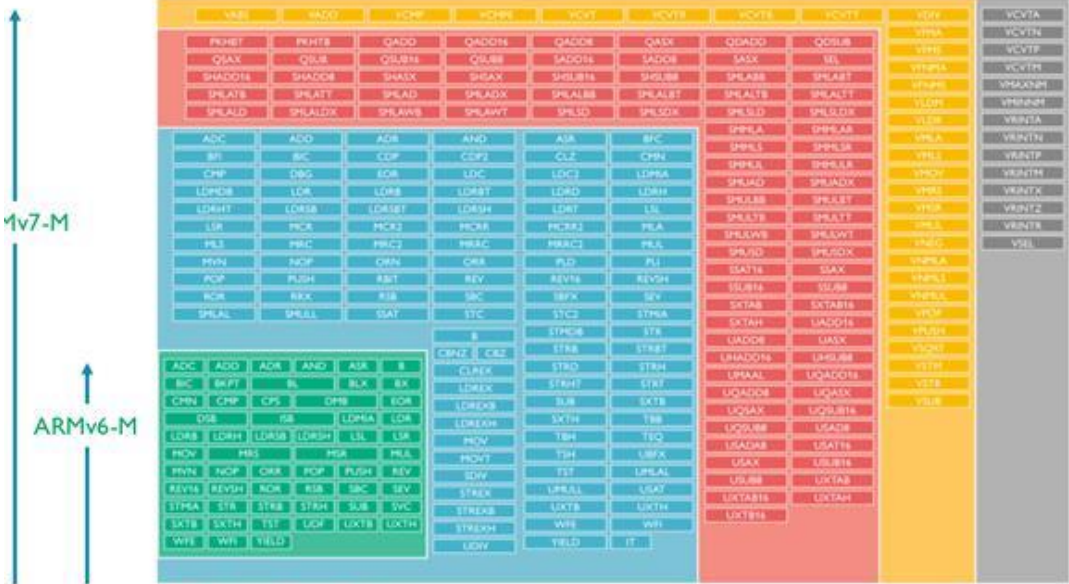


# 第5章 Cortex-Mx 指令系统

## Cortex-Mx 指令集概览

Cortex-Mx 指令集根据不同目标需求，有较大的伸缩

基本指令集较小-----RISC思想  
最小系统约56种， M3约97种

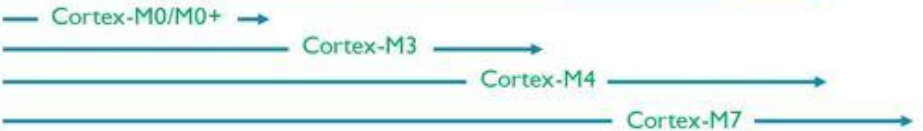


Floating Point

DSP (SIMD, fast MAC)

Advanced data processing  
bit field manipulations

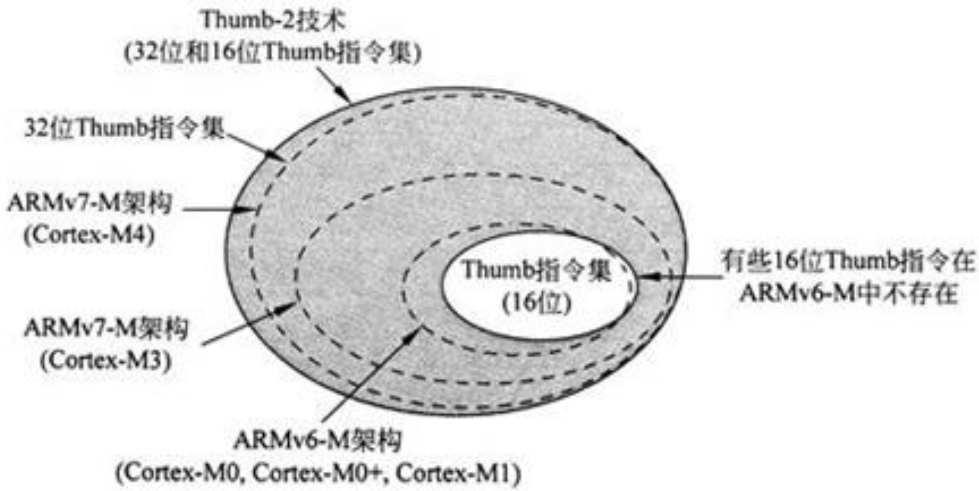
General data processing  
I/O control tasks



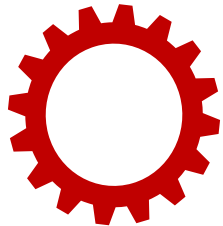
ADC	ADD	ADR	AND	ASR	B	CLZ
BFC	BFI	BIC	CDP	CLREX	CBNZ	CBZ
CMP				DBG	EOR	LDC
LDMIA				LDMDB	LDR	LDRB
LDRBT	BKPT	BLX	ADC	ADD	ADR	
LDREXH	BX	CPS	AND	ASR	B	
LDRSBT	DMB		BL	BIC		
MCR	DSB		CMN	CMP	EOR	
MCRR	ISB		LDR	LDRB	LDM	
MRC	MRS		LDRH	LDRSB	LDRSH	
NOP	MSR		LSL	LSR	MOV	
PLDW	NOP	REV	MUL	MVN	ORR	
RBIT	REV16	REVSH	POP	PUSH	ROR	
ROR	SEV	SXTB	RSB	SBC	STM	
SBFX	SXTH	UXTB	STR	STRB	STRH	
SMULL	UXTH	WFE	SUB	SVC	TST	
STMDB	WFI	YIELD				
STRD	STREX	STREXB	STREXH	STRH	STRBT	STRT
SUB	SXTB	SXTH	TBB	TBH	TEQ	TST
UBFX	UDIV	UMLAL	UMULL	USAT	UXTB	UXTH
WFE	WFI	YIELD	IT			

CORTEX-M0/M1

CORTEX-M3



Thumb 指令集和 Cortex-M 处理器实现的指令集间的差异



1、什么是Thumb？

Thumb指令能够看做是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。Thumb不是一个完整的体系结构，不能指望处理程序仅仅运行Thumb指令而不支持ARM指令集[5]。

2、为什么要有Thumb2,它与其他指令的关系如何？

注意上面的官方用语，用的是技术而不是说“Thumb-2指令集”。从官方角度说，并没有“Thumb-2指令集”。但[1]提及了，也算是合理的存在吧。

基于Thumb-2技术的Thumb指令集，不仅在原先的Thumb指令集基础上又添加了一些与ARM指令集中相同的指令（添加的这些指令大多是4字节编码的），而且降低了大多数原先Thumb指令集中运行条件的限制[3]。另外，为了提高架构间的软件移植性，并使得不同架构的ARM处理器符合同一汇编语言语法[1]，基于Thumb-2技术的Thumb指令集引入了全新的汇编语法——“统一汇编语言UAL”,从而实现了独立的ARM语法和Thumb语法的代替[3]。

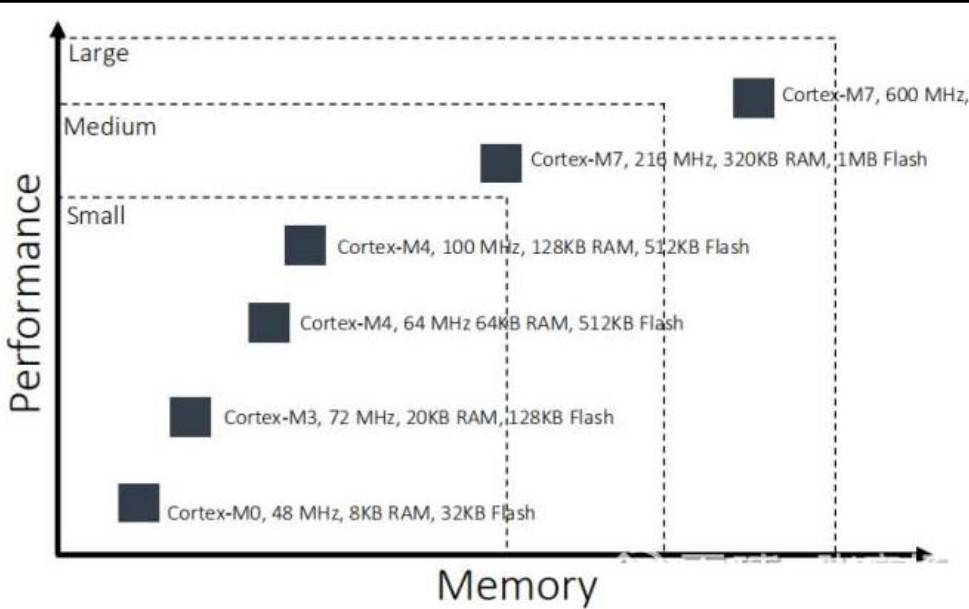
3、Cortex-M与Thumb-2

为了方便设计对成本敏感的设备，Cortex-M7处理器实现了紧密耦合的系统组件，减少了处理器面积，同时显著提高了中断处理和系统调试能力。Cortex-M7处理器实现了基于Thumb-2技术的Thumb®指令集的一个版本，确保了高的代码密度和降低的程序内存需求。Cortex-M7处理器指令集提供了现代32位架构所期望的卓越性能，比大多数8位和16位微控制器具有更好的代码密度[2]。

由于处理器支持Thumb-2指令集中的16和32指令，因此无须在Thumb状态（16位指令）和ARM状态（32位指令）间来回切换。

CortexM系列对Thumb-2指令集支持的程度是不同的，具体详见各自的手册。可以根据不同处理器支持的特性来初步判断是否有某些指令，如CortexM4支持浮点运算，因此

一点参考



	ARM Cortex-M0	ARM Cortex-M0+	ARM Cortex-M3	蜂鸟 E201	蜂鸟 E203	蜂鸟 E205
Dhystone(DMIPS/MHz)	0.84 (正式数据) 1.21 (经选项最大优化后数据)	0.94 (正式数据) 1.31 (经选项最大优化后数据)	1.25	1.171	1.23	1.355
CoreMark(CoreMark/MHz)	2.33	2.42	3.32	1.352	2.14	3.327
最小配置逻辑门数	12K	12K	36K	10K	12K	20K
流水线深度	3	2	3	2	2	2
乘法器	有	有	有	无	有	有
除法器	无	无	有	无	有	有
可扩展性	不支持	不支持	不支持	支持	支持	支持

注：Cortex-M0+的乘法器可以配置成单周期乘法器或者多周期迭代乘法器，Dhystone性能数据与CoreMark性能数据是采用单周期乘法还是多周期乘法器的信息不详。



## Cortex-Mx 基本指令类型

“最”基本 35 种

### 存储访问类

LDR	LDRB	LDRH	LDRSB	LDSH	LDM
STR	STRB	STRH			STM

### 数据传送类

MOV	MVN	PUSH	POP	ADR
-----	-----	------	-----	-----

### 算术、逻辑运算类

ADC	ADD	SBC	SUB	RSB
AND	ORR	EOR		MUL

### 比较运算类

CMP	CMN	TST
-----	-----	-----

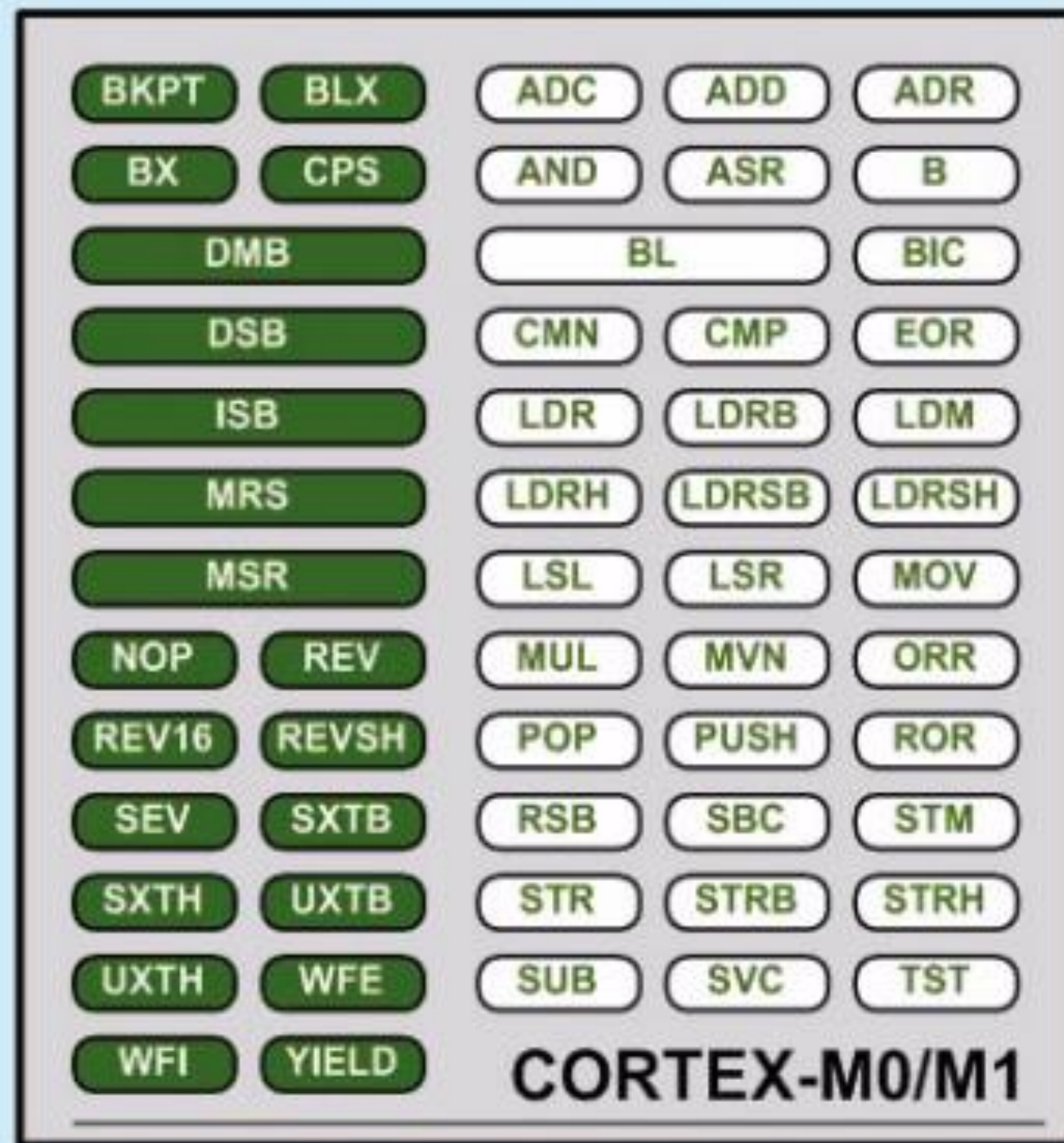
### 移位指令类

LSL	LSR	ROR	ASR	BIC
-----	-----	-----	-----	-----

### 分支类

B	BL	SVC
---	----	-----

为何选择M0指令集做介绍?



