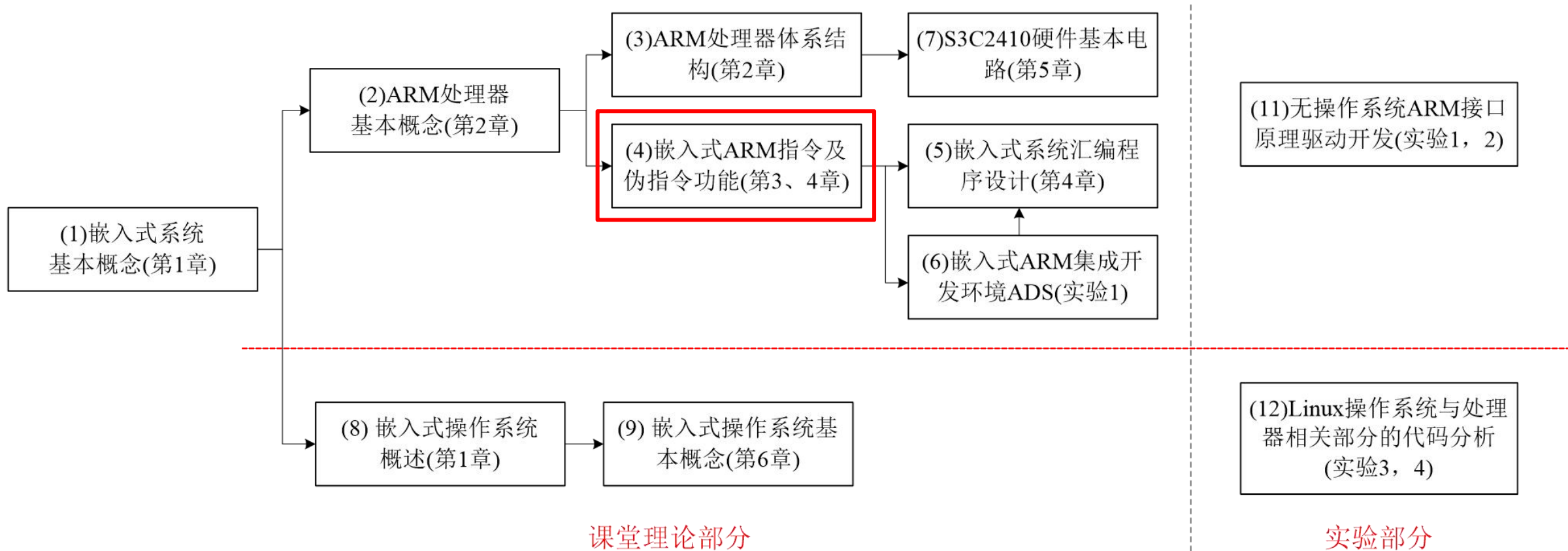
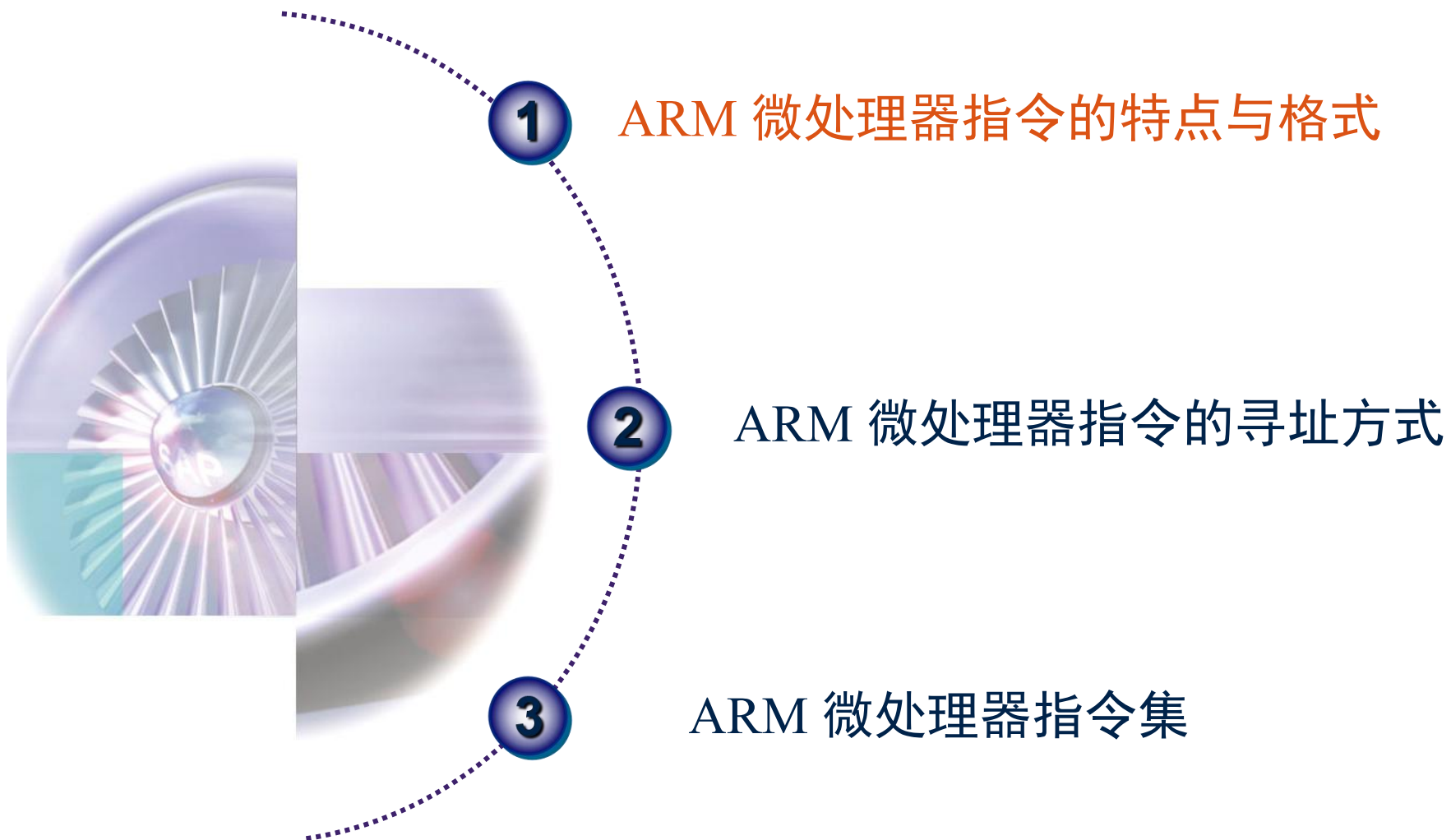




第三章 ARM指令集





① ARM指令都是32位的RISC

- 大多数指令执行都在单周期内完成；
- 所有指令都可以条件执行；

② ARM指令内存访问使用加载/存储（Load/Store）架构

③ ARM指令有9种寻址方式

- 立即寻址、寄存器寻址、寄存器移位寻址、寄存器间接寻址、基址寻址、堆栈寻址、多寄存器寻址、块复制寻址和相对寻址。

④ ARM指令分为5大类

- 数据处理指令、存储器访问指令、跳转指令、程序状态寄存器处理指令、协处理器指令和杂项指令

- ARM指令(数据处理指令类)语法格式如下：

< Opcode > <Cond> {S} Rd, Rn, {Operand2}

其中：“<>”内为必须项，“{ }”内为可选项

- 说明：

- Opcode：指令操作符
- Cond：指令执行的条件
- S：决定指令的操作是否影响CPSR的值，有S就影响CPSR，否则不影响
- Rd：目标寄存器
- Rn：第一操作数的寄存器
- Operand2：第二操作数

例如： **SUBNES R1,R1,#0x1D**

操作码	条件助记符	标志	含义
0000	EQ	Z=1	运算结果为0
0001	NE	Z=0	运算结果不为0
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)

C代码

```
if (r0 == 0)
{
    r1 = r1 + 1;
}
else
{
    r2 = r2 + 1;
}
```

ARM汇编指令
非条件执行

```
CMP r0, #0
BNE else
ADD r1, r1, #1
B end
else
    ADD r2, r2, #1
end
...
```

- 5条指令，占5个字空间
- 执行需要5~6个时钟周期

ARM汇编指令
条件执行

```
CMP r0, #0
ADDEQ r1, r1, #1
ADDNE r2, r2, #1
...
```

- 3条指令，占3个字空间
- 执行需要3个时钟周期

< Opcode > <cond> { S } Rd, Rn, {Operand2}

ARM指令采用固定的32位二进制编码，如下：

31	28	27	25	24	21	20	19	16	15	12	11	8	7	0
Cond		00I		Opcode		S	Rn		Rd		Shifter_operand			

Cond: 指令执行的条件编码

Opcode: 指令操作符编码

S: 决定指令的操作是否影响CPSR的值

Rd: 操作目标寄存器编码

Rn: 包含第一操作数的寄存器编码

Shifter_operand: 表示第二操作数，可寄存器或立即数

MOV R1, R0

31 28 27 25 24 21 20 19 16 15 12 11 8 7 0

Cond	00I	Opcode	S	Rn	Rd	Shifter_operand
1110	000	1101	0	0001	0000	000000000000

Cond: 1110

Opcode: 1101

I: 0

S: 0

Rd: 0001

Rn: 0000

Shifter_operand: 000000000000

MOV R1, #0x34

31 28 27 25 24 21 20 19 16 15 12 11 8 7 0

Cond	00I	Opcode	S	Rn	Rd	Shifter_operand
1110	001	1101	0	0001	0000	000000110100

Cond: 1110

Opcode: 1101

I: 1

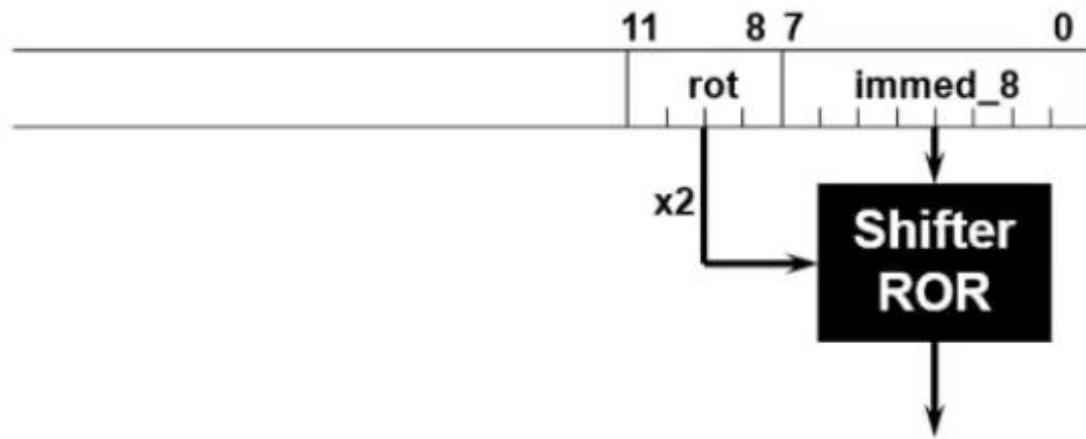
S: 0

Rd: 0001

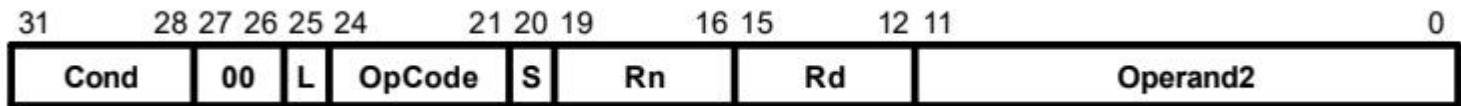
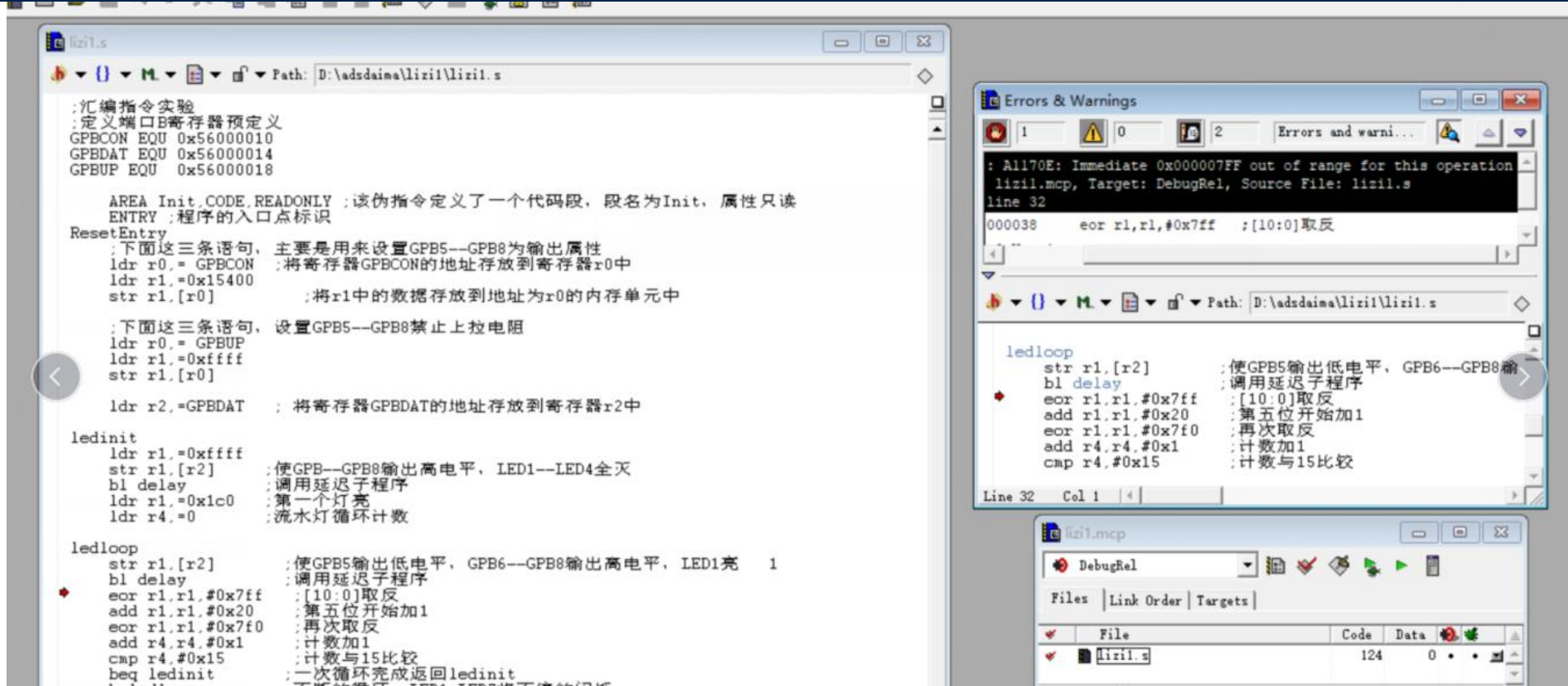
Rn: 总是0

