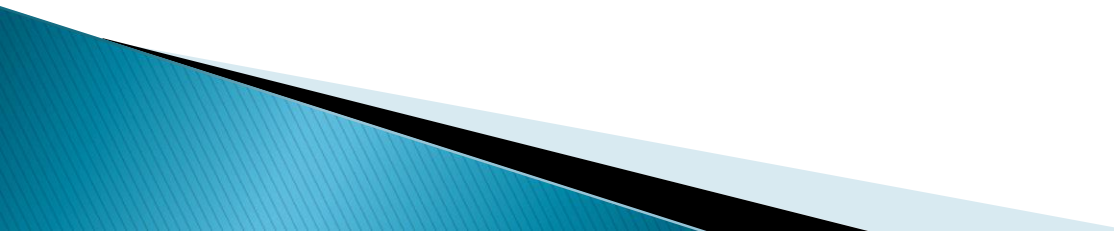


第3章 ARM指令集



目 录

- 3.1 ARM指令集概述
 - 3.2 ARM指令的寻址方式
 - 3.3 ARM指令简介
 - 3.4 Thumb指令简介
 - 3.5 ARM汇编语言简介
 - 3.6 C语言与汇编语言的混合编程
 - 3.7 本章小结
- 

在嵌入式系统开发中，目前最常用的编程语言是汇编语言和C语言。在较复杂的嵌入式软件中，由于C语言编写程序较方便，结构清晰，而且有大量支持库，所以大部分代码采用C语言编写，特别是基于操作系统的应用程序设计。但是在系统初始化、**BootLoadr**、中断处理等，对时间和效率要求较严格的地方仍旧要使用汇编语言来编写相应代码块。本章将介绍**ARM指令集指令及汇编语言**的相关知识。

3.1 **Part One**

ARM指令集概述

ARM处理器的指令集主要有：

- **ARM指令集**，是ARM处理器的原生**32位**指令集，所有指令长度都是**32位**，以字对齐（4字节边界对齐）方式存储；该指令集效率高，但是代码密度较低。
- **Thumb指令集**是**16位**指令集，**2字节**边界对齐，是ARM指令集的子集；在具有较高代码密度的同时，仍然保持ARM的大多数性能优势。
- **Thumb-2指令集**是对Thumb指令集的扩展，提供了几乎与ARM指令集完全相同的功能，同时具有**16位**和**32位**指令，既继承了Thumb指令集的高代码密度，又能实现ARM指令集的高性能；**2字节**边界对齐，**16位**和**32位**指令可自由混合。
- **Thumb-2EE指令集**是Thumb-2指令集的一个变体，用于动态产生的代码；不能与ARM指令集和Thumb指令集交织在一起。

除了上面介绍的指令集外，ARM处理器还有针对协处理器的扩展指令集，如普通协处理器指令、NEON和VFP扩展指令集、无线MMX技术扩展指令集等。

3.1.1 指令格式

ARM指令集的指令基本格式如下：

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

指令中“< >”内的项是必需的，“{ }”内的项是可选的。

符号	说明
opcode	操作码，即指令助记符，如MOV、SUB、LDR等
cond	条件码，描述指令执行的条件
S	可选后缀，指令后加上“S”，指令执行成功完成后自动更新CPSR寄存器中的条件标志位
Rd	目的寄存器
Rn	存放第1个操作数的寄存器
shift_operand	第2个操作数，可以是寄存器、立即数等

ARM指令的一般格式

ARM指令的编码格式

譬如一条加法指令，其语

ADDEQS

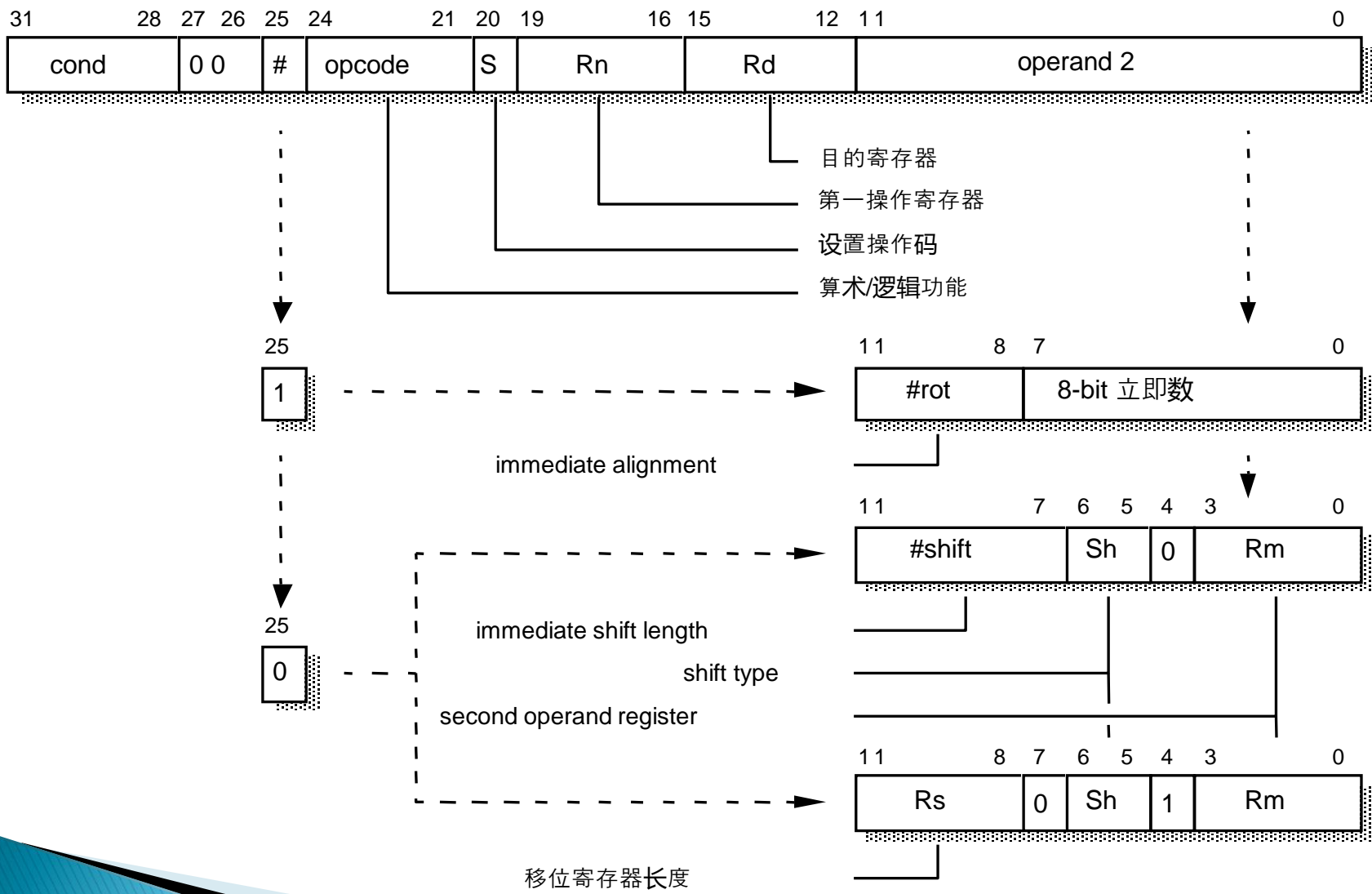
该指令的编码格式为：

机器指令编码格式
分为7个部分：

条件域，类别，操作
码，S域，第一源操作
数，目的操作数，第
二源操作数。

31~28	27~25	24~21	20	19~16	15~12	11~0
cond		opcode	S	Rn	Rd	op2
0000	000	0100	1	0001	0000	000000000010

ARM数据处理指令中第2操作数的编码格式图解



灵活的第2操作数

▶ 立即数型

- 格式： #<32位立即数>
- 也写成#immed_8r
- #<32位立即数>是取值为数字常量的表达式，并不是所有的32位立即数都是有效的。
- 有效的立即数很少。它必须由一个8位的立即数循环右移偶数位得到。原因是32位ARM指令中条件码和操作码等占用了一些必要的指令码位，32位立即数无法编码在指令中。
- 举例：
 - ADD r3, r7, #1020 ;#immed_8r型第2操作数,
;1020是0xFF循环右移30位后生成的32位立即数
;推导: $1020 = 0x3FC = 0x000003FC$

灵活的第2操作数（续1）

- 数据处理指令中留给Operand2操作数的编码空间只有12位，需要利用这12位产生32位的立即数。其方法是：**把指令最低8位（bit[7:0]）立即数循环右移偶数次**，循环右移次数由 $2 * \text{bit}[11:8]$ （bit[11:8]是Operand2的高4位）指定。
- 例如：MOV R4, #0x8000000A
其中的立即数#0x8000000A是由8位的0xA8循环右移0x4位得到。
- 又例如：MOV R4, #0xA0000002
其中的立即数#0xA0000002是由8位的0xA8循环右移0x6位得到。

灵活的第2操作数（续2）

▶ 寄存器移位型

- 格式：Rm{, <shift>}
- Rm是第2操作数寄存器，可对它进行移位或循环移位。
<shift>用来指定移位类型（LSL, LSR, ASR, ROR或RRX）和移位位数。其中移位位数有两种表示方式，一种是5位立即数（#shift），另外一种是指移量寄存器Rs的值。参看下面的例子。例子中的R1是Rm寄存器。
- ADD R5, R3, R1, LSL #2 ;R5 ← R3 + R1 * 4
- ADD R5, R3, R1, LSL R4 ;R5 ← R3 + R1 * 2^{R4}
;R4是Rs寄存器，Rs用于计算移位次数

3.1.2 指令的条件码

ARM指令集中几乎每条指令都可以是条件执行的，由cond可选条件码来决定，位于ARM指令的最高4位[31:28]，可以使用的条件码如表3-2所示。

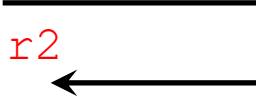
每种条件码的助记符由两个英文符号表示，在指令助记符的后面和指令同时执行。根据程序状态寄存器CPSR中的条件标志位[31:28]判断当前条件是否满足，若满足则执行指令。若指令中有后缀S，则根据执行结果更新程序状态寄存器CPSR中的条件标志位[31:28]。

指令条件码	助记符	CPSR条件标志位值	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 (不要执行)

条件执行及标志位

- ARM指令可以通过添加适当的条件码后缀来达到条件执行的目的。
 - 这样可以提高代码密度，减少分支跳转指令数目，提高性能。

```
CMP    r3, #0
BEQ    skip
ADD    r0, r1, r2
skip
```



```
CMP    r3, #0
ADDNE  r0, r1, r2
```

- 默认情况下，数据处理指令不影响程序状态寄存器的条件码标志位，但可以选择通过添加“S”来影响标志位。CMP不需要增加“S”就可改变相应的标志位。

```
loop
```

```
...
```

```
SUBS  r1, r1, #1
```

```
BNE  loop
```

R1减1，并设置标志位

如果 Z标志清零则跳转

条件执行示例

- ▶ 一系列的指令都使用条件指令

```
if (a==0) func(1);  
    CMP        r0, #0  
    MOVEQ      r0, #1; 把func()函数的参数赋给r0  
    BLEQ       func
```

- ▶ 置标志位，再使用不同的条件码

```
if (a==0) x=0    ;r0:a,r1:x  
if (a>0)  x=1;  
    CMP        r0, #0  
    MOVEQ      r1, #0  
    MOVGT      r1, #1
```

- ▶ 使用条件比较指令

```
if (a==4 || a==10) x=0;  
    CMP        r0, #4  
    CMPNE      r0, #10  
    MOVEQ      r1, #0
```

3.2 Part Two

ARM指令的寻址方式

寻址方式是指处理器根据指令中给出的地址信息，找出操作数所存放的物理地址，实现对操作数的访问。根据指令中给出的操作数的不同形式，ARM指令系统支持的寻址方式有：立即寻址、寄存器寻址、寄存器间接寻址、寄存器移位寻址、变址寻址、多寄存器寻址、堆栈寻址、块复制寻址和相对寻址等。

3.2.1 立即寻址

立即寻址也叫立即数寻址，指令的一部分不是操作数地址而是操作数本身。在立即数寻址中，操作数本身就在指令中，指令也就获得了操作数，这个操作

立即数在指令中以“#”为前缀，后面跟进制和实际数值。

~~#0x 16进制~~ 例：#0x55

~~#0d 或缺省 10进制 #0d25 / #25~~

2到9进制数，形式为#n_XXX,n的范围是2到9，XXX是具体数字。

例：#2_1010

例如

AD
MO

程序存储

MOV R0,#0x55

R0

0x55

其
“#”

MOV R0, #0x55

中要以

3.2.2 寄存器寻址

寄存器寻址是指将操作数放在寄存器中，指令中地址码部分给出寄存器编号。这是各类微处理器常用的一种有较高执行效率的寻址方式。

示例：

ADD	R0,	R1,	R2	;R0 \leftarrow R1 + R2
MOV	R0,	R1		;R0 \leftarrow R1

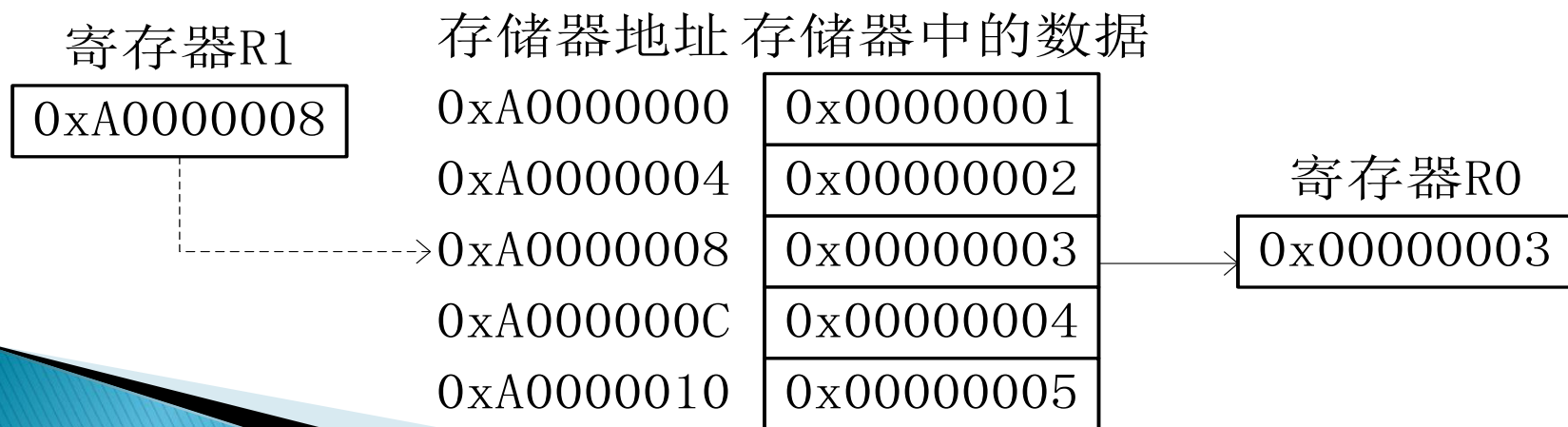
3.2.3 寄存器间接寻址

操作数存放在存储器中，并将所存放的存储单元地址放入某一通用寄存器中，在指令中的地址码部分给出该通用寄存器的编号。

示例：

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

该指令将寄存器R1中存放的值0xA0000008作为存储器地址，将该存储单元中的数据0x00000003传送到寄存器R0中，寻址示意图如图3-1所示。



3.2.4 寄存器移位寻址

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。

例如：

ADD
MOV

R1	0x00000001
R0	0b00001000

逻辑左移3位

0b00001000

3位

MOV R0, R1, LSL #3

RM指

移位
令集共有5种位移操作。如下所示：

寄存器移位寻址

- LSL逻辑左移 : Rx, LSL <op1>

(**L**ogical **S**hift **L**eft)

op1--为通用寄存器或立即数 (0~31)

MOV R0, R1, LSL # 5;

