rd←Rn+operand2+Carry	ADC{cond}{S}
SBC Rd, Rn, operand2	带进位减法指令	Rd←Rn-operand2-(NOT)Carry	SBC{cond}{S}
RSC Rd, Rn, operand2	带进位逆向减法指令	Rd←operand2-Rn-(NOT)Carry	RSC{cond}{S}

应用示例：

```
ADDS   R1,R1,#1                ;R1=R1+1，并影响标志位
ADDS   R3，R1，R2，LSL #2 ; R3＝R1+R2<<2
```

2、算术运算

助记符	说明	操作	条件码位置
ADD Rd, Rn, operand2	加法运算指令	Rd←Rn+operand2	ADD{cond}{S}
SUB Rd, Rn, operand2	减法运算指令	Rd←Rn-operand2	SUB{cond}{S}
RSB Rd, Rn, operand2	逆向减法指令	Rd←operand2-Rn	RSB{cond}{S}
ADC Rd, Rn, operand2	带进位加法	Rd←Rn+operand2+Carry	ADC{cond}{S}
SBC Rd, Rn, operand2	带进位减法指令	Rd←Rn-operand2-(NOT)Carry	SBC{cond}{S}
RSC Rd, Rn, operand2	带进位逆向减法指令	Rd←operand2-Rn-(NOT)Carry	RSC{cond}{S}

应用示例：

```
SUBS    R0,R0,#1            ;R0=R0-1
SUB   R6，R7，#0x10      ; R6＝R7-0x10
```

2、算术运算

助记符	说明	操作	条件码位置
ADD Rd, Rn, operand2	加法运算指令	Rd←Rn+operand2	ADD{cond}{S}
SUB Rd, Rn, operand2	减法运算指令	Rd←Rn-operand2	SUB{cond}{S}
RSB Rd, Rn, operand2	逆向减法指令	Rd←operand2-Rn	RSB{cond}{S}
ADC Rd, Rn, operand2	带进位加法	Rd←Rn+operand2+Carry	ADC{cond}{S}
SBC Rd, Rn, operand2	带进位减法指令	Rd←Rn-operand2-(NOT)Carry	SBC{cond}{S}
RSC Rd, Rn, operand2	带进位逆向减法指令	Rd←operand2-Rn-(NOT)Carry	RSC{cond}{S}

应用示例：

```
RSB   R3,R1,#0xFF00          ;R3=0xFF00-R1
RSBS  R1,R2,R2,LSL #2        ;R1=(R2<<2)-R2=R2×3
```

2、算术运算

助记符	说明	操作	条件码位置
ADD Rd, Rn, operand2	加法运算指令	Rd←Rn+operand2	ADD{cond}{S}
SUB Rd, Rn, operand2	减法运算指令	Rd←Rn-operand2	SUB{cond}{S}
RSB Rd, Rn, operand2	逆向减法指令	Rd←operand2-Rn	RSB{cond}{S}
ADC Rd, Rn, operand2	带进位加法	Rd←Rn+operand2+Carry	ADC{cond}{S}
SBC Rd, Rn, operand2	带进位减法指令	Rd←Rn-operand2-(NOT)Carry	SBC{cond}{S}
RSC Rd, Rn, operand2	带进位逆向减法指令	Rd←operand2-Rn-(NOT)Carry	RSC{cond}{S}

应用示例：

```
ADDS   R0,R0,R2          ;使用ADC实现64位加法
ADC  R1,R1,R3            ;(R1、R0)＝(R1、R0)+(R3、R2)
```

2、算术运算

助记符	说明	操作	条件码位置
ADD Rd, Rn, operand2	加法运算指令	Rd←Rn+operand2	ADD{cond}{S}
SUB Rd, Rn, operand2	减法运算指令	Rd←Rn-operand2	SUB{cond}{S}
RSB Rd, Rn, operand2	逆向减法指令	Rd←operand2-Rn	RSB{cond}{S}
ADC Rd, Rn, operand2	带进位加法	Rd←Rn+operand2+Carry	ADC{cond}{S}
SBC Rd, Rn, operand2	带进位减法指令	Rd←Rn-operand2-(NOT)Carry	SBC{cond}{S}
RSC Rd, Rn, operand2	带进位逆向减法指令	Rd←operand2-Rn-(NOT)Carry	RSC{cond}{S}

应用示例：

```
SUBS   R0,R0,R2          ;使用SBC实现64位减法
SBC    R1,R1,R3; (R1、R0)＝(R1、R0)-(R3、R2)
```

2、算术运算

助记符	说明	操作	条件码位置
ADD Rd, Rn, operand2	加法运算指令	Rd←Rn+operand2	ADD{cond}{S}
SUB Rd, Rn, operand2	减法运算指令	Rd←Rn-operand2	SUB{cond}{S}
RSB Rd, Rn, operand2	逆向减法指令	Rd←operand2-Rn	RSB{cond}{S}
ADC Rd, Rn, operand2	带进位加法	Rd←Rn+operand2+Carry	ADC{cond}{S}
SBC Rd, Rn, operand2	带进位减法指令	Rd←Rn-operand2-(NOT)Carry	SBC{cond}{S}
RSC Rd, Rn, operand2	带进位逆向减法指令	Rd←operand2-Rn-(NOT)Carry	RSC{cond}{S}