Shifter_operand: 000000110100

- ARM指令32位定长的，在数据处理指令格式中，第二操作数有12位来对应；



- ① 范围在0-255的8位立即数，或者8位立即数循环右移偶数位得到的数；
 - ② 4位的移位数乘以2组成了步长，范围在0-30的移位值；
- 若机器码为0xe3a004ff，则其对应的指令为MOV r0, #_____

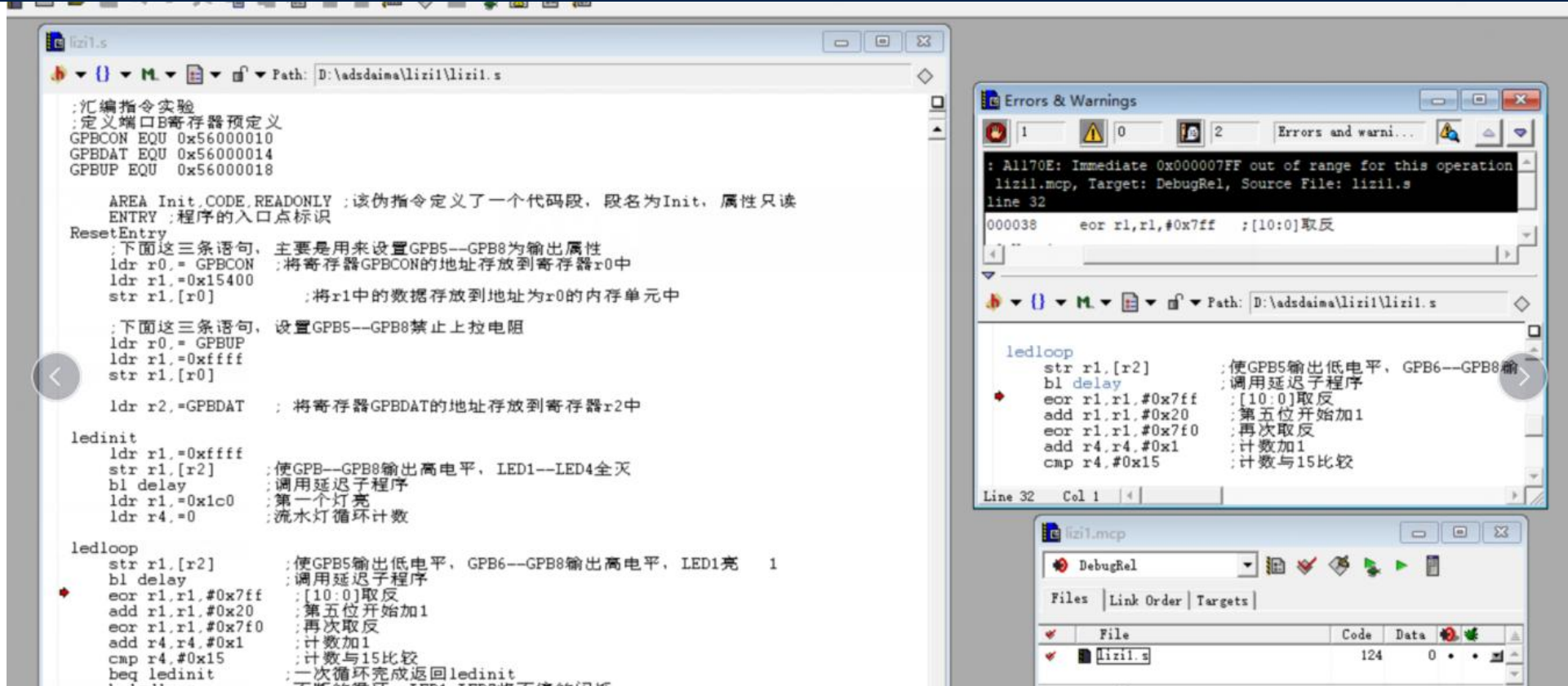


[24:21] Operation codes

0000 = AND-Rd: = Op1 AND Op2

0001 = **EOR**-Rd: = Op1 **EOR** Op2

0010 = SUB-Rd: = Op1-Op2



17:43:22

```

bl delay ;调用延迟子程序
eor r1,r1,#0x7ff ;[10:0]取反

```

17:43:28

这个取反

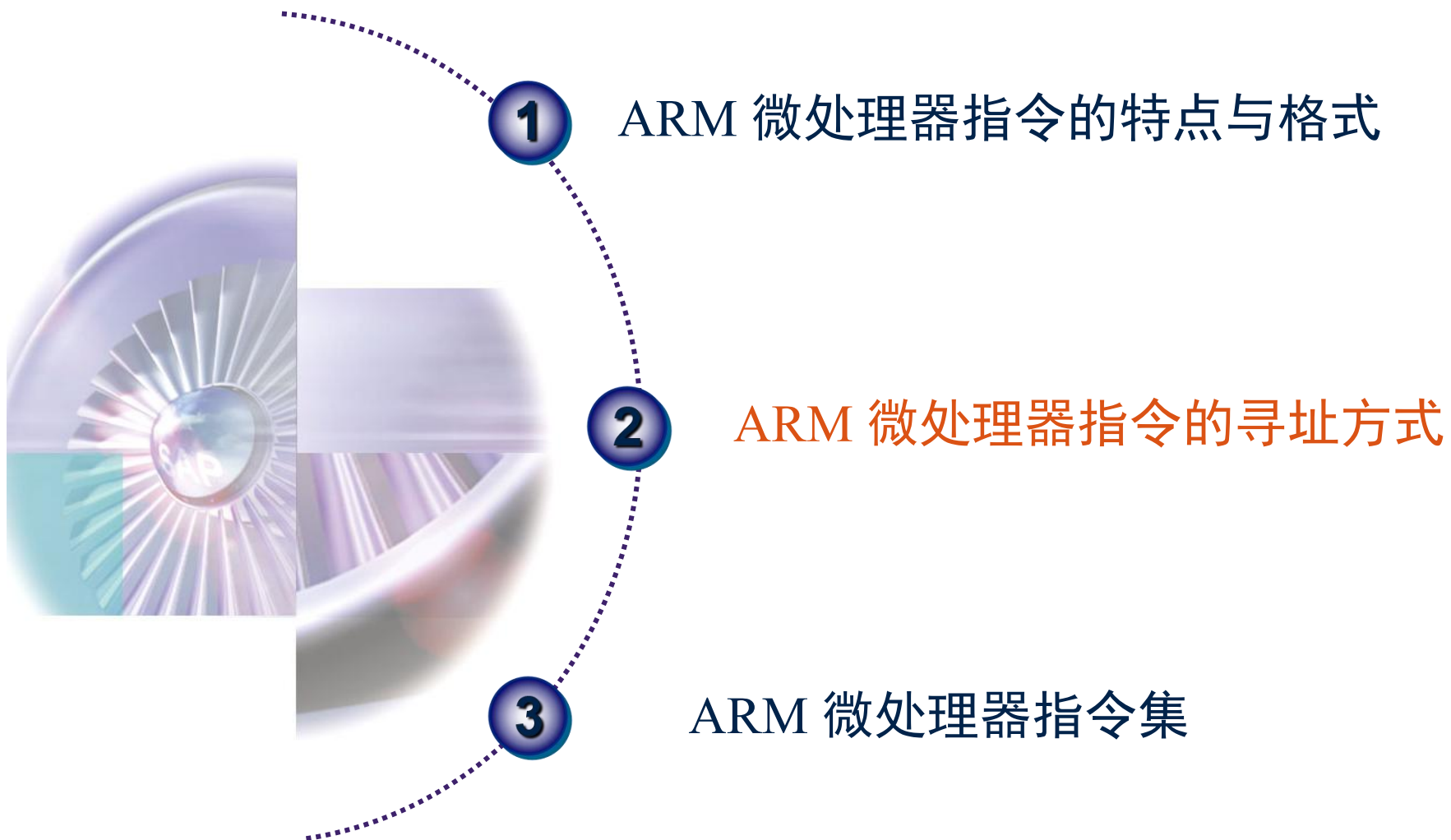
17:43:42

为啥会报Out of range的错

17:43:48

```
eor r1,r1,#0x7f0 ;再次取反
```

这个取反就没有



ARM指令系统支持如下几种常见的9种寻址方式：

- 立即寻址
- 寄存器寻址
- 寄存器移位寻址
- 寄存器间接寻址
- 基址变址寻址
- 多寄存器寻址
- 块复制寻址
- 堆栈寻址
- 相对寻址

■ 操作数本身就在指令中给出

ADD R0, R0, #1	/*R0←R0+1*/
ADD R0, R0, #0x3f	/*R0←R0+0x3f*/

Note:

- ✓ 立即数要求以 “#” 为前缀
- ✓ 十六进制表示的立即数，还要求在 “#” 后加上 “0x” 或 “&”。#0x3f
- ✓ 十进制表示的立即数，还要求在 “#” 后加上 “0d” 或省略。#0d35, #35
- ✓ 二进制表示的立即数，还要求在 “#” 后加上 “0b”。#0b1001

- 寄存器寻址就是利用寄存器中的数值作为操作数

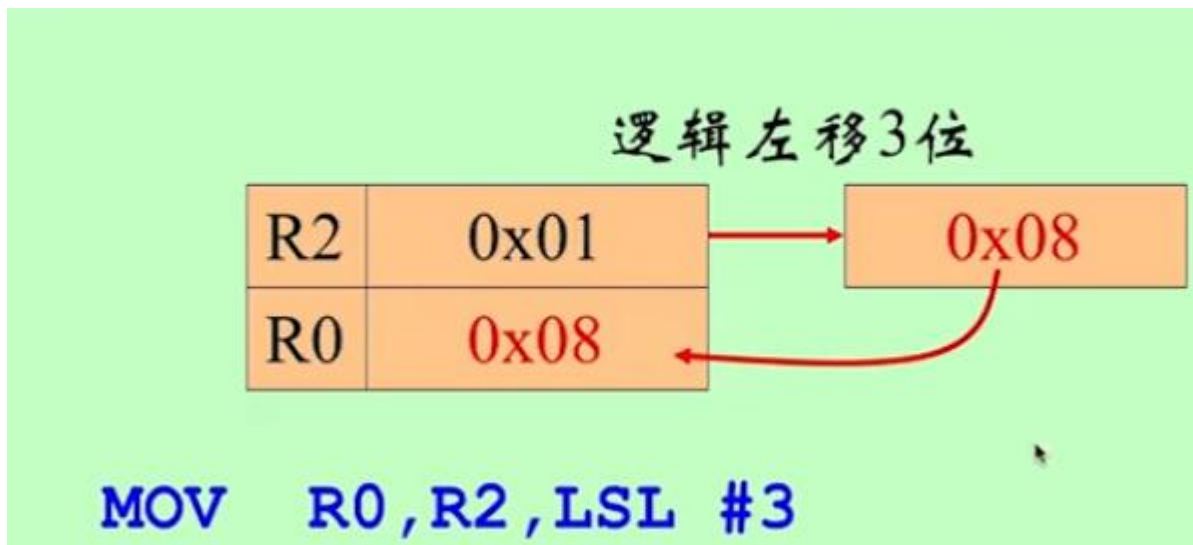
```
ADD    R0, R1, R2    /*R0←R1+R2*/
```

- 第二个操作数寄存器在与第一个操作数进行操作之前，先进行移位操作。如：

`ADD R3, R2, R1, LSL #2` ; $R3 \leftarrow R2 + R1 * 4$

`ADD R3, R2, R1, LSL R4` ; $R3 \leftarrow R2 + (R1 \text{左移} R4 \text{位})$

Note: R1左移一个偏移量后与R2相加，结果放R3



LSL : Logical Left Shift**LSR : Logical Shift Right**

- 1、移位的位数<32
- 2、可用立即数或寄存器方式给出

ASR: Arithmetic Right Shift**ROR: Rotate Right****RRX: Rotate Right Extended**

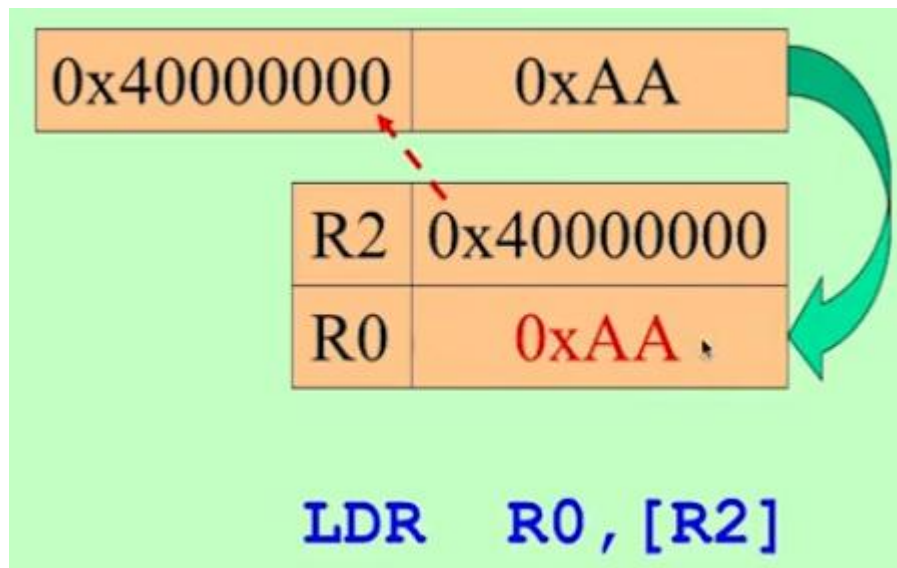
- 寄存器间接寻址就是以寄存器中的值作为操作数的地址，而操作数本身存放在存储器中。