R1的值左移5位后, 存入R0; 相当于R1的值 $\times 32$ 后, 存入R0。

- LSR逻辑右移 : Rx, LSR <op1>

(**L**ogical **S**hift **R**ight)

MOV R0, R1, LSR # 5;

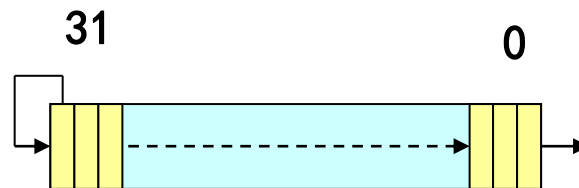
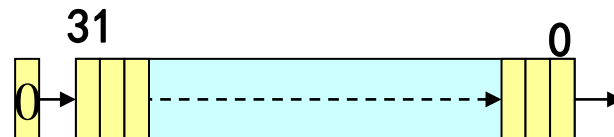
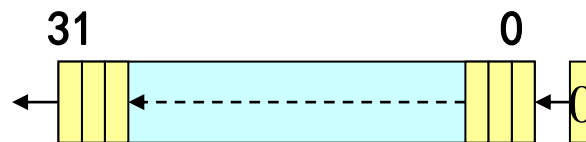
相当于R1的值除以32后, 存入R0。

- ASR算术右移 : Rx, ASR <op1>

(**A**rithmetic **S**hift **R**ight)

MOV R0, R1, ASR # 5;

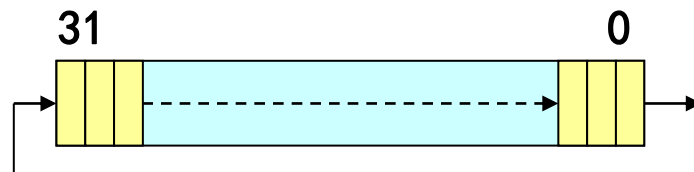
R1的值右移5位后, 存入R0; 最左端用第31位的值来填充。



寄存器移位寻址

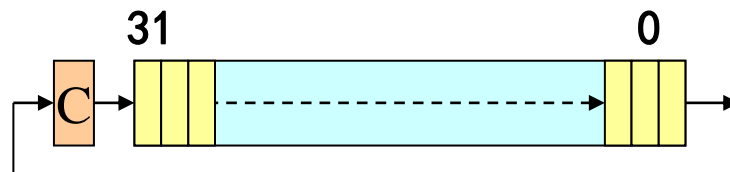
- **ROR**循环右移 : **R_x**, **ROR** <op1>
(**R**otate **R**ight)

MOV R0, R0, ROR #5;
R0的值循环右移5位。



- **RRX**带扩展的循环右移: **R_x**, **RRX**
(**R**otate **R**ight with **e**xtend)

MOV R0, R1, RRX;
R1的值扩展的循环右移, 最左端由进位标志位填充, 存入R0。



3.2.5 变址寻址

变址寻址方式是将某个寄存器（基址寄存器）的值与指令中给出的偏移量相加，形成操作数的有效地址，再根据该有效地址访问存储器。该寻址方式常用于访问在基址附近的存储单元。

示例：

```
LDR    R0,    [R1, #2]
```

偏移地址量可以是立即数或寄存器的内容。

该指令将R1寄存器的值0xA0000008加上位移量2，形成操作数的有效地址，将该有效地址单元中的数据传送到寄存器R0中。

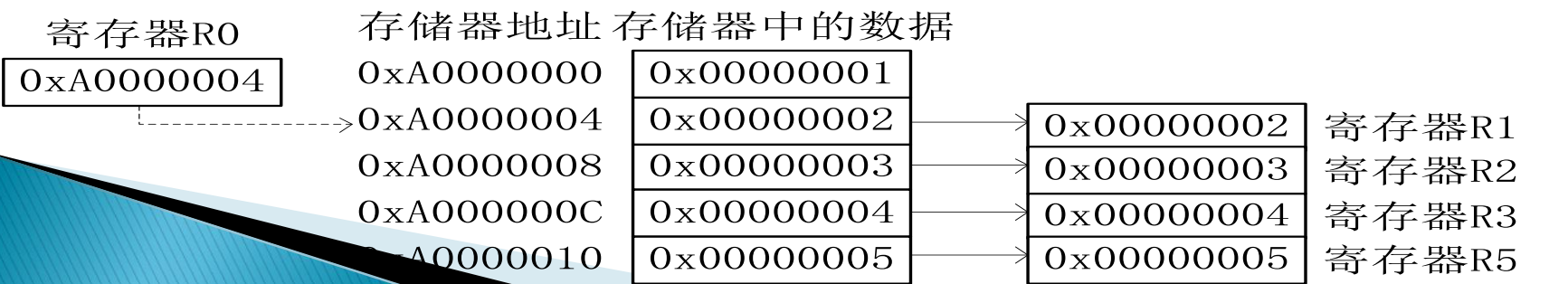
3.2.6 多寄存器寻址

多寄存器寻址方式可以在一条指令中传送多个寄存器的值，一条指令最多可以传送16个通用寄存器的值。连续的寄存器之间用“-”连接，不连续的中间用“,”分隔。

示例：

LDMIA R0! , {R1-R3, R5} ;R1 ← [R0]
 ;R2 ← [R0 + 4]
 ;R3 ← [R0 + 8]
 ;R5 ← [R0 + 12]

该指令将R0寄存器的值0xA0000004作为操作数地址，将存储器中该地址开始的连续单元中的数据传送到寄存器R1、R2、R3、R5中，寻址示意图如图3-2所示。



3.2.7 相对寻址

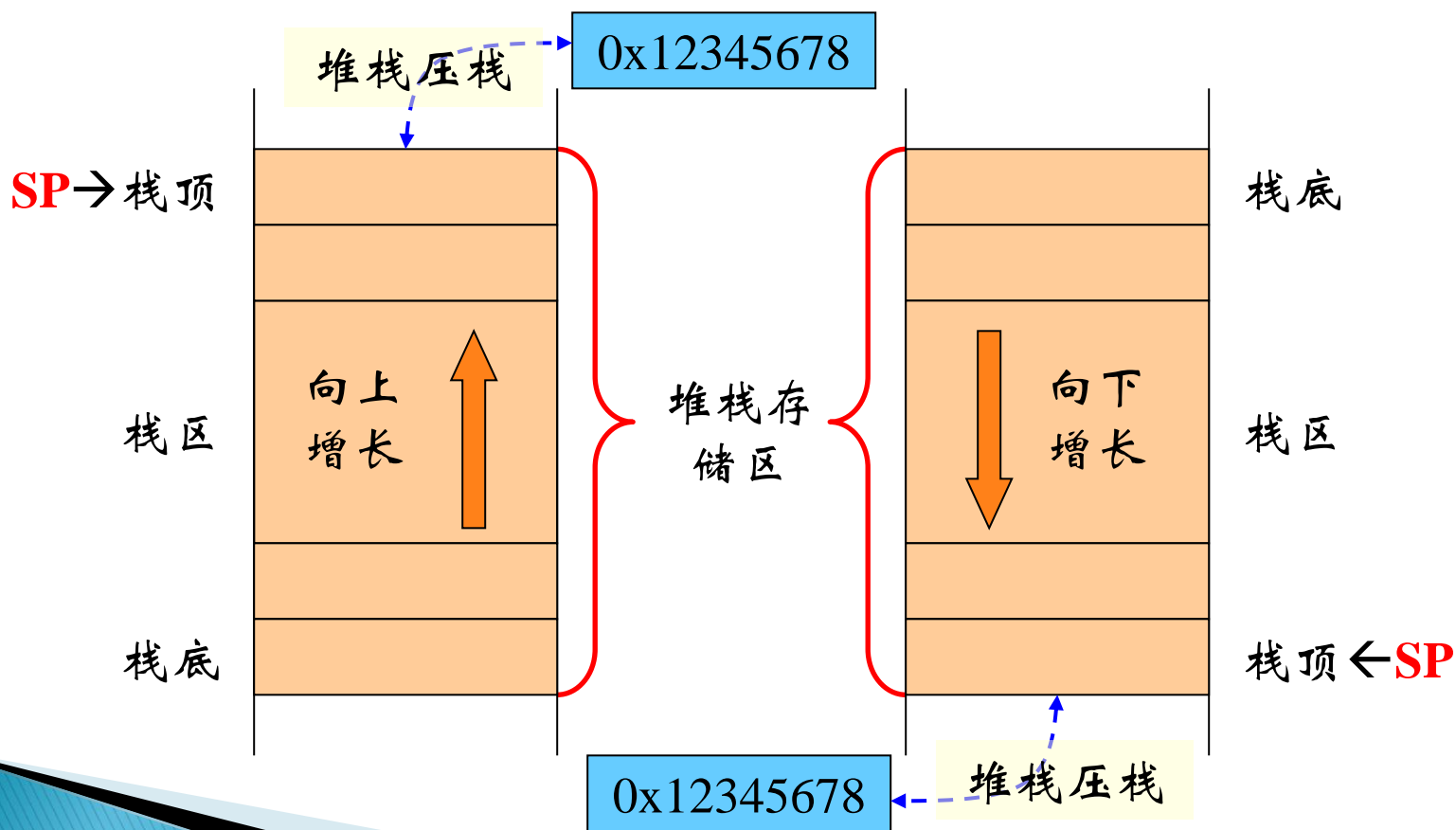
相对寻址方式就是以PC寄存器为基址寄存器，以指令中的地址标号为偏移量，两者相加形成操作数的有效地址。偏移量指出的是当前指令和地址标号之间的相对位置。子程序调用指令即是相对寻址方式。

示例：

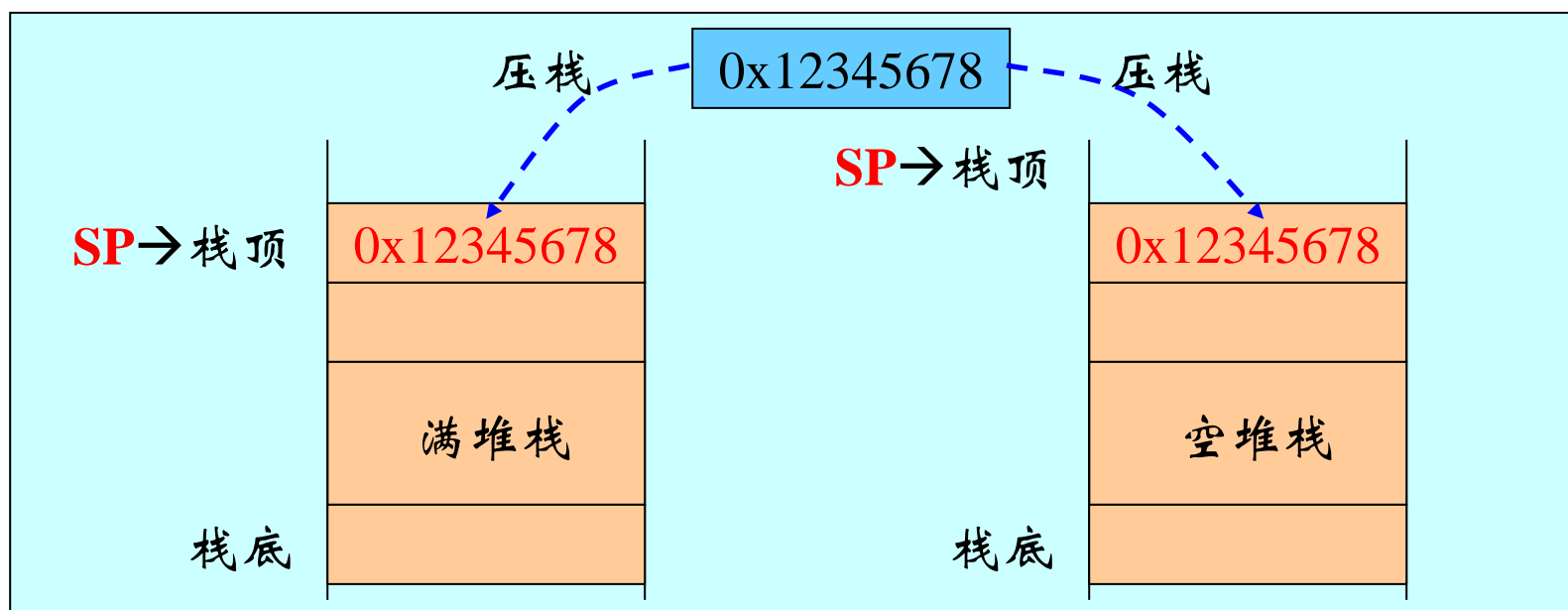
```
BL ADDR1                ;跳转到子程序ADDR1处执行
...
ADDR1:
...
MOV    PC,    LR        ;从子程序返回
```

3.2.8 堆栈寻址

堆栈是按“先进后出”或“后进先出”方式进行存取的存储区。堆栈寻址是隐含的，使用一个叫做堆栈指针（**SP**）的专门寄存器，指示当前堆栈的栈顶。



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**。



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。

STMFD	SP! , {R1-R3, LR} ;将寄存器R1-R3和LR压入堆栈，满递减堆栈
LDMFD	SP! , {R1-R3, LR} ;将堆栈数据出栈，放入寄存器R1-R3和LR

3.2.9 块复制寻址

块复制寻址方式是多寄存器传送指令LDM/STM的寻址方式。LDM/STM指令可以将存储器中的一个数据块复制到多个寄存器中，或则将多个寄存器中的值复制到存储器中。寻址操作中使用的寄存器可以是R0~R15这16个寄存器的所有或子集。

根据基地址的增长方向是向上还是向下，以及地址的增减与指令操作的先后顺序（操作先进行还是地址先增减）的关系，有四种寻址方式：

- IB（Increment Before）：地址先增加再完成操作，如STMIB、LDMIB。
- IA（Increment After）：先完成操作再地址增加，如STMIA、LDMIA。
- DB（Decrement Before）：地址先减少再完成操作，如STMDB、LDMDB。
- DA（Decrement After）：先完成操作再地址减少，如STMDA、LDMDA。

块复制寻址

块复制寻址可实现连续地址数据从存储器的某一位置拷贝到另一位置。

例如:

LDMIA R0, {R1-R5};

从以R0的值为起始地址的存储单元中取出5个字的数据

STMIA R1, {R1-R5};

将取出的数据存入以R1的值为起始地址的存储单元中

3.3

Part Three

ARM指令简介

ARM指令集主要有：跳转指令、数据处理指令、程序状态寄存器处理指令、加载/存储指令、协处理器指令和异常产生指令六大类。

ARM指令集是加载/存储型的，指令的操作数都存储在寄存器中，处理结果直接放入到目的寄存器中。采用专门的加载/存储指令来访问系统存储器。

3.3.1 跳转指令

跳转指令用于实现程序流程的跳转。在ARM程序中有两种方式可以实现程序流程的跳转：

- 直接向程序计数器PC中写入跳转地址，可以实现4G地址空间内的任意跳转。例如：

```
LDR    PC, [PC, # + 0x00FF]      ;PC ← [PC + 8 + 0x00FF]
```

- 使用专门的跳转指令。

ARM指令集中的跳转指令可以完成从当前指令向前或向后的32MB地址空间的跳转。跳转指令有：

1. B指令：

B {条件} 目标地址

跳转指令B是最简单的跳转指令，跳转到给定的目标地址，从那里继续执行。

示例：

B	WAITA	;无条件跳转到标号WAITA处执行
B	0x1234	;跳转到绝对地址0x1234处

2. BL指令

BL {条件} 目标地址

用于子程序调用，在跳转之前，将下一条指令的地址复制到链接寄存器**R14（LR）**中，然后跳转到指定地址执行。

示例：

BL FUNC1

将当前**PC**值保存到**R14**中，然后跳转到标号**FUNC1**处执行

3. BLX指令

BLX {条件} 目标地址

BLX指令从**ARM**指令集跳转到指定地址执行，并将处理器的工作状态由**ARM**状态切换到**Thumb**状态，同时将**PC**值保存到链接寄存器**R14**中。