Thumb 指令集 1

助记符	Thumb ISA	描 述	助记符	Thumb ISA	描 述
ADC	V1	带进位 32 位加	LSR	V1	逻辑右移
ADD	V1	32 位加	MOV	V1	数据传送
AND	V1	32 位逻辑与	MUL	V1	乘法指令
ASR	V1	算术右移	MVN	V1	取反传送
B	V1	分支指令	NEG	V1	取反
BIC	V1	32 位逻辑位清除	ORR	V1	逻辑或运算
BKPT	V2	断点指令	POP	V1	退栈
BL	V1	带链接的相对分支指令	PUSH	V1	压栈
BLX	V2	带交换的分支指令	ROR	V1	循环右移
CMN	V1	32 位相反数比较	SBC	V1	带进位减法
CMP	V1	32 位比较	STM	V1	多寄存器存储
EOR	V1	32 位逻辑异或	STR	V1	单寄存器数据存储
LDM	V1	多寄存器加载	SUB	V1	减法
LDR	V1	单寄存器加载	SWI	V1	软中断
LSL	V1	逻辑左移	TST	V1	位测试

## 寻址方式

立即寻址  
寄存器寻址  
寄存器偏移寻址  
寄存器间接寻址  
基址寻址  
多寄存器寻址  
堆栈寻址  
块拷贝寻址

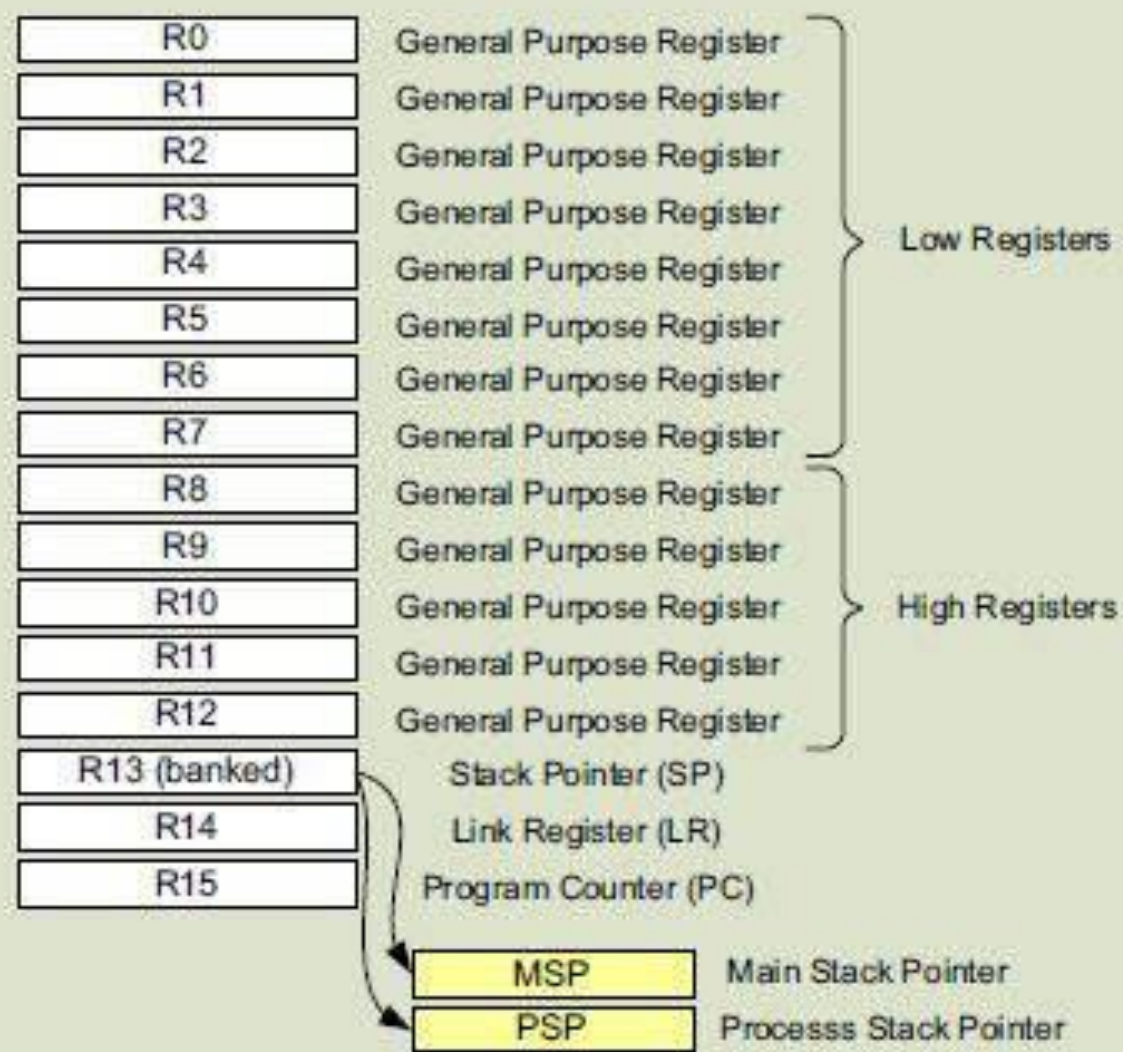
相对寻址

MOV	R0,	#0xFF00	; 0xFF00 -> R0
ADD	R0,	R0, #1	; R0 + 1 -> R0
SUB	R0,	R1, R2	; R1 - R2 -> R0
AND	R1,	R1, R2, LSL #3	; R2左移3位, 跟R1 与操作, 结果放入 R1
LDR	R1,	[R2]	; 将R2中的数值作为地址, 取出此地址中的数据保存在 R1 中
STR	R1,	[R0, #-2]	; 将R0中的数值减2作为地址, 把 R1的值保存到此地址中
LDMIA	R1!,	{R2-R7, R12}	; 将 R1的值读出到 R2-R7, R12, 过程中R1 自动加 1
STMFD	SP!,	{R1-R7, LR}	; 将 R1~R7, LR 入栈。满递减堆栈。
STMIA	R0!,	{R1-R7}	; 将R1~R7的数据保存到存储器中, 存储器指针在保存第一个值之后增加, 增长方向为向上增长
BL	ROUTE1		; 调用ROUTE1 子程序 -----(相对PC)

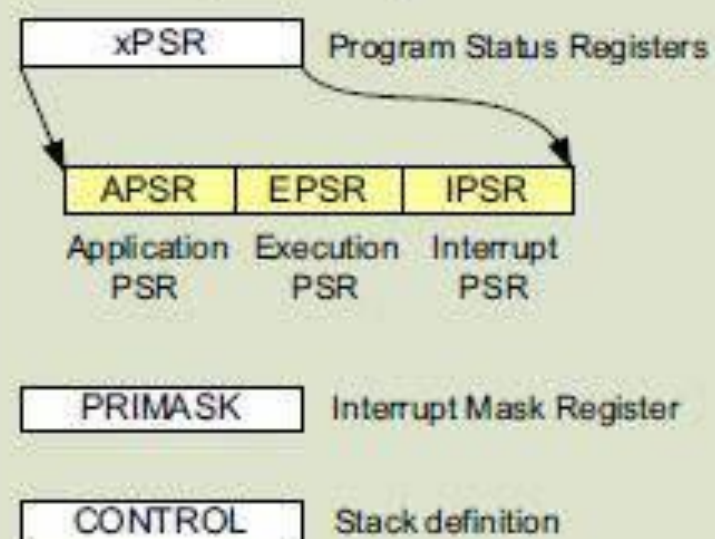
寄存器、地址  
是操作数的最常见形式

寻址方式是指令的重要组成部分  
也是指令多样性的因素

### Register bank



### Special Registers





## ARM的指令格式

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

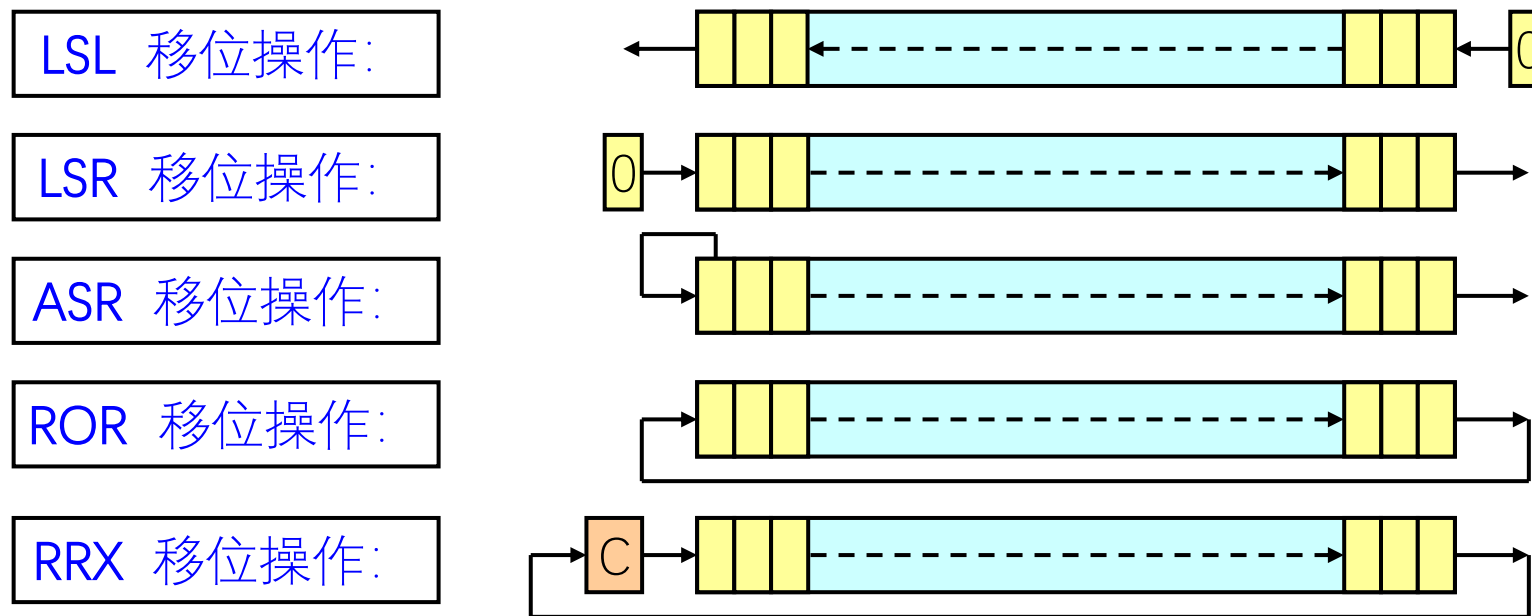
其中<>号内的项是必要的，{ }号是可选的

<b>opcode:</b>	指令助记符;
<b>cond:</b>	执行条件;
<b>S:</b>	是否影响CPSR寄存器的值;
<b>Rd:</b>	目标寄存器;
<b>Rn:</b>	第1个操作数的寄存器;
<b>operand2:</b>	第2个操作数;

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

■ 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值不变



**ADD     R1 , R1 , R1 , LSL #3     ; R1=R1+R1\*8=9R1**

**SUB     R1 , R1 , R2 , LSR R3     ; R1=R1-(R2/2<sup>R3</sup>)**



执行条件

cond:

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)