LDR R0, [R1]

/*R0←[R1]*/

STR R0, [R1]

/*[R1]←R0*/

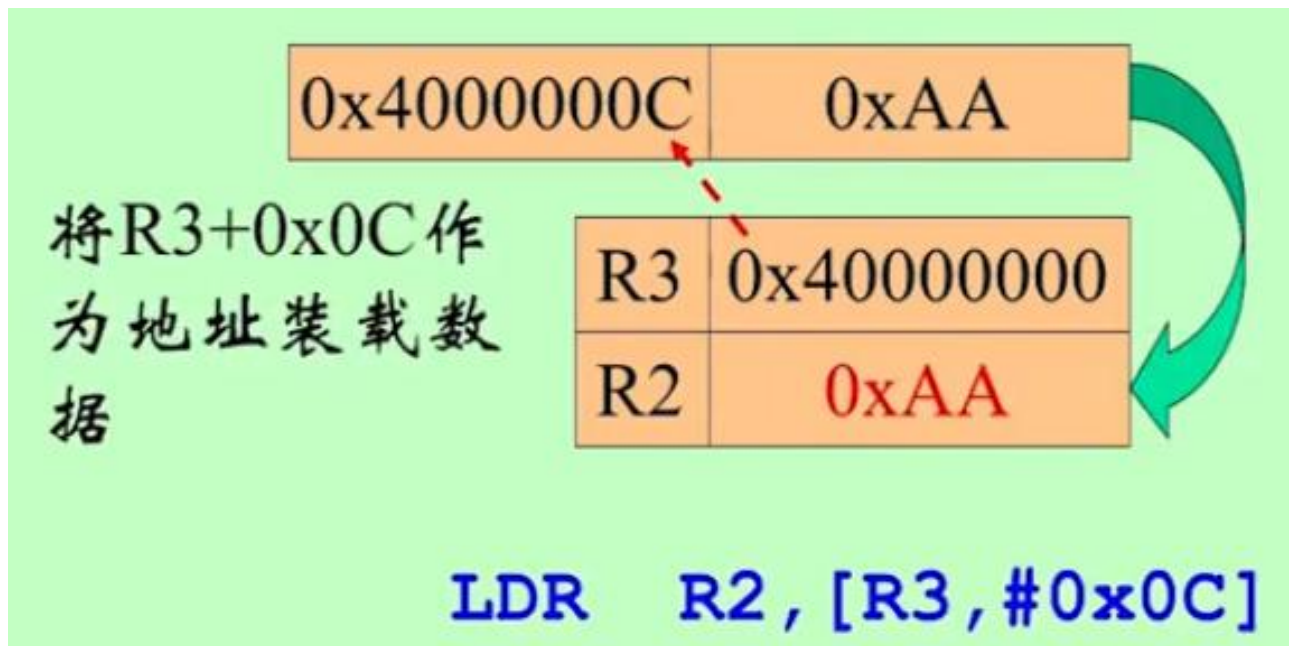


- 基址变址寻址就是将寄存器(该寄存器一般称作**基址寄存器**)的内容与指令中给出的**地址偏移量**相加, 从而得到一个操作数的有效地址。

LDR R0, [R1, # 4]	; R0←[R1+4] ; 前索引偏移, 基址不变
LDR R0, [R1, # 4]!	; R0←[R1+4]、R1←R1+4 ; 带写回的前索引偏移基址自动更新
LDR R0, [R1], # 4	; R0←[R1]、R1←R1+4 ; 后索引偏移, 基址后变
LDR R0, [R1, R2]	; R0←[R1+R2] ; 前索引偏移

Note: R1为基址寄存器

- 基址变址寻址就是将寄存器（该寄存器一般称作**基址寄存器**）的内容与指令中给出的**地址偏移量**相加，从而得到一个操作数的有效地址。



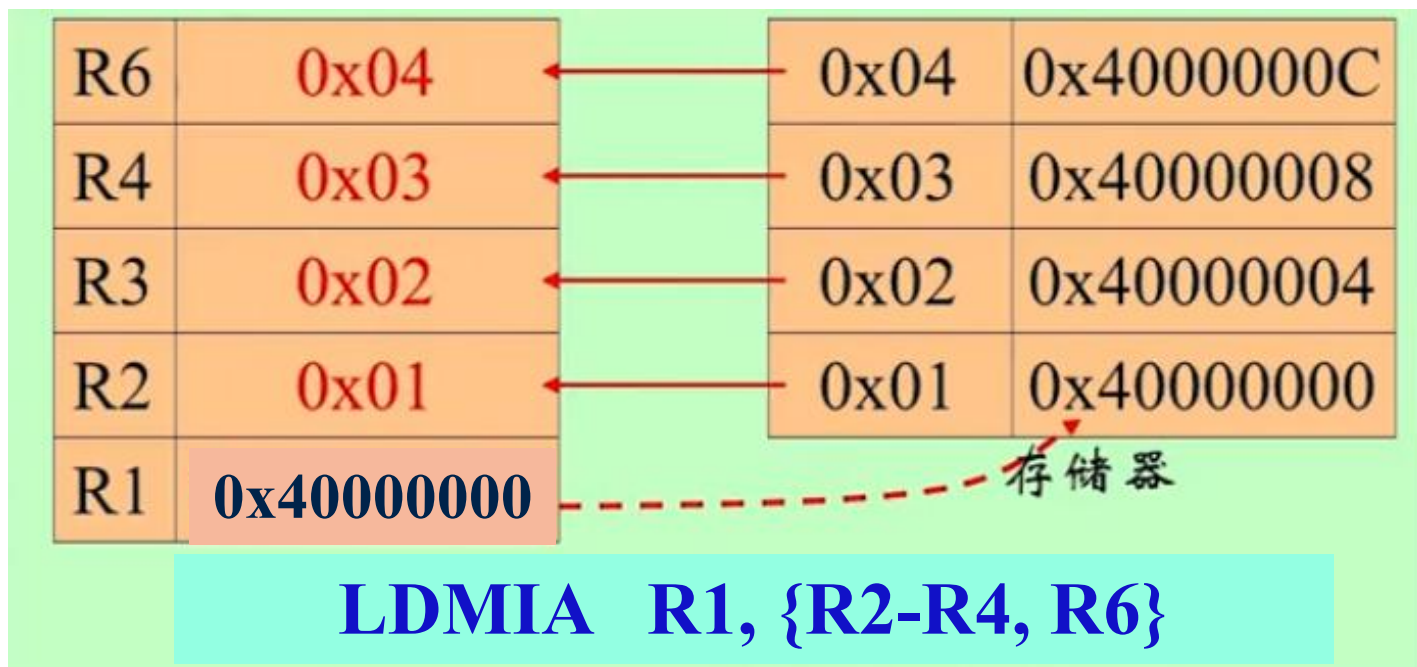
批量传输指令

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

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

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

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




```
STMIA R0 , {R3-R6, R10} ; [R0] ← R3  
                          ; [R0+4] ← R4  
                          ; [R0+8] ← R5  
                          ; [R0+12] ← R6  
                          ; [R0+16] ← R10
```

Note: 将R3~R6, R10中数据存到R0指向的地址处的内存单元, **R0不更新**。

```
STMIA R0!, {R3-R6, R10} ; [R0] ← R3 , R0 ← R0+4  
                        ; [R0] ← R4 , R0 ← R0+4  
                        ; [R0] ← R5 , R0 ← R0+4  
                        ; [R0] ← R6 , R0 ← R0+4  
                        ; [R0] ← R10 , R0 ← R0+4
```

Note: 将R3~R6, R10中数据存到R0指向的地址处的内存单元, **R0更新**。

批量传输指令

- 把存储器中的一个数据块加载到多个寄存器中，或者把多个寄存器的值保存到内存块中。**即多寄存器寻址。**

STMLA R0!, {R1-R7} ; 将R1~R7的数据保存到存储器中

STMIB R0!, {R1-R7} ;

Note:该指令的**后缀IB**表示在每次执行加载/存储操作**之前**，R0按字长度增加，因此，指令可将连续存储单元的值传送到R1~R7。

- 堆栈寻址是一种**特殊的**多寄存器寻址，使用一个专门的寄存器（**堆栈指针SP**）指向一块存储区域（**堆栈**）。
- ✓ **堆栈指针**指向最后压入堆栈的数据时，称为**满堆栈**（Full Stack）
- ✓ **堆栈指针**指向下一个将要放入数据的空位置时，称为**空堆栈**（Empty Stack）。

- 堆栈寻址是隐含的，使用一个专门的寄存器（**堆栈指针SP**）指向一块存储区域（**堆栈**）。

- ✓ 当堆栈由**低地址向高地址生成**时，称为**递增堆栈(Ascending)**。
- ✓ 当堆栈由**高地址向低地址生成**时，称为**递减堆栈(Descending)**。

- 堆栈寻址是隐含的，使用一个专门的寄存器（**堆栈指针SP**）指向一块存储区域（**堆栈**）。
- ✓ **堆栈指针**指向最后压入堆栈的数据时，称为**满堆栈**（Full Stack）
- ✓ **堆栈指针**指向下一个将要放入数据的空位置时，称为**空堆栈**(Empty Stack)。
- ✓ 当堆栈由**低地址向高地址生成**时，称为**递增堆栈**(Ascending)。
- ✓ 当堆栈由**高地址向低地址生成**时，称为**递减堆栈**(Descending)。

■ 这样就有四种类型的堆栈工作方式

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

STMFD SP! , {R1-R7, LR} ; 将LR, R7-R1 入栈, 满递减堆栈
LDMFD SP! , {R1-R7, LR} ; 数据出栈到 R1-R7, LR寄存器, 满
; 递减堆栈

■ ARM堆栈操作通过块传送指令来完成:

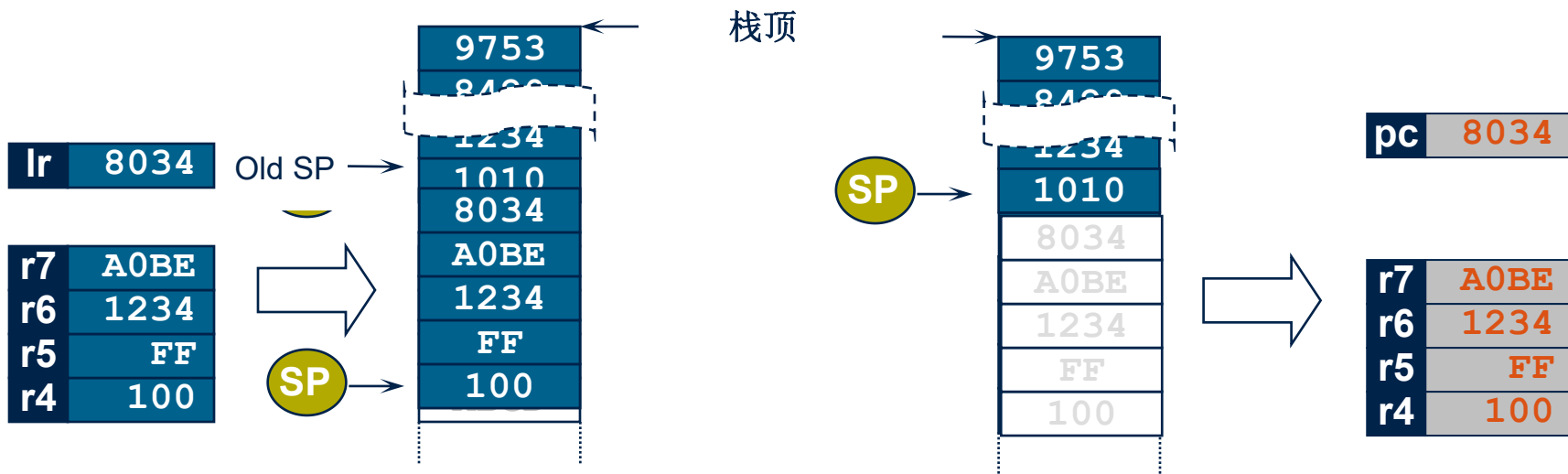
- **STMFD** (Push) 块存储- **F**ull **D**escending stack [STMDB]
- **LDMFD** (Pop) 块装载- **F**ull **D**escending stack [LDMIA]

入栈：先减址，再顺序入栈数据

出栈：先弹出数据，后增栈址

STMFD sp!,{r4-r7,lr}

LDMFD sp!,{r4-r7,pc}



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

BL **NEXT** ; 跳转到子程序NEXT处执行

.....

NEXT

.....