应用示例：

```
RSBS  R2,R0,#0
RSC   R3,R1,#0  ;使用RSC指令实现求64位数值的负数
```

3、逻辑运算指令

助记符	说明	操作	条件码位置
AND Rd, Rn, operand2	逻辑与操作指令	Rd←Rn & operand2	AND{cond}{S}
ORR Rd, Rn, operand2	逻辑或操作指令	Rd←Rn | operand2	ORR{cond}{S}
EOR Rd, Rn, operand2	逻辑异或操作指令	Rd←Rn ^ operand2	EOR{cond}{S}
BIC Rd, Rn, operand2	位清除指令	Rd←Rn & (~operand2)	BIC{cond}{S}

3、逻辑运算指令

助记符	说明	操作	条件码位置
AND Rd, Rn, operand2	逻辑与操作指令	Rd←Rn & operand2	AND{cond}{S}
ORR Rd, Rn, operand2	逻辑或操作指令	Rd←Rn | operand2	ORR{cond}{S}
EOR Rd, Rn, operand2	逻辑异或操作指令	Rd←Rn ^ operand2	EOR{cond}{S}
BIC Rd, Rn, operand2	位清除指令	Rd←Rn & (~operand2)	BIC{cond}{S}

应用示例：

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

3、逻辑运算指令

助记符	说明	操作	条件码位置
AND Rd, Rn, operand2	逻辑与操作指令	Rd←Rn & operand2	AND{cond}{S}
ORR Rd, Rn, operand2	逻辑或操作指令	Rd←Rn | operand2	ORR{cond}{S}
EOR Rd, Rn, operand2	逻辑异或操作指令	Rd←Rn ^ operand2	EOR{cond}{S}
BIC Rd, Rn, operand2	位清除指令	Rd←Rn & (~operand2)	BIC{cond}{S}

逻辑或操作指令——ORR指令将operand2的值与寄存器Rn的值按位作逻辑“或”操作，结果保存到Rd中。指令格式如下：

```
ORR{cond}{S}    Rd,Rn, operand2
```

应用示例：

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

3、逻辑运算指令

助记符	说明	操作	条件码位置
AND Rd, Rn, operand2	逻辑与操作指令	Rd←Rn & operand2	AND{cond}{S}
ORR Rd, Rn, operand2	逻辑或操作指令	Rd←Rn | operand2	ORR{cond}{S}
EOR Rd, Rn, operand2	逻辑异或操作指令	Rd←Rn ^ operand2	EOR{cond}{S}
BIC Rd, Rn, operand2	位清除指令	Rd←Rn & (~operand2)	BIC{cond}{S}

逻辑异或操作指令——EOR指令将operand2的值与寄存器Rn的值按位作逻辑“异或”操作，结果保存到Rd中。指令格式如下：

```
EOR{cond}{S}    Rd,Rn, operand2
```

应用示例：

```
EOR     R1,R1,#0x0F    ;将R1的低4位取反
EORS    R0,R5,#0x01    ; 将R5和0x01进行逻辑异或，
                       ;结果保存到R0，并影响标志位
```

3、逻辑运算指令

助记符	说明	操作	条件码位置
AND Rd, Rn, operand2	逻辑与操作指令	Rd←Rn & operand2	AND{cond}{S}
ORR Rd, Rn, operand2	逻辑或操作指令	Rd←Rn | operand2	ORR{cond}{S}
EOR Rd, Rn, operand2	逻辑异或操作指令	Rd←Rn ^ operand2	EOR{cond}{S}
BIC Rd, Rn, operand2	位清除指令	Rd←Rn & (~operand2)	BIC{cond}{S}

位清除指令——BIC指令将寄存器Rn的值与operand2的值的反码按位作逻辑“与”操作，结果保存到Rd中。指令格式如下：

```
BIC{cond}{S}    Rd,Rn, operand2
```

应用示例：

```
BIC    R1,R1,#0x0F      ;将R1的低4位清零，其它位不变
```

例：写出以下ARM汇编代码

- 设R0=0X55667788
  - 1.取出R0 的高八位(25-32位)，结果放在R1中
  - 2.将R0的第12位设置为1，其它位保持不变
  - 3.将R0的第13位取反，其它位保持不变
  - 4.将R0的次高八位（第 16到23位清零），其它位保持不变。
  - ANDS　R1, R0, #0xFF000000
  - **ORR　R0,R0,#0x000000800**
  - **EOR　R0,R0,#0x000001000**
  - BIC　R0, R0, #0x00FF000000

4、比较指令

助记符	说明	操作	条件码位置
CMP Rn, operand2	比较指令	标志N、Z、C、V←Rn-operand2	CMP{cond}
CMN Rn, operand2	负数比较指令	标志N、Z、C、V←Rn+operand2	CMN{cond}
TST Rn, operand2	位测试指令	标志N、Z、C、V←Rn & operand2	TST{cond}
TEQ Rn, operand2	相等测试指令	标志N、Z、C、V←Rn ^ operand2	TEQ{cond}