MOV[<cond>][s] <Rd>, <Op2>	数据传送	$Rd \leftarrow Op2$
MOV[{cond}]<Rd>, #imm16	数据传送	$Rd \leftarrow imm16$
MOVT[{cond}]<Rd>, #imm16	数据传送到顶部	Rd高8位 $\leftarrow imm16$
MVN[<cond>][s] <Rd>, <Op2>	数据取反传送	$Rd \leftarrow \sim Op2$
ADD[<cond>][s] <Rd>, <Rn>, <Op2>	加法运算	$Rd \leftarrow Rn + Op2$
ADC[<cond>][s] <Rd>, <Rn>, <Op2>	带进位加法运算	$Rd \leftarrow Rn + Op2 + C$
SUB[<cond>][s] <Rd>, <Rn>, <Op2>	减法运算	$Rd \leftarrow Rn - Op2$
SBC[<cond>][s] <Rd>, <Rn>, <Op2>	带借位减法运算	$Rd \leftarrow Rn - Op2 - NOT(C)$
RSB[<cond>][s] <Rd>, <Rn>, <Op2>	逆向减法运算	$Rd \leftarrow Op2 - Rn$
RSC[<cond>][s] <Rd>, <Rn>, <Op2>	带借位逆向减法运算	$Rd \leftarrow Op2 - Rn - NOT(C)$
CMP[<cond>] <Rd>, <Op2>	比较	影响标志NZCV $\leftarrow Rn - Op2$
CMN[<cond>] <Rd>, <Op2>	负数比较	影响标志NZCV $\leftarrow Rn + Op2$
AND[<cond>][s] <Rd>, <Rn>, <Op2>	逻辑与运算	$Rd \leftarrow Rn \&Op2$
ORR[<cond>][s] <Rd>, <Rn>, <Op2>	逻辑或运算	$Rd \leftarrow Rn  Op2$
EOR[<cond>][s] <Rd>, <Rn>, <Op2>	逻辑异或运算	$Rd \leftarrow Rn \wedge Op2$
BIC[<cond>][s] <Rd>, <Rn>, <Op2>	位清零	$Rd \leftarrow Rn \&(\sim Op2)$
TST[<cond>] <Rn>, <Op2>	位测试	影响标志NZCV $\leftarrow Rn \&Op2$
TEQ[<cond>] <Rn>, <Op2>	测试相等	影响标志NZCV $\leftarrow Rn - Op2$

## 指令编码格式参考

[illegible]



## 指令实例参考

Register	Value
<b>Core</b>	
R0	0x0000008A
R1	0x0000002A
R2	0x00000000
R3	0x200000E8
R4	0x20000000
R5	0x40011000
R6	0x00000000
R7	0x200000E8
R8	0x40010808
R9	0x0000000A
R10	0x08002970
R11	0x000000FF
R12	0x00000F00
R13 (SP)	0x20000590
R14 (LR)	0x0800291D
R15 (PC)	0x08002914
<b>xPSR</b>	0x01000000
N	0
Z	0
C	0
V	0
Q	0
T	1
IT	Disabled
ISR	0
<b>Banked</b>	
<b>System</b>	
<b>Internal</b>	
Mode	Thread
Privi...	Privileged
Stack	MSP
States	805432252
Sec	80.54322520

```

0x080028E8 F04F0BFF MOV    r11,#0xFF
0x080028EC 6067    STR    r7,[r4,#0x04]
149:          if (!(GPIOA->IDR&1))      i++;      else
0x080028EE F8D80000 LDR    r0,[r8,#0x00]
0x080028F2 07C0    LSL    r0,r0,#31
150:          {          i+=9;          }          //T1_5: 有UP键按下(P
151:          GPIOC->BSRR=LEDcd[i%10]&0xff;          //T1_1:  LE
0x080028F4 8820    LDRH    r0,[r4,#0x00]
0x080028F6 D01D    BEQ    0x08002934
0x080028F8 3009    ADDS    r0,r0,#0x09
0x080028FA B201    SXTH    r1,r0
0x080028FC FB91F2F9 SDIV    r2,r1,r9
0x08002900 FB091012 MLS    r0,r9,r2,r1
0x08002904 8020    STRH    r0,[r4,#0x00]
0x08002906 F81A1000 LDRB    r1,[r10,r0]
0x0800290A 6129    STR    r1,[r5,#0x10]
152:          Delay(500+i*23);          GPIOC->BRR=0xff;
153:  //-----
0x0800290C EBC001C0 RSB    r1,r0,r0,LSL #3
0x08002910 EB011000 ADD    r0,r1,r0,LSL #4
0x08002914 F50070FA ADD    r0,r0,#0x1F4
0x08002918 F7FD0CFE BL.W    Delay (0x08000318)
0x0800291C F8C5B014 STR    r11,[r5,#0x14]
154:          md.sBUF[i]=i*3+x;
0x08002920 F9B40000 LDRSH    r0,[r4,#0x00]
0x08002924 8862    LDRH    r2,[r4,#0x02]
0x08002926 EB000140 ADD    r1,r0,r0,LSL #1
0x0800292A 4411    ADD    r1,r1,r2
0x0800292C F8271010 STRH    r1,[r7,r0,LSL #1]
155:          if (i==0)          {          md.sBUF[14]=md.sBUF[15];          md.
0x08002930 B110    CBZ    r0,0x08002938
0x08002932 E004    B      0x0800293E
0x08002934 1C40    ADDS    r0,r0,#1
0x08002936 E7E0    B      0x080028FA
0x08002938 8BF9    LDRH    r1,[r7,#0x1E]
0x0800293A 83B9    STRH    r1,[r7,#0x1C]
0x0800293C 83FE    STRH    r6,[r7,#0x1E]
156:          md.sBUF[15]+=md.sBUF[i];
0x0800293E 8BF9    LDRH    r1,[r7,#0x1E]
0x08002940 F8370010 LDRH    r0,[r7,r0,LSL #1]
0x08002944 4408    ADD    r0,r0,r1
0x08002946 83F8    STRH    r0,[r7,#0x1E]
146:  while (1)
0x08002948 E7D1    B      0x080028EE

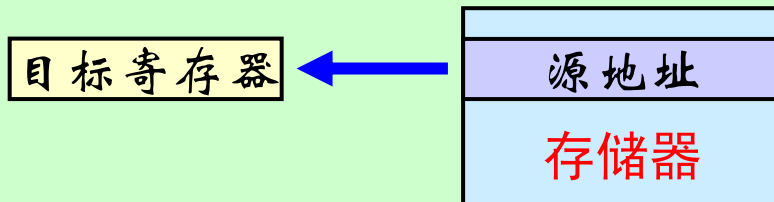
```



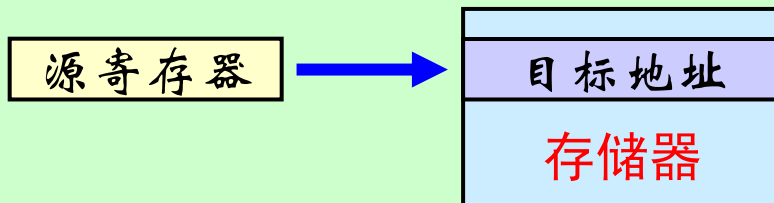
## 1. 存储器访问指令

- 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。
- 若使用LDR指令加载数据到PC寄存器，则实现程序跳转功能

装载(Load)指令: LDR 目标寄存器, 源地址



存储(Store)指令: STR 源寄存器, 目标地址



助记符	说明	条件码位置
LDR Rd,addressing	加载 <b>字</b> 数据	LDR {cond}
LDR <b>B</b> Rd,addressing	加载无符号 <b>字节</b> 数据	LDR {cond} B
LDR <b>T</b> Rd,addressing	以 <b>用户模式</b> 加载字数据	LDR {cond} T
LDR <b>BT</b> Rd, addressing	以 <b>用户模式</b> 加载 <b>无符号</b> 字节数据	LDR {cond} BT
LDR <b>H</b> Rd, addressing	加载 <b>无符号半字</b> 数据	LDR {cond} H
LDR <b>SB</b> Rd, addressing	加载 <b>有符号字节</b> 数据	LDR {cond} SB
LDR <b>SH</b> Rd, addressing	加载 <b>有符号半字</b> 数据	LDR {cond} SH

助记符	说明	条件码位置
STR Rd, addressing	存储 <b>字</b> 数据	STR {cond}
STR <b>B</b> Rd,addressing	存储 <b>字节</b> 数据	STR {cond} B
STR <b>T</b> Rd,addressing	以 <b>用户模式</b> 存储字数据	STR {cond} T
STR <b>BT</b> Rd,addressing	以 <b>用户模式</b> 存储 <b>字节</b> 数据	STR {cond} BT
STR <b>H</b> Rd,addressing	存储 <b>半字</b> 数据	STR {cond} H

装载指令: **LDR**      目标寄存器, 源地址

保存指令: **STR**      源寄存器, 目标地址

**立即数**: 立即数可以是一个无符号的数值。这个数据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。  
如: **LDR R1,[R0,#0x12]**

**寄存器**: 寄存器中的数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。  
如: **LDR R1,[R0,R2]**

**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。  
如: **LDR R1,[R0,R2,LSL #2]**

## LDR加载/STR存储指令可以有以下变化

- 零偏移                      如：LDR   Rd, [Rn]
- 程序相对偏移              如：LDR   Rd, label
- 前索引偏移                 如：LDR   Rd, [Rn, #0x04]!

将Rn+0x04内存地址中的数据加载到Rd中，然后 $Rn = Rn + 0x04$ ，如果没有“! ”，Rn的值将得不到更新。

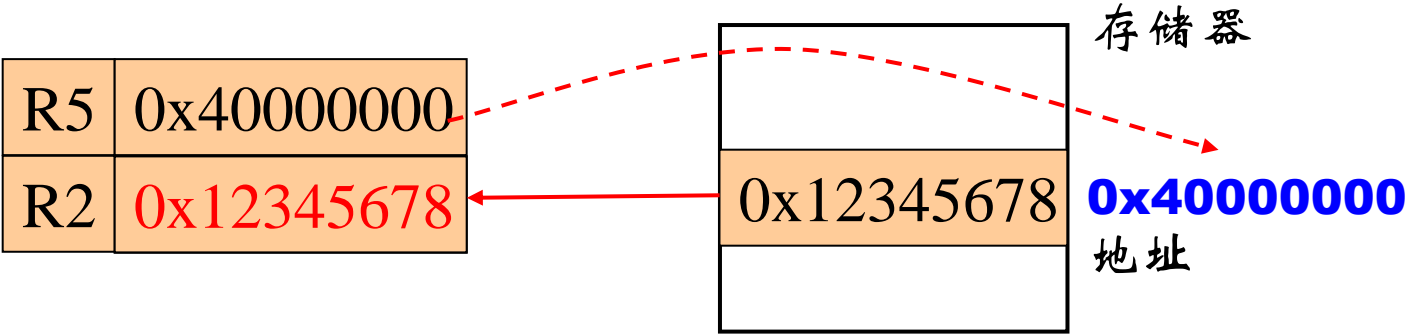
- 后索引偏移                 如：LDR   Rd, [Rn], #0x04

将Rn地址指向的内存中的数据加载到Rd中，然后 $Rn = Rn + 0x04$

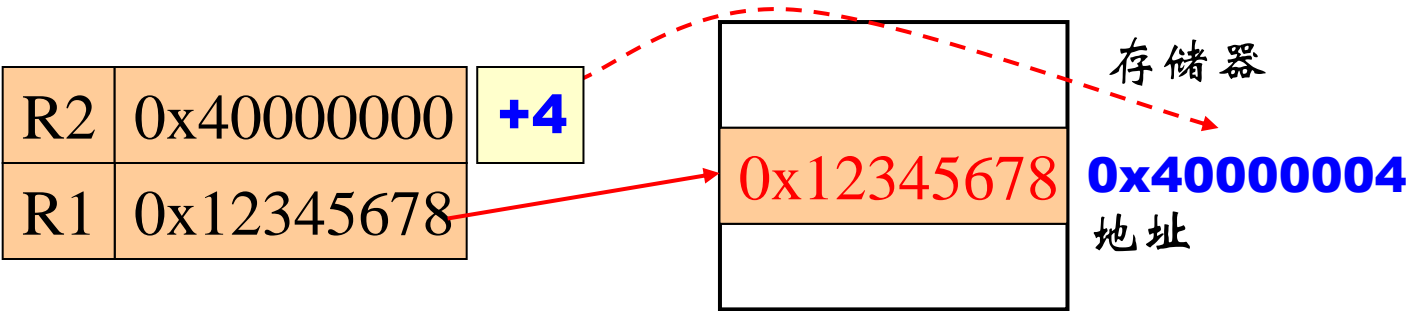


# 应用示例

**LDR R2,[R5]** ; 将R5指向地址的字数据存入R2



**STR R1,[R2,#0x04]** ; 将R1的数据存储到R2+0x04地址



## 应用示例

**LDRX**

**STRX**

```
156:                                md.sBUF[15] += md.sBUF[i];
```

```
157: //自行扩展Begin
```

```
0x08002940 8BF9          LDRH      r1, [r7, #0x1E]
```

```
0x08002942 F8373010    LDRH      r3, [r7, r0, LSL #1]
```

```
0x08002946 4419          ADD       r1, r1, r3
```

```
0x08002948 83F9          STRH      r1, [r7, #0x1E]
```

```
158: md.Cnt00 = (md.sBUF[12] | x) & (md.Cnt02 * md.sBUF[10]);
```

```
0x0800294A F9B71066    LDRSH    r1, [r7, #0x66]
```

```
0x0800294E F8B7C014    LDRH      r12, [r7, #0x14]
```

```
0x08002952 FB01F30C    MUL        r3, r1, r12
```

```
0x08002956 F8B7C018    LDRH      r12, [r7, #0x18]
```

```
0x0800295A EA4C0C02    ORR        r12, r12, r2
```

```
0x0800295E EA03030C    AND        r3, r3, r12
```