MOV **PC, LR** ; 从子程序返回

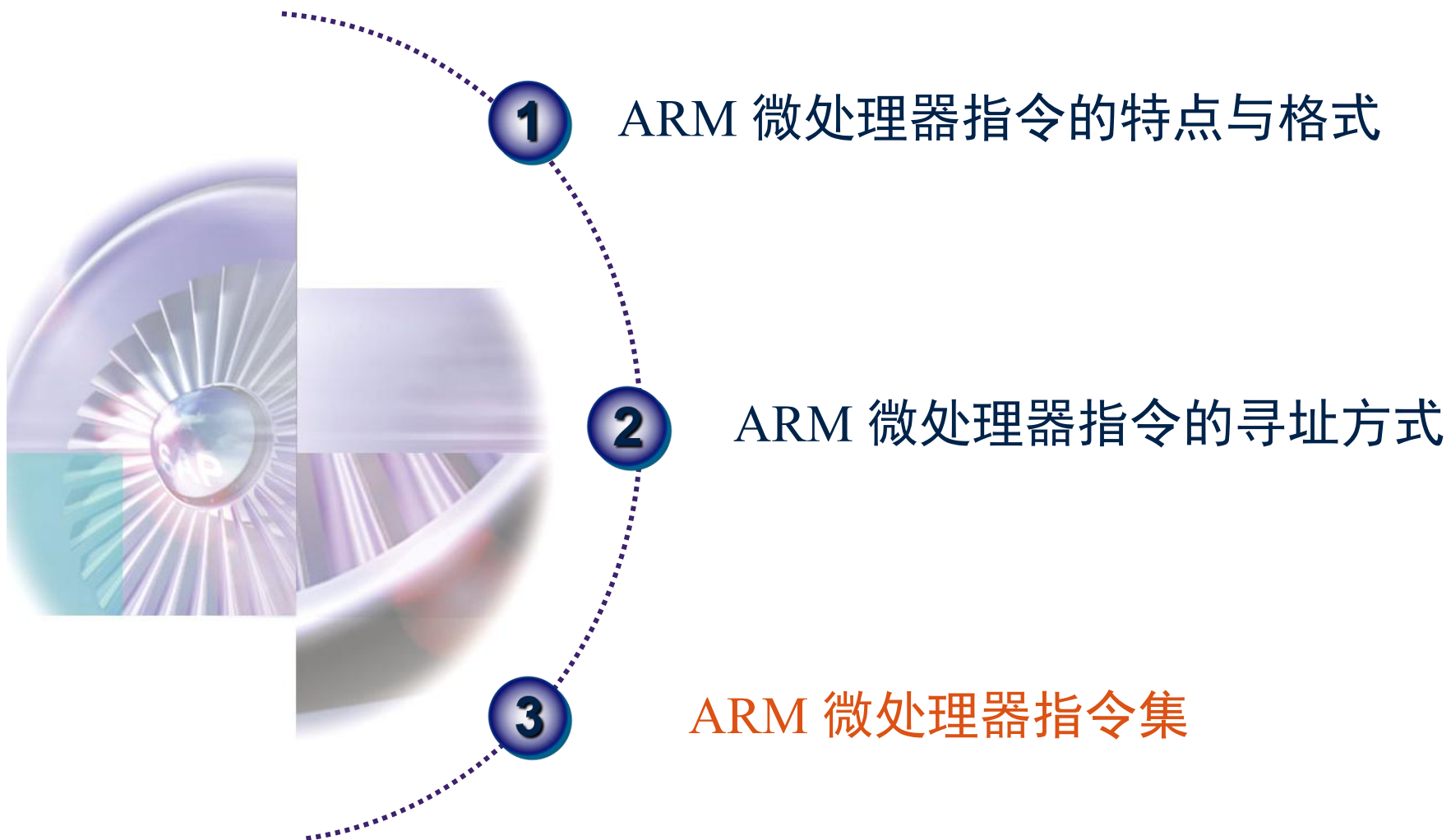
Note: 以上程序段完成子程序的调用和返回，跳转指令BL采用了相对寻址方式。

ARM指令系统支持如下几种常见的9种寻址方式：

- 立即寻址
- 寄存器寻址
- 寄存器移位寻址
- 寄存器间接寻址
- 基址变址寻址
- 多寄存器寻址
- 块复制寻址
- 堆栈寻址
- 相对寻址

寄存器相关

存储器相关



ARM微处理器的指令集可以分为五大类

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

1. 跳转指令
2. 数据处理指令
3. 乘法指令与乘加指令
4. 程序状态寄存器访问指令
5. 加载/存储指令
6. 批量数据加载/存储指令
7. 数据交换指令
8. 移位指令（操作）
9. 协处理指令
10. 异常产生指令

ARM微处理器的指令集可以分为五大类

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

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的地址空间的跳转，包括以下4条指令：

- ✓ **B** 跳转指令
- ✓ **BL** 带返回的跳转指令
- ✓ **BLX** 带返回和状态切换的跳转指令
- ✓ **BX** 带状态切换的跳转指令

B forward

ADD r1, r2, #4

ADD r0, r6, #2

ADD r3, r7, #4

forward

SUB r1, r2, #4

backward

ADD r1, r2, #4

SUB r1, r2, #4

ADD r4, r6, r7

B backward

(2) BL——带链接的转移指令

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←下一条指令地址
CMP R1, #5        ; 转至子程序SUB1处
...
SUB1 ...
MOV PC, LR        ; 子程序返回
```

Note：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。

(3) BX——带状态切换的转移指令

指令格式如下：

31	28	27				20	19				8	7			4	3	0
cond		0 0 0 1 0 0 1 0					全为1					0 0 0 1		Rm			

BX{cond} Rm

BX指令跳转到Rm指定的地址执行程序。

Note:

- 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**。

(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)

Note: 转移地址限制在当前指令的±32MB的范围内。

ARM微处理器的指令集可以分为五大类

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

2. 数据处理指令大致分为5类：

2.1 数据传送指令：MOV、MVN

2.2 算术运算指令：ADD、ADC、SUB、SBC、RSB、RSC、MUL、MULA、UMULL.....

2.3 逻辑运算指令：AND、ORR、EOR、BIC

2.4 比较指令：CMP、CMN

2.5 测试指令：TST、TEQ

31	28	27	26	25	24	21	20	19	16	15	12	11	0
cond		0	0	I	opcode	S	Rn		Rd		operand2		

数据处理指令：

- 这些指令仅针对寄存器内的数据操作，不能使用内存数据。

- 语法：

第1操作数 第2操作数

< Opcode > < Cond > {S} Rd, Rn, Operand2

- 比较指令仅仅设置条件标志位，无需指定Rd

CMP r0, r3

- 数据传送指令

MOV r0, r1; MOV R0, #0x01

- 第2个操作数可以为一个寄存器或一个立即数

SUB r0, r1, r2

AND r1, r4, #0xff

□ 语法:

第二个操作数通过移位器传输到ALU，从而可以移位操作

< Opcode > < Cond > { S } Rd, Rn, Operand2

- 若第2操作数为寄存器，则可附加移位；
- 主要移位操作包括LSL, LSR, ASR, ROR, RRX;



LSL : Logical Left Shift



LSR : Logical Shift Right



1、移位的位数<32

2、可用立即数或寄存器方式给出

ASR: Arithmetic Right Shift



ROR: Rotate Right



RRX: Rotate Right Extended



□ 语法：第二个操作数通过移位器传输到ALU，从而可以移位操作

< Opcode > < Cond > {S} Rd, Rn, Operand2

- 若第2操作数为寄存器，则可附加移位；
- 主要移位操作包括LSL, LSR, ASR, ROR, RRX;
- 移位的值可以为立即数或寄存器
- 例如：

ADD r0, r1, r2, LSL #1

ADD R3, R2, R1, LSL R4

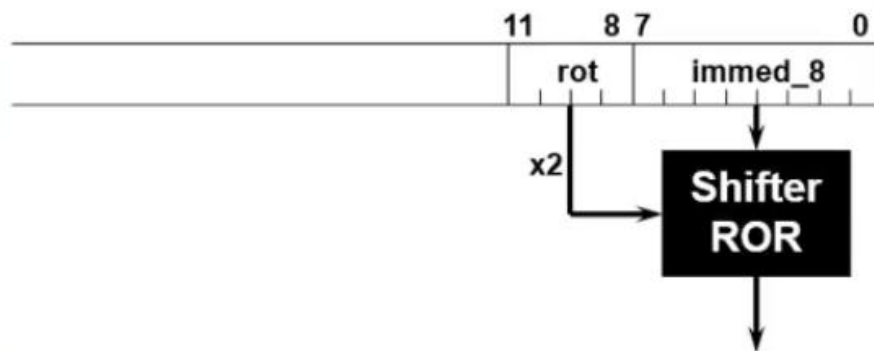


□ 语法：

第二个操作数是立即数

< Opcode > < Cond > {S} Rd, Rn, Operand2

- 范围在0-255的8位立即数，或者8位立即数循环右移偶数位得到的数；
- ARM指令32位定长的，在数据处理指令格式中，第二操作数有12位来对应；



- 4位的移位数乘以2组成了步长，范围在0-30的移位值；

指令格式如下：

数据传送指令：

MOV{<cond>}{S} <Rd>,<源操作数>

数据取反传送指令：

MVN{<cond>}{S} <Rd>,<源操作数>

即把一个被取反的值传送到目的寄存器中。

指令举例如下：

MOV R1, R0 ; 例

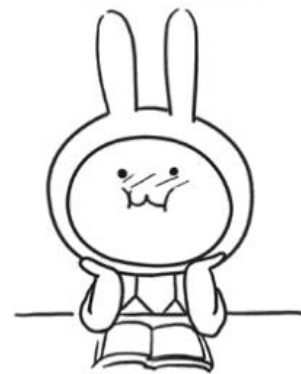
MOV R1, R0, LSL #2 ; R1=R0*4

MOVS R1, R0, LSL #1 ; R1=R0*2

MVN R0, #0 ; R0=0xFFFFFFFF

Note: 其中S决定指令的操作是否影响CPSR中条件标志位的值，当没有S时指令不更新CPSR中条件标志位的值。

沉迷自学
无法自拔



(1) 加法指令

指令格式如下：

ADD{<cond>}{S} <Rd>,<Rn>,<op2>

加法指令， $Rd=Rn+op2$

ADC{<cond>}{S} <Rd>,<Rn>,<op2>

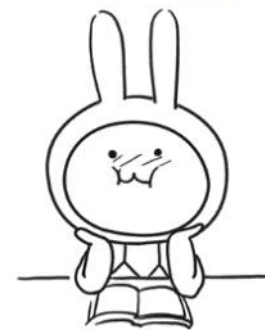
带进位的加法指令， $Rd=Rn+op2+carry$ (再加上CPSR中的C条件标志位的值)

指令举例如下：

ADDS R4, R0, R2 ;(R5 R4=R1R0+R3R2)

ADC R5, R1, R3 ;64位加法

沉迷自学
无法自拔



(2) 减法指令

指令格式如下：

SUB{<cond>}{S} <Rd>,<Rn>,<op2>

减法指令， $Rd = Rn - op2$

SBC{<cond>}{S} <Rd>,<Rn>,<op2>

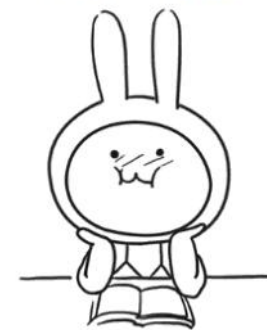
带借位减法指令， $Rd = Rn - op2 - carry$ (再减去CPSR中的C条件标志位的反码)

指令举例如下：

SUBS R4, R0, R2 ;(R5 R4=R1R0-R3R2)

SBC R5, R1, R3 ;64位减法

沉迷自学
无法自拔



指令格式如下：

RSB{<cond>}{S} <Rd>,<Rn>,<op2>

反向减法指令， $Rd = op2 - Rn$ 。

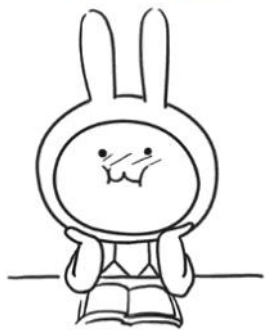
RSC{<cond>}{S} <Rd>,<Rn>,<op2>

带借位的反向减法指令， $Rd = op2 - Rn - !carry$ ，**RSC**指令用于把操作数2减去操作数1，再减去CPSR中的C条件标志位的反码，并将结果存放到目的寄存器中。

指令举例如下：

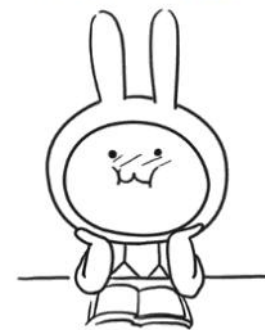
```
RSB    R5,R2,R1    ;/*R1减R2的值传送到R5*/  
RSC    R7,R1,R2    ;/*R2减R1的值再减去!C位后传送  
                  ;到R7*/
```

沉迷自学
无法自拔

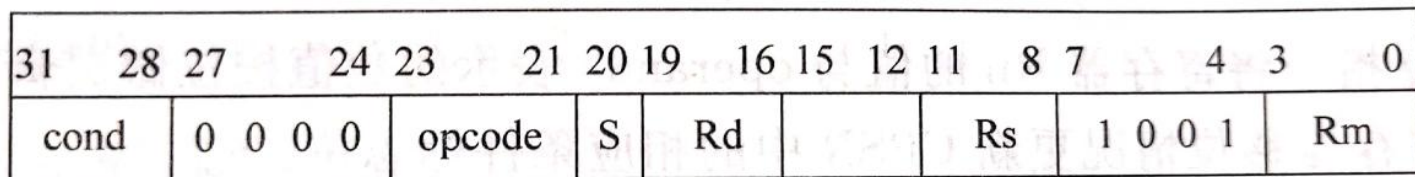


- ARM微处理器支持的乘法指令与乘加指令共有6条，可分为运算结果为32位和运算结果为64位两类，与前面的数据处理指令不同，指令中的所有操作数、目的寄存器**必须为通用寄存器**，**不能对操作数使用立即数或被移位的寄存器**，同时目的寄存器和操作数1必须是不同的寄存器。
- 乘法指令与乘加指令共有以下6条：
 - ✓ MUL 32位乘法指令
 - ✓ MLA 32位乘加指令
 - ✓ SMULL 64位有符号数乘法指令
 - ✓ SMLAL 64位有符号数乘加指令
 - ✓ UMULL 64位无符号数乘法指令
 - ✓ UMLAL 64位无符号数乘加指令

沉迷自学
无法自拔



32位乘法指令



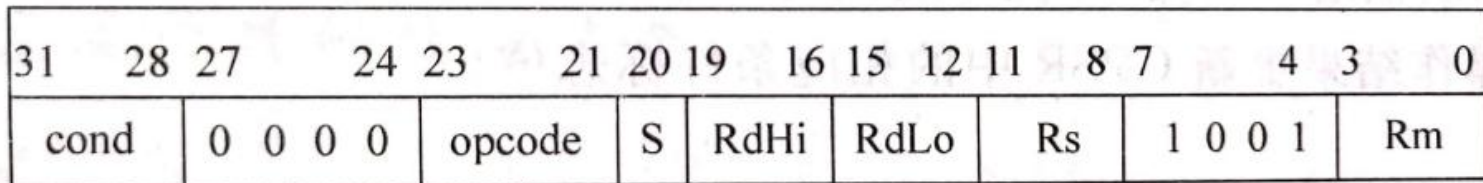
000: MUL

001: MLA

MUL: 全为0

MLA: Rn

64位乘法指令



100: UMULL

101: UMLAL

110: SMULL

111: SMLAL

沉迷自学
无法自拔



(1) MUL指令——32位乘法指令

指令格式如下：