4、比较指令

助记符	说明	操作	条件码位置
CMP Rn, operand2	比较指令	标志N、Z、C、V←Rn-operand2	CMP{cond}
CMN Rn, operand2	负数比较指令	标志N、Z、C、V←Rn+operand2	CMN{cond}
TST Rn, operand2	位测试指令	标志N、Z、C、V←Rn & operand2	TST{cond}
TEQ Rn, operand2	相等测试指令	标志N、Z、C、V←Rn ^ operand2	TEQ{cond}

比较指令——CMP指令将寄存器Rn的值减去operand2的值，根据操作的结果更新CPSR中的相应条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。指令格式如下：

```
CMP{cond}   Rn, operand2
```

应用示例：

```
CMP  R1,#10                    ;R1与10比较，设置相关标志位
```

4、比较指令

助记符	说明	操作	条件码位置
CMP Rn, operand2	比较指令		
CMN Rn, operand2	负数比较指		
TST Rn, operand2	位测试指令		ond}
TEQ Rn, operand2	相等测试指令	perand2	TEQ{cond}

注意：
CMN指令与ADDS指令的区别在于CMN指令不保存运算结果。
位。

负数比较指令——CMN指令使用寄存器Rn的值加上operand2的值，根据操作的结果更新CPSR中的相应条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。指令格式如下：

应用示例：

```
CMN   R0，#1  ；R0+1，判断R0是否为1的补码。若是，则Z
              ；位置1。
```

4、比较指令

助记符	说明	操作	条件码位置
CMP Rn, operand2	比较指令	标志N Z V	MP{cond}
CMN Rn, operand2	负数比较		nd}
TST Rn, operand2	位测试		d}
TEQ Rn, operand2	相等		ond}

注意：

TST指令与**ANDS**指令的区别在于TST指令不保存运算结果。

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

位测试指令——TST指令将Rn的值按位作逻辑“与”操作，根据操作的结果更新CPSR中的条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。指令格式如下：

```
TST{cond}    Rn, operand2
```

应用示例：

```
TST R0,#0x01        ; 判断R0的最低位是否为0
TST R1,#0x0F        ; 判断R1的低4位是否为0
```

4、比较指令

助记符	说明		位置
CMP Rn, operand2	比较指令		
CMN Rn, operand2	负数比较		
TST Rn, operand2	位测试指		
TEQ Rn, operand2	相等测试		d}

注意：

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

相等测试指令——TEQ指令将[illegible]2的值按位作逻辑“异或”操作，根据操作的结[illegible]更新CPSR中的相应条件标志位，以便后面的指令根据相应的条件标志来判断是否执行。指令格式如下：

应用示例：

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

ARM指令集

(1).指令格式
(2).条件码
(3).ARM存储器访问指令
(4)ARM数据处理指令
(5) ARM分支指令
(6) 乘法指令
(7).协处理器指令
(8).杂项指令
(9).伪指令

5、分支指令（跳转）

在ARM中有两种方式可以实现程序的跳转：

- **一种是使用分支转移指令直接跳转；**
- **另一种则是直接向PC寄存器赋值来实现跳转。**

ARM的分支转移指令，可以从当前指令向前或向后的32MB的地址空间跳转，根据完成的功能它可以分为以下4种：

- **B　　分支指令**
- **BL　带链接的分支指令**
- **BX　带状态切换的分支指令**
- **BLX　带链接和状态切换的分支指令**

（1）B—转移指令

指令格式如下：

B{cond}　label

B指令跳转到指定的地址执行程序。

指令举例如下：

B　WAITA　；跳转到WAITA标号处

B　0x1234　；跳转到绝对地址0x1234处

转移指令B限制在当前指令的±32 MB的范围内。

（1）B—转移指令（举例）

无条件跳转：

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

❖执行10次循环：

```
        MOV     R0, #10
LOOP
        ……
        SUBS    R0，R0, #1
        BNE     LOOP
```

（2）BL—带链接转移指令

指令格式如下：

BL{cond}　label

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

指令举例如下：

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

注意：转移地址限制在当前指令的±32 MB的范围内。

BL指令用于子程序调用。

```
        BL          SUB1
          ……
SUB1   STMFD    R13！,{R0-R3,R14}
          ……
        BL          SUB2
          ……
SUB2    ……
```

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

（3）BX—带状态切换的转移指令

指令格式如下：

BX{cond}　Rm

BX指令跳转到Rm指定的地址执行程序。

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

处理器状态切换

```
    CODE32        ;ARM状态下的代码
    LDR R0, =Into_Thumb+1
                  ;产生跳转地址并且设置最低位
    BX R0         ;Branch Exchange 进入Thumb状态
    …
    CODE16        ;Thumb状态下的子函数
Into_Thumb
    …
    LDR R3, =Back_to_ARM
    ;产生字对齐的跳转地址，最低位被清除
    BX R3         ;Branch Exchange 返回到ARM状态
    CODE32        ;ARM状态下的子函数
Bach_to_ARM
```