示例：

BLX FUNC1

;将当前**PC**值保存到**R14**中，然后跳转到标号**FUNC1**处执行，
;并切换到**Thumb**状态

BLX R0

;将当前**PC**值保存到**R14**中，然后跳转**R0**中的地址处执行，
;并切换到**Thumb**状态

4. BX指令

BX {条件} 目标地址

带状态切换的跳转指令，跳转到指定地址执行。若目标地址寄存器的位[0]为1，处理器的工作状态切换为Thumb状态，同时将CPSR中的T标志位置1，目标地址寄存器的位[31:1]复制到PC中；若目标地址寄存器的位[0]为0，处理器的工作状态切换为ARM状态，同时将CPSR中的T标志位清0，目标地址寄存器的位[31:1]复制到PC中。

示例：

BX R0

；跳转R0中的地址处执行，如果R0[0]=1，切换到Thumb状态

3.3.2 数据处理指令

数据传输指令主要完成寄存器中数据的各种运算操作。数据处理指令的使用原则：

- 所有操作数都是**32**位，可以是寄存器或立即数。
- 如果数据操作有结果，结果也为**32**位，放在目的寄存器中。
- 指令使用“两操作数”或“三操作数”方式，即每一个操作数寄存器和目的寄存器分别指定。
- 数据处理指令只能对寄存器的内容进行操作。指令后都可以选择**S**后缀来影响标志位。比较指令不需要后缀**S**，这些指令执行后都会影响标志位。

1. MOV指令

MOV {条件}{S} 目的寄存器, 源操作数

MOV指令将一个立即数、一个寄存器或被移位的寄存器传送到目的寄存器中。后缀S表示指令的操作是否影响标志位。如果目的寄存器是寄存器PC可以实现程序流程的跳转；寄存器PC做为目的寄存器且后缀S被设置，则在跳转的同时，将当前处理器工作模式下的SPSR值复制到CPSR中。

示例：

MOV R0, #0x01	;将立即数0x01装入到R0
MOV R0, R1	;将寄存器R1的值传送到R0
MOVS R0, R1, LSL #3	
;将寄存器R1的值左移3位后传送到R0，并影响标志位	
MOV PC, LR	;将链接寄存器LR的值传送到PC中，
用于子程序返回	

2. MVN指令

MVN {条件}{S} 目的寄存器, 源操作数

MVN指令将一个立即数、一个寄存器或被移位的寄存器的值先按位求反，再传送到目的寄存器中。后缀**S**表示是否影响标志位。

示例：

```
MVN R0, #0x0FF
```

;将立即数0xFF按位求反后装入R0，操作后R0=0xFFFFFFFF00

```
MVN R0, R1
```

;将寄存器R1的值按位求反后传送到R0

3. ADD指令

ADD {条件}{S} 目的寄存器, 操作数1, 操作数2

ADD指令将两个操作数相加后, 结果放入目的寄存器中。
同时根据操作的结果影响标志位。

示例:

ADD R0, R0, #1	;R0 = R0 + 1
ADD R0, R1, R2	;R0 = R1 + R2
ADD R0, R1, R2, LSL #3	;R0 = R1 + (R2 << 3)

4. SUB指令

SUB {条件}{S} 目的寄存器, 操作数1, 操作数2

SUB指令用于把操作数1减去操作数2，将结果放入目的寄存器中。同时根据操作的结果影响标志位。

示例：

SUB R0, R0, #1	;R0 = R0 - 1
SUB R0, R1, R2	;R0 = R1 - R2
SUB R0, R1, R2, LSL #3	;R0 = R1 - (R2 << 3)

5. RSB指令

RSB {条件}{S} 目的寄存器, 操作数1, 操作数2

RSB指令称为逆向减法指令，用于把操作数2减去操作数1，将结果放入目的寄存器中。同时根据操作的结果影响标志位。

示例：

RSB R0, R0, #0xFFFF	;R0 = 0xFFFF - R0
RSB R0, R1, R2	;R0 = R2 - R1

6. ADC指令

ADC {条件}{S} 目的寄存器, 操作数1, 操作数2

ADC指令将两个操作数相加后，再加上CPSR中的C标志位的值，将结果放入目的寄存器中。同时根据操作的结果影响标志位。

示例：

```
ADDS R0, R0, R2
```

```
ADC R1, R1, R3
```

;用于64位数据加法， $(R1, R0) = (R1, R0) + (R3, R2)$

7. SBC指令

SBC {条件}{S} 目的寄存器, 操作数1, 操作数2

SBC指令用于操作数1减去操作数2，再减去CPSR中的C标志位值的反码，将结果放入目的寄存器中。同时根据操作的结果影响标志位。

示例：

```
SUBS R0, R0, R2
```

```
SBC R1, R1, R3
```

;用于64位数据减法， $(R1, R0) = (R1, R0) - (R3, R2)$

8. RSC指令

RSC {条件}{S} 目的寄存器, 操作数1, 操作数2

RSC指令用于操作数2减去操作数1，再减去CPSR中的C标志位值的反码，将结果放入目的寄存器中。同时根据操作的结果影响标志位。

示例：

```
RSBS R2, R0, #0
```

```
RSC R3, R1, #0
```

```
RSC R0, R1, R2
```

;用于求64位数据的负数

;R0 = R2 - R1 -!C

9. AND指令

AND {条件}{S} 目的寄存器, 操作数1, 操作数2

AND指令实现两个操作数的逻辑与操作，将结果放入目的寄存器中。同时根据操作的结果影响标志位。常用于将操作数某些位清0。

示例：

```
AND R0, R1, R2
```

```
;R0 = R1 & R2
```

```
AND R0, R0, #3
```

```
;R0的位0和位1不变，其余位清0
```

10. ORR指令

ORR {条件}{S} 目的寄存器, 操作数1, 操作数2

ORR指令实现两个操作数的逻辑或操作，将结果放入目的寄存器中。同时根据操作的结果影响标志位。常用于将操作数某些位置1。

示例：

```
ORR R0, R0, #3           ;R0的位0和位1置1，其余位不变
```


11. EOR指令

EOR {条件}{S} 目的寄存器, 操作数1, 操作数2

EOR指令实现两个操作数的逻辑异或操作，将结果放入目的寄存器中。同时根据操作的结果影响标志位。常用于将操作数某些位置取反。

示例：

```
EOR R0, R0, #0F
```

;R0的低4位取反

12. BIC指令

BIC {条件}{S} 目的寄存器, 操作数1, 操作数2

BIC指令用于清除操作数1的某些位，将结果放入目的寄存器中。同时根据操作的结果影响标志位。操作数2为32位掩码，掩码中设置了哪些位则清除操作数1中这些位。

示例：

```
BIC R0, R0, #0F    ;将R1的低4位清0，其他位不变
```

13. CMP指令

CMP {条件} 操作数1, 操作数2

CMP指令用于把一个寄存器的值减去另一个寄存器的值或立即数，根据结果设置**CPSR**中的标志位，但不保存结果。
示例：

CMP R1, R0

;将R1的值减去R0的值，并根据结果设置**CPSR**的标志位

CMP R1, #0x200

;将R1的值减去200，并根据结果设置**CPSR**的标志位

14. CMN指令

CMN {条件} 操作数1, 操作数2

CMN指令用于把一个寄存器的值减去另一个寄存器或立即数取反的值，根据结果设置CPSR中的标志位，但不保存结果。该指令实际完成两个操作数的加法。

示例：

CMN R1, R0

;将R1的值和R0的值相加，并根据结果设置CPSR的标志位

CMN R1, #0x200

;将R1的值和立即数200相加，并根据结果设置CPSR的标志位

15. TST指令

TST {条件} 操作数1, 操作数2

TST指令用于把一个寄存器的值和另一个寄存器的值或立即数进行按位与运算，根据结果设置CPSR中的标志位，但不保存结果。该指令常用于检测特定位的值。

示例：

```
TST R1, #0x0F
```

;检测R1的低4位是否为0

16. TEQ指令

TEQ {条件} 操作数1, 操作数2

TEQ指令用于把一个寄存器的值和另一个寄存器的值或立即数进行按位异或运算，根据结果设置CPSR中的标志位，但不保存结果。该指令常用于检测两个操作数是否相等。

示例：

TEQ R1, R2

;将R1的值和R2的值进行异或运算，并根据结果设置CPSR的标志位

补充：乘法类指令

①MUL 32位乘法指令：

MUL {条件} {S} Rd , Rm , Rs;

MUL是32位正数乘法指令，其功能是将Rm与Rs相乘，并将乘积的低32位保存在Rd中，如果操作数是有符号的，可以假定结果也是有符号的。

MULEQS R0 , R1 , R2 ; 若Z=1, 执行R0 :=R1*R2, 并设置N位和Z位

②MLA 带累加的乘法：

MLA {条件} {S} Rd , Rm , Rs , Rn;

MLA指令的作用与MUL类似，不同的是把Rn的值加到结果上，并把结果存到目的寄存器Rd中。其中，Rm，Rs都是32位有符号或无符号数，该指令再在求累加和时很方便。

值得注意的是：目的寄存器Rd与操作数寄存器Rm不能相同，R15可用于操作数或目的寄存器。

MLA R0 , R1 , R2 , R3 ; R0 :=R3+R1*R2

3.3.3 程序状态寄存器处理指令

MRS指令和**MSR**指令用于在状态寄存器和通用寄存器间传输数据。状态寄存器的值要通过“读取→修改→写回”三个步骤操作来实现，可先用**MRS**指令将状态寄存器的值复制到通用寄存器中，修改后再通过**MSR**指令把通用寄存器的值写回状态寄存器。

示例：

```
MRS R0, CPSR           ;将CPSR的值复制到R0中
ORR R0, R0, #C0
;R0的位6和位7置1，即屏蔽外部中断和快速中断
MSR CPSR, R0           ;将R0值写回到CPSR中
```


- MRS和MSR允许传送CPSR / SPSR中的内容到/从一个通用寄存器中。
MRS指令和MSR指令的格式如下：

- $\text{MRS}\{\langle\text{cond}\rangle\} \text{ Rd}, \langle\text{psr}\rangle \quad ; \text{ Rd} = \langle\text{psr}\rangle$
- $\text{MSR}\{\langle\text{cond}\rangle\} \langle\text{psr}[_\text{fields}]\rangle, \text{Rm} ; \langle\text{psr}[_\text{fields}]\rangle = \text{Rm}$

在这里：

- $\langle\text{psr}\rangle = \text{CPSR or SPSR}$
- $[_\text{fields}] = \text{'fsxc' 的任意组合}$

其中，MSR指令中的fields（域）可用于设置程序状态寄存器中需要操作的位：

- 位[31: 24]为条件标志位域，用f表示。
- 位[23: 16]为状态位域，用s表示。
- 位[15: 8]为扩展位域，用x表示。
- 位[7: 0]为控制位域，用c表示。

- 也允许送一个立即数到 psr_fields

- $\text{MSR}\{\langle\text{cond}\rangle\} \langle\text{psr_fields}\rangle, \# \text{Immediate}$

- 用户模式下，所有位均可以被读取，但只有条件标志位 (c)可被写。

示例：

MSR CPSR_cxsf, R3

MSR指令举例

▶ MSR指令举例如下：

MSR CPSR_c, #0xD3

； CPSR[7…0]=0xD3，即切换到管理模式，0b11010011

MSR CPSR_cxsf, R3

； CPSR=R3

MSR指令说明

- ▶ 程序中不能通过MSR指令直接修改CPSR中的T控制位来实现ARM状态/Thumb状态的切换，必须使用BX指令完成处理器状态的切换(因为BX指令属分支指令，它会打断流水线状态，实现处理器状态切换)。
- ▶ MRS与MSR配合使用，实现CPSR或SPSR寄存器的读—修改—写操作，可用来进行处理器模式切换、允许/禁止IRQ/FIQ中断等设置，如下面的程序清单所示。

使能IRQ中断（开中断）

ENABLE_IRQ

```
MRS      R0, CPSR
BIC      R0, R0, #0x80
MSR      CPSR_c, R0
MOV      PC, LR
```

I位=0 开中断

禁能IRQ中断（关中断）

DISABLE_IRQ

MRS R0 CPSR

ORR R0, R0, #0x80

MSR CPSR_c, R0

MOV PC, LR

I位=1 关中断

堆栈指令初始化

INITSTACK

MOV R0, LR

; 保存返回地址

MSR CPSR_c, #0xD3

LDR SP, StackSvc

; 设置管理模式堆栈, M[4:0]=0b10011

MSR CPSR_c, #0xD2

LDR SP, StackIrq

; 设置中断模式堆栈, M[4:0]=0b10010

MOV PC, R0

3.3.4 Load/Store指令

- ▶ Load/Store指令用于寄存器和内存间数据的传送，Load用于把内存中的数据装载到寄存器中，而Store则用于把寄存器中的数据存入内存。

- Load/Store指令分为三类：

- 单一数据传送指令（LDR和STR等）；
- 多数据传送指令（LDM和STM）；
- 数据交换指令（SWP和SWPB）。

LDR字数据加载指令

- 1 **[Rn + 偏移/ Rm]**所指数据装入后， **Rn**不变
LDR Rd, [Rn] ; 把内存中地址为Rn的字数据装入寄存器Rd中;
LDR Rd, [Rn, Rm]; 将内存中地址为Rn+Rm的字数据装入寄存器Rd中;
LDR Rd, [Rn, #index] ; 将内存中地址为Rn+index的字数据装入寄存器Rd中;
LDR Rd, [Rn, Rm, LSL #5] ; 将内存中地址为Rn+Rm×32的字数据装入寄存器Rd;

偏移量（变址），为12位的无符号数。

- 1、操作数地址为基址加变址，执行后基址不变。

addr: [Rn, Rm]

$((Rn) + (Rm)) \rightarrow (Rd)$; 执行后Rn不变

LDR字数据加载指令

2、操作数地址为基址加变址，执行后基址改变

addr: $[R_n, R_m]!$

$((R_n) + (R_m)) \rightarrow (R_d);$ 执行后 $(R_n) + (R_m) \rightarrow (R_n);$

$[R_n + \text{偏移} / R_m]$ 所指数据装入后， **$R_n = R_n + \text{偏移} / R_m$**

LDR $R_d, [R_n, R_m]!$; 将内存中地址为 $R_n + R_m$ 的字数据装入寄存器 R_d ，并将新地址 $R_n + R_m$ 写入 R_n ；

LDR $R_d, [R_n, \#index]!$; 将内存中地址为 $R_n + index$ 的字数据装入寄存器 R_d ，并将新地址 $R_n + index$ 写入 R_n ；

LDR $R_d, [R_n, R_m, LSL \#5]!$; 将内存中地址为 $R_n + R_m \times 32$ 的字数据装入寄存器 R_d ，并将新地址 $R_n + R_m \times 32$ 写入 R_n

LDR字数据加载指令

3、操作数地址为基址，执行后基址改变

addr: [Rn], Rm

((Rn)) \rightarrow (Rd); 执行后 (Rn) + (Rm) \rightarrow (Rn);

[Rn]所指数据装入后, $Rn = Rn + \text{偏移} / Rm$

LDR Rd, [Rn], Rm ; 将内存中地址为Rn的字数据装入寄存器Rd, 并将新地址 $Rn + Rm$ 写入Rn;

LDR Rd, [Rn], #index ; 将内存中地址为Rn的字数据装入寄存器Rd, 并将新地址 $Rn + index$ 写入Rn;

LDR Rd, [Rn], Rm, LSL #5; 将内存中地址为Rn的字数据装入寄存器Rd, 并将新地址 $Rn + Rm \times 32$ 写入Rn。

LDR字数据加载指令

例如

LDR R0, [R1, R2, LSL # 5]!;
 $((R1) + (R2) \times 32) \rightarrow R0$; $(R1) + (R2) \times 32 \rightarrow R1$

LDR R1, [R0, #0x12];
 $((R0) + (12)) \rightarrow R1$; (R0) 不变

LDR R1, [R0, -R2];
 $((R0) - (R2)) \rightarrow R1$; (R0) 不变

LDR R1, [R0]
 $((R0)) \rightarrow R1$; (R0) 不变

LDR R0, [R1], R2, LSL #2;
 $((R1)) \rightarrow R0$, 执行后 $(R1) + (R2) \times 4 \rightarrow R1$

MOV R1, #UARTADD; 地址UARTADD装入R1

LDR R0, [R1]; 将外设端口数据输入到R0

LDRB字节数据加载指令

2. LDRB字节数据加载指令:

LDR{<cond>}B <Rd>,<addr>;

- ▶ 功能：同LDR指令，但该指令只是从内存读取一个8位的字节数据而不是一个32位的字数据，并将Rd的高24位清0。
- 针对**小端**模式配置，如果提供的地址在一个字边界上，则字节数据使用在 0 至 7 位上的数据，如果在一个字边界加上一个字节地址上，则使用 8 至 15 位，以此类推。
LDRB R0, [R1]; 将内存中起始地址为R1的一个字节数据装入R0
- 可以在这些指令上使用条件执行。但要注意**条件标志要先于字节标志**，LDREQB Rx, <address> (不是 LDRBEQ...)。

LDRH半字数据加载指令:

LDR {<cond>} H Rd , <地址>;

LDRH用于存储器中将一个16位数据装载到指定的目标寄存器，目标寄存器的高16位自动清零，指令要求内存地址是半字对齐，当Rd为PC时，从存储器中读取的子数据被当做目的地址，程序自动实现跳转。

LDRH R0 , [R1 , #5] ; 将[R1, #5]指向的2字节加载到R0，高16位清零。

STR字数据存储指令

3. STR字数据存储指令:

STR{<cond>} <Rd>,<addr>;

- 地址addr寻址方式同LDR指令。

例如:

STR R0, [R1, #5]!