```
MUL{cond}{S} Rd, Rm, Rs
```

MUL指令完成将操作数1与操作数2的乘法运算，并把结果放置到目的寄存器中，同时可以根据运算结果设置CPSR中相应的条件标志位。其中，操作数1和操作数2均为32位的有符号数或无符号数。

指令举例如下：

```
MUL R0, R1, R2      ; R0=R1×R2  
MULSR0, R1, R2      ; R0=R1×R2, 同时设置CPSR  
                     ; 中的N位和Z位。
```

沉迷自学
无法自拔



(2) MLA指令——32位乘加指令

指令格式如下：

```
MLA{cond}{S} Rd, Rm, Rs, Rn
```

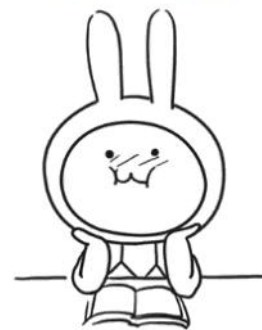
MLA指令完成将Rm与Rs的乘法运算，再将乘积加上Rn，并把结果放置到Rd中，同时可以根据运算结果设置CPSR中相应的条件标志位。其中Rm和Rs均为32位的有符号数或无符号数。

指令举例如下：

```
MLA R0, R1, R2, R3 ; R0=(R1*R2)+R3
```

```
MLAS R0, R1, R2, R3
```

沉迷自学
无法自拔



(3) SMULL指令——64位有符号乘法指令

指令格式如下：

SMULL{cond}{S} RdLo, RdHi, Rm, Rs

SMULL指令完成将Rm与Rs的乘法运算，并把结果的低32位放置到RdLo中，结果的高32位放置到RdHi中，同时可以根据运算结果设置CPSR中相应的条件标志位。其中，Rm和Rs均为32位的有符号数。

指令举例如下：

SMULL R0, R1, R2, R3 ; R1:R0 = R2 × R3

沉迷自学
无法自拔



(4) SMLAL指令——64位有符号乘加指令

指令格式如下：

`SMLAL{cond}{S} RdLo, RdHi, Rm, Rs`

SMLAL指令完成将Rm与操作数Rs的乘法运算，并把结果的低32位同RdLo中的值相加后又放置到RdLo中，结果的高32位同RdHi中的值相加后又放置到RdHi中，同时可以根据运算结果设置CPSR中相应的条件标志位。其中，Rm和Rs均为32位的有符号数。

指令举例如下：

`SMLAL R0, R1, R2, R3 ; (R1, R0)=R2×R3+(R1,R0)`

沉迷自学
无法自拔



(5) UMULL指令——64位无符号乘法指令

指令格式如下：

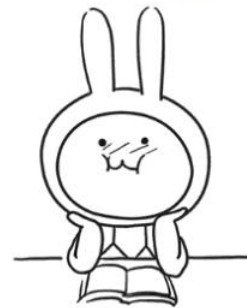
```
UMULL{cond}{S} RdLo, RdHi, Rm, Rs
```

UMULL指令完成将操作数Rm与操作数Rs的乘法运算，并把结果的低32位放置到目的寄存器RdLo中，结果的高32位放置到目的寄存器RdHi中，同时可以根据运算结果设置CPSR中相应的条件标志位。其中，操作数Rm和操作数Rs均为32位的无符号数。

指令举例如下：

```
UMULL    R0, R1, R2, R3 ; (R1, R0)=R2×R3
```

沉迷自学
无法自拔



(5) UMLAL指令——64位无符号乘加指令

指令格式如下：

```
UMLAL{cond}{S} RdLo, RdHi, Rm, Rs
```

UMLAL指令完成将Rm与Rs的乘法运算，并把结果的低32位同RdLo中的值相加后又放置到RdLo中，结果的高32位同RdHi中的值相加后又放置到RdHi中，同时可以根据运算结果设置CPSR中相应的条件标志位。其中Rm和Rs均为32位的无符号数。

指令举例如下：

```
UMLAL R0, R1, R2, R3 ; (R1, R0)=R2×R3+(R1,R0)
```

沉迷自学
无法自拔



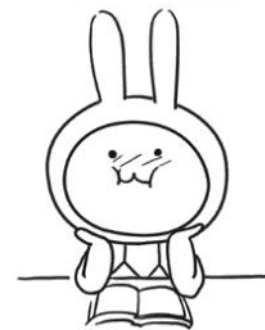
指令格式如下：

- **AND** {<cond>}{S} <Rd>,<Rn>,<op2>
逻辑与指令 Rd=Rn AND op2
- **ORR** {<cond>}{S} <Rd>,<Rn>,<op2>
逻辑或指令 Rd=Rn OR op2
- **EOR** {<cond>}{S} <Rd>,<Rn>,<op2>
逻辑异或指令 Rd=Rn EOR op2
- **BIC** {<cond>}{S} <Rd>,<Rn>,<op2>
位清除指令 Rd=Rn AND (!op2)

指令举例如下：

```
BIC    R0, R0, #0x0F           ; R0低4位清0
```

沉迷自学
无法自拔



BIC——清零指令

指令格式如下：

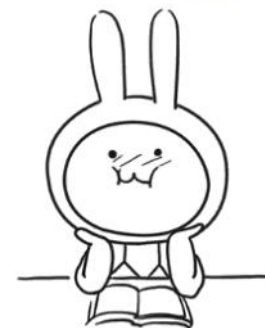
```
BIC {cond} {S}  Rd, Rn, operand2
```

BIC指令用于清除操作数1(Rn)中的某些位，将结果放入目标寄存器Rd中，同时根据结果影响标志位。操作数2为32位掩码，决定哪些位清0。

指令举例如下：

```
BIC R0, R0, #0x0F
```

沉迷自学
无法自拔



根据Rn - Op2设置条件码

(1) CMP——比较指令

指令格式如下：

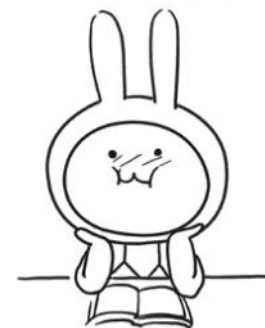
```
CMP {cond} Rn, operand2
```

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。

指令举例如下：

```
CMP R1, #5
```

沉迷自学
无法自拔



(2) CMN——负数比较指令

指令格式如下：

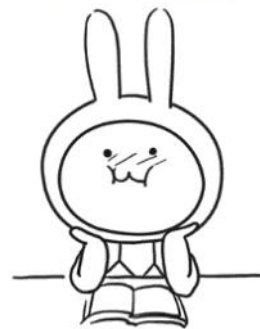
```
CMN {cond} Rn, operand2
```

CMN指令将寄存器Rn的值与operand2取反后的值进行比较（相加），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。

指令举例如下：

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

沉迷自学
无法自拔



(1) TST指令——位测试指令

指令的格式为：

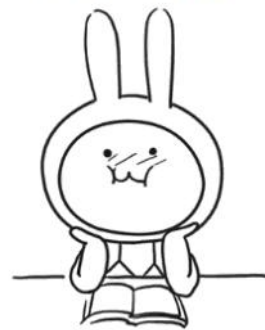
```
TST {cond} Rn, operand2
```

TST指令用于把一个Rn的内容和operand2的内容进行按位逻辑与运算，并根据运算结果更新CPSR中条件标志位的值。操作数1是要测试的数据，而操作数2是一个位掩码，该指令一般用来检测是否设置了特定的位。

指令的举例为：

```
TST R1, #1      ; 测试R1最低位是否为0，若为0，  
                ; 则EQ有效  
TST R1, #0x0f   ; 测试R1最低4位是否为0，若都  
                ; 为0，则EQ有效，只要有一个不  
                ; 为0，则NE有效
```

沉迷自学
无法自拔



(2) TEQ指令——测试相等指令

指令的格式为：

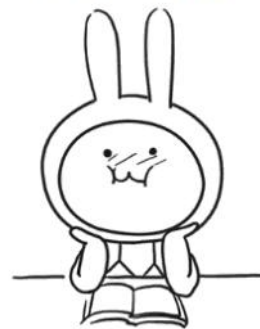
```
TEQ {cond} Rn, operand2
```

TEQ指令用于把Rn的内容和operand2的内容进行按位的异或运算，并根据运算结果更新CPSR中条件标志位的值。该指令通常用于比较操作数1和操作数2是否相等。

指令的举例为：

```
TEQ R1, #1; R1=R2? Y,EQ有效; N, NE有效
```

沉迷自学
无法自拔



ARM微处理器的指令集可以分为五大类：

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

指令实现存储器与寄存器之间的数据传输。

■ 单加载/存储指令

LDR {<cond>} <Rd>,<address>

字数据**读**取指令。

STR {<cond>} <Rd>,<address>

字数据**写**入指令。

■ 批量加载/存储命令

LDM {<cond>} <addressing_mode> <Rn>{!},<Registers>

批量内存字数据**读**取指令。

STM {<cond>} <addressing_mode> <Rn>{!},<Registers>

批量内存字数据**写**入。

加载指令	存储指令	操作字长
LDR	STR	Word
LDRB	STRB	Byte
LDRH	STRH	Halfword
LDRSB		带符号的byte
LDRSH		带符号的halfword

Note: 存储器系统必须支持所有访问宽度

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	0
cond		0	1	I	P	U	0	W	L	Rn	Rd		addressing_mode		

cond 为指令执行的条件。

$I = \begin{cases} 1: \text{偏移量为寄存器移位形式} \\ 0: \text{偏移量为 12 位立即数} \end{cases}$

$P = \begin{cases} 1: \text{前变址操作} \\ 0: \text{后变址操作} \end{cases}$

$U = \begin{cases} 1: \text{内存地址 address 为基址寄存器 Rn 值加上地址偏移量} \\ 0: \text{内存地址 address 为基址寄存器 Rn 值减去地址偏移量} \end{cases}$

$W = \begin{cases} 1: \text{执行基地址寄存器回写操作} \\ 0: \text{不执行基地址寄存器回写操作} \end{cases}$

$L = \begin{cases} 1: \text{执行 Load 操作} \\ 0: \text{执行 Store 操作} \end{cases}$

(1) LDR指令

指令语法格式如下：

```
LDR{<cond>}{<size>} Rd, <address>
```

功能：将指定地址上的内存数据，按指令中规定的大小(LDR字；或LDRB低8位字节，高24位=0；或LDRH半字，高16位=0)，加载到寄存器Rd中。

指令的举例如下：

```
LDR r0, [r1]
```


(2) STR指令

指令语法格式如下：

STR{<cond>}{<size>} Rd, <address>

功能：将寄存器Rd中的数据按指令中规定的大小（STR字；或STRB低8位字节，高24位=0；或STRH半字，高16位=0），存储到指定地址上的内存中。

STR r0, [r1]

LDR/STR:采用寄存器间接寻址或基址变址寻址的方式。其不同的索引方式如下：(举例)

1. 零偏移;

```
LDR r0, [r1]
```

```
STR r0, [r1]
```

3. 后索引偏移（后变址）;

```
LDR r0, [r1], #0x04
```

; 例(r1)送r0, r1=r1+4

2. 前索引偏移（前变址）;

```
OP{cond} Rd, [Rn, offset] {!}
```

```
LDR r0, [r1, #0x04]; 例 (r1+4) 送r0
```

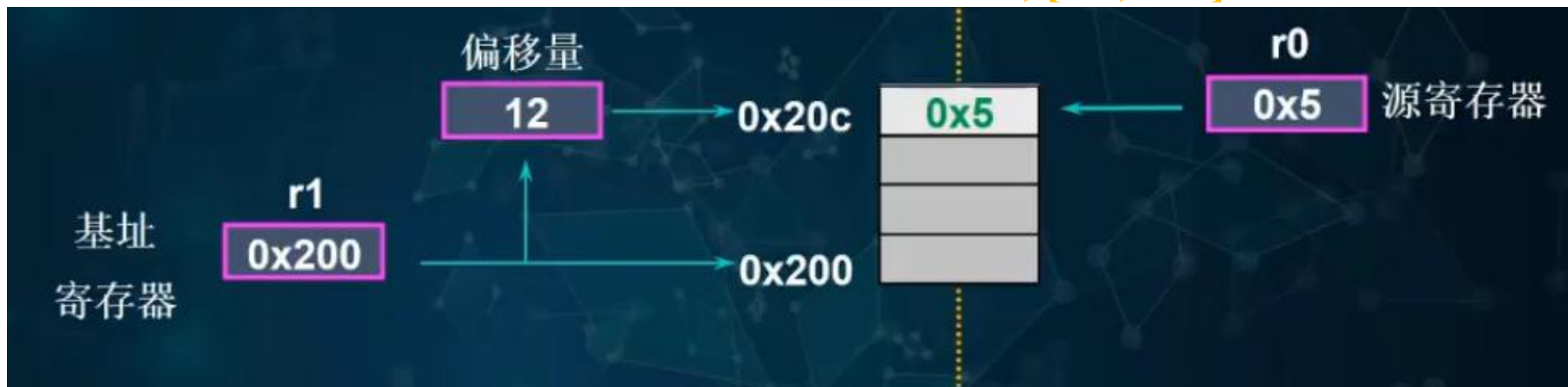
```
LDR r0, [r1, #-4]!
```

; 例 (r1-4) 送r0, r1=r1-4

```
LDR r0, [r1, r2, ASR #0x04]!
```

; 取地址 (r1+r2右移4位) 内容送r0, 然后r1更新为r1+r2右移4位

前索引偏移: $\text{STR R0}, [\text{R1}, \#12] \longrightarrow \text{STR R0}, [\text{R1}, \#12]!$



后索引偏移: $\text{STR R0}, [\text{R1}], \#12$



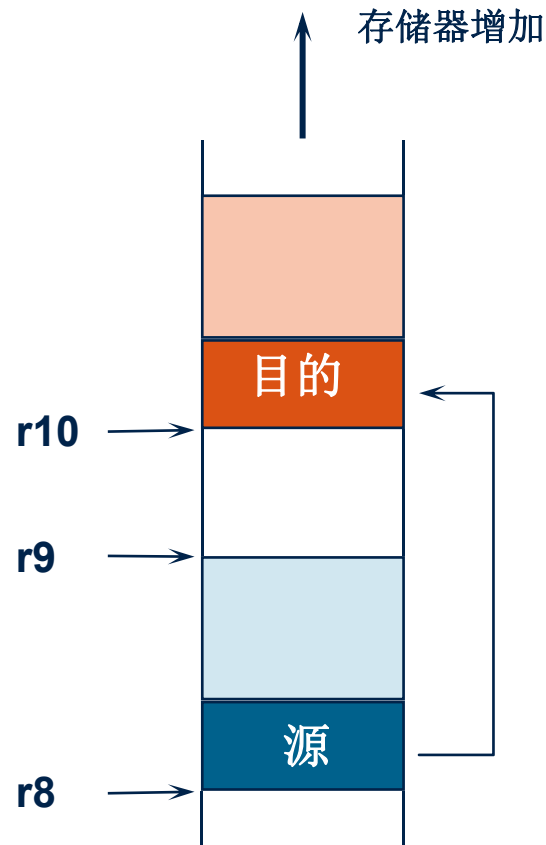
例如：后续寻址可用来实现拷贝一个内存块，每次循环拷贝一个字；

；r8指向源数据起始地址

；r9指向源数据尾地址

；r10指向目的数据起始地址