（4）BLX—带链接和状态切换的转移指令

指令格式如下：

BLX　<target address>

BLX指令先将下一条指令的地址拷贝到R14（即LR)连接寄存器中，然后跳转到指定地址处执行程序。(只有V5T及以上体系 支持BLX)

转移地址限制在当前指令的±32MB的范围内。

6、乘法指令（自学为主）

ARM7TDMI具有三种乘法指令，分别为：

- 32×32位乘法指令；
- 32× 32位乘加指令；
- 32× 32位结果为64位的乘/乘加指令。

6、乘法指令

助记符	说明	操作	条件码位置
MUL Rd, Rm, Rs	32位乘法指令	Rd←Rm*Rs (Rd≠Rm)	MUL{cond}{S}
MLA Rd, Rm, Rs, Rn	32位乘加指令	Rd←Rm*Rs+Rn (Rd≠Rm)	MLA{cond}{S}
UMULL RdLo, RdHi, Rm, Rs	64位无符号乘法指令	(RdLo, RdHi) ←Rm*Rs	UMULL{cond}{S}
UMLAL RdLo, RdHi, Rm, Rs	64位无符号乘加指令	(RdLo, RdHi) ←Rm*Rs+(RdLo, RdHi)	UMLAL{cond}{S}
SMULL RdLo, RdHi, Rm, Rs	64位有符号乘法指令	(RdLo, RdHi) ←Rm*Rs	SMULL{cond}{S}
SMLAL RdLo, RdHi, Rm, Rs	64位有符号乘加指令	(RdLo, RdHi) ←Rm*Rs+(RdLo, RdHi)	SMLAL{cond}{S}

6、乘法指令

助记符	说明	操作	条件码位置
MUL Rd, Rm, Rs	32位乘法指令	Rd←Rm*Rs (Rd≠Rm)	MUL{cond}{S}
MLA Rd, Rm, Rs, Rn	32位乘加指令	Rd←Rm*Rs+Rn (Rd≠Rm)	MLA{cond}{S}
UMULL RdLo, RdHi, Rm, Rs	64位无符号乘法指令	(RdLo, RdHi) ←Rm*Rs	UMULL{cond}{S}
UMLAL RdLo, RdHi, Rm, Rs	64位无符号乘加指令	(RdLo, RdHi) ←Rm*Rs+(RdLo, RdHi)	UMLAL{cond}{S}
SMULL RdLo, RdHi, Rm, Rs	64位有符号乘法指令	(RdLo, RdHi) ←Rm*Rs	SMULL{cond}{S}
SMLAL RdLo, RdHi, Rm, Rs	64位有符号乘加指令	(RdLo, RdHi) ←Rm*Rs+(RdLo, RdHi)	SMLAL{cond}{S}

32位乘法指令——MUL指令将Rm和Rs中的值相乘，结果的低32位保存到Rd中。指令格式如下：

应用示例：

```
MUL   R1,R2,R3    ;R1=R2×R3
MULS  R0,R3,R7    ;R0=R3×R7，同时影响CPSR中的N位和Z位
```

6、乘法指令

助记符	说明	操作	条件码位置
MUL Rd, Rm, Rs	32位乘法指令	Rd←Rm*Rs (Rd≠Rm)	MUL{cond}{S}
MLA Rd, Rm, Rs, Rn	32位乘加指令	Rd←Rm*Rs+Rn (Rd≠Rm)	MLA{cond}{S}
UMULL RdLo, RdHi, Rm, Rs	64位无符号乘法指令	(RdLo, RdHi) ←Rm*Rs	UMULL{cond}{S}
UMLAL RdLo, RdHi, Rm, Rs	64位无符号乘加指令	(RdLo, RdHi) ←Rm*Rs+(RdLo, RdHi)	UMLAL{cond}{S}
SMULL RdLo, RdHi, Rm, Rs	64位有符号乘法指令	(RdLo, RdHi) ←Rm*Rs	SMULL{cond}{S}
SMLAL RdLo, RdHi, Rm, Rs	64位有符号乘加指令	(RdLo, RdHi) ←Rm*Rs+(RdLo, RdHi)	SMLAL{cond}{S}

32位乘加指令——MLA指令将Rm和Rs中的值相乘，再将乘积加上第3个操作数，结果的低32位保存到Rd中。指令格式如下：

应用示例：

```
MLA    R1,R2,R3,R0          ; R1=R2×R3+R0
```

6、乘法指令（自学为主）

助记符	说明	操作	条件码位置
MUL Rd, Rm, Rs	32位乘法指令	Rd←Rm*Rs (Rd≠Rm)	MUL{cond}{S}
MLA Rd, Rm, Rs, Rn	32位乘加指令	Rd←Rm*Rs+Rn (Rd≠Rm)	MLA{cond}{S}
UMULL RdLo, RdHi, Rm, Rs	64位无符号乘法指令	(RdLo, RdHi) ←Rm*Rs	UMULL{cond}{S}
UMLAL RdLo, RdHi, Rm, Rs	64位无符号乘加指令	(RdLo, RdHi) ←Rm*Rs+(RdLo, RdHi)	UMLAL{cond}{S}
SMULL RdLo, RdHi, Rm, Rs	64位有符号乘法指令	(RdLo, RdHi) ←Rm*Rs	SMULL{cond}{S}
SMLAL RdLo, RdHi, Rm, Rs	64位有符号乘加指令	(RdLo, RdHi) ←Rm*Rs+(RdLo, RdHi)	SMLAL{cond}{S}