STR R2, [R1, #16]

STR R0, [R7], #-8

MOV R1, #UARTADD; 将外设端口地址UARTADD装入R1中

STR R0, [R1]; 将数据输出到外设端口寄存器中;

▶ 应用举例： (1) 用ARM指令实现 $x = (a+b) - c$:

LDR R4, =a; (R4) =a, a为内存变量地址

LDR R0, [R4];((R4))→R0, (a)→R0

LDR R5, =b;

LDR R1, [R5]; ((R5))→R1, (b)→R1

ADD R3, R0, R1; (a)+(b)→R3

LDR R4, =c

LDR R2, [R4]; ((R4))→R2, (c)→R2

SUB R3, R3, R2; a+b-c → R3

LDR R4, =x ;(R4)=x

STR R3, [R4]; (a+b) - c存入x变量

(2) 用ARM指令实现 $x=a*(b+c)$

```
ADR    R4,  b;    变量b的地址装入R4中; LDR R4, =b
LDR    R0,  [R4]   ; (b)→R0
ADR    R4,  c      ; LDR R4, =c
LDR    R1,  [R4]   ; (c)→R1
ADD    R2,  R0, R1 ; b+c →R2
ADR    R4,  a      ; LDR R4, =a
LDR    R0,  [R4]   ; (a)→R0
MUL    R3,  R2, R0 ; (b+c)*a→R2
ADR    R4,  x      ; LDR R4, =x
STR    R3,  [R4]   ; (b+c)*a→x
```


STRB字节数据存储指令

4. STRB字节数据存储指令:

STR{<cond>}B <Rd>,<addr>;

- ▶ 功能：把寄存器Rd中的低8位字节数据保存到addr所表示的内存地址中。其他用法同STR指令。

例如：

STRB R0, [R1];
入R1表示的内存地址中

将寄存器R0中的低8位数据存

数据加载/存储指令

变址模式	数据	基址寄存器	示例
回写前变址	mem[base+offset]	基址寄存器加上偏移	LDR r0, [r1, #4]!
前变址	mem[base+offset]	不变	LDR r0, [r1, #4]
后变址	mem[base]	基址寄存器加上偏移	LDR r0, [r1], #4

例子

寄存器

r0
r1

存储器

0x01010101
0x02020202

地址
0x00009000
0x00009004

▶ PRE

$r0 = 0x00000000$, $r1 = 0x00009000$,

$\text{Mem32}[0x00009000] = 0x01010101$

$\text{Mem32}[0x00009004] = 0x02020202$

回写型前变址寻址: `LDR r0, [r1, #0x4]!`

▶ **POST** $r0 = 0x02020202$,
 $r1 = 0x00009004$

前变址寻址: `LDR r0, [r1, #0x4]`

▶ **POST** $r0 = 0x02020202$, $r1 = 0x00009000$

后变址寻址: `LDR r0, [r1], #0x4`

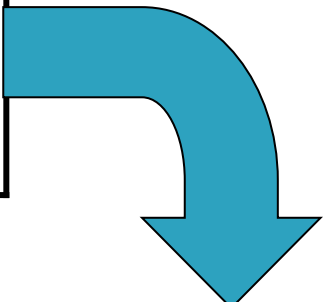
▶ **POST** $r0 = 0x01010101$, $r1 = 0x00009004$

例子

```
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:...
```

批量数据加载/存储指令

5. LDM批量数据加载指令:

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

- ▶ 功能：从一片连续的内存单元读取数据到各个寄存器中，内存单元的起始地址为基址寄存器Rn的值，各个寄存器由寄存器列表regs表示。该指令一般用于多个寄存器数据的出栈。
- **{!}**：若选用了此后缀，则当指令执行完毕后，将最后的地址写入基址寄存器。
- 在存储R15到内存中时，自动保存**CPSR** 位。在重新装载R15 的时候，除非你要求否则不恢复 CPSR 位。要求的方法是在寄存器列表后跟随一个 ‘^’ 。

批量数据加载/存储指令

6. STM批量数据存储指令:

$STM\{\langle cond \rangle\}\{\langle type \rangle\} \langle Rn \rangle\{!\}, \langle regs \rangle;$

- ▶ 功能：将各个寄存器的值存入一片连续的内存单元中，内存单元的起始地址为基址寄存器Rn的值，各个寄存器由寄存器列表regs表示。该指令一般用于多个寄存器数据的入栈。

批量数据加载/存储指令

- ▶ 提供一个有用的简写，要包含一个范围的寄存器，可以简单的只写第一个和最后一个，并在其间加一个横杠。

例如：

R0-R3 等同与 R0, R1, R2, R3,

STMFD R13!, {R0-R12, R14}

...

LDMFD R13!, {R0-R12, PC}

寄存器列表无顺序，
低存储单元对应低标
号寄存器

- 保存所有的寄存器，做一些事情，接着重新装载所有的寄存器。从 R14 装载 PC，不触及 CPSR 标志。

批量数据加载/存储指令

- 指令中，type字段有以下几种：

FD: 满递减堆栈；后地址加4；

FA: 满递增堆栈；前地址加4；
ID: 满递减堆栈；前地址减4；
(一个满栈的栈指针指向上次写的最后一个数据单元)

DA: 每次传送后地址减4；

ED: 空递减堆栈；

DB: 每次传送前地址减4；

EA: 空递增堆栈；（空栈的栈指针指向第一个空闲单元。）

批量数据加载/存储指令

- ▶ 4 种 ‘类型’ 就变成了 8 个指令：

栈

其他

STMEAD

STMIOB

预先增加索引

STMED

STMIA

过后增加索引

STMEDB

STMDB

预先减少索引

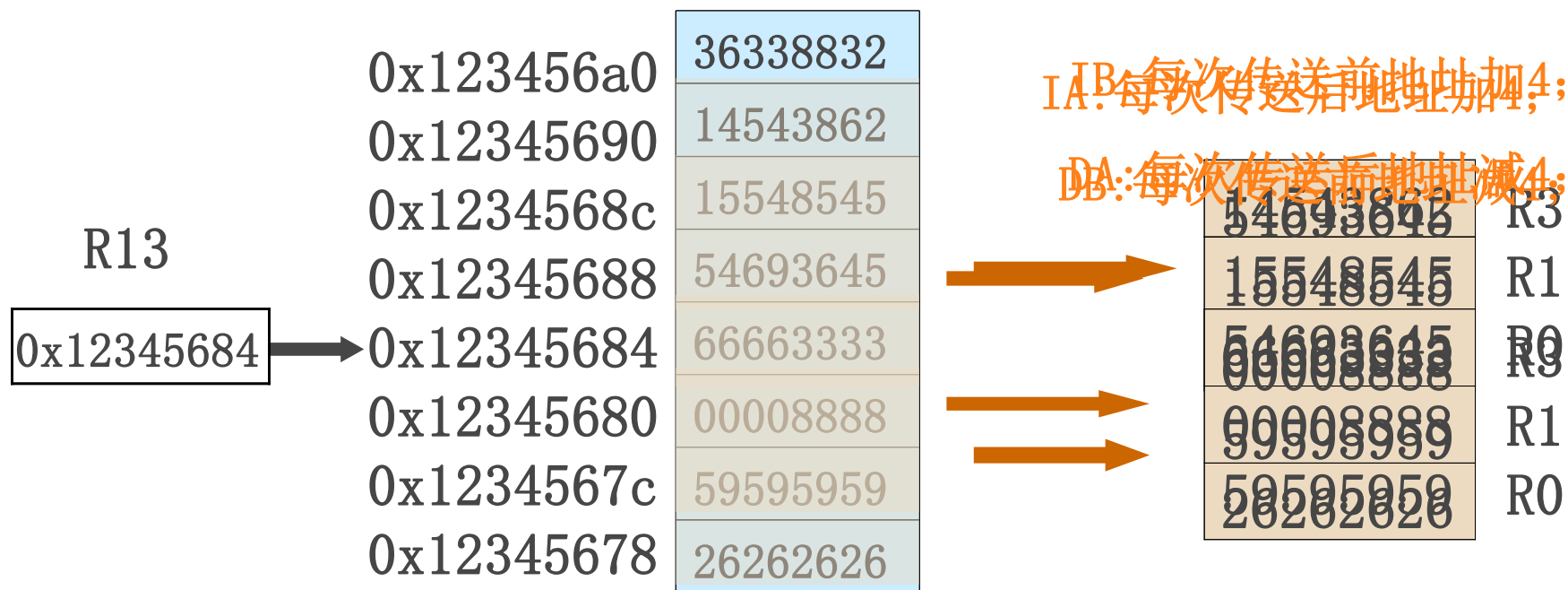
STMEDB

STMDB

过后减少索引

批量数据加载/存储指令

- IA、IB、DA、DB指定了地址加还是减，是传送前还是传送后。例如：LDMIA/IB/DA/DB R13!, {R0-R1, R3}; 各指令执行完后，结果如图所示：



内存中的数据

多寄存器传送指令的寻址模式

寻址模式	描述	起始地址	结束地址	$R_n!$
IA	执行后增加	R_n	$R_n+4*N-4$	R_n+4*N
IB	执行前增加	R_n+4	R_n+4*N	R_n+4*N
DA	执行后减少	$R_n-4*N+4$	R_n	R_n-4*N
DB	执行前减少	R_n-4*N	R_n-4	R_n-4*N

注：！ 决定 R_n 的值是否随着传送而改变

例:将存储器中的连续数据装载到寄存器

▶ PRE

mem32[0x80018]=0x03,
mem32[0x80014]=0x02,
mem32[0x80010]=0x01,
r0=0x00080010, r1=0x00000000,
r2=0x00000000, r3=0x00000000

执行指令: **LDMIA r0!, {r1-r3}**

▶ POST

r0=0x0008001c,
r1=0x00000001,
r2=0x00000002,
r3=0x00000003

地址指针 存储地址 数据

r0=0x80010 →

0x80020	0x00000005
0x8001c	0x00000004
0x80018	0x00000003
0x80014	0x00000002
0x80010	0x00000001
0x8000c	0x00000000

r3=0x00000000

r2=0x00000000

r1=0x00000000

例子

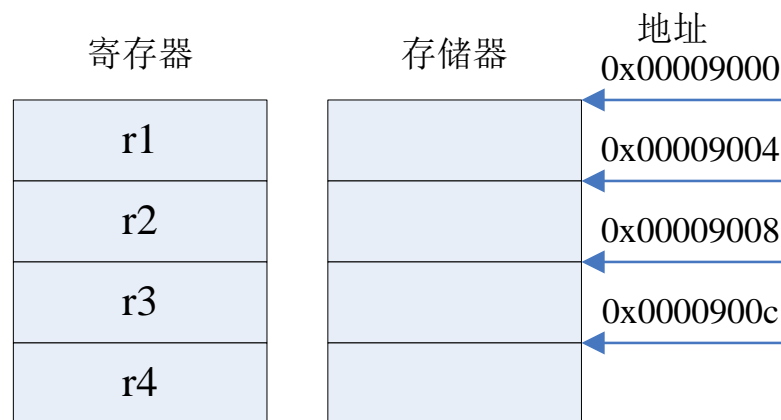
要求:

保存r1~r3到内存地址0x9000~0x900c,
并且更新基址寄存器r4

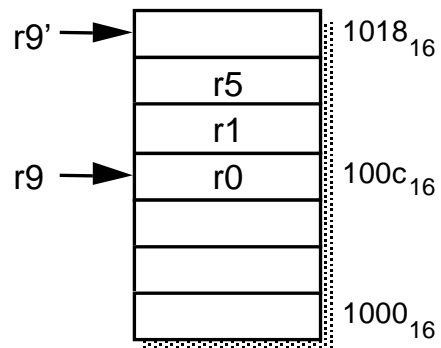
PRE: r1=0x00000001, r2=0x00000002,
r3=0x00000003, r4=0x9000

执行操作: STMIA r4!, {r1, r2, r3}(执行
后增加)

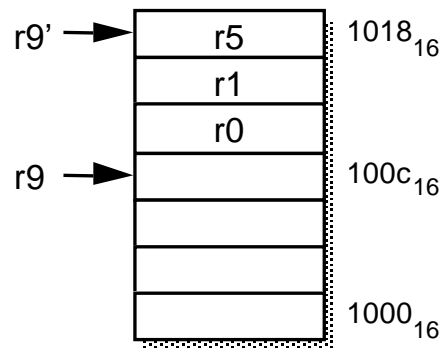
POST: mem32[0x9000]=0x00000001
mem32[0x9004]=0x00000002
mem32[0x9008]=0x00000003
r4=0x900c