```
loop    LDR    r0, [r8], #4    ; load data
        STR    r0, [r10], #4   ; store data
        CMP    r8, r9         ; check for the end
        BLT    loop           ; loop until done
```



- ARM微处理器所支持批量数据加载/存储指令可以一次在一片连续的存储器单元和多个寄存器之间传送数据，批量加载指令用于将一片连续的存储器中的数据传送到多个寄存器，批量数据存储指令则完成相反的操作。常用的加载存储指令如下：

指令语法格式如下：

```
LDM {<cond>} <addressing_mode> <Rn>{!}, <Registers> {^}
```

批量加载指令。批量内存字数据读取指令。

```
STM {<cond>} <addressing_mode> <Rn>{!}, <Registers> {^}
```

批量存储指令。批量内存字数据写入。

31	28	27	25	24	23	22	21	20	19	16	15	0
cond	1	0	0	P	U	0	W	L	Rn	register_list		

cond 为指令执行的条件。

$P = \begin{cases} 1: \text{前变址操作} \\ 0: \text{后变址操作} \end{cases}$

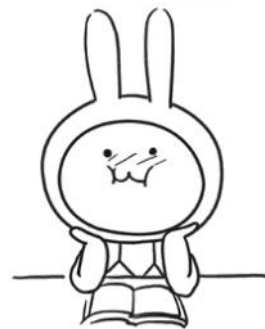
U 表示地址变化的方向：

$U = \begin{cases} 1: \text{地址向上变化(upwards)} \\ 0: \text{地址向下变化(downwards)} \end{cases}$

$W = \begin{cases} 1: \text{执行基地址寄存器回写操作} \\ 0: \text{不执行基地址寄存器回写操作} \end{cases}$

$L = \begin{cases} 1: \text{执行 Load 操作} \\ 0: \text{执行 Store 操作} \end{cases}$

沉迷自学
无法自拔



STMFD R13!, {R0, R4-R12, LR}

；子程序入口，保存现场到堆栈

LDMFD R13!, {R0, R4-R12, PC}

；子程序出口，恢复现场

LDMIA R0!, {R1~R3} ；见例

Note:

- 基址寄存器Rn：保存“传送数据”的起始地址，不允许为R15；

STMFD R13!, {R0, R4-R12, LR}

; 子程序入口, 保存现场到堆栈

LDMFD R13!, {R0, R4-R12, PC}

; 子程序出口, 恢复现场

LDMIA R0!, {R1~R3} ;见例

Note:

- **Register-List:** 可包含多个寄存器, 用“,”隔开, 寄存器按由小到大的顺序排列;

LDM/STM {<cond>} <addressing_mode> <Rn>{!}, <Registers> {^}

mode:

内存操作

IA 每次传送后地址加
IB 每次传送前地址加
DA 每次传送后地址减
DB 每次传送前地址减

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

LDMIB R0, {R1, R2, R3, R4} ; R1←[R0+4]
 ; R2←[R0+8]
 ; R3←[R0+12]
 ; R4←[R0+16]

LDMIA / STMIA	increment after
LDMIB / STMIB	increment before
LDMDA / STMDA	decrement after
LDMDB / STMDB	decrement before

LDM/STM {<cond>} <mode> <Rn>{!}, <Registers> {^}

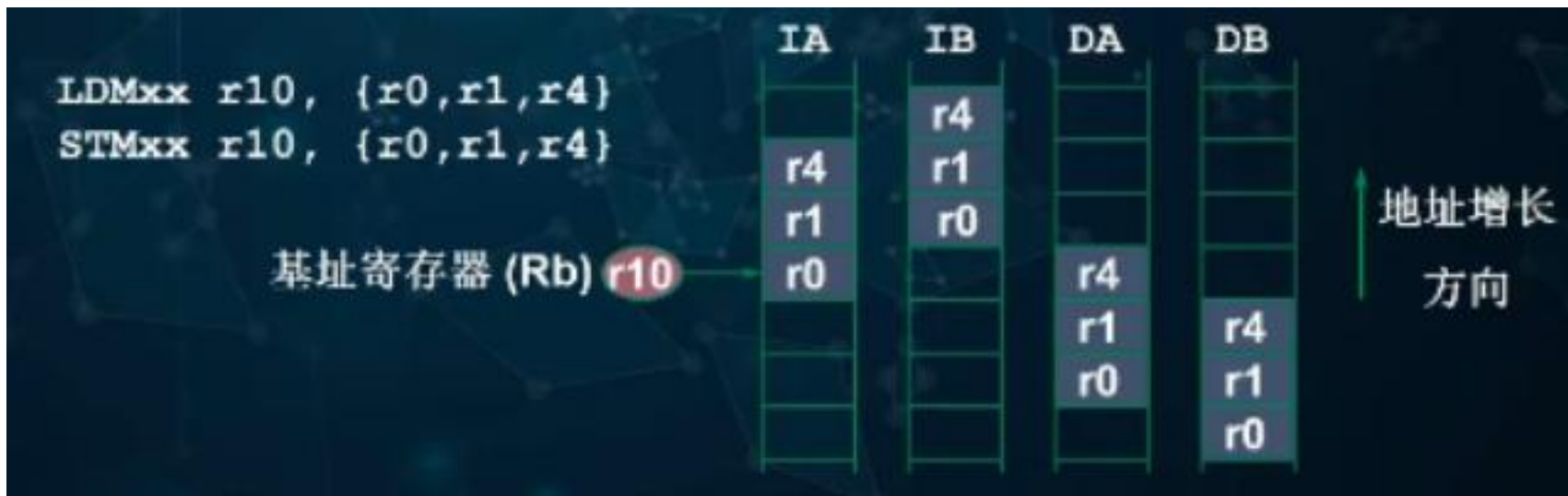
mode:

内存操作

IA 每次传送后地址加
IB 每次传送前地址加
DA 每次传送后地址减
DB 每次传送前地址减

Note:

- 寄存器存入顺序始终遵循最低编号寄存器存入最低地址。



LDM/STM {<cond>} <mode> <Rn> {!}, <Registers> {^}

mode:

内存操作

堆栈操作

IA 每次传送后地址加
IB 每次传送前地址加
DA 每次传送后地址减
DB 每次传送前地址减

FD 满递减堆栈
ED 空递减堆栈
FA 满递增堆栈
EA 空递增堆栈

LDM/STM {<cond>} <mode> <Rn>{!}, <Registers> {^}



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

```
STMIA R0 , {R3-R6, R10} ; [R0] ← R3
                        ; [R0+4] ← R4
                        ; [R0+8] ← R5
                        ; [R0+12] ← R6
                        ; [R0+16] ← R10
```