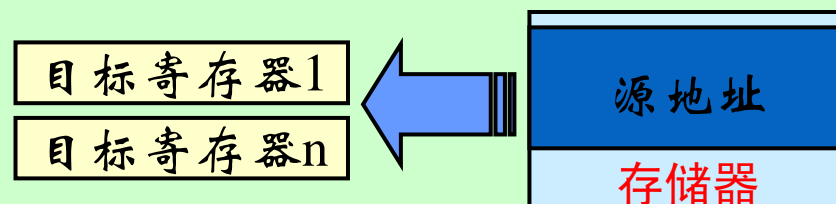
```
0x08002962 B21A          SXTB      r2, r3
```

```
0x08002964 F8A72062    STRH      r2, [r7, #0x62]
```

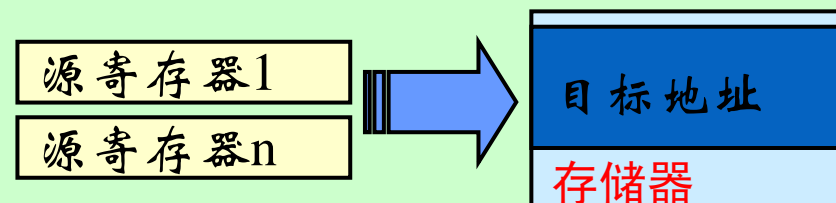
## 1.1 存储器访问指令-多寄存器

- 多寄存器加载/存储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；
- 允许一条指令传送16个寄存器的任何子集或所有寄存器；
- 主要用于现场保护、数据复制、常数传递等。

装载指令: **LDMx** 源地址,目标寄存器列表



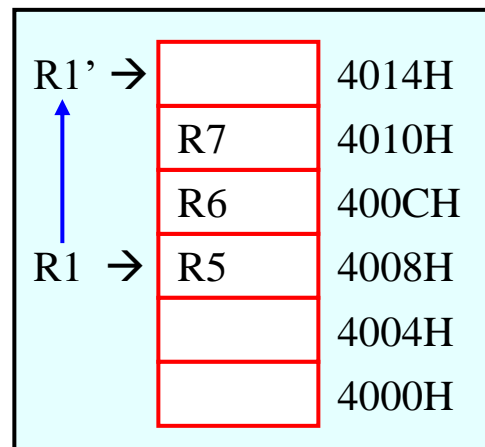
存储指令: **STMx** 目标地址,源寄存器列表



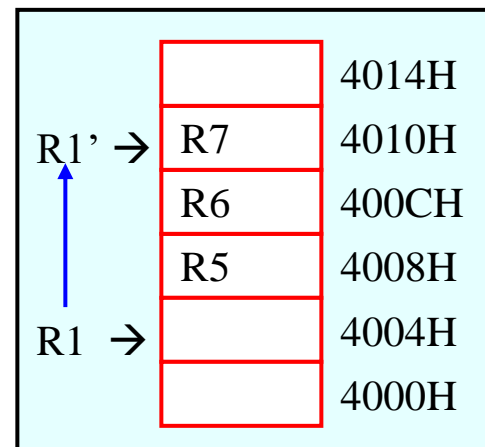
**IA:** 每次传送后地址加4  
**IB:** 每次传送前地址加4  
**DA:** 每次传送后地址减4  
**DB:** 每次传送前地址减4

## -多寄存器

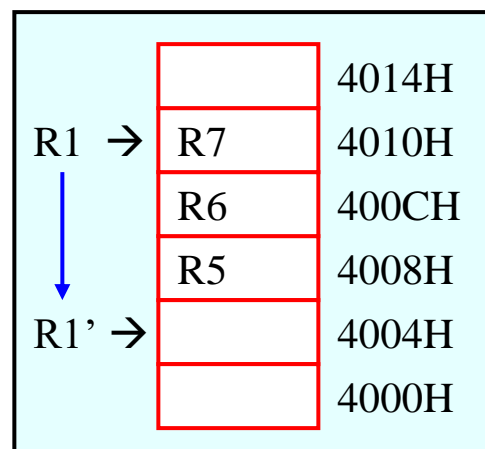
数据块传送指令操作过程如右图所示，其中R1为指令执行前的基址寄存器，R1'则为指令执行后的基址寄存器。



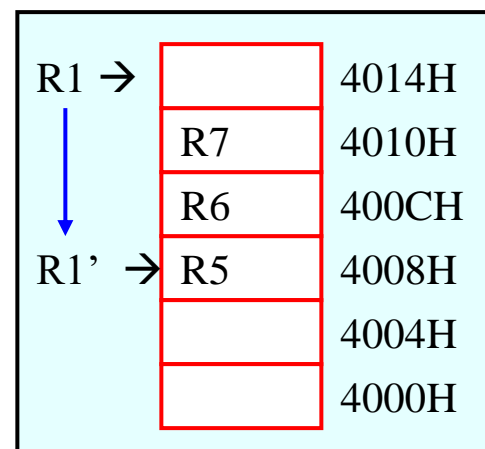
指令STM**IA** R1!,{R5-R7}



指令STM**IB** R1!,{R5-R7}



指令STM**DA** R1!,{R5-R7}



指令STM**DB** R1!,{R5-R7}



# -多寄存器

## 选项

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	

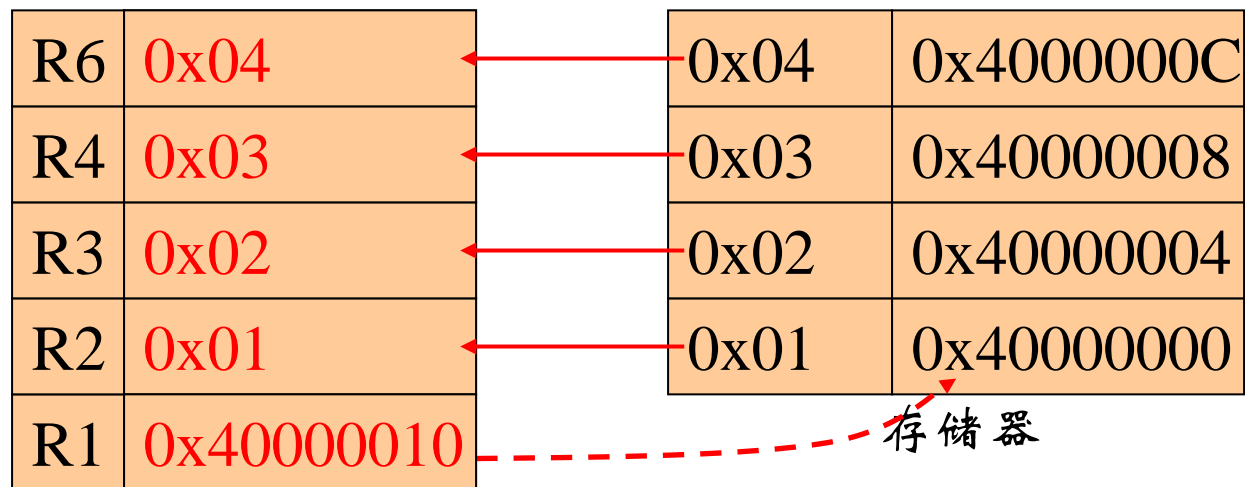
数据块传送 存储	堆栈操作 压栈	说明
STM <b>DA</b>	STM <b>ED</b>	空递减
STM <b>IA</b>	STM <b>EA</b>	空递增
STM <b>DB</b>	STM <b>FD</b>	满递减
STM <b>IB</b>	STM <b>FA</b>	满递增

数据块传送 加载	堆栈操作 出栈	说明
LDM <b>DA</b>	LDM <b>FA</b>	满递减
LDM <b>IA</b>	LDM <b>FD</b>	满递增
LDM <b>DB</b>	LDM <b>EA</b>	空递减
LDM <b>IB</b>	LDM <b>ED</b>	空递增

## 应用示例

### LDMIA R1!, {R2-R4,R6}

将R1指向的内存数据读取到R2-R4和R6寄存器中

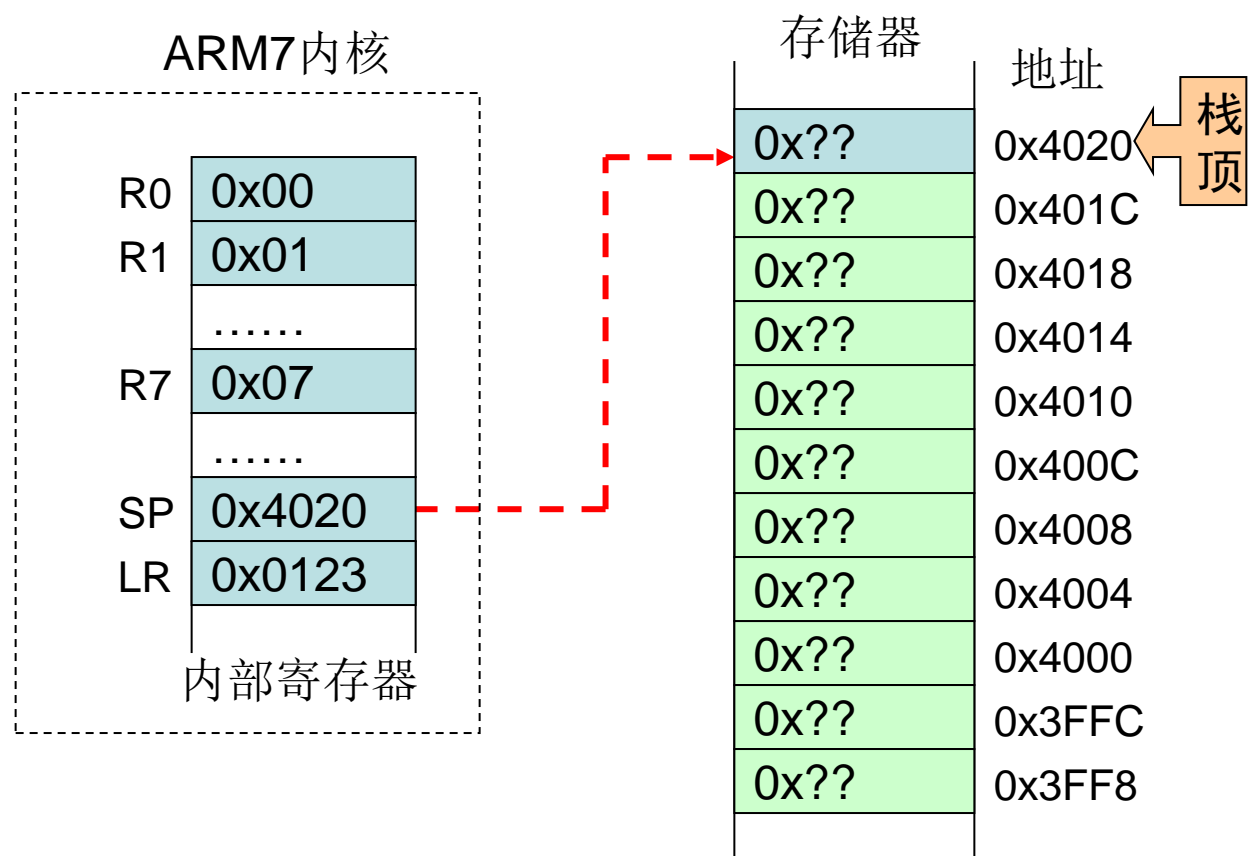


# 应用示例--满递减压栈操作

## STMFD SP!,{R0-R7,LR}

1.压栈操作前寄存器和堆栈区的状态;

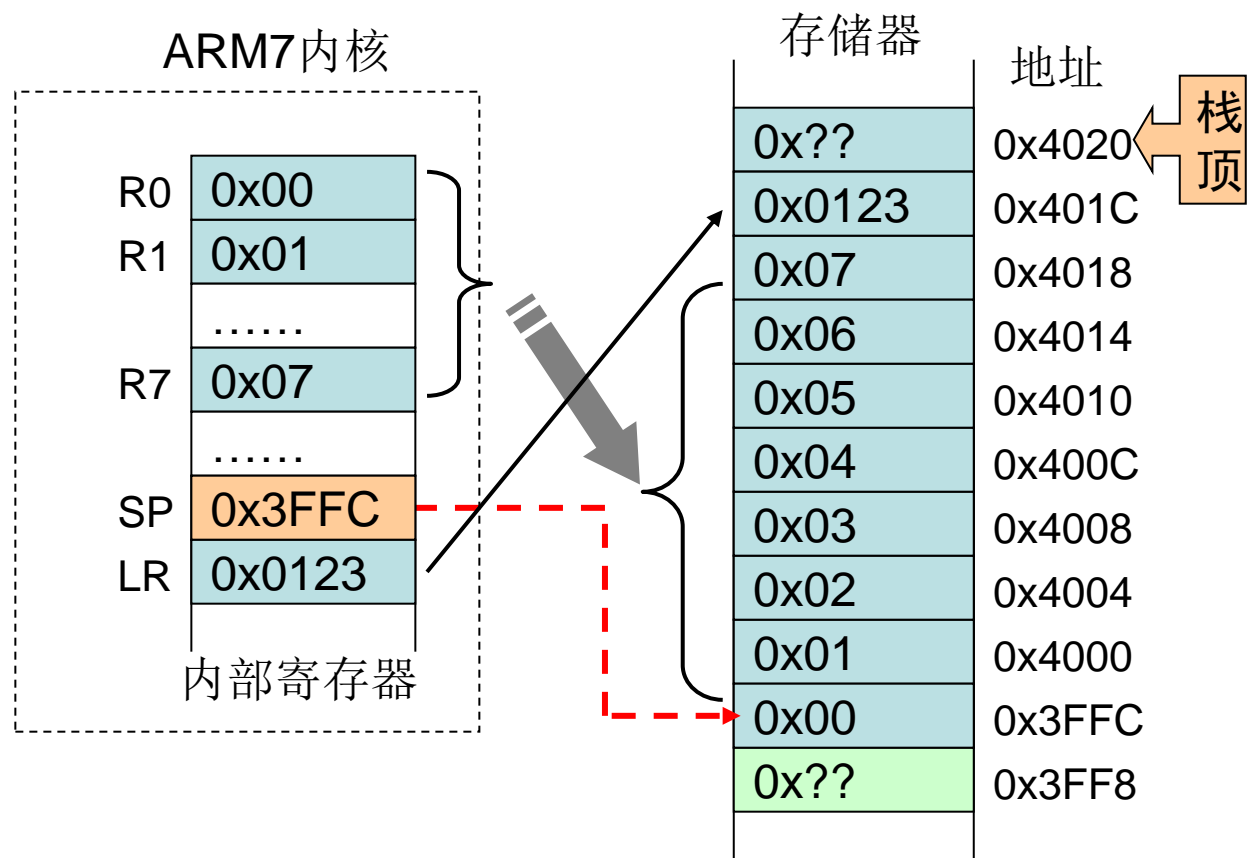
2.压栈操作前堆栈指针指向栈顶;