64位无符号乘法指令——UMULL指令将Rm和Rs中的值作无符号数相乘，结果的低32位保存到RdLo中，而高32位保存到RdHi中。指令格式如下

应用示例：

```
UMULL   R0,R1,R5,R8  ; (R1、R0)=R5×R8
```

6、乘法指令（自学为主）

助记符	说明	操作	条件码位置
MUL Rd, Rm, Rs	32位乘法指令	Rd←Rm*Rs (Rd≠Rm)	MUL{cond}{S}
MLA Rd, Rm, Rs, Rn	32位乘加指令	Rd←Rm*Rs+Rn (Rd≠Rm)	MLA{cond}{S}
UMULL RdLo, RdHi, Rm, Rs	64位无符号乘法指令	(RdLo, RdHi) ←Rm*Rs	UMULL{cond}{S}
UMLAL RdLo, RdHi, Rm, Rs	64位无符号乘加指令	(RdLo, RdHi) ←Rm*Rs+(RdLo, RdHi)	UMLAL{cond}{S}
SMULL RdLo, RdHi, Rm, Rs	64位有符号乘法指令	(RdLo, RdHi) ←Rm*Rs	SMULL{cond}{S}
SMLAL RdLo, RdHi, Rm, Rs	64位有符号乘加指令	(RdLo, RdHi) ←Rm*Rs+(RdLo, RdHi)	SMLAL{cond}{S}

64位无符号乘加指令——UMLAL指令将Rm和Rs中的值作无符号数相

,32

应用示例：

```
UMLAL   R0,R1,R5,R8  ;(R1、R0)=R5×R8+(R1、R0)
```

6、乘法指令（自学为主）

助记符	说明	操作	条件码位置
MUL Rd, Rm, Rs	32位乘法指令	Rd←Rm*Rs (Rd≠Rm)	MUL{cond}{S}
MLA Rd, Rm, Rs, Rn	32位乘加指令	Rd←Rm*Rs+Rn (Rd≠Rm)	MLA{cond}{S}
UMULL RdLo, RdHi, Rm, Rs	64位无符号乘法指令	(RdLo, RdHi) ←Rm*Rs	UMULL{cond}{S}
UMLAL RdLo, RdHi, Rm, Rs	64位无符号乘加指令	(RdLo, RdHi) ←Rm*Rs+(RdLo, RdHi)	UMLAL{cond}{S}
SMULL RdLo, RdHi, Rm, Rs	64位有符号乘法指令	(RdLo, RdHi) ←Rm*Rs	SMULL{cond}{S}
SMLAL RdLo, RdHi, Rm, Rs	64位有符号乘加指令	(RdLo, RdHi) ←Rm*Rs+(RdLo, RdHi)	SMLAL{cond}{S}

64位有符号乘法指令——SMULL指令将Rm和Rs中的值作有符号数相乘，结果的低32位保存到RdLo中，而高32位保存到RdHi中。指令格式如下：

应用示例：

SMLAL R2,R3,R7,R6 ；(R3、R2)=R7×R6+(R3、R2)

6、乘法指令（自学为主）

助记符	说明	操作	条件码位置
MUL Rd, Rm, Rs	32位乘法指令	Rd←Rm*Rs (Rd≠Rm)	MUL{cond}{S}
MLA Rd, Rm, Rs, Rn	32位乘加指令	Rd←Rm*Rs+Rn (Rd≠Rm)	MLA{cond}{S}
UMULL RdLo, RdHi, Rm, Rs	64位无符号乘法指令	(RdLo, RdHi) ←Rm*Rs	UMULL{cond}{S}
UMLAL RdLo, RdHi, Rm, Rs	64位无符号乘加指令	(RdLo, RdHi) ←Rm*Rs+(RdLo, RdHi)	UMLAL{cond}{S}
SMULL RdLo, RdHi, Rm, Rs	64位有符号乘法指令	(RdLo, RdHi) ←Rm*Rs	SMULL{cond}{S}
SMLAL RdLo, RdHi, Rm, Rs	64位有符号乘加指令	(RdLo, RdHi) ←Rm*Rs+(RdLo, RdHi)	SMLAL{cond}{S}

64位有符号乘加指令——SMLAL指令将Rm和Rs中的值作有符号数相乘，64位乘积与RdHi、RdLo相加，结果的低32位保存到RdLo中，而高32位保存到RdHi中。指令格式如下：