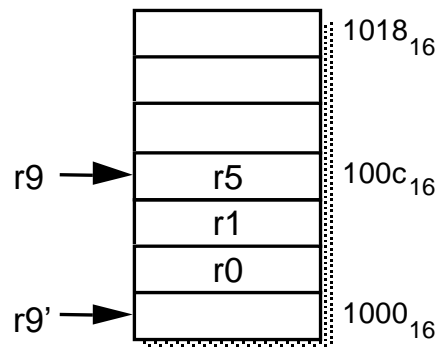
多寄存器传送寻址模式



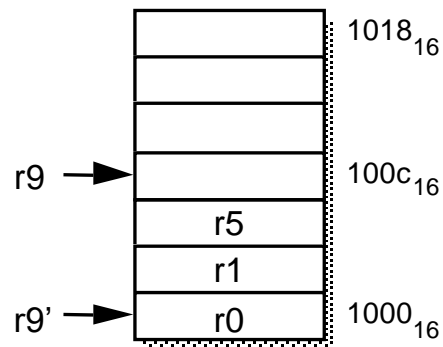
STMIA $r9!$, { $r0,r1,r5$ }



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



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



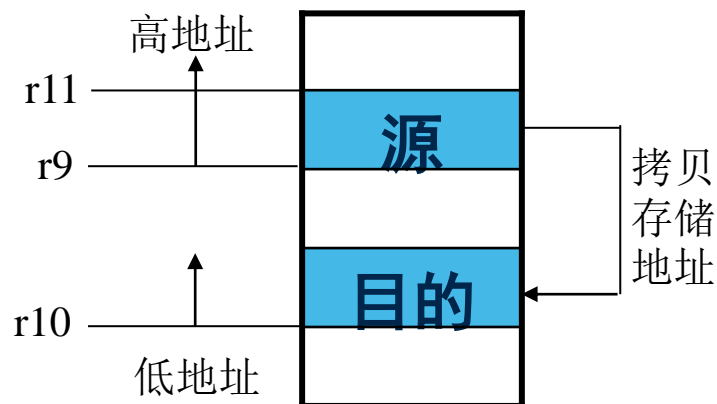
STMDB $r9!$, { $r0,r1,r5$ }

例：完成一个存储器数据块拷贝

- 注：r9——存放源数据的起始地址
r10——存放目标起始地址
r11——存放源结束地址

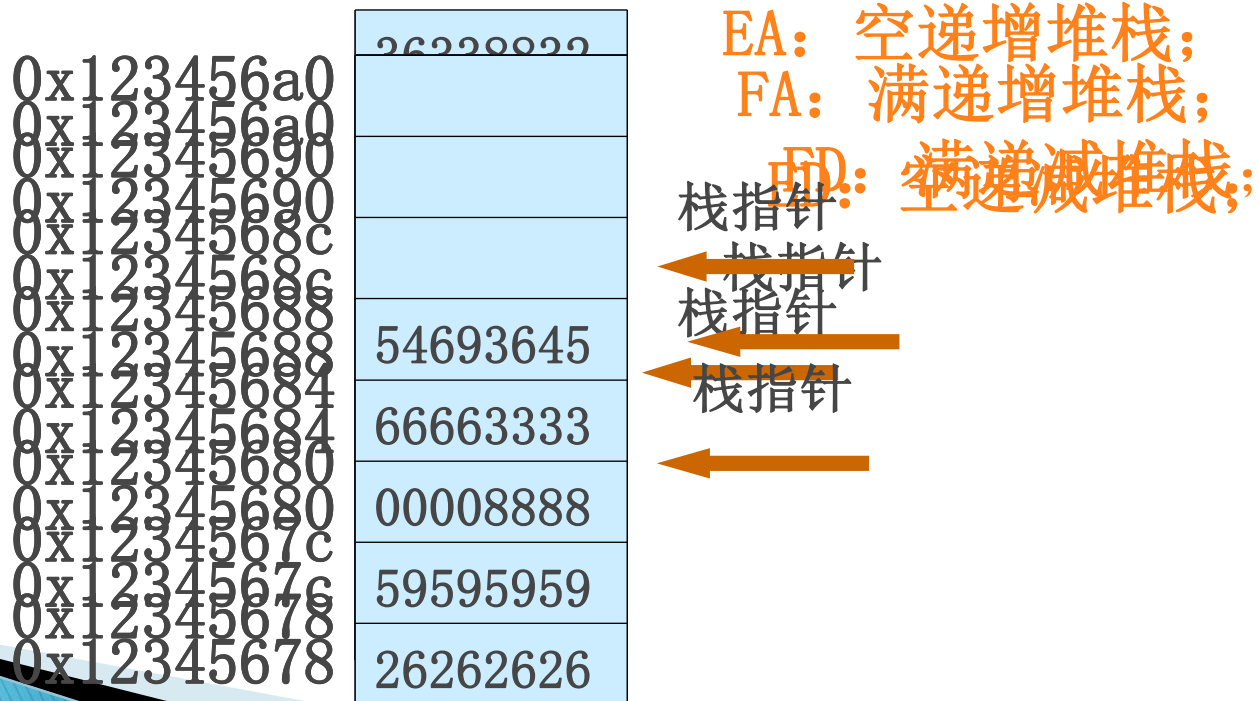
loop

```
LDMIA r9!, {r0-r7}    ;装载32字节并更新r9指针  
STMIA r10!, {r0-r7}    ;存储32字节并更新r10指针  
CMP  r9,  r11          ;是否到达结束地址  
BNE  loop              ;不相等跳转
```



批量数据加载/存储指令

- 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。一个满栈的栈指针指向上次写的最后一个数据单元，而空栈的栈指针指向第一个空闲单元。一个降序栈是在内存中反向增长而升序栈在内存中正向增长。如下图所示，数据以16进制存储。



堆栈操作寻址方式

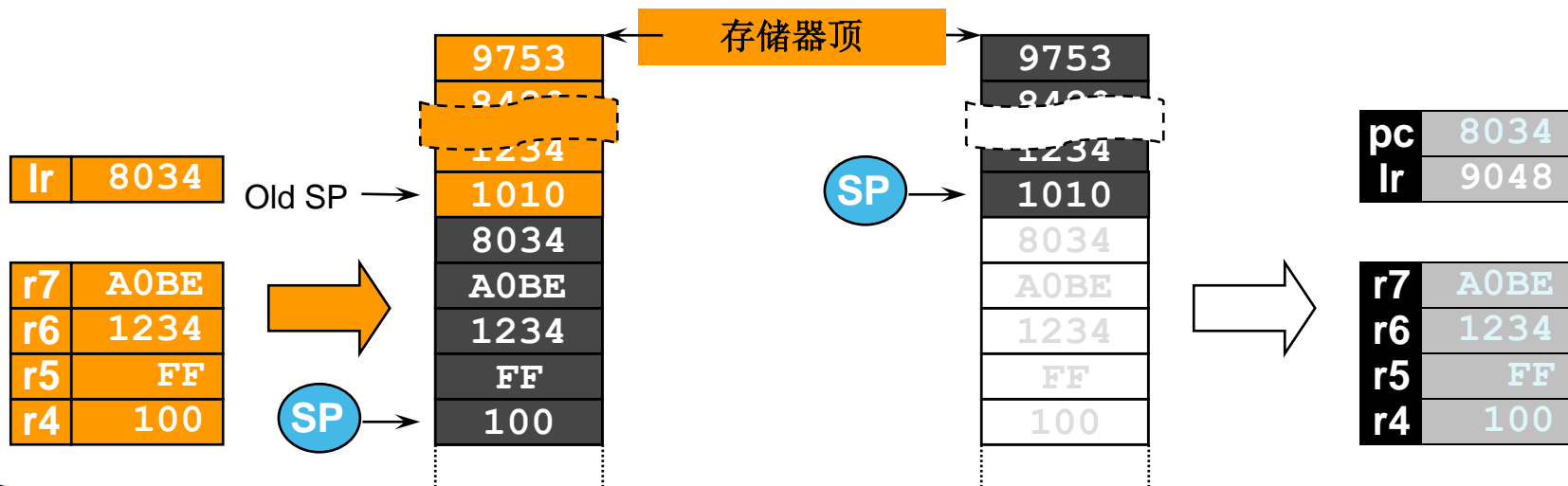
寻址方式	说明	pop	=LDM	push	=STM
FA	递增满	LDMFA	LDMDA	STMFA	STMIB
FD	递减满	LDMFD	LDMIA	STMFD	STMDB
EA	递增空	LDMEA	LDMDB	STMEA	STMIA
ED	递减空	LDMED	LDMIB	STMED	STMDA

堆栈

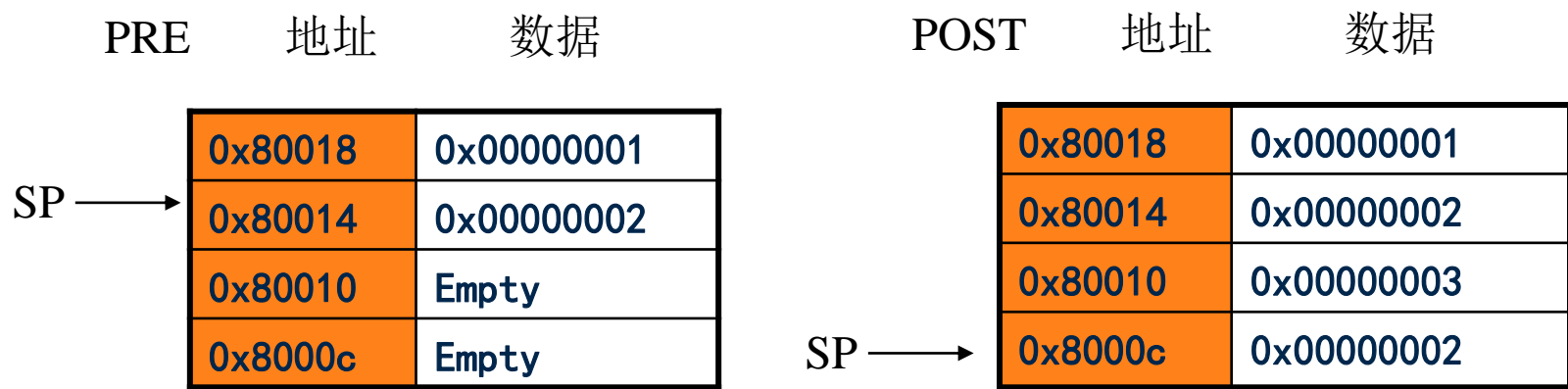
- ▶ ARM堆栈操作通过块传送指令来完成:
 - STMFD (Push) 块存储- Full Descending stack [STMDB]
 - LDMFD (Pop) 块装载- Full Descending stack [LDMIA]

STMFD sp!, {r4-r7, lr}

LDMFD sp!, {r4-r7, pc}



例：把寄存器内容放入堆栈，更新sp



PRE : r1=0x00000002, r4=0x00000003, sp=0x00080014

执行指令： STMFD sp!, {r1,r4}

POST: r1=0x00000002, r4=0x00000003, sp=0x0008000c

SWP字数据交换指令

7. SWP字数据交换指令:

$SWP\{<cond>\} <Rd>, <op1>, [<op2>];$

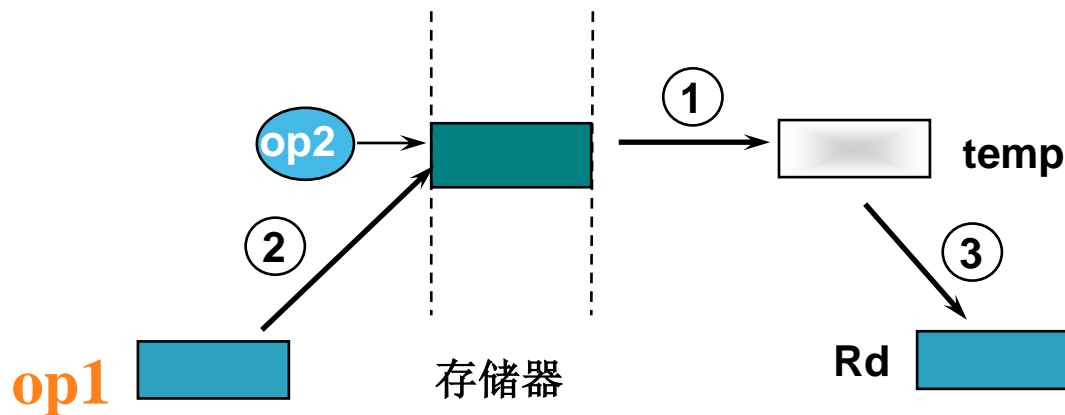
- 功能：从op2所表示的内存装载一个字并把这个字放置到目的寄存器Rd中，然后把寄存器op1的内容存储到同一内存地址中，即 $Rd=[op2], [op2]=op1$ 。其中op1，op2均为寄存器

例如：

- 如果目的和操作数 1 是同一个寄存器，则把寄存器的内容和给定内存位置的内容进行交换。
 $SWP R0, R1, [R2]$ 将R2所表示的内存单元中的字数据装载到R0，然后将R1中的字数据保存到R2所表示的内存单元中。

SWP

- 在寄存器和存储器之间，由一次存储器读和一次存储器写组成的原子操作。完成一个字节或字的交换。



- 可用作信号量
- 不能由armcc编译产生，必须使用汇编器。

SWP指令应用示例

Spin

MOV r1, =semaphore

MOV r2, #1

SWP r3, r2, [r1]

CMP r3, #1

BEQ spin

注：信号量指向的单元是0或1，如果为1，则表示该服务被另一个过程使用，程序继续循环，直至为0

SWPB字节数据交换指令

8. SWPB字节数据交换指令:

SWP{<cond>}B <Rd>, <op1>, [<op2>];

- ▶ 功能：从op2所表示的内存装载一个字节并把这个字节放置到目的寄存器Rd的低8位中，Rd的高24位设置为0；然后将寄存器op1的低8位数据存储在到同一内存地址中。

例如：

SWPB R0, R1, [R2]; 将R2所表示的内存单元中的一个字节数据装载到R0的低8位，然后将R1中的低8位字节数据保存到R2所表示的内存单元中。

3.3.5 协处理指令

ARM体系结构允许通过增加协处理器来扩展指令集。ARM协处理器具有自己专用的寄存器组，它们的状态由控制ARM状态的指令的镜像指令来控制。程序的控制流指令由ARM处理器来处理，所有的协处理器指令只能同数据处理的和数据传送有关。

ARM协处理器指令可完成下面三类操作：

- ARM协处理器的数据处理操作。
- ARM处理器和协处理器的寄存器之间数据传输。
- ARM协处理器的寄存器和存储器之间数据传输。

ARM协处理器指令主要包括5条，它们的格式和功能如下：

助记符	说明	功能
CDP coproc,opcode1,CRd,CRn,CRm{,opcode2}	协处理器数据操作指令	用于ARM处理器通知协处理器执行特定的操作
LDC{L} coproc,CRd <地址>	协处理器数据读取指令	从某一连续的存储单元将数据读取到协处理器的寄存器中
STC{L} coproc,CRd <地址>	协处理器数据写入指令	将协处理器的寄存器数据写入到某一连续的存储单元中
MCR coproc,opcode1,Rd,CRn{,opcode2}	ARM寄存器到协处理器寄存器的数据传输指令	将ARM处理器的寄存器中的数据传输到协处理器的寄存器中
MRC coproc,opcode1,Rd,CRn{,opcode2}	协处理器寄存器到ARM寄存器的数据传输指令	将协处理器的寄存器中的数据传输到ARM处理器的寄存器中

3.3.6 异常产生指令

ARM处理器有两条异常产生指令，软中断指令（**SWI**）和断点中断指令（**BKPT**）。

1. SWI指令

SWI {条件} 24位立即数

SWI指令用于产生**SWI**异常中断，实现从用户模式切换到管理模式，**CPSR**保存到管理模式下的**SPSR**中，执行转移到**SWI**向量，；其他模式下也可使用**SWI**指令，同样切换到管理模式。该指令不影响条件码标志。

示例：

```
SWI 0x02
```

```
;软中断，调用操作系统编号为02 的系统例程
```

2. BKPT指令

BKPT 16位立即数

BKPT指令产生软件断点中断，软件调试程序可以使用该中断。立即数会被ARM硬件忽视，但能被调试工具利用来得到有用的信息。

示例：

```
BKPT 0xFF32
```

3.4 Part Four

Thumb指令简介

为了兼容存储系统总线宽度为**16**位的应用系统，**ARM**体系结构中提供了**16**位**Thumb**指令集，它可以看作是**ARM**指令压缩形式的子集，是针对代码密度的问题而提出的，它具有**16**位的代码密度，这对于嵌入式系统来说至关重要。

Thumb不是一个完整的体系结构，不能指望处理器只执行**Thumb**指令而不支持**ARM**指令集。因此，**Thumb**指令只需要支持通用功能，必要时可以借助完善的**ARM**指令集。只要遵循一定的调用规则，**Thumb**子程序和**ARM**子程序可以互相调用。当处理器在执行**ARM**程序段时，称**ARM**处理器处于**ARM**工作状态；当处理器在执行**Thumb**程序段时，称**ARM**处理器处于**Thumb**工作状态。

Thumb指令集没有协处理器指令、信号量指令以及访问**CPSR**或**SPSR**的指令，没有乘加指令及**64**位乘法指令等，并且指令的第二操作数受到限制；除了分支指令**B**有条件执行功能外，其他指令均为无条件执行；大多数**Thumb**数据处理指令采用**2**地址格式。

Thumb指令集与ARM指令集的区别一般有如下4点：

1. 跳转指令

程序相对转移，特别是条件跳转与ARM代码下的跳转相比，在范围上有更多的限制，转向子程序是无条件的转移。

2. 数据处理指令

数据处理指令是对通用寄存器进行操作，在大多数情况下，操作的结果须放入其中一个操作数寄存器中，而不是第3个寄存器中。数据处理操作比ARM状态的更少。访问R8~R15受到一定限制：除MOV和ADD指令访问寄存器R8~R15外，其他数据处理指令总是更新CPSR中的ALU状态标志；访问寄存器R8~R15的Thumb数据处理指令不能更新CPSR中的ALU状态标志。

3. 单寄存器加载和存储指令

在Thumb状态下，单寄存器加载和存储指令只能访问寄存器R0～R7。

4. 多寄存器加载和多寄存器存储指令

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指令可以加载程序计数器PC。

3.5

Part Five

ARM汇编语言程序简介

ARM源程序文件

使用简单的文本编辑器或者其他的编程开发环境进行编辑.