## 应用示例--满递减压栈操作:

**STMFD SP!,{R0-R7,LR}**

- 1.压栈操作前寄存器和堆栈区的状态;
- 2.压栈操作前堆栈指针指向栈顶;
- 3.执行压栈操作指令保存R0-R7和LR



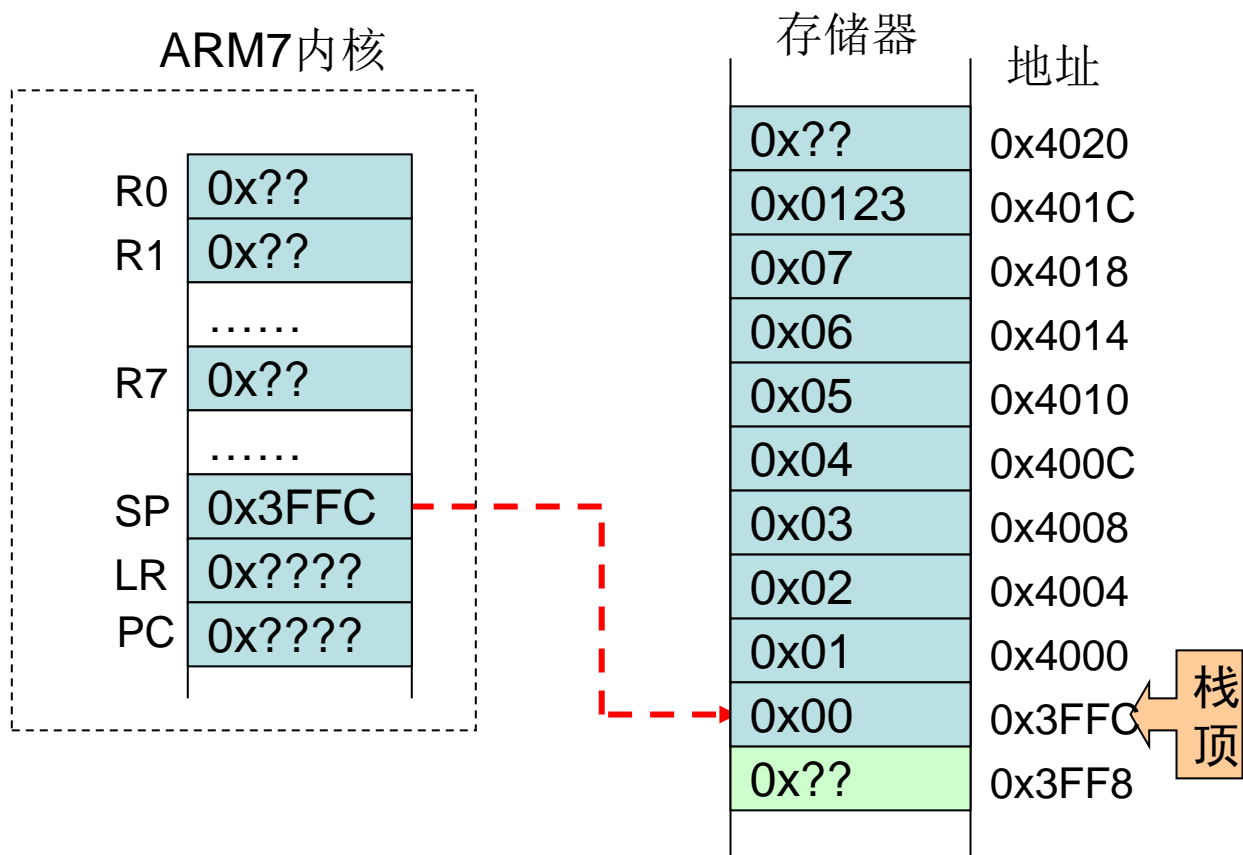


应用示例--满递减出栈操作：

**LDMFD SP!,{R0-R7,PC}**

1.出栈操作前寄存器和堆栈区的状态；

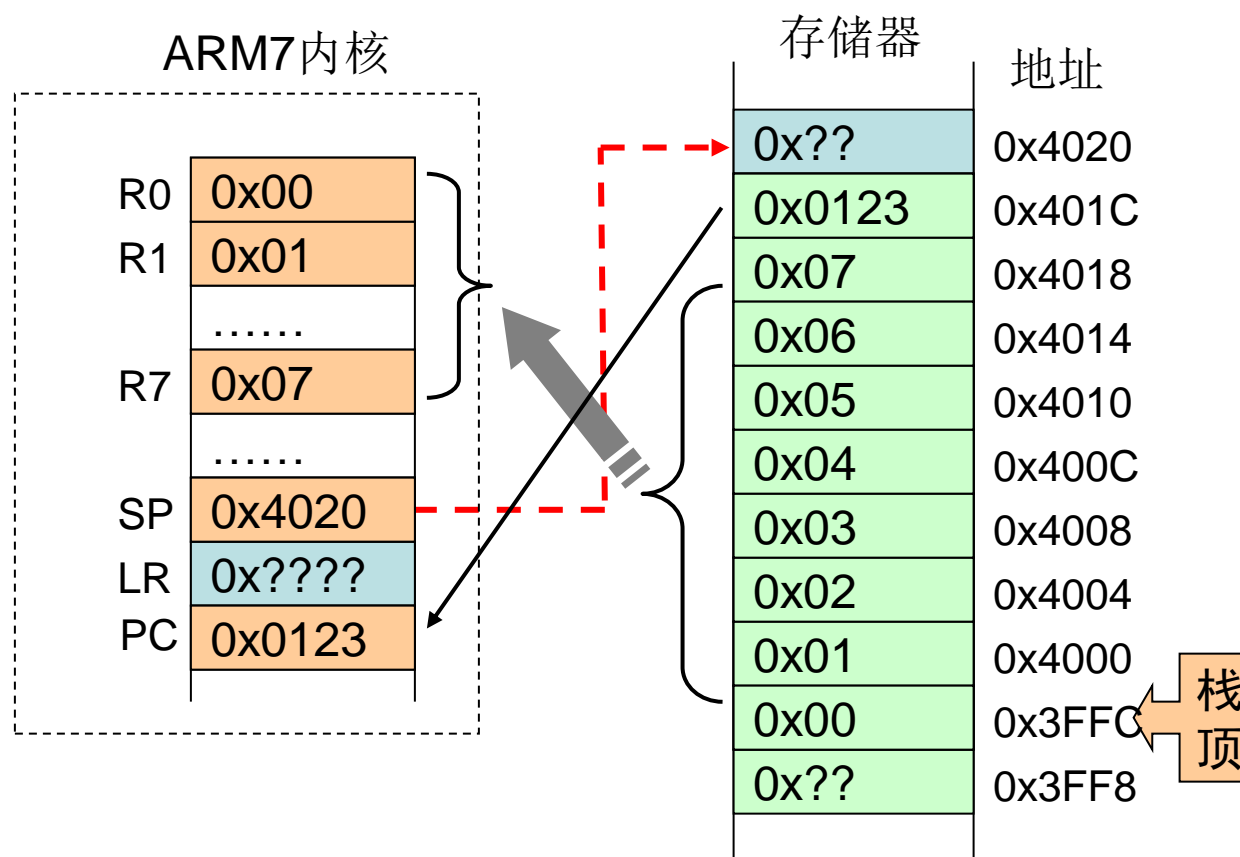
2.出栈操作前堆栈指针指向栈顶；



## 应用示例--满递减出栈操作：

**LDMFD SP!,{R0-R7,PC}**

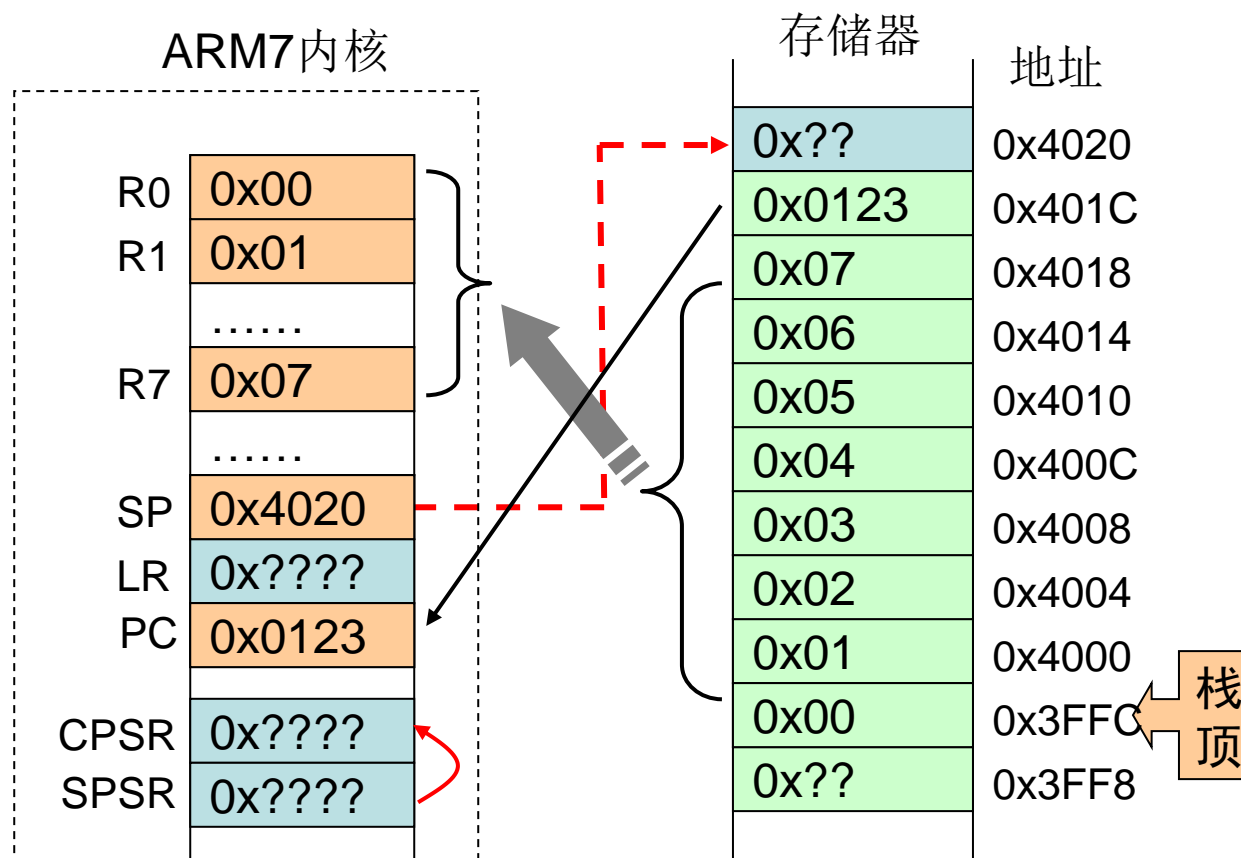
- 1.出栈操作前寄存器和堆栈区的状态；
- 2.出栈操作前堆栈指针指向栈顶；
- 3.执行出栈操作指令恢复R0-R7和PC