应用示例：

```
SMULL    R2,R3,R7,R6        ; (R3、R2)=R7×R6
```

7、协处理器指令（自学为主）

助记符	说明	操作	条件码位置
CDP coproc,opcode1,CRd,CRn, CRm{,opcode2}	协处理器数据操作指令	取决于协处理器	CDP{cond}
LDC{L} coproc, CRd,<地址>	协处理器数据读取指令	取决于协处理器	LDC{cond}{L}
STC{L} coproc, CRd,<地址>	协处理器数据写入指令	取决于协处理器	STC{cond}{L}
MCR coproc,opcode1,Rd,CRn, CRm{,opcode2}	ARM寄存器到协处理器寄存器的数据传送指令	取决于协处理器	MCR{cond}
MRC coproc,opcode1,Rd,CRn, CRm{,opcode2}	协处理器寄存器到ARM寄存器到的数据传送指令	取决于协处理器	MCR{cond}

ARM920T 有两个具体协处理器

1. CP14调试通信通道协处理器

2. CP15系统控制协处理器

CP15 —系统控制协处理器 （the system control coprocessor）他通过协处理器指令MCR和MRC提供具体的寄存器来配置和控制caches、MMU、保护系统、配置时钟模式

7、协处理器指令（自学为主）

ARM处理器通过CDP指令通知ARM协处理器执行特定的操作。该操作由协处理器完成，即对命令的参数的解释与协处理器有关，指令的使用取决于协处理器。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。指令格式如下：

```
CDP{cond}    coproc,opcode1,CRd,CRn,CRm{,opcode2}
```

应用示例：

```
CDP     p7,0,c0,c2,c3,0      ;对协处理器7操作，操作码为0，
                             ;可选操作码为0
CDP     p6,1,c3,c4,c5        ;对协处理器6操作，操作码为1
```

7、协处理器指令（自学为主）

协处理器数据存取指令LDC/STC指令可以将某一连续内存单元的数据读取到协处理器的寄存器中，或者将协处理器的寄存器数据写入到某一连续的内存单元中，传送的字数由协处理器来控制。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。

数据读取指令格式

```
LDC{cond}{L}        coproc, CRd,<地址>
```

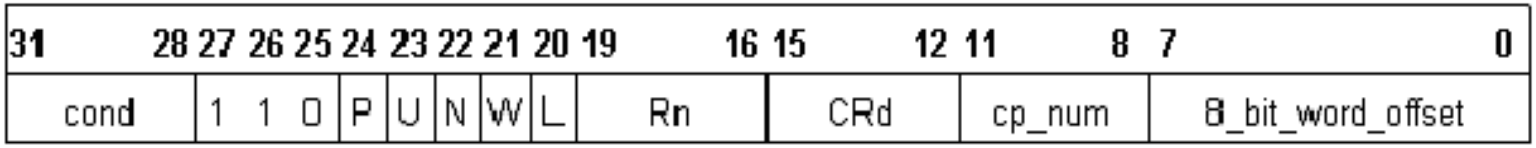
数据存储指令格式

```
STC{cond}{L}        coproc, CRd,<地址>
```

7、协处理器指令（自学为主）

协处理器数据存取指令LDC/STC指令可以将某一连续内存单元的数据读取到协处理器的寄存器中，或者将协处理器的寄存器数据写入到某一连续的内存单元中，传送的字数由协处理器来控制。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。

数据操作指令编码



应用示例：

```
LDC     p5,c2,[R2,#4]
LDC     p6，c2，[R1]
STC     p5,c1,[R0]
STC     p5，cl，[R0，#-0x04]
```

7、协处理器指令（自学为主）

如果需要在ARM处理器中的寄存器与协处理器中的寄存器之间进行数据传送，那么可以使用MCR/MRC指令。MCR指令用于将ARM处理器的寄存器中的数据传送到协处理器的寄存器。MRC指令用于将协处理器的寄存器中的数据传送到ARM处理器的寄存器中。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。

MCR指令格式（ARM→协处理器）

```
MCR{cond}    coproc,opcode1,Rd,CRn,CRm{,opcode2}
```

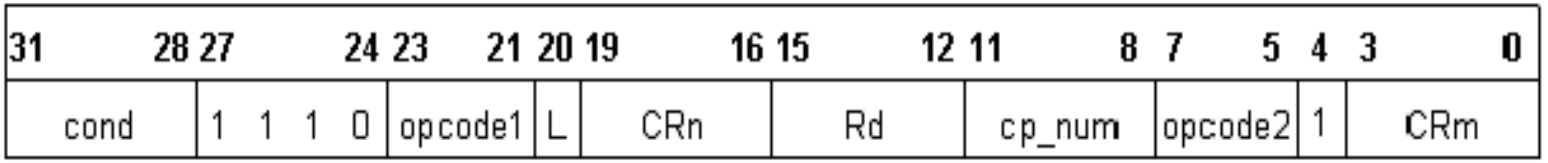
MRC指令格式（协处理器→ ARM ）

```
MRC{cond}    coproc,opcode1,Rd,CRn,CRm{,opcode2}
```