文件类型	扩展名
汇编语言文件	.s
C 语言源文件	.c
C++ 源文件	.cpp
引入文件	.INC
头文件	.h

3.5.1 伪操作

ARM汇编语言程序是由**机器指令、伪指令和伪操作**组成的。伪操作是ARM汇编语言程序里的一些特殊的指令助记符，和指令系统中的助记符不同，这些助记符没有相应的操作码。伪操作主要是为完成汇编程序做一些准备工作，在源程序汇编过程中起作用，一旦汇编完成，伪操作的使命就完成了。

宏是一段独立的程序代码，通过伪操作定义，在程序中使用宏指令即可调用宏。当程序被汇编时，汇编程序校对每个宏调用进行展开，用宏定义代替源程序中的宏指令。

1. 符号定义伪操作

符号定义伪操作用于定义ARM汇编程序中的变量、对变量赋值及定义寄存器名称等。常用伪操作有：

- **GBLA**、**GBLL**和**GBLS**：定义全局变量。
- **LCLA**、**LCLL**和**LCLS**：定义局部变量。
- **SETA**、**SETL**和**SETS**：为变量赋值。
- **RLIST**：为通用寄存器列表定义名称。
- **CN**：为协处理器的寄存器定义名称。
- **CP**：为协处理器定义名称。
- **DN**和**SN**：为VFP的寄存器定义名称。
- **FN**：为FPA的浮点寄存器定义名称。

伪操作

- LCLA、LCLL、LCLS

伪指令用于定义一个汇编程序中的局部变量，并初始化

格式： **LCLA/LCLL/LCLS** 局部变量名

↓
定义一个局部的数字变量，初始化为0

↓
定义一个局部的逻辑变量，初始化为F

↓
定义一个局部的字符串变量，初始化为空串

➤这三条伪指令用于声明局部变量，在其局部作用范围内变量名必须唯一。

伪操作

例:

- LCLA num1 ;声明一个局部的数字变量，变量名为
 ;num1
- LCLL l2 ;声明一个局部的逻辑变量，变量名为l2
- LCLS str3 ;定义一个局部的字符串变量，变量名为
 ;str3
- num1 SETA 0xabcd ;将该变量赋值为0xabcd
- l2 SETL {FALSE} ;将该变量赋值为假
- str3 SETS "Hello!" ;将该变量赋值为"Hello!"
-

在宏内定义局部变量后，若在宏外使用该指令时编译会出错。

伪操作

- GBLA、GBLL、GBLS

伪操作定义一个汇编程序中的全局变量，并初始化

格式：GBLA/GBLL/GBLS 变量名

↓
定义一个全局的数字变量，并初始化为0

↓
定义一个全局的逻辑变量，并初始化为F

↓
定义一个全局字符串变量，并初始化为空串

➤这三条伪指令用于定义全局变量，因此在整个程序范围内变量名必须唯一。

伪操作

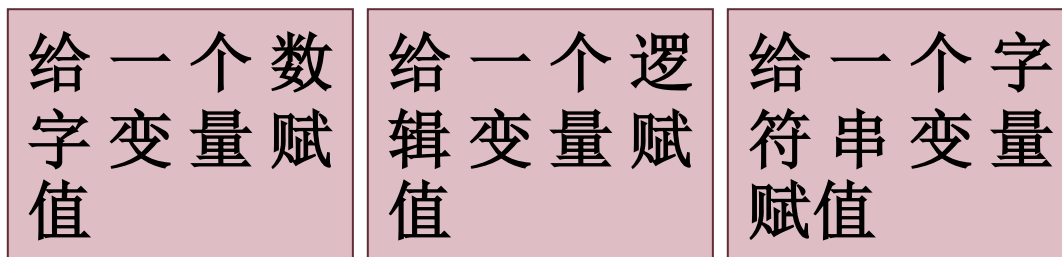
例:

- `GBLA num1` ;定义一个全局的数字变量num1
- `num1 SETA 0xabcd`
- `GBLL l2` ;定义一个全局的逻辑变量, 变量名为l2
- `l2 SETL {FALSE}` ;将该变量赋值为假
- `GBLS str3` ;定义一个全局的字符串变量str3
- `str3 SETS "Hello!"`

伪操作

- SETA、SETL、SETS

格式：变量名 SETA/SETL/SETS 表达式



➤格式中的变量名必须为已经定义过的全局或局部变量，表达式为将要赋给变量的值。

伪操作

例：

- LCLA num1 ;定义局部数字变量
- num1 SETA 0x1234 ;赋值为0x1234
- LCLS str3 ;定义局部字符串变量
- str3 SETS "Hello!" ;赋值为"Hello!"


伪操作

变量代换

- 如果在字符串变量的前面有一个\$字符，在汇编时编译器将用该字符串变量的内容代替该串变量。

例：

LCLS	str1
LCLS	str2
str1	SETS "book"
str2	SETS "It is a \$str1"



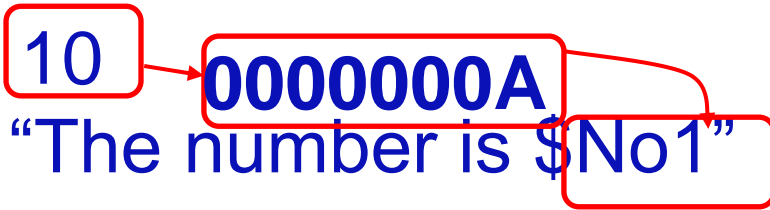
则在汇编后，str2的值为” It is a book”。

伪操作

- 如果在数字变量前面有一个代换操作符“\$”，编译器会将该数字变量的值转换为十六进制的字符串，并用该十六进制的字符串代换“\$”后的数字变量。

例：

No1 SETA 10 0000000A
Str1 SETS “The number is \$No1”

A diagram illustrating the substitution process. A red box highlights the value '10' in the 'SETA' instruction. An arrow points from this box to the hexadecimal string '0000000A'. Another red box highlights this string, and an arrow points from it to the '\$No1' part of the 'SETS' instruction string, showing how the value is substituted into the string.

则汇编后**Str1**的值为 “The number is 0000000A”。

伪操作

- 如果需要将“\$”字符 加入到字符串中，可以用“\$\$”代替，此时编译器将不再进行变量代换，而是把“\$\$”看作一个“\$”。

例：

Str SETS “The character is \$\$”

则编译后**Str**字符串的值为
“The character is \$”。

➤使用 “.” 来表示字符串中变量名的结束。

汇编程序的变量代换

```
AREA ||.text||, CODE, READONLY ;代码段,名称为||.text||,属性为只读
GBLS str1 ;声明str1为全局字符串
GBLS str2 ;声明str2为全局字符串
GBLL l1 ;声明l1为全局逻辑变量
```

```
GBLA num1
```

运行后结果:

aaa str1:bbb l1:T,a1:0000004Fccc

```
l1 SETL
```

{TRUE}

```
num1 SETA
```

0x4f

```
str1 SETS
```

"bbb"

```
str2 SETS
```

"aaa str1:\$str1, l1:\$l1, a1:\$num1.ccc"

;str2包含了多个变量

;函数main

;保存必要的寄存器和返回地址到数据栈

```
main PROC
STMFD sp!,{lr}
```

```
ADR r0, strhello
```

```
BL _printf
```

;调用C运行时库的_printf函数打印字符串

```
LDMFD sp!,{pc}
```

;恢复寄存器值

;strhello代表本地字符串的地址

;定义一段字节空间

;函数main结束

```
strhello
```

```
DCB "$str2\n\0"
```

```
ENDP
```

2. 数据定义伪操作

数据定义伪操作用于数据表定义、文字池定义、数据空间分配等。常用伪操作有：

- **LTORG**：声明一个数据缓冲池的开始。
- **MAP**：定义一个结构化的内存表的首地址。
- **FIELD**：定义结构化内存表的一个数据域
- **SPACE**：分配一块内存空间，并用0初始化。
- **DCB**：分配一段字节的内存单元，并用指定的数据初始化。
- **DCD**和**DCDU**：分配一段字的内存单元，并用指定的数据初始化。
- **DCFD**和**DCFDU**：分配一段双字的内存单元，并用双精度的浮点数据初始化。
- **DCFS**和**DCFSU**：分配一段字的内存单元，并用单精度的浮点数据初始化。
- **DCQ**和**DCQU**：分配一段双字的内存单元，并用64位整型数据初始化。
- **DCW**和**DCWU**：分配一段半字的内存单元，并用指定的数据初始化。

伪操作

- DCB

DCB用于分配一块字节单元并用伪指令中指定的表达式进行初始化。

格式：标号/变量 **DCB** **表达式**

DCB 可用
“=” 代替

表达式可以为
使用双引号的
字符串或 0 -
255的数字

例：

Array1 DCB 1,2,3,4,5

str1 DCB “Your are welcome! ”

;数组
;构造字符串
;并分配空间

伪操作

- DCW/DCWU

格式：标号/变量 **DCW/DCWU** 表达式

DCW分配一段半字存储单元并用表达式值初始化，它定义的存储空间是半字对齐的

DCWU 功能跟 **DCW** 类似，只是分配的字存储单元不严格半字对齐

例：

**Arrayw1 DCW 0xa,-0xb,0xc,-0xd ;构造固定数组并分
;配半字存储单元**

伪操作

- DCD/DCDU

格式：标号/变量

DCD

DCDU

表达式

DCD用于分配一块字存储单元并用伪指令中指定的表达式初始化，它定义的存储空间是字对齐的。也可用“&”代替

DCDU只是分配的存储单元不严格字对齐

例：

Arrayd1	DCD	1334, 234, 345435	;构造固定数组并分配
			;字为单元的存储单元
Label	DCD	str1	;该字单元存放str1的地址

伪操作

- SPACE

格式： 标号 **SPACE** **表达式**

SPACE用于分配一片连续的存储区域并初始化为**0**，也可用“**%**”代替

表 达 式 为
要 分 配 的
字 节 数

例：

Freespace SPACE 1000 ;分配1000字节的存储空间

伪操作

- MAP

格式: **MAP**



MAP 定义一个结构化的内存表的首地址，可以用“^”来代替

表达式



表达式可以为程序中的标号或数学表达式

[,基址寄存器]



基址寄存器为可选项，当基址寄存器选项不存在时，表达式的值即为内存表的首地址，当该选项存在时，内存表的首地址为表达式的值与基址寄存器的和

➤ **MAP**可以与**FIELD**伪操作配合使用来定义结构化的内存表。

➤ 例：

MAP 0x130, R2

；内存表首地址为 **$0x130 + R2$**

伪操作

- **FILED**

格式：标号 **FIELD** 字节数

↓
FIELD用于定义一个结构化内存表中的数据域，可用“#”来代替**FILED**

➤ **FIELD**常与**MAP**配合使用来定义结构化的内存表：**FIELD**伪指令定义内存表中的各个数据域，**MAP**则定义内存表的首地址，并为每个数据域指定一个标号以供其他的指令引用。

➤ **注意**：**MAP**和**FIELD**伪指令仅用于定义数据结构，并不分配存储单元。

伪操作

例：

MAP	0xF10000	;定义结构化内存表首地址为 ;0xF10000
count	FIELD 4	;定义count的长度为4字节， ;位置为0xF10000
x	FIELD 4	;定义x的长度为4字节， 位置 ;为0xF10004
y	FIELD 4	;定义y的长度为4字节， 位置 ;为0xF10008

3.汇编控制伪操作

汇编控制伪操作用于条件汇编、宏定义、重复汇编控制等，常用伪操作有：

- **IF、ELSE和ENDIF**：根据条件把一段源程序代码包括在汇编程序内或排除在程序之外。
- **WHILE和WEND**：根据条件重复汇编相同的源程序代码段。
- **MACRO和MEND**：**MACRO**标识宏定义的开始，**MEND**标识宏定义结束。用**MACRO**和**MEND**定义一段代码，称为宏定义体，在程序中可以通过宏指令多次调用该代码段。
- **MEXIT**：用于从宏中跳转出去。

伪操作

- IF、ELSE、ENDIF

IF、ELSE、ENDIF伪操作能根据逻辑表达式的成立与否决定是否在编译时加入某个指令序列。

格式： IF 逻辑表达式

假 代码段1
ELSE
 代码段2
ENDIF



如果IF后面的逻辑表达式为真，则编译代码段1，否则编译代码段2。

➤ **ELSE**及代码段2也可以没有，这时，当**IF**后面的逻辑表达式为真，则加入代码段1，否则继续编译后面的指令。

伪操作

例:

```
GBLL    isbig    ;声明一个全局的逻辑变量,  
                ;变量名为isbig
```

```
isbig SETL    ...
```

```
IF    isbig  
        add r0,r0,1
```

```
ELSE  
        sub r0,r0,1
```

```
ENDIF
```

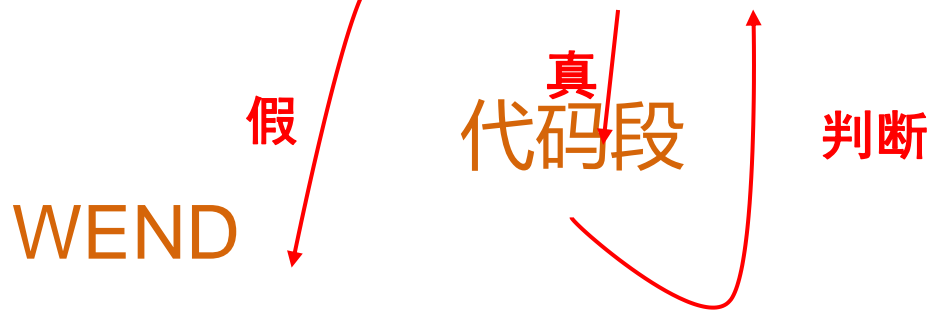
**IF、ELSE、
ENDIF 伪指令可
以嵌套使用**

伪操作

- WHILE、WEND

WHILE和WEND伪指令能根据逻辑表达式的成立与否决定是否循环执行这个代码段。

格式：WHILE 逻辑表达式



当**WHILE**后面的逻辑表达式为真，则执行代码段，该代码段执行完毕后，再判断逻辑表达式的值，若为真则继续执行，一直到逻辑表达式的值为假。

➤ **WHILE、WEND**伪指令可以嵌套使用。

伪操作

例：

GBLA num ;声明全局的数字变量num

num SETA 9 ;由num控制循环次数

.....