## 带状态寄存器恢复的出栈操作：

**LDMFD SP!,{R0-R7,PC}^**

1. 出栈操作前寄存器和堆栈区的状态；
2. 出栈操作前堆栈指针指向栈顶；
3. 执行出栈操作指令恢复R0-R7和PC



## 2 数据传送指令

数据处理指令只能对寄存器进行操作，不能对内存数据进行操作。  
数据处理指令均可选择S后缀，并影响状态标志。

**MOV** 将8位立即数或寄存器传送到目标寄存器Rd

**MOV** 目标寄存器, 操作数

目标寄存器

操作数



**MVN** 目标寄存器, 操作数

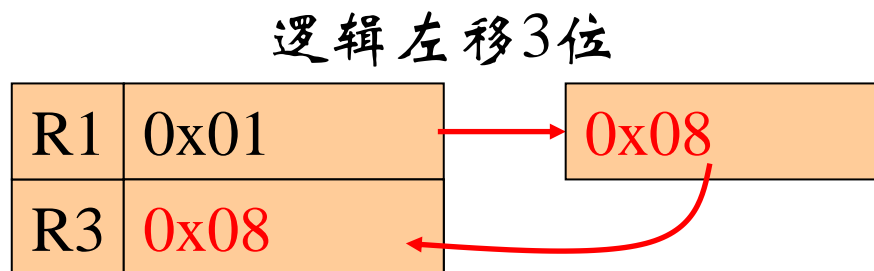
目标寄存器

操作数取反



应用示例：

**MOV R3,R1,LSL #3** ;R3=R1×8

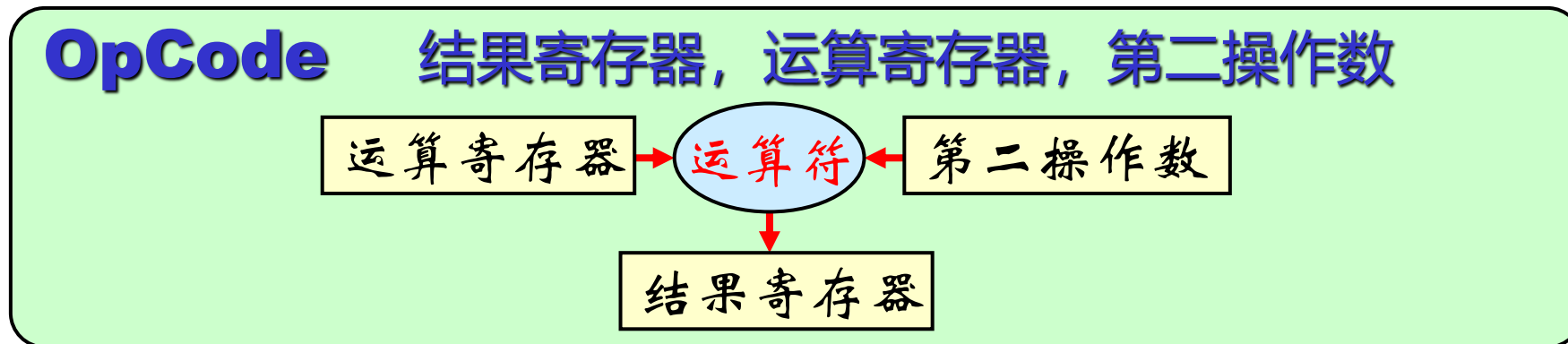


MOV指令用于将寄存器的值或常数传送到另一个寄存器中，不能访问内存。

LDR指令用于从内存中读取数据放入寄存器中。

### 3 算术逻辑运算指令

包括“加/减” 及“与/或/异或”等指令，格式如下：



部分**算术**运算符：

**ADD**：加法运算

**ADC**：带进位加法运算

**SUB**：减法运算

**RSB**：逆向减法运算

**SBC**：带进位减法运算

**RSC**：带进位逆向减法运算

部分**逻辑**运算符：

**AND**：逻辑“与”运算

**ORR**：逻辑“或”运算

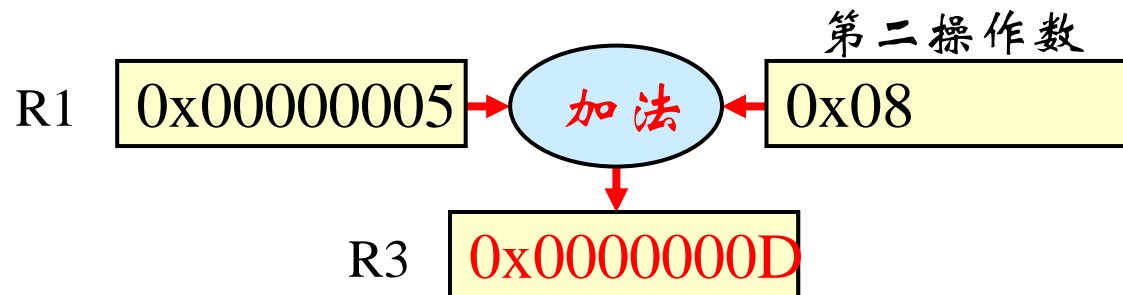
**EOR**：逻辑“异或”运算

**BIC**：位清除运算

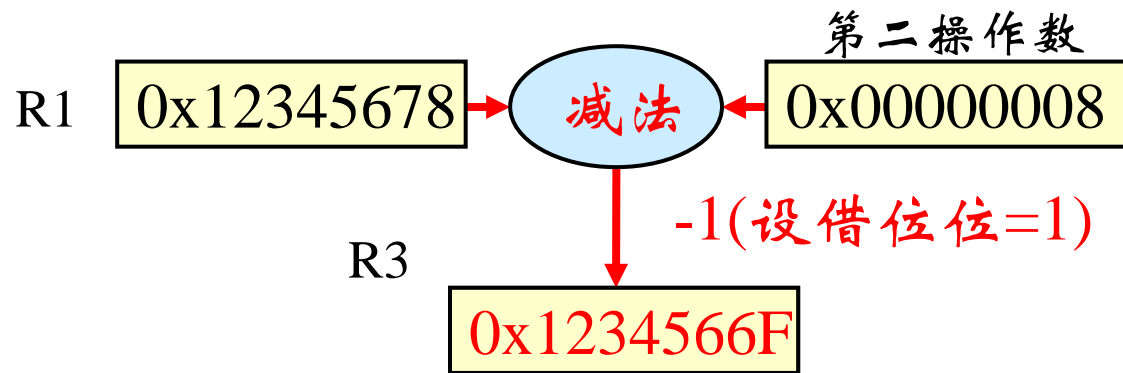


应用示例：

**ADD R3,R1, #0x08** ;R3=R1+8

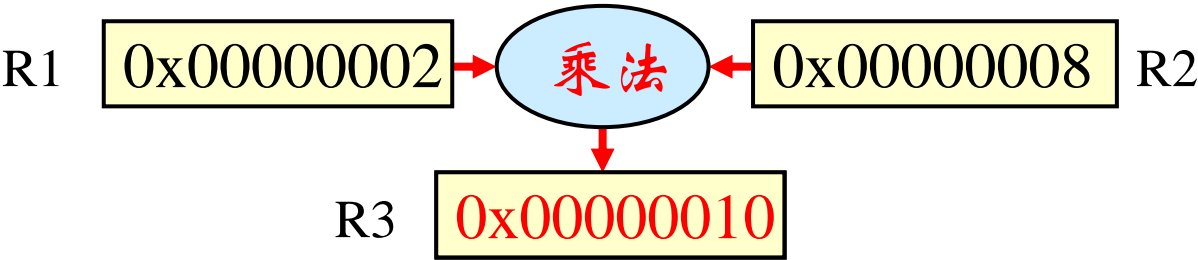


**SBC R3,R1, #0x08** ;R3=R1 - 0x00000008 - ! carry



乘法 MUL

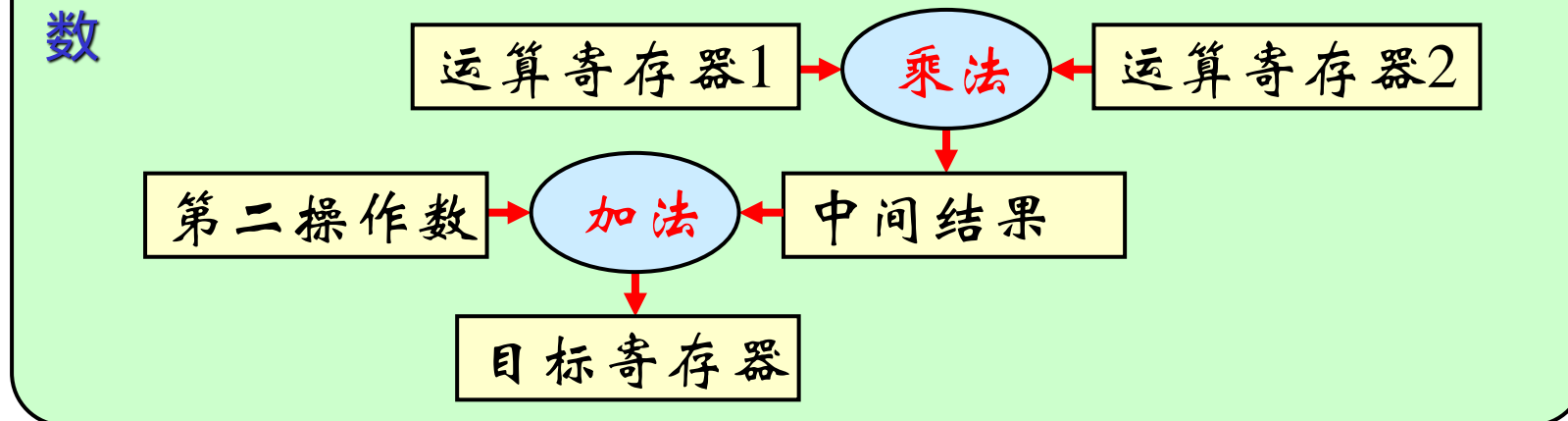
MUL R3,R2,R1 ; R3=R2×R1



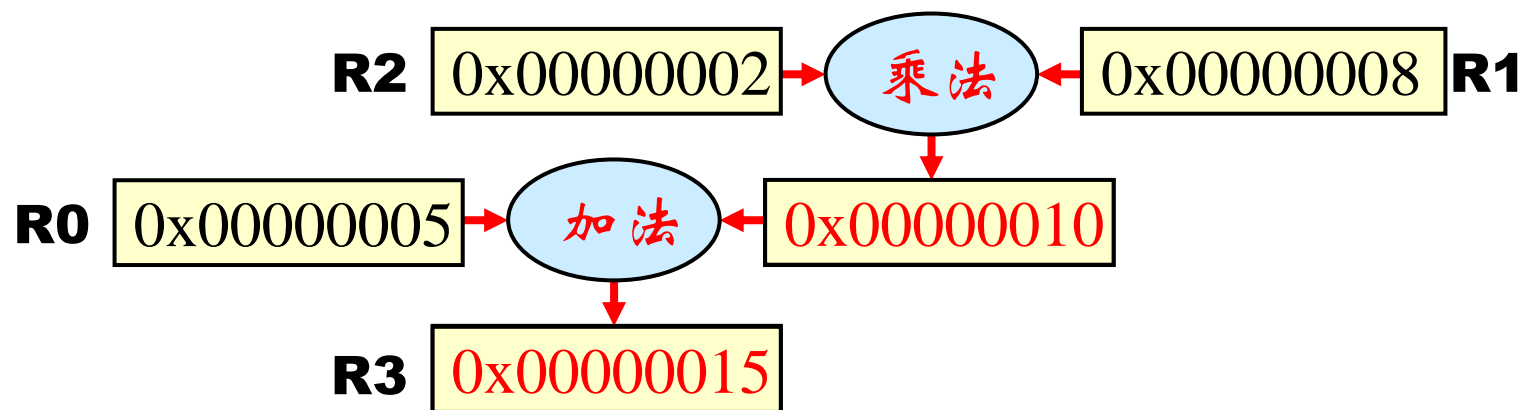
助记符	说明	操作	条件码位置
MUL Rd,Rm,Rs	32位乘法指令	$Rd \leftarrow Rm * Rs$ ( $Rd \neq Rm$ )	MUL {cond} {S}
MLA Rd,Rm,Rs,Rn	32位乘加指令	$Rd \leftarrow Rm * Rs + Rn$ ( $Rd \neq Rm$ )	MLA {cond} {S}
UMULL RdLo,RdHi,Rm,Rs	64位无符号乘法指令	$(RdLo, RdHi) \leftarrow Rm * Rs$	UMULL {cond} {S}
UMLAL RdLo,RdHi,Rm,Rs	64位无符号乘加指令	$(RdLo,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