```
STMIA R0! , {R3-R6, R10} ; [R0] ← R3 , R0 ← R0+4
                        ; [R0] ← R4 , R0 ← R0+4
                        ; [R0] ← R5 , R0 ← R0+4
                        ; [R0] ← R6 , R0 ← R0+4
                        ; [R0] ← R10 , R0 ← R0+4
```

LDM/STM {<cond>} <mode> <Rn>{!}, <Registers> {^}




{^} 为可选后缀，是否拷贝SPSR到CPSR。用于异常返回的处理。

- **Rn**: 基址寄存器，装有传送数据的初始地址，不允许为R15。
- 寄存器列表可以为R0~R15的任意组合。

■ 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]

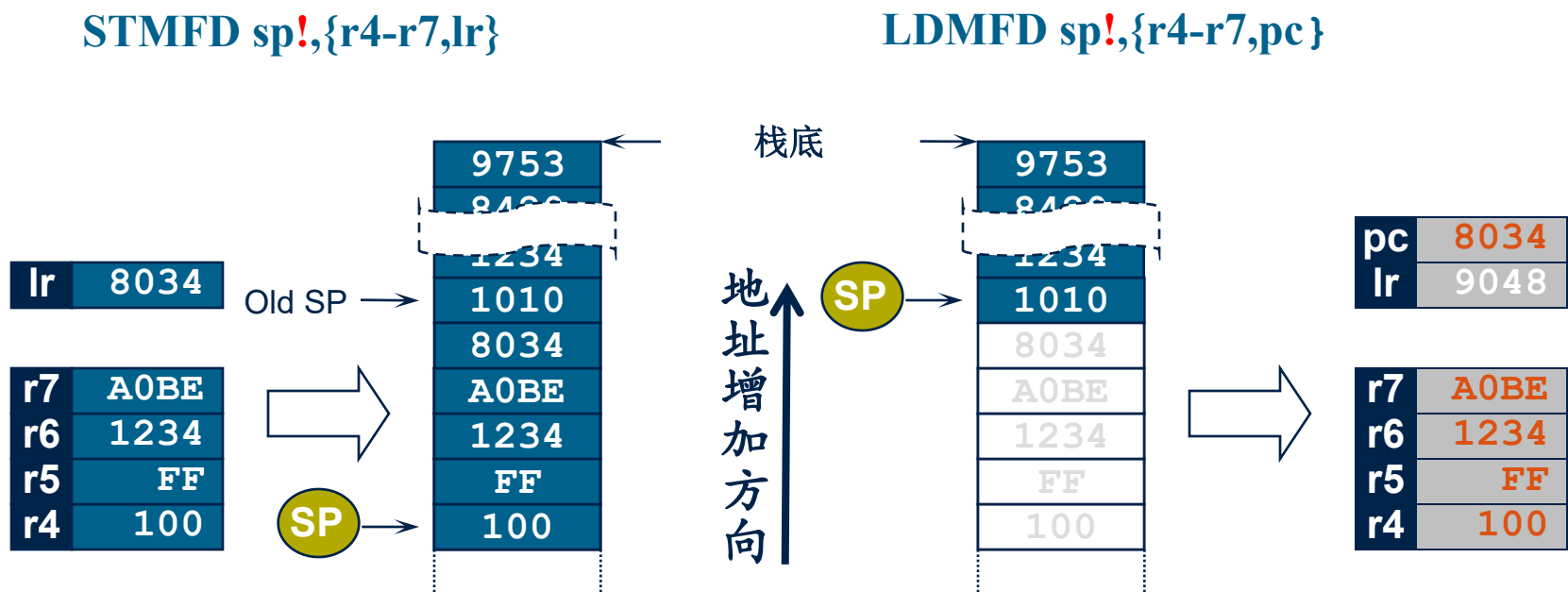


注意: 这里寄存器在压栈时的顺序按照寄存器编号由高到低的顺序进行, 与被指定的寄存器顺序无关。

STMFD sp!,{r4-r7,lr}

LDMFD sp!,{r4-r7,pc}

- ARM堆栈操作通过块传送指令来完成：
 - **STMFD**(Push) 块存储- **F**ull **D**escending stack [STMDB]
 - **LDMFD**(Pop) 块装载- **F**ull **D**escending stack [LDMIA]

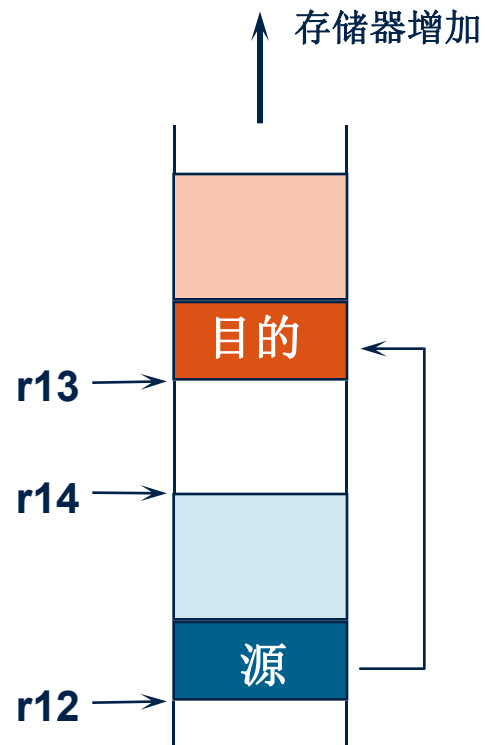


; r12指向源数据起始地址

; r14指向源数据尾地址

; r13指向目的数据起始地址

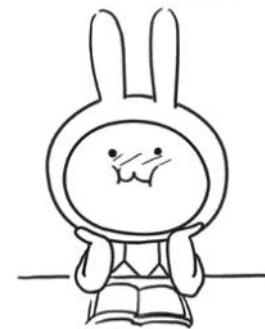
```
loop    LDMIA r12!, {r0-r7}    ; load data
        STMIA r13!, {r0-r7}    ; store data
        CMP    r12, r14        ; check for the end
        BNE    loop            ; loop until done
```



■ ARM微处理器所支持数据交换指令能在**存储器**和**寄存器**之间交换数据。数据交换指令有如下两条：

1. SWP **字**数据交换指令
2. SWPB **字节**数据交换指令

沉迷自学
无法自拔

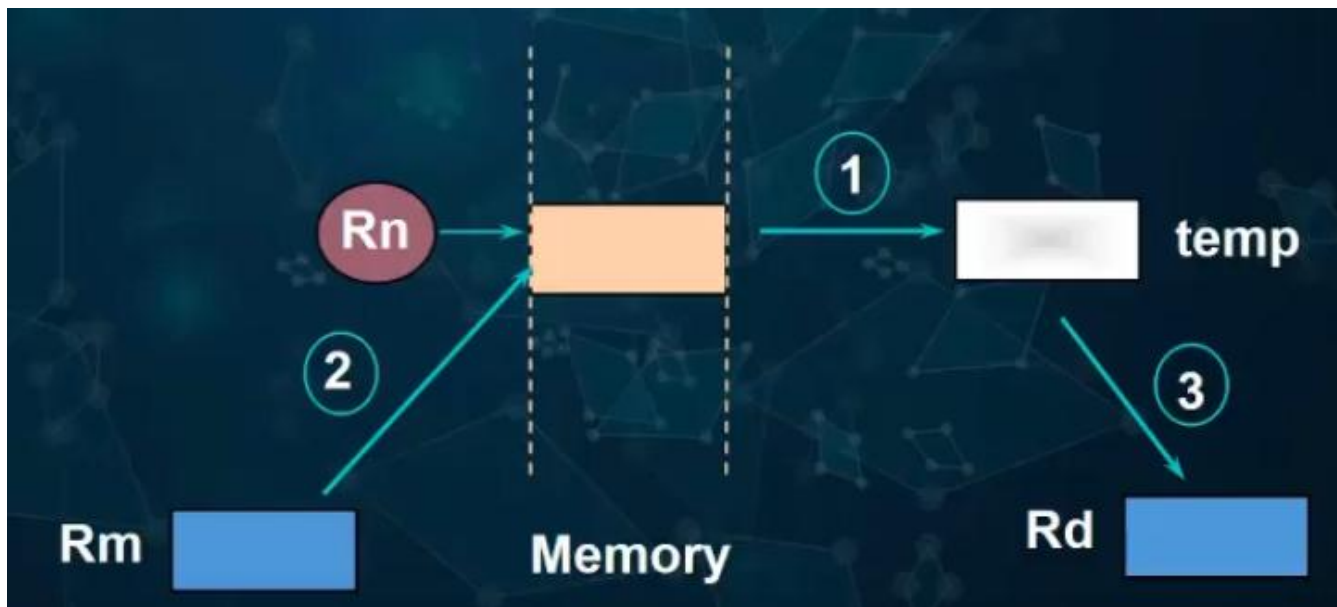


在寄存器和存储器之间，由一次存储器读和一次存储器写组成的操作。完成一个字节或字的交换。

■ 指令的语法格式为：

SWP{<cond>}{B} Rd , Rm , [Rn]

字数据交换， $Rd=[Rn]$ ， $[Rn]=Rm$ 。



沉迷自学
无法自拔

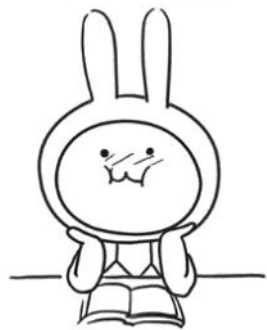


■ 指令的语法格式为：

SWPB{<cond>} Rd, Rm , [Rn]

字节数据交换指令。从Rn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位数据存储到同一内存地址中。

沉迷自学
无法自拔



SWP R0, R1, [R2]

;R0=R2所表示的内存单元的数据，例
;将R1数据保存到R2所表示的内存单元

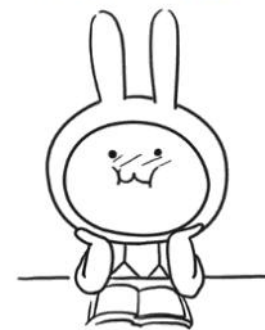
SWP R1, R1, [R10]

; 交换[R10]和R1的内容。Rd=Rm时

SWPB R3, R2, [R10]

;R3=R10所表示的内存单元的数据的低
;8位，R3高24位置0再将R2的数据的低
;8位保存到R10所表示的内存单元

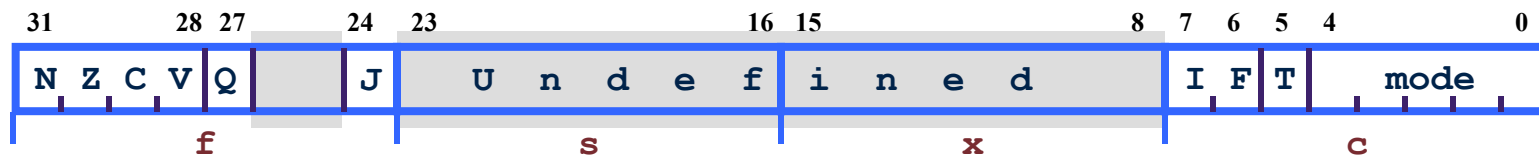
沉迷自学
无法自拔



ARM微处理器的指令集可以分为五大类

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

- ARM微处理器支持程序状态寄存器访问指令，用于在程序状态寄存器CPSR/SPSR和通用寄存器之间传送数据，程序状态寄存器访问指令包括以下两条：
 - MRS 程序状态寄存器到通用寄存器的数据传送指令
 - MSR 通用寄存器到程序状态寄存器的数据传送指令
- 用来进行处理器模式切换、允许/禁止IRQ/FIQ中断等。



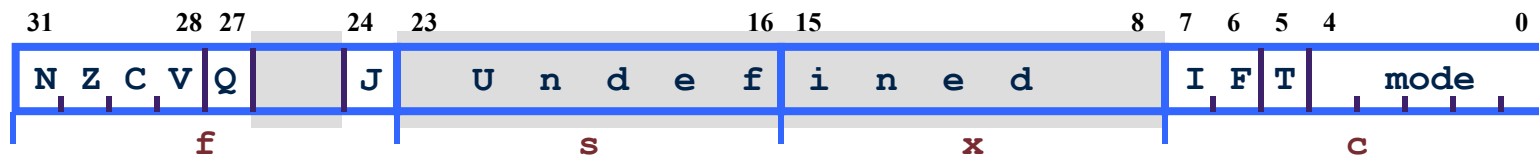
■ 指令的语法格式为：

MRS{<cond>} Rd,<psr> ; Rd = <psr>

MSR{<cond>} <psr[_fields]>,Rm ; <psr[_fields]> = Rm

MRS R0, CPSR ;/*保存 cpsr 到 R0*/

MSR SPSR, R1 ;/*通过 R1修改 spsr */



■ 指令的语法格式为：

MRS{<cond>} Rd,<psr> ; Rd = <psr>

MSR{<cond>} <psr[_fields]>,Rm ; <psr[_fields]> = Rm

fields: 指定传送区域。

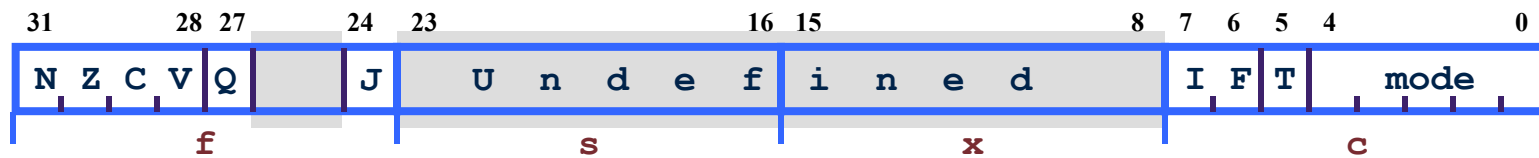
c: 控制域 (psr[7:0])

x: 扩展域 (psr[15:8])

s: 状态域 (psr[23:16])

f: 标志位域 (psr[31:24])

Note: 用户模式下，所有位均可以被读取，但不可被写回。



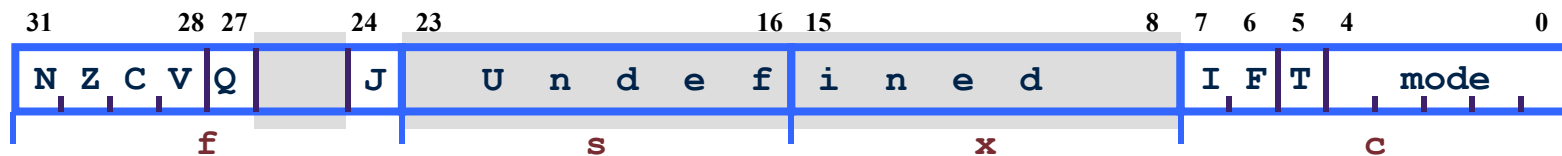
■ 指令的语法格式为：

MRS{<cond>} Rd,<psr> ; Rd = <psr>

MSR{<cond>} <psr[_fields]>,Rm ; <psr[_fields]> = Rm

MSR CPSR_f, R0 ;/*CPSR_f ← R0[31:24]*/

MSR CPSR_c, #0xf0 ;/*N,Z,C,V位均被置1



修改状态寄存器一般通过“读取-修改-写回”三步骤来实现。

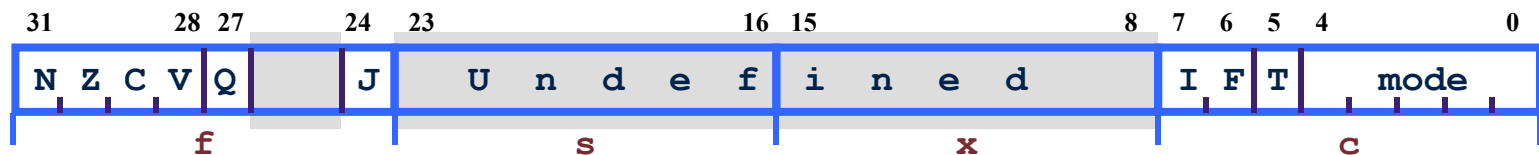
使能IRQ中断：

ENABLE_IRQ

MRS R0, CPSR

BIC R0, R0, #0X80

MSR CPSR, R0 ;/ R0只去改变cpsr的控制位*/



切换到IRQ模式:

```
MRS R0, CPSR
```

```
BIC R0, R0, #0X1F
```

```
ORR R0, R0, #0X12
```

```
MSR CPSR, R0 ;/ R0只去改变cpsr的控制位*/
```

Mode位(处理器模式位):

- 0b10000 User
- 0b10001 FIQ
- 0b10010 IRQ
- 0b10011 Supervisor
- 0b10111 Abort
- 0b11011 Undefined
- 0b11111 System

ARM微处理器的指令集可以分为六大类

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

■ ARM微处理器所支持的异常指令有如下两条：

- SWI 软件中断指令
- BKPT 断点中断指令



(1) SWI指令——软件中断指令

- 软件中断指令 (SWI) 用于进入**管理模式**，用于用户程序调用操作系统的API。
- 软件中断处理程序执行以下指令可以**从SWI模式返回**，无论是在ARM状态还是Thumb状态：