WHILE num > 0

sub r0,r0,1

add r1,r1,1

num SETA num-1

WEND



4. 其它伪操作

其它伪操作常用的有段定义伪操作、入口点设置伪操作、包含文件伪操作、标号导出或引入声明等：

- **ALIGN**：边界对齐。
- **AREA**：段定义。
- **CODE16**和**CODE32**：指令集定义。
- **END**：汇编结束。
- **ENTRY**：程序入口。
- **EQU**：常量定义。
- **EXPORT**和**GLOBAL**：声明一个符号可以被其它文件引用。
- **IMPORT**和**EXTERN**：声明一个外部符号。
- **GET**和**INCLUDE**：包含文件。
- **INCBIN**：包含不被汇编的文件。
- **RN**：给特定的寄存器命名。
- **ROUT**：标记局部标号使用范围的界限。

其他伪操作

▶ AREA

格式: **AREA** 段名 属性,

AREA用于定义一个代码段、数据段或者特定属性的段

属性部分表示该代码段/数据段的相关属性，多个属性可以用“，”分隔。

● 常见属性如下:

- DATA: 定义数据段。
- CODE: 定义代码段。
- READONLY: 表示本段为只读。
- READWRITE: 表示本段可读写。

其他伪操作

- ▶ 一个汇编程序至少应该包含一个段，当程序太长时，也可以将程序分为多个代码段和数据段。
- ▶ 例：
 - `AREA test, CODE, READONLY`
 - `AREA ||.text||, CODE, READONLY`

其他伪操作

▶ CODE16、CODE32

格式: **CODE16** **CODE32**

CODE16伪操作指示
编译器后面的代码为
16位的Thumb指令

CODE32伪操作指示
编译器后面的代码为
32位的ARM指令

▶如果在汇编源代码中同时包含**Thumb**和**ARM**指令时，可以用“**CODE32**”通知编译器其后的指令序列为**32位的ARM指令**，用“**CODE16**”伪指令通知编译器其后的指令序列为**16位的Thumb指令**。

▶在使用**ARM**指令和**Thumb**指令混合编程的代码里，这两条伪指令后面的代码类型是不同的，但它们并不能对处理器进行状态的切换。

其他伪操作

▶ ENTRY

格式: ENTRY

ENTRY用于指定汇编程序的入口。
在一个完整的汇编程序中至少要有一个ENTRY，也可以有多个。

➤下面的代码使用了ENTRY:

```
AREA subrout, CODE, READONLY
ENTRY                      ;指定程序入口

start
    MOV    r0, #10        ;设置参数
    MOV    r1, #3
    BL     doadd           ;调用子函数
```

其他伪操作

▶ END

格式: **END**

END告诉编译器
已经到了源程序
的结尾

例:

AREA constdata, DATA, READONLY

.....

END

;结尾

其他伪操作

▶ EQU

格式：名称

EQU

表达式[, 类型]

EQU用于将程序中的数字常量、标号、基于寄存器的值赋予一个等效的名称，这一点类似于C语言中的**#define**，可用“*”代替**EQU**

如果表达式为**32**位的常量，我们可以指定表达式的数据类型，类型域可以有以下三种：
CODE16/CODE32/DATA

例：

num1 EQU 1234 ;定义num1为1234

addr5 EQU str1+0x50

d1 EQU 0x2400, CODE32

;定义d1的值为0x2400,且
;该处为**32**位的**ARM**指令

其他伪操作

▶ EXPORT

格式: **EXPORT** 标号[,WEAK]

EXPORT 在程序中声明一个全局标号，其他文件中的代码可以引用该标号。用户也可以用**GLOBAL**代替**EXPORT**

[,WEAK]可选项声明其他文件有同名的标号，则该同名标号优先于该标号被引用

例:

```
AREA    ||.text||, CODE, READONLY
main    PROC
```

```
.....
        ENDP
```

```
EXPORT main
END
```

;声明一个可全局引用的函数main

其他伪操作

▶ IMPORT

格式: **IMPORT** 标号 **[, WEAK]**

IMPORT告诉编译器这个标号要在当前源文件中使用, 但标号是在其他的源文件中定义的。不管当前源文件是否使用过该标号, 这个标号都会加入到当前源文件的符号表中

[, WEAK]选项表示如果所有的源文件都没有找到这个标号的定义, 编译器也不会提示错误信息。编译器在多数情况下将该标号置为0, 如果这个标号被**B**或**BL**指令引用, 则将**B**或**BL**指令替换为**NOP**操作

例:

```
AREA    mycode, CODE, READONLY
IMPORT      _printf    ;通知编译器当前文件要引用函
                        ;数_printf
END
```

其他伪操作

▶ EXTERN

格式: **EXTERN** 标号 **[,WEAK]**

EXTERN告诉编译器所使用的标号要在当前源文件中引用,但该标号是在其他的源文件中定义的。与**IMPORT**不同的是,如果当前源文件实际上没有引用该标号,该标号就不会被加入到当前文件的符号表中

[, WEAK]选项表示如果所有的源文件都没有找到这个标号的定义,编译器也不会提示错误信息。编译器在多数情况下将该标号置为0,如果这个标号被**B**或**BL**指令引用,则将**B**或**BL**指令替换为**NOP**操作

例:

```
AREA    ||.text||, CODE, READONLY
```

```
EXTERN  _printf ,WEAK
```

```
END
```

;告诉编译器当前文件要引用标号,如果找不到,则不提示错误

其他伪操作

► INCBIN

格式: **INCBIN** 文件名

INCBIN 将一个数据文件或者目标文件包含到当前的源文件中，编译时被包含的文件不作任何变动的存放在当前文件中，编译器从后面开始继续处理。

例:

```
AREA    constdata, DATA, READONLY
INCBIN          data1.dat      ;源文件包含文件data1.dat
INCBIN  E: \DATA\data2.bin    ;源文件包含文件
                                   ;E: \DATA\data2.bin
END
```

3.5.2 伪指令

ARM中的伪指令并不是真正的ARM或Thumb指令，这些伪指令在汇编编译器对源程序进行汇编处理时被替换成对应ARM或Thumb指令（序列）。

常用的伪指令有：

➤ **ADR**：小范围的地址读取伪指令，该指令将基于PC的相对偏移地址或基于寄存器的相对偏移地址读取到寄存器中。格式为：

ADR {cond} register, expr

cond是可选的指令执行条件。

register是目的寄存器。

expr是基于PC或基于寄存器的地址表达式，当地址值是字节对齐时，取值范围在-255~255B；当地址值是字对齐时，取值范围在-1020~1020B；当地址值是16字节对齐时，取值范围更大。

➤ **ADRL**: 中等范围的地址读取伪指令，该指令比**ADR**的取值范围更大。格式为：

ADRL {cond} register, expr

cond是可选的指令执行条件。

register是目的寄存器。

expr是基于**PC**或基于寄存器的地址表达式，当地址值是字节对齐时，取值范围在**-64~64KB**；当地址值是字对齐时，取值范围在**-256~256KB**；当地址值是**16**字节对齐时，取值范围更大；在**32**位的**Thumb-2**指令中，取值范围在可达**-1~1MB**。

➤ **LDR**: 大范围的地址读取伪指令，将一个**32**位的常数或者一个地址值读取到寄存器中。格式为：

LDR {cond} register, = [expr|label-expr]

cond是可选的指令执行条件。

register是目的寄存器。

expr是**32**位常量。

➤ **NOP**空操作伪指令，在汇编时被替换成**ARM**中的空操作。

3.5.3 汇编语言格式

ARM（Thumb）汇编语言的语句格式为：

[标号] <指令|条件|S> <操作数>[:注释]

在ARM汇编程序中，ARM指令、伪操作、伪指令、伪操作的助记符全部用大写字母，或者全部用小写字母，不能既有大写也有小写字母。

所有标号在一行的顶格书写，后面不要添加“：”，所有指令不能顶格书写。

注释内容以“；”开头到本行结束。

源程序中允许有空行，如果单行太长，可以用字符“\”将其分开，“\”后不能有任何字符，包括空格和制表符等。

变量的设置，常量的定义，其标识符必须在一行顶格书写。

3.5.4 汇编语言的程序结构

段（**section**）是ARM汇编语言组织源文件的基本单位，是独立的、具有特定名称的、不可分割的指令或数据序列。段分为代码段和数据段，代码段存放执行代码，数据段存放代码执行时需要的数据。一个ARM汇编程序至少需要一个代码段，较大的程序可以包含多个代码段和数据段。

ARM汇编语言源程序经过汇编后生成可执行的映像文件，格式有axm、bin、elf、hex等。可执行的映像文件通常包括三个部分：

- 一个或多个代码段，代码段的属性为只读。
- 零个或多个包含初始化数据的数据段，属性为可读可写。
- 零个或多个不包含初始化数据的数据段，属性为可读可写。

连接器根据一定的规则将各个段安排到内存的相应位置。源程序中段之间的相对位置与可执行的映像文件中段的相对位置不一定相同。

下面的程序说明了**ARM**标准汇编语言程序的基本结构：

AREA	BUF, DATA, READWRITE	;声明数据段BUF
	count DCB 30	;定义一个字节单元count
AREA	EXAMPLE1, CODE, READONLY	;声明代码段EXAMPLE1
ENTRY		;程序入口
CODE32		;声明32位ARM指令
START		
	LDRB R0, count	;R0 = count
	MOV R1, #10	;R1 = 8
	ADD R0, R0, R1	;R0 = R0 + R1
	B START	
END		

标准汇编语言程序的结构

汇编源程序示例第一部分 (test0源程序)

CODE32

AREA CODE

ARM的汇编语言程序

一般由几个段组成，

每个段均由AREA伪操

;32位的ARM指令段

;代码段，名称codesec,属性为

·只读

main F

本程序定义了两个段，

第一个段为代码段codesec;

第二个段为数据段constdatasec

main

必要的寄存器和返回地址

据栈

签strhello代表的地址值

C运行时库的_printf函数

“Hello world!” 字符串

子函数welcomfun

BL

属性为READONLY，数据段的默
认属性为READWRITE。

BL

LDMFD

sp!,{pc}

;恢复寄存器值

strhello

;strhello代表本地字符串的地址

DCB

"Hello world!\n\0"

;定义一段字节空间

ENDP

;函数main结束

汇编语言程序的结构

汇编源程序示例第二部分

welcomefun			;子函数welcomfun
STMFD	sp!,{lr}		;保存必要的寄存器和返回地
			;址到数据栈
ADR	r0,adrstrarm		;取adrstrarm的地址放到寄存器r0中
LDR	r0,[r0,#0]		;将strarm的值放到r0中
BL	_printf		;调用C运行时库的_printf函数打印
			" world!"字符串
LD			
adrstrarm			
DCI			

本程序定义了两个段，
第一个段为代码段codesec
第二个段为数据段constdatasec

AREA constdatasec, DATA, READONLY,ALIGN=0

;数据段，名称为constdatasec, 属性
;为只读

汇编语言程序的结构

汇编源程序示例第三部分

strarm

**DCB "Welcome to ARM world!\n\0" ;存放 “Welcome to ARM
; world!” 字符串**

EXPORT main ;导出main函数供外部调用

;引入三个C运行时库函数和ARM库

IMPORT _main

IMPORT __main

IMPORT _printf

IMPORT ||Lib\$\$Request\$\$armlib||, WEAK

END

;程序结束

GNU ARM汇编器汇编命令

1 汇编环境:Linux系统环境

2 用途:

(1) 规范ARM汇编语言格式

(2) 提供GNU ARM汇编语言汇编器的汇编命令。

GNU汇编命令是用于指示编译器操作方式的伪操作/伪指令，所有伪操作名称都以“.”为前缀，随后的命令名称要求使用小写字母。

GNU ARM汇编器汇编命令:ARM GNU汇编命令格式

**label: instruction or directive or pseudo-instruction
@comment**

- (1) label:标号字段。Linux ARM 汇编语言中，任何以冒号结尾的标识符都被认为是一个标号。
- (2) instruction or directive or pseudo-instruction: 指令或伪操作或伪指令字段。
由上一节介绍的ARM汇编**指令**或者用于GNU编译器编译过程的**伪指令**构成。
- (3) @comment: 是注释字段。“@”以后的所有字符在编译过程中均被认为是注释标识符，不参与编译过程。可以使用“@”或使用C语言风格的注释（/*.....*/）来代替分号“;”。

GNU ARM汇编器汇编命令:专有符号

“.”	用于定义标号
“@”	当前位置到行尾为注释字符。
“#”	整行注释字符。
“;”	新行分隔符。
“#” 或 “\$”	直接操作数前缀。
“.arm”	以arm格式编译，同code32
“.thumb”	以thumb格式编译，同code16
“.code16”	以thumb格式编译
“.code32”	以arm格式编译

常量编译控制伪操作：

伪操作	语法格式	作用
.byte	.byte expr {, expr} ...	分配一段字节内存单元，并用expr初始化。
.hword/.short	.hword expr {, expr} ...	分配一段半字内存单元，并用expr初始化。
.ascii	.ascii expr {, expr} ...	定义字符串 <i>expr</i> （非零结束符）。
.asciz /.string	.asciz expr {, expr} ...	定义字符串 <i>expr</i> （以/0为结束符）。
.float/.single	.float expr {, expr} ...	定义一个32bit IEEE 浮点数expr。
.double	.double expr {, expr} ...	定义64bit IEEE浮点数expr。
.word/.long /.int	.word expr {, expr} ...	分配一段字内存单元，并用expr初始化。
.fill	.fill repeat {, size}{, value}	分配一段内存单元，用size长度value填充repeat次。
.zero	.zero size	分配一段字节内存单元，并用0填充内存。
.space/.skip	.space size {, value}	分配一段字节内存单元，用value将内存单元初始化。

汇编程序代码控制伪操作

伪操作	语法格式	作用
<code>.section</code>	<code>.section expr</code>	定义域中包含的段。
<code>.text</code>	<code>.text {subsection}</code>	将操作符开始的代码编译到代码段或代码段子段。
<code>.data</code>	<code>.data {subsection}</code>	将操作符开始的数据编译到数据段或数据段子段。
<code>.bss</code>	<code>.bss {subsection}</code>	将变量存放到.bss段或.bss段的子段。
<code>.code 16/.thumb</code>	<code>.code 16</code> <code>.thumb</code>	表明当前汇编指令的指令集选择Thumb指令集。
<code>.code 32/.arm</code>	<code>.code 32</code> <code>.arm</code>	表明当前汇编指令的指令集选择ARM指令集。
<code>.end</code>	<code>.end</code>	标记汇编文件的结束行，即标号后的代码不作处理。
<code>.include</code>	<code>.include "filename"</code>	将一个源文件包含到当前源文件中。
<code>.align/.balign</code>	<code>.align {alignment} {, #max}</code>	通过添加填充字节使当前位置满足一定的对齐方式。

汇编程序代码控制伪操作

伪操作	语法格式	作用
<code>.equ</code>	<code>.equ symbol, expr</code>	定义常量表达式，定义符号为表达式值。
<code>.set</code>	<code>.set symbol, expr</code>	定义符号为表达式值。
<code>.equiv</code>	<code>.equiv symbol, exp</code>	<code>.equiv</code> 与 <code>.equ</code> 类似，不同之处在于如果 <code>symbol</code> 已定义，则会产生错误
<code>.global/globl</code>	<code>.global symbol</code>	声明全局符号（函数名），用于外部程序跳转或调用。
<code>.extern</code>	<code>.extern symbol</code>	声明外部符号（函数名），用于调用外部程序。

宏及条件编译控制伪操作

伪操作	语法格式	作用
<code>.macro</code> 、 <code>.exitm</code> 及 <code>.endm</code>	<code>.macro acroname</code> <code>{parameter {,</code> <code>parameter} ...}</code> <code>...</code> <code>.endm</code>	<code>.macro</code> 伪操作标识宏定义的开始， <code>.endm</code> 标识宏定义的结束。用 <code>.macro</code> 及 <code>.endm</code> 定义一段代码，称为宏定义体。 <code>.exitm</code> 伪操作用于提前退出宏。
<code>.ifdef</code> 、 <code>.else</code> 及 <code>.endif</code>	<code>.ifdef condition</code> <code>...</code> <code>.else</code> <code>...</code> <code>.endif</code>	当满足某条件时对一组语句进行编译，而当条件不满足时则编译另一组语句。其中 <code>else</code> 可以缺省。