## 乘加指令MLA

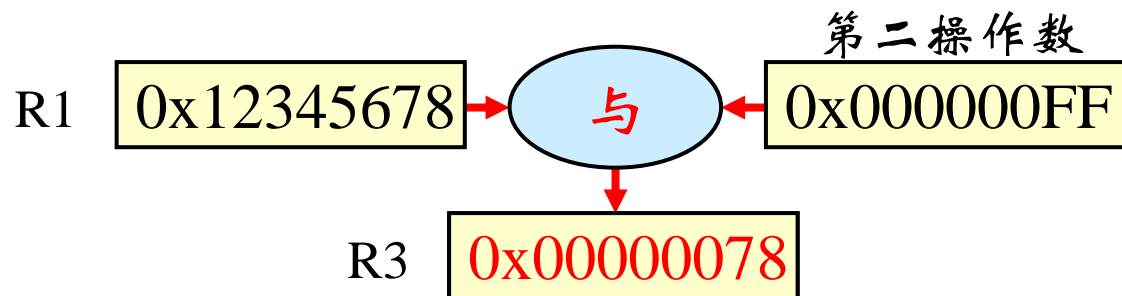
**MLA** 目标寄存器, 运算寄存器1, 运算寄存器2, 第二操作数



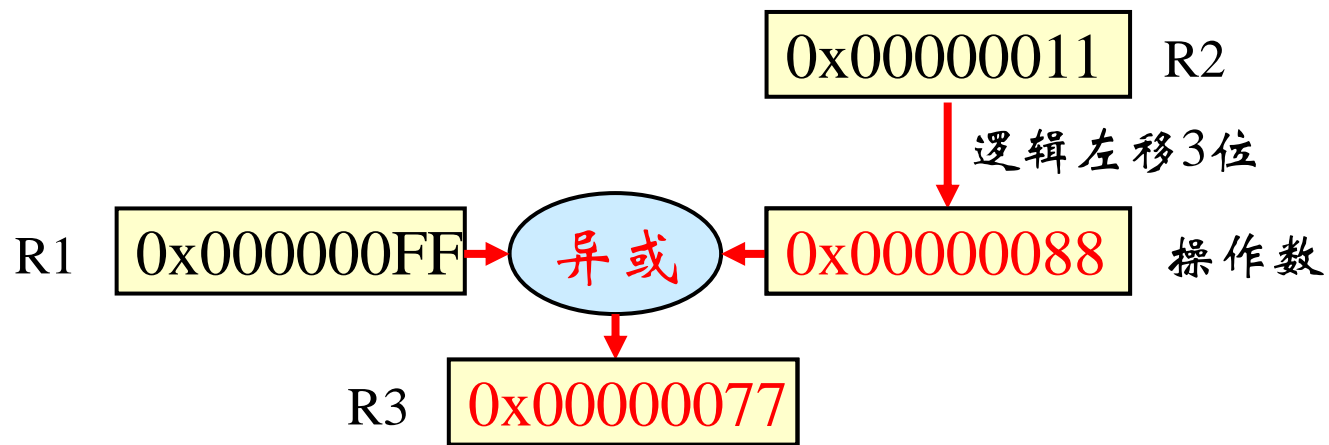
**MLA R3,R2,R1,R0** ;  $R3 = R2 \times R1 + R0$



**AND R3,R1, #0xFF** ;R3=R1 & 0x000000FF



**EOR R3,R1, R2,LSL 0x03** ;R3=R1 ^ (R2 × 8)

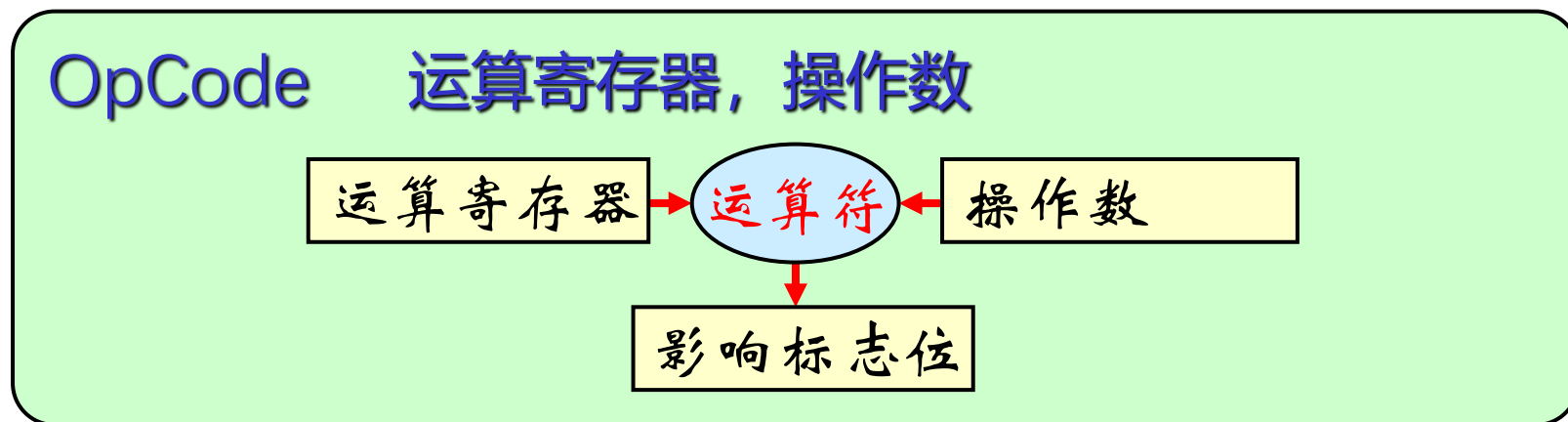


## 算术逻辑运算参考

```
156:                md.sBUF[15] += md.sBUF[i];
157: //自行扩展Begin
0x08002940 8BF9      LDRH      r1, [r7, #0x1E]
0x08002942 F8373010  LDRH      r3, [r7, r0, LSL #1]
0x08002946 4419      ADD      r1, r1, r3
0x08002948 83F9      STRH      r1, [r7, #0x1E]
158: md.Cnt00 = (md.sBUF[12] | x) & (md.Cnt02 * md.sBUF[10]);
0x0800294A F9B71066  LDRSH      r1, [r7, #0x66]
0x0800294E F8B7C014  LDRH      r12, [r7, #0x14]
0x08002952 FB01F30C  MUL      r3, r1, r12
0x08002956 F8B7C018  LDRH      r12, [r7, #0x18]
0x0800295A EA4C0C02  ORR      r12, r12, r2
0x0800295E EA03030C  AND      r3, r3, r12
0x08002962 B21A      SXTH      r2, r3
0x08002964 F8A72062  STRH      r2, [r7, #0x62]
```

## 4. 比较指令

**比较指令**将两个数值进行特定运算，根据运算结果影响CPSR相关标志位，用于后面程序的条件执行，但是运算结果不予保存。



比较运算符：

**CMP**：数值比较

**TST**：位测试

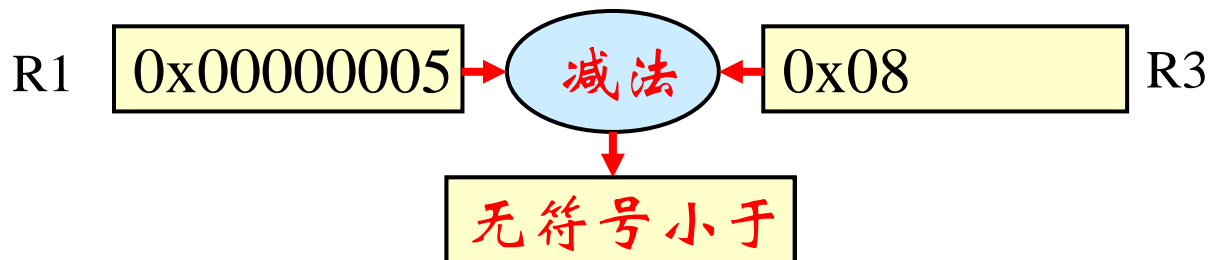
**CMN**：负数比较

**TEQ**：相等测试

应用示例：

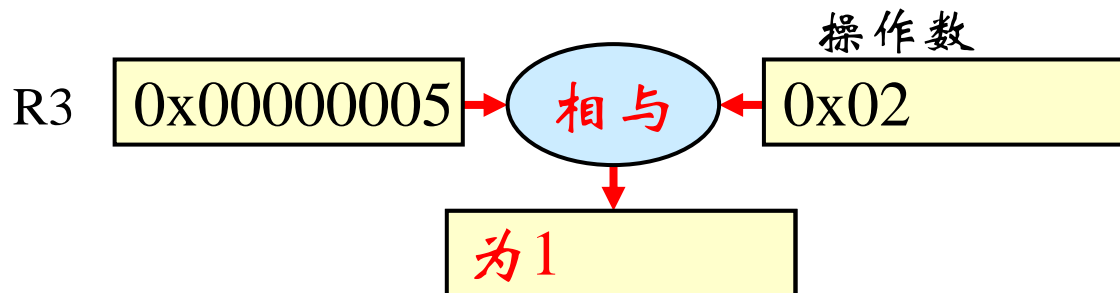
**CMP R3,R1**

;R3减R1并影响标志位



**TST R3,#0x02**

;测试R3的bit\_2是否为0并影响标志位





## CMP

## 示例参考

```
159: if(md.Cnt00<i|| i<0)                md.Cnt02*=5;           else    md.Cnt02/=3;
```

0x08002968	4282	CMP	r2,r0
0x0800296A	DB01	BLT	0x08002970
0x0800296C	2800	CMP	r0,#0x00
0x0800296E	DA02	BGE	0x08002976
0x08002970	EB010081	ADD	r0,r1,r1,LSL #2
0x08002974	E002	B	0x0800297C
0x08002976	2003	MOVS	r0,#0x03
0x08002978	FB91F0F0	SDIV	r0,r1,r0
0x0800297C	F8A70066	STRH	r0,[r7,#0x66]
0x08002980	E7B5	B	0x080028EE
0x08002982	0000	DCW	0x0000