```
MOVS PC, R14_svc
```

Note: 以上指令恢复PC(从R14_svc)和CPSR(从SPSR_svc)的值，并返回到SWI的下一条指令。

你尽管复习



考到了算我输

指令的语法格式为：

SWI number

SWI{cond} <immed_24>

- SWI指令用于产生软件中断，以便用户程序能调用操作系统的系统例程。
- 操作系统在SWI的异常处理程序中提供相应的系统服务，指令中24位的立即数指定用户程序调用系统例程的类型，相关参数通过通用寄存器(r0~r3)传递。
- 当指令中24位的立即数被忽略时，用户程序调用系统例程的类型由通用寄存器R0的内容决定，同时，参数通过其他通用寄存器传递。

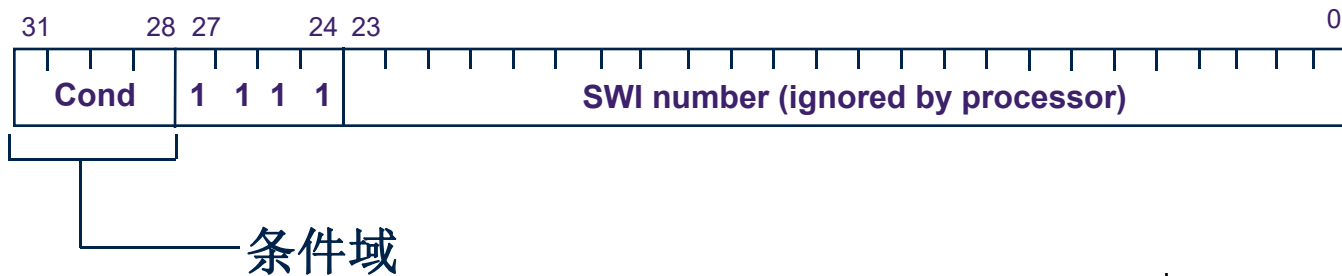
指令举例如下：

SWI 0x02 ; 调用操作系统编号为02的API

你尽管复习



考到了算我输



- 产生一个异常事件，跳转到SWI 向量。
- **SWI 处理程序**需要检测**SWI号**，从而决定采取何种操作。
- 通过SWI机制，**运行在用户模式下的应用程序**，可请求操作系统执行一系列特权操作。

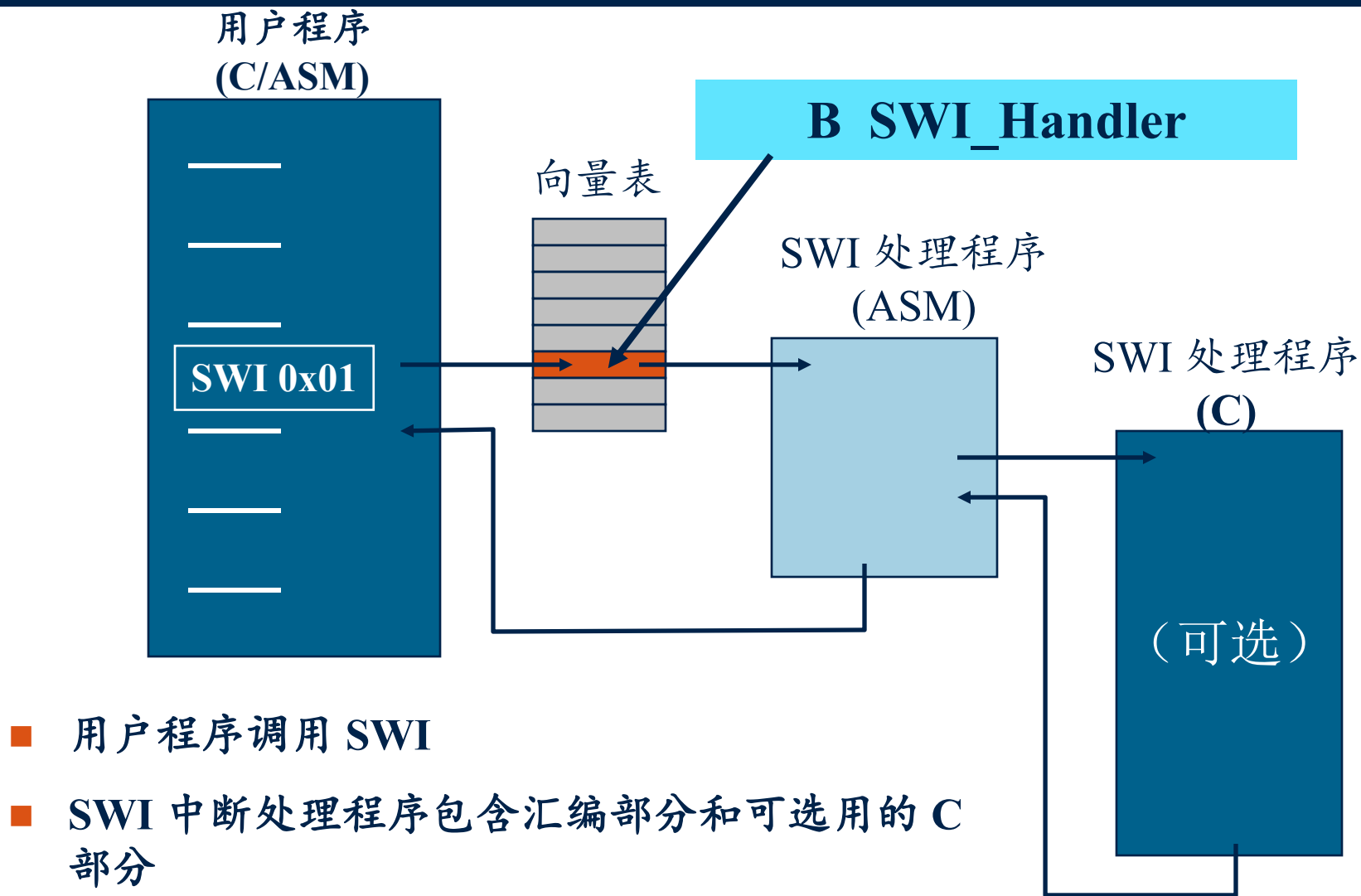
0x1C	FIQ
0x18	IRQ
0x14	(Reserved)
0x10	Data Abort
0x0C	Prefetch Abort
0x08	Software Interrupt
0x04	Undefined Instruction
0x00	Reset

Vector Table

你尽管复习



考到了算我输



- 用户程序调用 SWI
- SWI 中断处理程序包含汇编部分和可选用的 C 部分

你尽管复习



出题老师

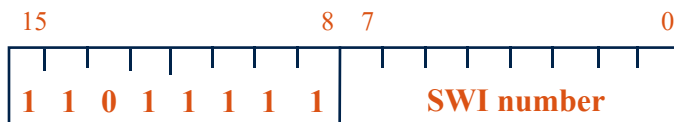
考到了算我输

- ARM 内核不提供直接传递软中断(SWI)号到处理程序的机制：
 - SWI 处理程序必须定位SWI 指令，并提取SWI指令中的常数域。
- 为此, SWI 处理程序必须确定SWI 调用是在哪一种状态 (ARM/Thumb).
 - 检查 SPSR 的 T-bit。

ARM 态格式:



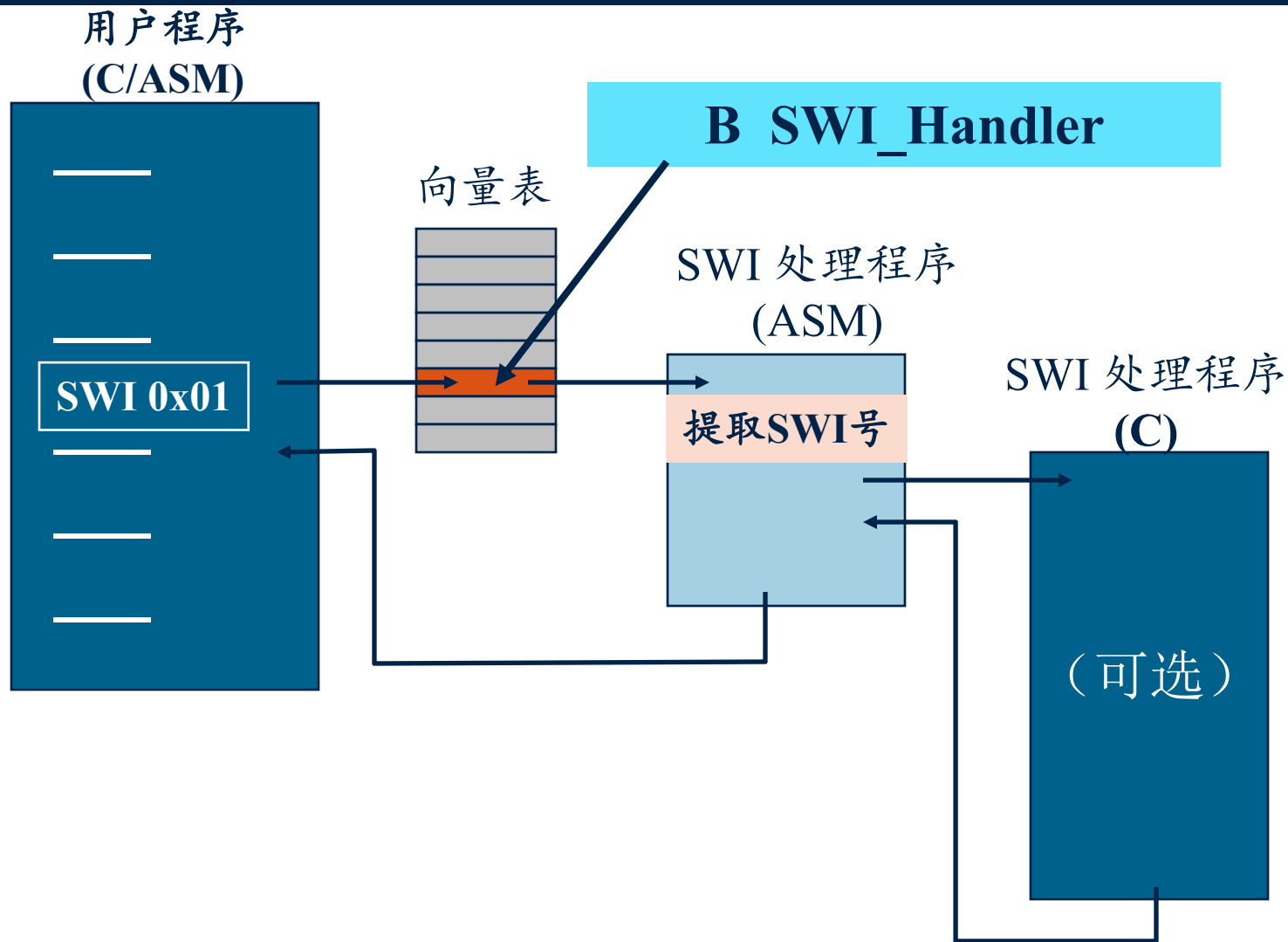
Thumb 态格式:



你尽管复习



考到了算我输



你尽管复习



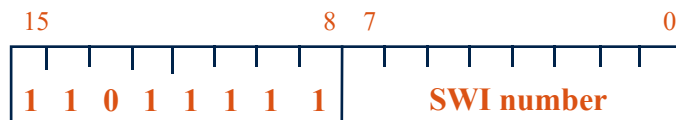
考到了算我输

- 为此, SWI 处理程序必须**确定SWI 调用是在哪一种状态** (ARM/Thumb).
 - 检查 SPSR 的 T-bit.
 - SWI 指令在ARM 状态下在 LR-4 位置, Thumb 状态下在 LR-2位置。
 - SWI 指令按相应的格式译码:

ARM 态格式:



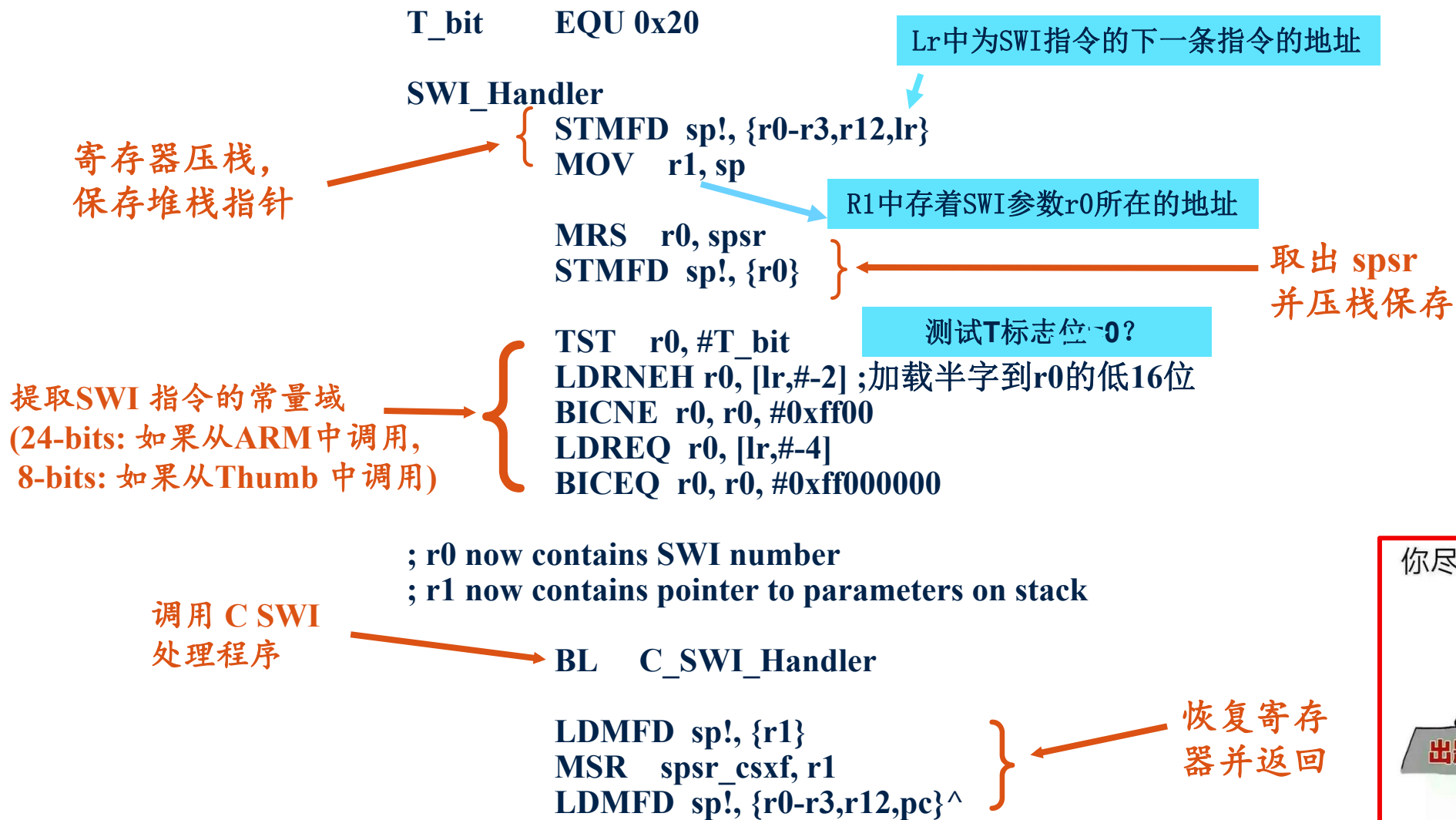
Thumb 态格式:



你尽管复习



考到了算我输



你尽管复习



考到了算我输

```
// Memory mapped registers
volatile unsigned parallel_output, parallel_input;

void C_SWI_Handler (unsigned number, int *param)
// r0 = SWI number
// r1 = pointer to SWI parameters in memory
{
    switch (number)
    {
        case 0: parallel_output = param[0];
                break;
        case 1: param[0] = parallel_input;
                break;
        default : break;
    }
}
```



(2) BKPT指令——断点中断指令

■ 指令的语法格式为：

BKPT {immed_16}

- 执行该指令将引起预取指令(Prefetch Abort)异常，使处理器进入调试状态；
- 断点是调试代理在RAM中设置的。

Note: BKPT指令产生软件断点中断，可用于程序的调试，V5T以上体系结构使用。

你尽管复习



考到了算我输

ARM体系**支持16个协处理器**。如果系统中没有协处理器，则会触发未定义指令异常。

■ 三种协处理器指令：

■ 协处理器数据处理指令

- CDP：初始化协处理器数据处理操作

■ 协处理器寄存器传送指令

- MRC：从 ARM 寄存器移到协处理器寄存器
- MCR：从协处理器寄存器移到ARM 寄存器

■ 协处理器存储器传送指令

- LDC：从寄存器所指向的存储器装载到协处理器寄存器
- STC：从协处理器寄存器存储到寄存器所指向的存储器

你尽管复习

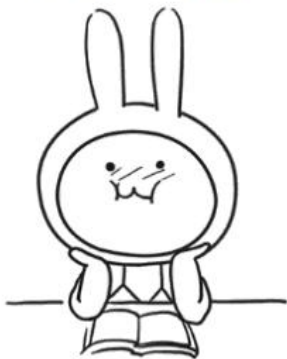


考到了算我输

条件码	后缀	标 志	含 义
0000	EQ	Z置位	相等
0001	NE	Z清零	不相等
0010	CS	C置位	无符号数大于或等于
0011	CC	C清零	无符号数小于
0100	MI	N置位	负数
0101	PL	N清零	正数或零
0110	VS	V置位	溢出
0111	VC	V清零	未溢出

条件码	后缀	标 志	含 义
1001	LS	C清零Z置位	无符号数小于或等于
1010	GE	N等于V	带符号数大于或等于
1011	LT	N不等于V	带符号数小于
1100	GT	Z清零且（N等于V）	带符号数大于
1101	LE	Z置位或（N不等于V）	带符号数小于或等于
1110	AL	忽略	无条件执行

沉迷自学
无法自拔



自学要求掌握

你尽管复习



考到了算我输

自学但不考