例: Hello World!

```
.data

greeting:
    .asciz "Hello World"

.balign 4
return: .word 0

.text

.global main
main:
    ldr r1, addr_of_return
    str lr, [r1]

    ldr r0, addr_of_greeting

    bl puts

    ldr r1, addr_of_return
    ldr lr, [r1]
    bx lr

addr_of_greeting: .word greeting
addr_of_return: .word return

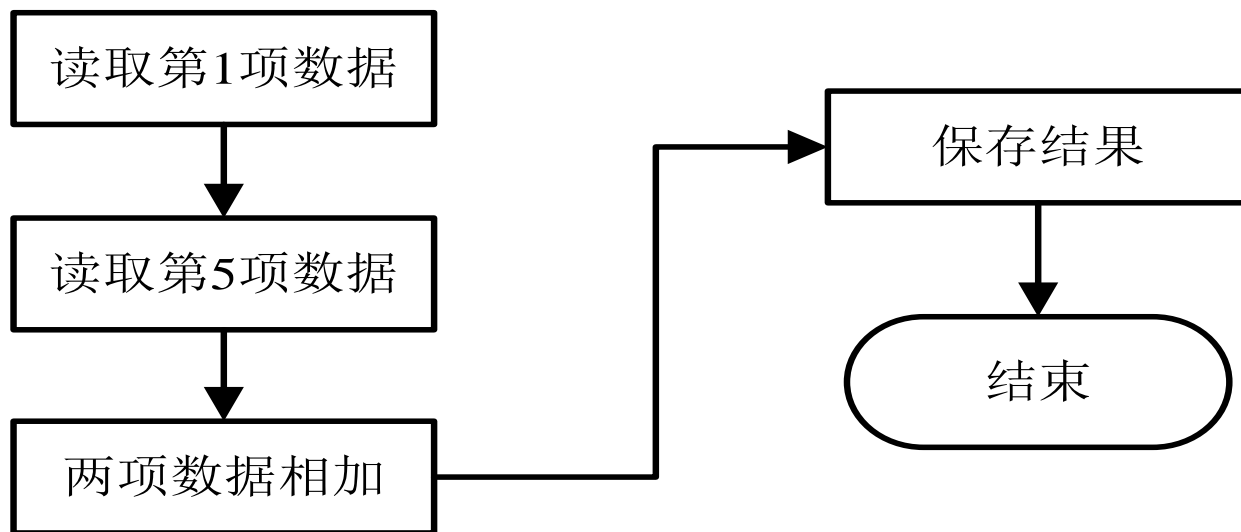
.global puts
```

汇编语言程序设计及举例

1、顺序程序设计

最简单的程序是没有分支、没有循环的顺序运行程序。

例：通过查表操作实现数组中的第1项数据和第5项数据相加，结果保存到数组第9项中。



程序清单

```
        .data                @定义数据段
array:  .word                0x11, 0x22, 0x33, 0x44 @定义12个字的数组
        .word                0x55, 0x66, 0x77, 0x88
        .word                0x00, 0x00, 0x00, 0x00

        .text
        .globl               main
main:    ldr                  r0, =array          @取得数组Array首地址
        ldr                  r2, [r0]           @装载数组第1项字数据给R2
        mov                  r1, #4
        ldr                  r3, [r0, r1, lsl #2] @装载数组第5项字数据给R3
        add                  r2, r2, r3         @R2 + R3→R2
        mov                  r1, #8             @R1=8
        str                  r2, [r0, r1, lsl #2] @保存结果到数组第9项
        .end
```

2、分支程序设计

例. 编写汇编程序实现C语言if,else分支程序, C语言程序如下:

```
static int a = 10;
static int b = 4;
static int x;

int main()
{
    if ( a < b ) x = 1;
    else
        x = 0;
    .
    .
    .
}
```


If-Then-Else（条件执行）

```
1      .data
2 a:    .word 10    @ static int a=10;
3 b:    .word 4     @ static int b=4;
4 x:    .word 0     @ static int x;
5      .text
6      .globl main
7 main: ldr      r0, =a      @ load pointer to 'a'
8       ldr      r1, =b      @ load pointer to 'b'
9       ldr      r0, [r0]    @ load 'a'
10      ldr      r1, [r1]    @ load 'b'
11      cmp      r0, r1      @ compare them
12      movlt    r0, #1      @ THEN section - load 1 into r0
13      movge    r0, #0      @ ELSE section - load 0 into r0
14      ldr      r1, =x      @ load pointer to 'x'
15      str      r0, [r1]    @ store r0 in 'x'
```

If-Then-Else (分支指令)

```
1      .data
2  a:   .word 10    @ static int a=10;
3  b:   .word 4     @ static int b=4;
4  x:   .word 0     @ static int x;
5
6      .text
7      .globl main
8  main: ldr      r0, =a        @ load address of 'a'
9        ldr      r1, =b        @ load address of 'b'
10       ldr      r0, [r0]      @ load 'a'
11       ldr      r1, [r1]      @ load 'b'
12       cmp      r0, r1        @ compare them
13       bge      else         @ if a >= b then goto else_code
14       mov      r0, #1        @ THEN section - load 1 into r0
15       b        after        @ skip the else section
16  else: mov      r0, #0        @ ELSE section - load 0 into r0
17  after: ldr      r1, =x        @ load pointer to 'x'
        str      r0, [r1]      @ store r0 in 'x'
```

3、循环程序设计

循环结构由以下两部分组成：

1、循环体：

要求重复执行的程序段部分；

2、循环结束条件：

在循环程序中必须给出循环结束条件，否则程序就会进入死循环。

例：编写汇编程序实现计数循环。

计数循环用C语言表达：

```
for(i=0; i<10; i++ )    x++ ;
```

汇编语言程序：

```

    ...
    mov     r0, #0           @初始化r0=0
    mov     r2, #0           @设置r2=0; r2控制循环次数
for:  cmp     r2, #10         @判断r2<10?
     bhs    for_e            @若条件失败，退出循环
     add     r0, r0, #1       @循环体，r0=r0 + 1
     add     r2, r2, #1       @r2 + 1
     b      for
for_e:  ...
```

例：汇编程序实现条件循环。

条件循环用C语言表达如下：

```
while( x<= y)  x* = 2;
```

汇编语言程序：

```
...
mov    r0, #1           @初始化r0 = 1
mov    r1, #20          @初始化r1 = 20
b      w_2              @首先要判断条件
w_1:   mov    r0, r0, lsl #1 @循环体,  $r0 * 2$ 
w_2:   cmp    r0, r1      @判断 $r0 \leq r1$ ? , 即 $x \leq y$ ?
      bls    w_1         @若条件正确, 继续循环
w_end:
...
```

例：汇编程序实现条件循环输出字符串。

```
int main()
{
    int i; i = 0;
    while(i<10)
    {
        printf("Hello World - %d\n", i);
        i++;
    }
    return 0;
}
```

汇编实现循环语句

```
1      .data
2  str:  .asciz "Hello World - %d\n"
3
4      .text
5      .globl main
6  main: @ We are going to use r4 and make a function call, so
7         stmfd sp!, {r4, lr} @ push lr and r4 onto stack
8         mov     r4, #0      @ use r4 for i; i=0
9  loop: cmp     r4, #10      @ perform comparison
10        bge     done        @ end loop if i >= 10
11        ldr     r0, =str     @ load pointer to format string
12        mov     r1, r4       @ copy i into r1
13        bl      printf      @ printf("Hello World - %d\n", i);
14        add     r4, r4, #1   @ i++
15        b       loop        @ repeat loop test
16  done: mov     r0, #0       @ move return code into r0
17        ldmfdd sp!, {r4, lr} @ pop lr and r4 from stack
18        mov     pc, lr      @ return from main
19
20      .end
```

调用标准C库函数

```
1      .data
2  str1: .asciz  "%d"
3  str2: .asciz  "You entered %d\n"
4  n:    .word   0
5
6      .text
7      .globl  main
8  main: stmfd   sp!, {lr}      @ push link register onto stack
9        ldr     r0, =str1      @ load address of format string
10       ldr     r1, =n          @ load address of int variable
11       bl      scanf          @ call scanf("%d",&n)
12       ldr     r0, =str2      @ load address of format string
13       ldr     r1, =n          @ load address of int variable
14       ldr     r1, [r1]        @ load int variable
15       bl      printf          @ call printf("You entered %d\n",n)
16       mov     r0, #0          @ load return value
17       ldmfd   sp!, {lr}      @ pop link register from stack
18       mov     pc, lr         @ return from main
```

例：编写循环语句实现数据块复制。

ldr	r0, =data_dst	@指向数据目标地址
ldr	r1, =data_src	@指向数据源地址
mov	r10, #20	@复制数据个数 $20 \times n$ 个字
		@n为ldm指令操作数据个数
loop:	ldmia r1!, {r2-r9}	@从数据源读取8个字到r2~r9
	stmia r0!, {r2-r9}	@将r2~r9的数据保存到目标地址
	subs r10, r10, #1	@r10-1
	bne loop	

3.6

Part Six

C语言与汇编语言的混合编程

在嵌入式开发中，C语言是一种常见的程序设计语言。C语言程序可读性强，易维护，可移植性和可靠性高。ARM体系结构不仅支持汇编语言也支持C语言，在一些情况下需要采用汇编语言和C语言混合编程。

3.6.1 C程序中内嵌汇编

在C语言程序中嵌入汇编可以完成一些C语言不能完成的操作，同时代码效率也比较高。

在ARM C语言程序中使用关键词__asm来标识一段汇编指令代码，格式为：

```
__asm                                //asm前2个下划线
{
    instruction [; instruction]
    ...
    [instruction]
}
```

如果一行有多条汇编指令，指令间用“；”隔开；如果一条指令占多行，要使用续行符号“\”。

在ARM C语言程序中也可以使用关键词asm来标识一段汇编指令代码，格式为：

```
asm("instruction [; instruction]");
```

GNU C程序中内嵌汇编

在ARM C语言程序中使用关键词__asm来标识一段汇编指令代码，格式为：

1. 基本内联形式

```
__asm [volatile] (code);
```

例：

```
__asm volatile ("mov r0, r0");
```

1. 扩展内联形式

```
__asm [volatile] (code_template  
:outputs  
[:inputs  
[:clobber_list]]  
);
```

GNU C程序中内嵌汇编示例

```
#include <stdio.h>
int add(int i, int j)
{
    int res = 0;
    __asm ("ADD %[result], %[input_i], %[input_j]"
        : [result] "=r" (res)
        : [input_i] "r" (i), [input_j] "r" (j)
        );
    return res;
}
int main(void)
{
    int a = 1;
    int b = 2;
    int c = 0;
    c = add(a,b);
    printf("Result of %d + %d = %d\n", a, b, c);
}
```

- **code**

汇编指令，例如"ADD R0, R1, R2".

- **code_template**

- 汇编指令的模板，例如"ADD %[result], %[input_i], %[input_j]".

- **outputs**

输出操作数列表，以逗号分隔。每个操作数由方括号中的符号名称、约束字符串和小括号中的C语言表达式。在此示例中，有一个输出操作数：**[result]**
"=r" (res);

输出操作数列表也可以为空。例如：

```
__asm ("ADD R0, %[input_i], %[input_j] "
```

```
: /* 输出操作数为空 */
```

```
: [input_i] "r" (i), [input_j] "r" (j)
```

```
);
```

- **inputs**

输入操作数的可选列表，以逗号分隔。输入操作数使用与输出操作数相同。在此示例中，有两个输入操作数：**[input_i] "r" (i), [input_j] "r" (j)**.输入操作数列表也可以为空。

- **clobber_list** 寄存器破坏描述符

以逗号分隔的字符串列表。每个字符串都是汇编代码可能修改的寄存器的名称。寄存器破坏描述符通知编译器我们使用了哪些寄存器（程序员自己知道内嵌汇编代码中使用了哪些寄存器），否则对这些寄存器的使用就有可能导致错误，修改描述部分可以起到这种作用。

例如，如果一个寄存器包含一个临时值，则将其包含在 **clobber** 列表中。编译器避免使用此列表中的寄存器作为输入或输出操作数，或者在执行汇编代码时使用它来存储另一个值。

该列表可以为空。除了寄存器，列表还可以包含特殊参数：

"cc "

该指令修改条件代码标志。

"memory "

该指令访问未知的内存地址。

其中的寄存器 **clobber_list** 必须使用小写字母而不是大写字母。

示例：

```
__asm ("ADD R0, %[input_i], %[input_j] "  
: /*输出操作数为空 */  
: [input_i] "r" (i), [input_j] "r" (j)  
: "r5", "r6", "cc", "memory "  
);
```

例子2

```
#include <stdio.h>
int main(void)
{
    int result ,value;
    value=1;
    printf("old value is %x",value);
    __asm("mov %0,%1,ror #1": "=r"(result):"r"(value));
    printf("new result is %x\n",result)    ;

    return 1;
}
```


3.6.2 汇编中访问C语言程序变量

在C语言中声明的全局变量可以被汇编语言通过地址间接访问。

例如，在C语言程序中已经声明了一个全局变量glovbvar，通过IMPORT伪指令声明外部变量的方式访问：

```
AREA  EXAMPLE2, CODE, REDAONLY
IMPORT      glovbvar          ;声明外部变量glovbvar
START
    LDR     R1, =glovbvar      ;装载外部变量地址
    LDR     R0, [R1]           ;读出全局变量glovbvar数据
    ADD     R0, R0, #1
    STR     R0,[R1]            ;保存变量值
    MOV     PC, LR
END
```

3.6.3 ARM中的汇编和C语言相互调用

为了使单独编译的C语言程序和汇编语言程序之间能够相互调用，必须遵守ATPCS规则。ATPCS即ARM-THUMB procedure call standard（ARM-Thumb过程调用标准）的简称。

ATPCS规定了应用程序的函数可以如何分开地写，分开地编译，最后将它们连接在一起，所以它实际上定义了一套有关过程（函数）调用者与被调用者之间的协议。基本ATPCS规定了在子程序调用时的一些基本规则，包括3个方面的内容：

- 各寄存器的使用规则机器相应的名称。
- 数据栈的使用规则。
- 参数传递的规则。

1. C程序调用汇编程序

C程序调用汇编程序首先通过**extern**声明要调用的汇编程序模块，声明中形参个数要与汇编程序模块中需要的变量个数一致，且参数传递要满足**ATPCS**规则，然后在C程序中调用。

示例：

```
#include<stdio.h>
extern void strcpy(char *d, char *s); //使用关键词声明
int main()
{
    char *srcstr = "first";
    char dststr[20] = "second";
    strcpy(dststr, srcstr);           //汇编模块调用;
}
```

被调用的汇编程序：

```
.text
.globl strcpy                //使用globl伪操作声明本汇编程序
strcpy:
    ldrb    r2, [r1], #1
    strb    r2, [r0], #1
    cmp     r2, #0
    bne     strcpy
    mov     pc, lr
.end
```

2. 汇编程序调用C程序

在调用之前必须根据C语言模块中需要的参数个数，以及ATPCS参数规则，完成参数传递，即前四个参数通过R0~R3传递，后面的参数通过堆栈传递，然后再利用B、BL指令调用。子程序结果返回规则：1.结果为一个32位的整数时,可以通过寄存器R0返回；2.结果为一个64位整数时,可以通过R0和R1返回，依此类推。

```
int g(int a,int b,int c,int d,int e)      //C语言函数原型
```

```
{  
    return (a+b+c+d+e);  
}
```

//汇编程序调用C程序g()计算1+2+3+4+5的结果；

```
.text
```

```
.extern g
```

```
.globl main
```

main:

```
    str lr,[sp,#-4]!    @保存返回地址
```

```
    mov r0,#1           @设置参数1
```

```
    mov r1,#2           @设置参数2
```

```
    mov r2,#3           @设置参数3
```

```
    mov r3,#4           @设置参数4
```

```
    mov r4,#5           @参数5通过数据栈传递
```

```
    str r4, [sp,#-4]!
```

```
    bl g                @调用c程序g(),其结果从r0返回
```

```
    add sp,sp,#4        @调整数据栈指针,准备返回
```

```
    ldr pc,[sp],#4      @返回
```

```
end
```

3.7

Part Seven

本章小结

本章首先对**ARM**处理器指令的九种寻址方式进行了说明，并详细介绍了**ARM**指令集中各种指令的格式、功能和使用方法，简单介绍了**16**位的**Thumb**指令。随后介绍了**ARM**汇编语言的伪操作、伪指令和汇编语句格式，通过示例讲述了汇编语言程序的结构。最后讲述了**C**语言和汇编语言混合编程的规则和方法。通过本章，读者了解到**ARM**程序设计的基本知识，为基于**ARM**处理器的嵌入式软件开发奠定基础。