## 5. 分支类指令 B (Branch)

1. 使用分支指令跳转
2. 直接向PC寄存器赋值实现跳转

例：            MOV            PC, R14

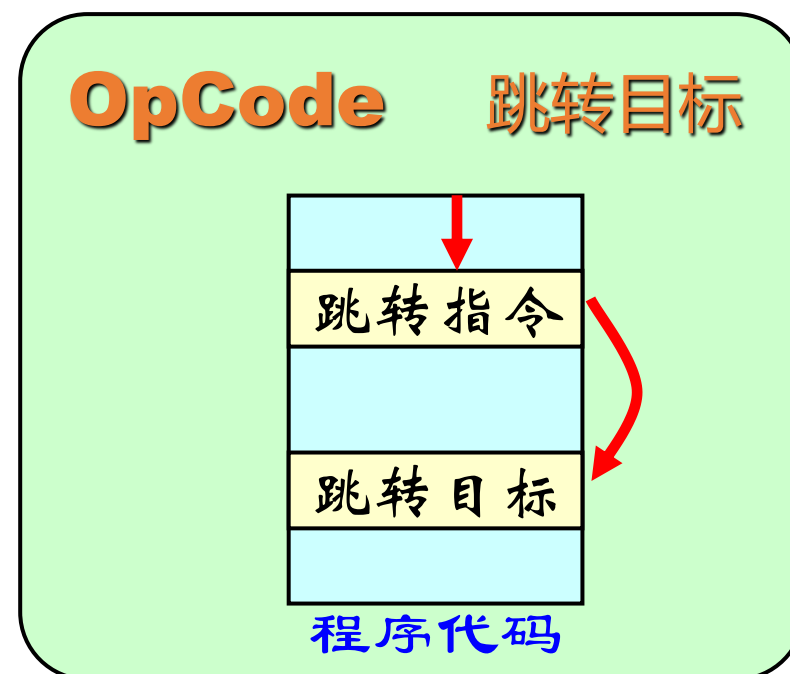
分支指令种类：

**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

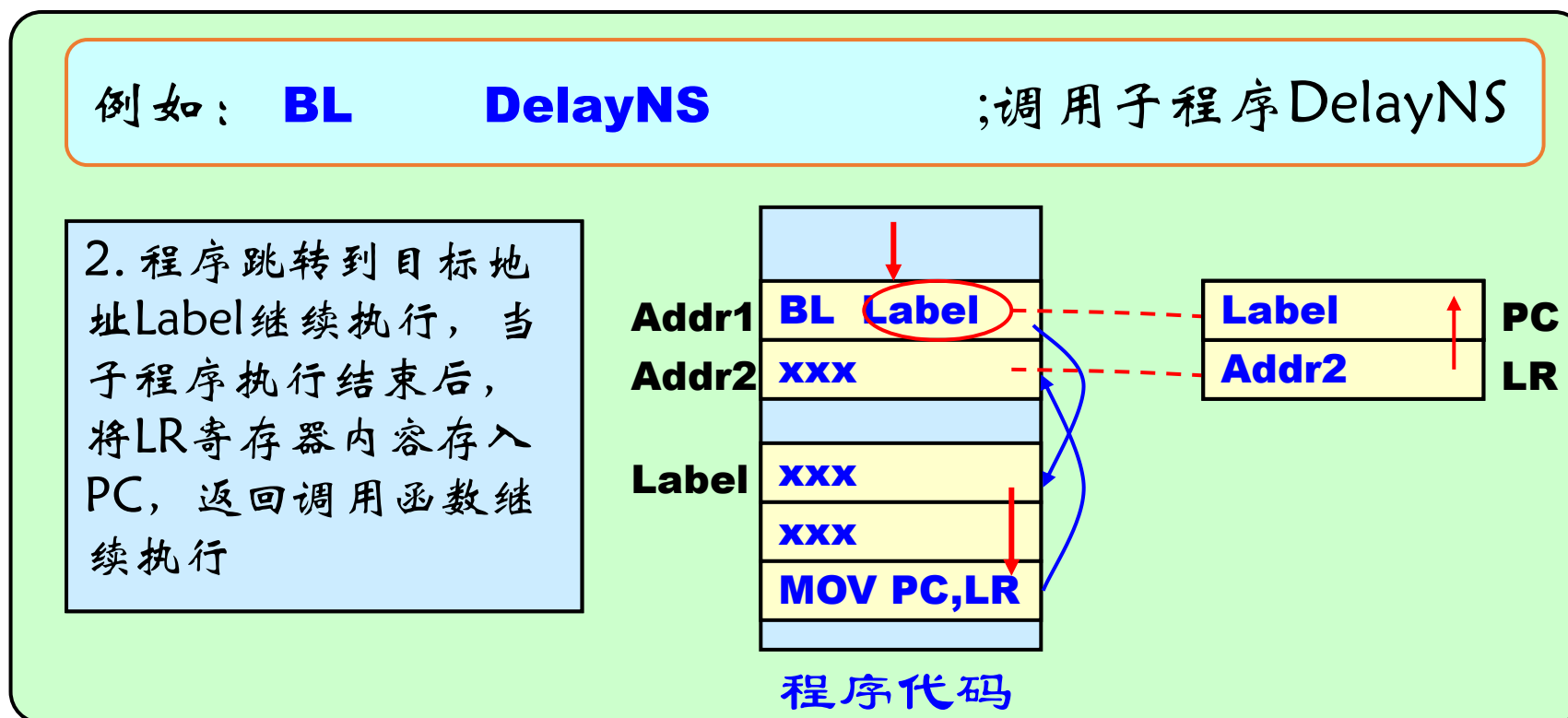
分支类指令是程序流程控制的重要内容  
条件、循环



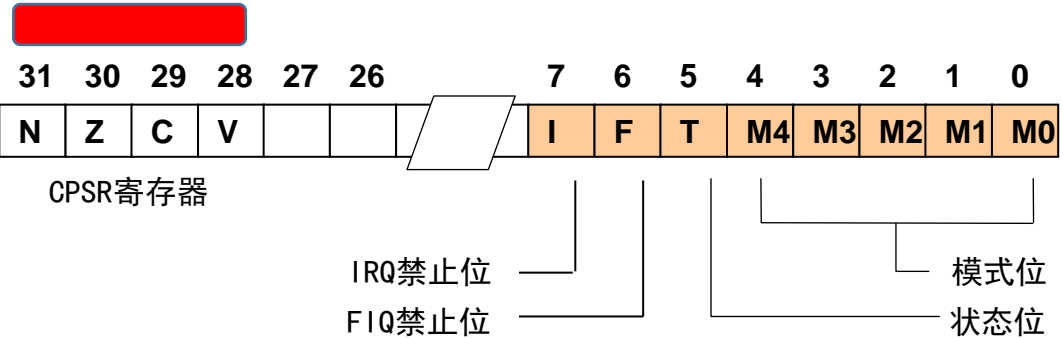
## BL指令

除了具有跳转功能，还能在跳转之前将下一条指令的地址拷贝到R14(LR)链接寄存器中，它适用于子程序调用。

示意如下：



# 分支指令 Bzz



```
149:      if (!(GPIOA->IDR&1))      i++;
0x080028EE F8D80000 LDR        r0,[r8,#0x00]
0x080028F2 07C0    LSLS        r0,r0,#31
150:      {                          }
151:      GPIOC->BSRR=LEDcd[i%10]&0xff;
0x080028F4 8820    LDRH        r0,[r4,#0x00]
0x080028F6 D01E    BEQ        0x08002936
0x080028F8 3009    ADDS        r0,r0,#0x09
0x080028FA B201    SXTB        r1,r0
0x080028FC FB91F2F9 SDIV        r2,r1,r9
0x08002900 FB091012 MLS        r0,r9,r2,r1
0x08002904 8020    STRH        r0,[r4,#0x00]
0x08002906 F81A1000 LDRB        r1,[r10,r0]
0x0800290A 6129    STR        r1,[r5,#0x10]
```

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)

分支指令  
Bxx  
典型形式

157: //自行扩展Begin				
0x08002940	8BF9	LDRH	r1, [r7, #0x1E]	
0x08002942	F8373010	LDRH	r3, [r7, r0, LSL #1]	
0x08002946	4419	ADD	r1, r1, r3	
0x08002948	83F9	STRH	r1, [r7, #0x1E]	
158: md.Cnt00=(md.sBUF[12] x)&(md.Cnt02 * md.sBUF[10]);				
0x0800294A	F9B71066	LDRSH	r1, [r7, #0x66]	
0x0800294E	F8B7C014	LDRH	r12, [r7, #0x14]	
0x08002952	FB01F30C	MUL	r3, r1, r12	
0x08002956	F8B7C018	LDRH	r12, [r7, #0x18]	
0x0800295A	EA4C0C02	ORR	r12, r12, r2	
0x0800295E	EA03030C	AND	r3, r3, r12	
0x08002962	B21A	SXTH	r2, r3	
0x08002964	F8A72062	STRH	r2, [r7, #0x62]	
159: if (md.Cnt00<i   i<0) md.Cnt02*=5; else md.Cnt02/=3;				
0x08002968	4282	CMP	r2, r0	
0x0800296A	DB01	BLT	0x08002970	
0x0800296C	2800	CMP	r0, #0x00	
0x0800296E	DA02	BGE	0x08002976	
0x08002970	EB010081	ADD	r0, r1, r1, LSL #2	
0x08002974	E002	B	0x0800297C	
0x08002976	2003	MOVS	r0, #0x03	
0x08002978	FB91F0F0	SDIV	r0, r1, r0	
0x0800297C	F8A70066	STRH	r0, [r7, #0x66]	
0x08002980	E7B5	B	0x080028EE	

## 6 地址读取ADR

;查表应用示例:

**ADR R0,DISP\_TAB** ;加载转换表地址

**LDRB R1,[R0,R2]** ;使用R2作为参数, 进行查表

...

**DISP\_TAB**

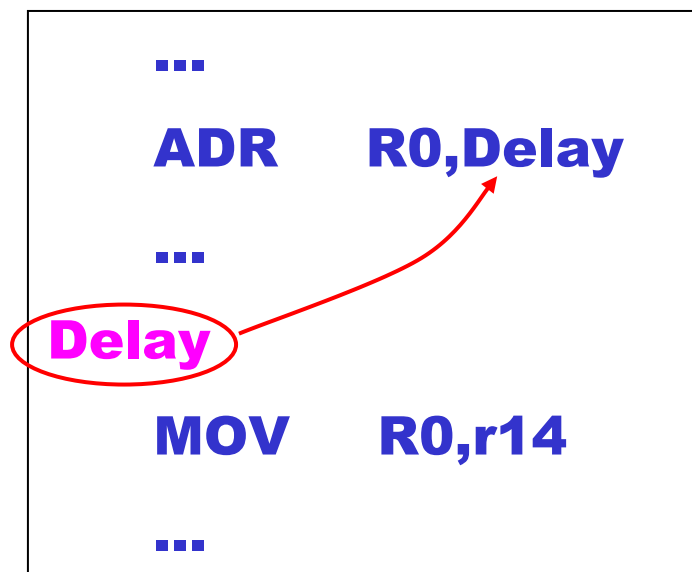
**DCB 0xC0,0xF9,0xA4,0xB0,0x99, 0x92,0x82,0xF8**



## 地址读取ADR

应用示例（源程序）：

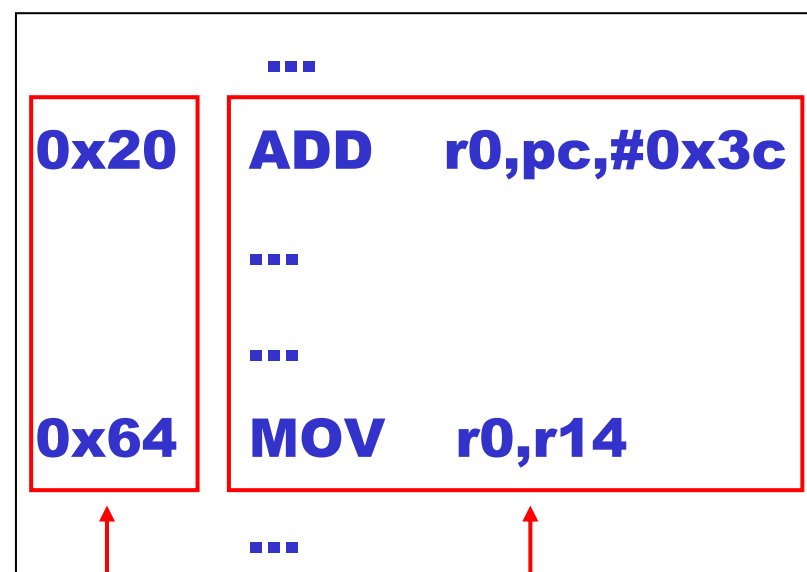
```
...  
ADR    R0,Delay  
...  
Delay  
MOV    R0,r14  
...
```



使用伪指令将程序标号Delay的地址存入R0

编译后的反汇编代码：

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

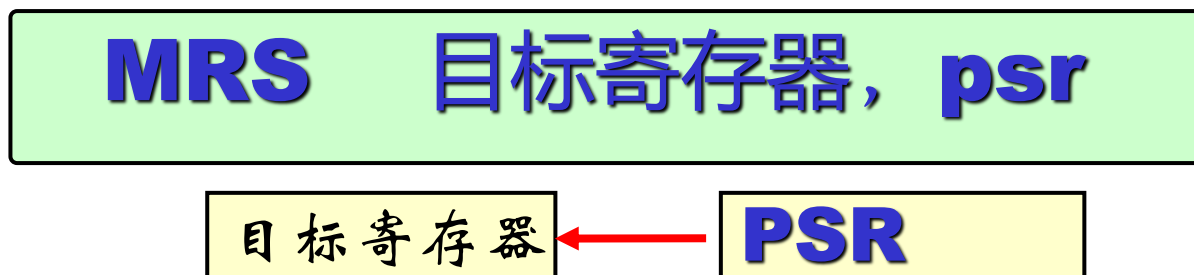


地址

程序代码

## 7. 杂项指令-状态寄存器读指令MRS

在ARM中，只有MRS指令可以对状态寄存器CPSR和SPSR进行读操作。通过读CPSR可以了解当前处理器的工作状态。读SPSR寄存器可以了解到进入异常前的处理器状态。指令格式如下所示：



示例：

**MRS R1,CPSR** ; 读取CPSR状态寄存器到R1

**MRS R2,SPSR** ; 读取SPSR状态寄存器到R2

## 杂项指令-状态寄存器写指令MSR

在ARM中，只有MSR指令可以对状态寄存器CPSR和SPSR进行写操作。与MRS配合使用，可以实现对CPSR或SPSR寄存器的读-修改-写操作，可以切换处理器模式等操作。

**MSR**      **psr\_field, 操作数**

PSR寄存器被分为四个8位的域：

1. 状态位域：用 **s** 表示
2. 扩展位域：用 **x** 表示
3. 条件标志位域：用 **f** 表示
4. 控制位域：用 **c** 表示

操作数分为两种：

1. 寄存器
2. 8位图立即数

示例：将R0的内容写入CPSR寄存器的控制位域

**MSR CPSR\_c, R0**

启动代码堆栈初始化应用示例：

```
INITSTACK
```

```
MOV R0, LR
```

;设置管理模式堆栈

```
MSR CPSR_C, #0xD3
```

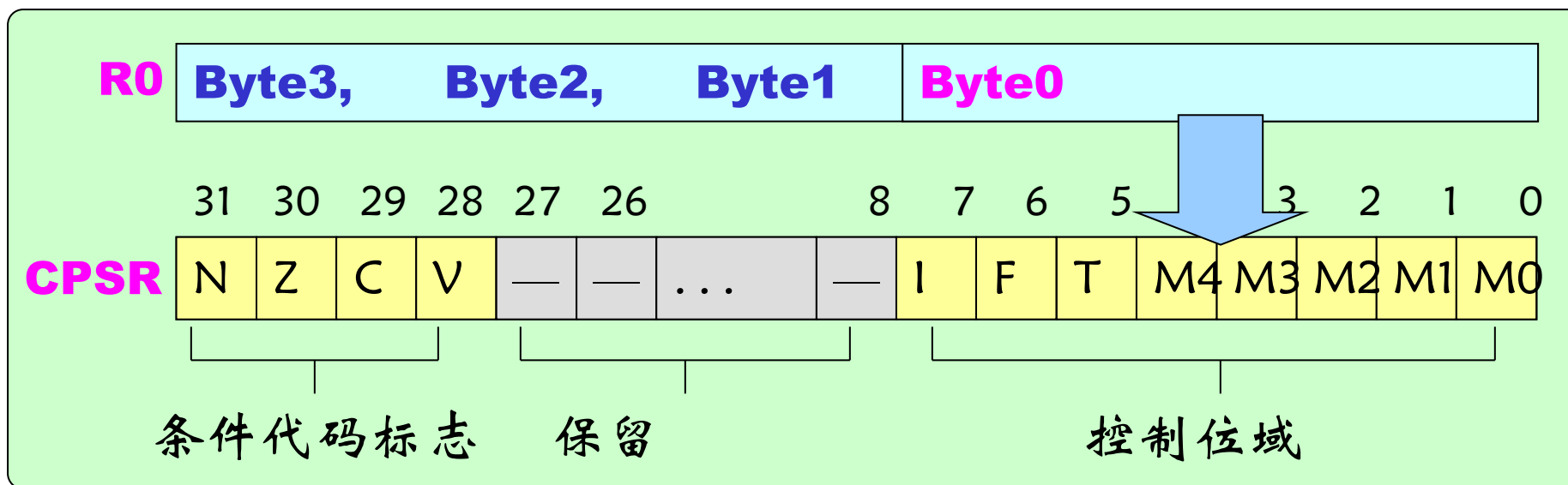
```
LDR SP, StackSvc
```

;设置中断模式堆栈

```
MSR CPSR_C, #0xD2
```

```
LDR SP, StackIrq
```

```
.....
```



其他少量杂项指令

。。。。。。不再介绍

整理、小结

为何学习指令？

用机器语言、汇编编程？  
No!

是学习、理解CPU（微处理器系统）  
如何工作的一个方面

指令的设计是如何考虑的？  
先有鸡还是先有蛋

更多问题与思考？？？