7、协处理器指令（自学为主）

如果需要在ARM处理器中的寄存器与协处理器中的寄存器之间进行数据传送，那么可以使用MCR/MRC指令。MCR指令用于将ARM处理器的寄存器中的数据传送到协处理器的寄存器。MRC指令用于将协处理器的寄存器中的数据传送到ARM处理器的寄存器中。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。

寄存器传送指令编码



应用示例：

```
MCR     p6,2,R7,c1,c2
MCR     p7, 0, R1, c3, c2, 1
MRC     p5,2,R2,c3,c2
MRC     p7, 0, R0, c1, c2, 1
```

MMU的管理

- ARM处理器的CP15协处理器用于系统存储管理，即MMU功能的实现。
- CP15可以包含16个32位的寄存器，其编号为0-15。实际上对于某些编号的寄存器可能对应有多个物理寄存器。在指令中指定特定的标志位来区分这些物理寄存器。有些类似于ARM寄存器中，处于不同的处理器模式时，ARM某些寄存器可能不同。

1. 编写C对应的汇编程序， R1 ，R2中分别存一个32位的数，

1> 判断如果R1的第8位为1，则将R2的第12位取反.

2> 判断如果R1的第12位为1，则将R2的高8位，即25到32位设置为1，

3> 判断如果R1的第18位为1，则将R2的次高位，即17位到24位设置为0，其它位保持不变.

```
int a,b;
If(a &(0x01 <<7)
{
      b ^=(0x01 <<11)
}
If(a &(0x01 <<11)
{
     b |=(0xff <<24)
}
If(a &(0x01 <<17)
{
     b &=~(0xff <<16)
}
```

```
int data_src[100];
int data_dest[100];
For(i=0;i<100;i++)
  data_dest[i]=data_src[i];
```

- Void data_Wcopy(int* src,int *dest,int num)

问题:如何实现一次拷贝8个字呢?

提示:(8的整数倍的时候:

movs r3,r2 lsr #3)使用

```
ldmia  r0!,{r4-r11}
stmia  r1!,{r4-r11}
```

(不能被8的整除的时候:

ands r2,r2,#7)使用

```
ldr    r3,[r0],#4
str    r3,[r0],#4
```

```
Blockcopy
    MOVS        R3,R2,LSR #3
    BEQ         Wordcopy
    STMFD       SP!,{R4-R11}
Copy
    LDMIA       R0!,{R4-R11}
    STMIA       R1!,{R4-R11}
    SUBS        R3,R3,#1
    BNE         Copy
Wordcopy
    ANDS        R2,R2,#7
    MOVEQ       PC,LR
Copyloop
    LDR         R3,[R0],#4
     STR        R3,[R1],#4
    SUBS        R2,R2,#1
    BNE  Copyloop
     MOV        PC,LR
```

8、软中断指令（自学为主）

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
    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
    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
        ...
    LDMFD   SP!, {R0-R3, R12, PC}^ ; SWI异常中断返回
```

9、 伪指令、伪操作和宏指令（自学）

- **伪指令**——是汇编语言程序里的特殊指令助记符，在汇编时被合适的机器指令替代。
- **伪操作**——为汇编程序所用，在源程序进行汇编时由汇编程序处理，只在汇编过程起作用，不参与程序运行。
- **宏指令**——通过伪操作定义的一段独立的代码。在调用它时将宏体插入到源程序中。**也就是常说的宏**。

说明：所有的伪指令、伪操作和宏指令，均与具体的开发工具中的编译器有关。

（1）ARM汇编伪指令

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

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

ADR——小范围的地址读取

ADR伪指令功能：将基于PC相对偏移的地址值或基于寄存器相对偏移的地址值读取到寄存器中。

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

语法格式：

ADR{cond}　register，expr

其中：

- register：加载的目标寄存器。
- expr：地址表达式。
  - 地址非字对齐,-255~255
  - 地址字对齐,-1020~1020

例1：

```
        ……
(0x20)  ADR  R1,Delay
        ……
Delay
(0x64)  MOV  R0,R14
        ……
```

使用ADR将程序标号Delay所表示的地址存入R1。

编译后的反汇编代码：

```
        ……
ADD  R1,PC,#0x3C
        ……

MOV R0,R14
```

PC+0x3C
=0x20+0x08+0x3C
=0x64

❖ 例2：查表

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

ADRL——中等范围的地址读取

ADRL伪指令功能：将基于PC相对偏移的地址值或基于寄存器相对偏移的地址值读取到寄存器中，比ADR伪指令可以读取更大范围的地址。

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

语法格式：

ADRL{cond}　　register,expr

其中：

- **register：加载的目标寄存器。**
- **expr：地址表达式。**
  - **地址非字对齐,-64KB~64KB**
  - **地址字对齐,-256KB~256KB**

例3：

……

(0x20)　ADRL　R1,Delay

……

Delay

(0x64)　MOV　R0,R14

……

使用ADRL将程序标号Delay所表示的地址存入R1。

编译后的反汇编代码：

……

ADD　R1,PC,#0x3C

ADD　R1,R1,#0

……

MOV R0,R14

ADRL伪指令被汇编成两条指令，尽管第2条指令并没有意义。

ADRL—中等范围的地址读取伪指令

示例：

```
start    MOV  r0,#10      ; PC值为当前指令地址值加8字节
         ADRL r4,start+60000
         ; 本ADRL伪指令将被编译器替换成下面两条指令
         ; ADD  r4,pc,#0xe800
         ; ADD  r4,r4,#0x254
         ; 60000=0xEA60
```

LDR ——大范围的地址读取

LDR伪指令功能：用于加载32位立即数或一个地址值到指定的寄存器。

LDR伪指令功能实现方法：在汇编编译源程序时，LDR伪指令被编译器替换成一条合适的指令。

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

语法格式：

LDR{cond}　register,=expr

其中：

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

例4：

```
        ……
(0x060) LDR  R1,=Delay
        ……
Delay
(0x102) MOV  R0,R14
        ……
```

使用LDR将程序标号Delay所表示的地址存入R1。

编译后的反汇编代码：

```
        ……
        LDR  R1,stack
        ……
Delay
        MOV R0,R14
        ……
        LTORG
stack   DCD    0x102
```

LDR伪指令被汇编成一条LDR指令,并在文字池中定义一个常量,该常量为标号Delay的地址。

注意：

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

NOP——空操作伪指令

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

用途：NOP可用于延时操作。

语法格式：　NOP

例：延时子程序

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

GNU 伪指令集

- 数据定义（Data Definition）伪操作
- 汇编控制伪操作
- 杂项伪操作
- GNU汇编书写格式

数据定义（Data Definition）伪操作

- 数据定义伪操作一般用于为特定的数据分配存储单元，同时可完成已分配存储单元的初始化。常见的数据定义伪操作有如下几种：

.byte　单字节定义	.byte　0x12,'a',23
.short　定义双字节数据	.short　0x1234,65535
.long /.word 定义4字节数据	.word　0x12345678
.quad　定义8字节	.quad　0x1234567812345678
.float 定义浮点数	.float　0f3.2
.string/.asciz/.ascii 定义字符串	.ascii "abcd\0",

注意：.ascii 伪操作定义的字符串需要每行添加结尾字符'\0'，其他不需要

汇编控制伪操作

- 汇编控制伪操作用于控制汇编程序的执行流程，常用的汇编控制伪操作包括以下几条：
- .if、.else　.endif伪操作能根据条件的成立与否决定是否执行某个指令序列。当.if后面的逻辑表达式为真，则执行.if后的指令序列，否则执行.else后的指令序列；　.if、.else、.endif伪指令可以嵌套使用。
- 语法格式：

```
.if  logical-expressing
…
.else
…
.endif
```

汇编控制伪操作

- .macro，.endmmacro伪操作可以将一段代码定义为一个整体，称为宏指令，然后就可以在程序中通过宏指令多次调用该段代码。其中，$标号在宏指令被展开时，标号会被替换为用户定义的符号。宏操作可以使用一个或多个参数，当宏操作被展开时，这些参数被相应的值替换。宏操作的使用方式和功能与子程序有些相似，子程序可以提供模块化的程序设计、节省存储空间并提高运行速度。但在使用子程序结构时需要保护现场，从而增加了系统的开销，因此，在代码较短且需要传递的参数较多时，可以使用宏操作代替子程序
- 语法格式：
- .macro
- {$label} macroname {$parameter{,$parameter}…}
- ……..code
- .endm

GNU汇编书写格式

- 代码行中的注释符号: ‘@’
  整行注释符号: ‘#’
  语句分离符号: ‘;’
  直接操作数前缀: ‘#’ 或 ‘$’
- 标号：标号只能由a～z，A～Z，0～9，“.”，_等（由点、字母、数字、下划线等组成，除局部标号外，不能以数字开头）字符组成，标号的后面加“：”。
- 段内标号的地址值在汇编时确定；
- 段外标号的地址值在连接时确定。
- 局部标号:局部标号 主要在局部范围内使用而且局部标号可以重复出现。它由两部组成开头是一个0-99直接的数字局部标号 后面加“:”
- F：指示编译器只向前搜索
- B：指示编译器只向后搜索

习题

1 编写一简单ARM汇编程序段，实现1+2+…+100的运算。

2 将存储器中0x40000000开始的200字的数据，传送到0x48000000开始的区域。

3 编写一程序，存储器中从0x40000000开始有一个64位数。（1）将取反，再存回原处；（2）求其补码，存放到0x40000008处 。

4 教材上P56上1、4、5

编程习题：LDR,STR 的使用

- int data_dest[10];
- int data_src[10];
- void datacopy(int* src,int* dest){

```
 (1) for(i=0;i<10;i++)
       dest[i]=src[i];
  (2) for(i=0;i<10;i++)
      *(dest++)=*(src++);
(3) *dest=*src;
      for(i=0;i<9;i++)
      *(++dest)=*(++src);
}
```

提示使用

```
ldr   r3,[r0,r2 lsl #2]
 ldr   r3,[r0,#4]!
 ldr   r3,[r0],#4
```