Principles and Practices of Microcontroller (Embedded System Design I) -Arm Architecture and ISA

Gang Chen (陈刚)

Associate Professor

Institute of Unmanned Systems

School of data and computer science

Sun Yat-Sen University

https://www.usilab.cn/team/chengang/

中山大學 SUN YAT-SEN UNIVERSITY 数据科学与计算机学院 School of Data and Computer Science

例子

```
;文件名：TEST1.S
;功能：实现两个寄存器相加
;说明：使用ARMulate软件仿真调试
        AREA    Example1,CODE,READONLY    ;声明代码段Example1
        ENTRY                             ;标识程序入口
        CODE32                            ;声明32位ARM指令
START   MOV     R0,#0                     ;设置参数
        MOV     R1,#10
LOOP    BL      ADD_SUB                   ;调用子程序ADD_SUB
        B       LOOP                      ;跳转到LOOP
ADD_SUB
        ADDS    R0,R0,R1                  ;R0 = R0 + R1
        MOV     PC,LR                     ;子程序返回
        END                               ;文件结束
```

Cortex指令分类

1. 数据传送指令
2. 数据处理指令
3. 子程序调用和跳转指令
4. 饱和运算指令
5. 隔离指令
6. 其他指令等

指令概述

- **ARM是三地址指令格式，指令基本格式**

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

其中<>号内的项是必须的，{}号内的项是可选的。

opcode：指令助记符； cond：执行条件；

S：是否影响CPSR寄存器的值；

Rd：目标寄存器； Rn：第1个操作数的寄存器；

operand2：第2个操作数；

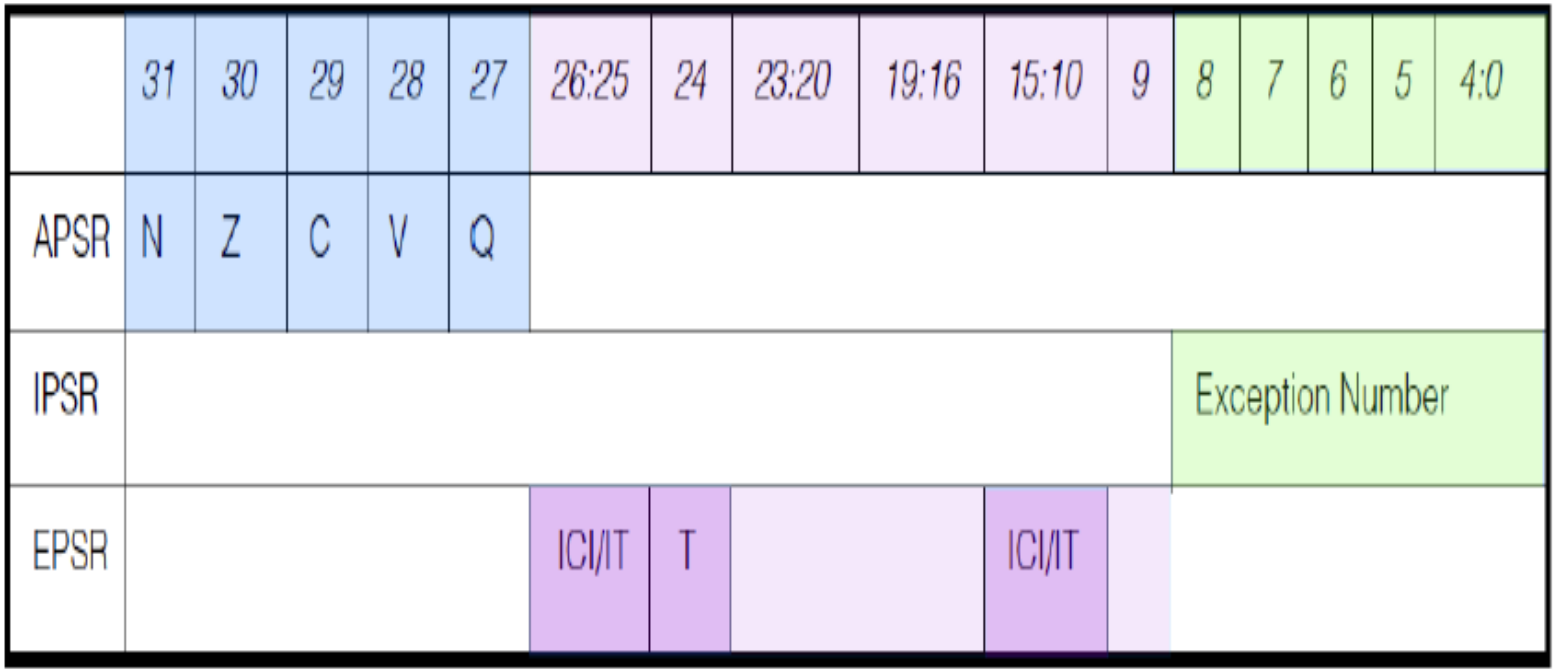
指令举例

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

指令语法	目标寄存器（Rd）	源寄存器1 (Rn)	源寄存器2 (Rm)
ADD r3,r1,r2 ADDS r0,r1	r3 r0	r1 r0	r2 r1

CM3的状态寄存器

- 应用程序 PSR（APSR）
- 中断号 PSR（IPSR）
- 执行 PSR（EPSR）



后缀的含义

后缀名	含义
S	要求更新 APSR 中的标志 s，例如： ADDS R0, R1 ；根据加法的结果更新 APSR 中的标志
EQ,NE,LT,GT 等	有条件地执行指令。EQ=Euqal, NE= Not Equal, LT= Less Than, GT= Greater Than。还有若干个其它的条件。例如： BEQ <Label> ；仅当 EQ 满足时转移

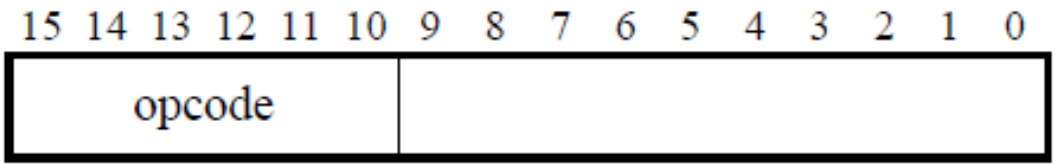
条件码

条件码	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)

跳转指令的应用——条件判断语句

```
CMP R0,#0;
CMPNE R1,#1;
ADDEQ R2,R3,R4;
```

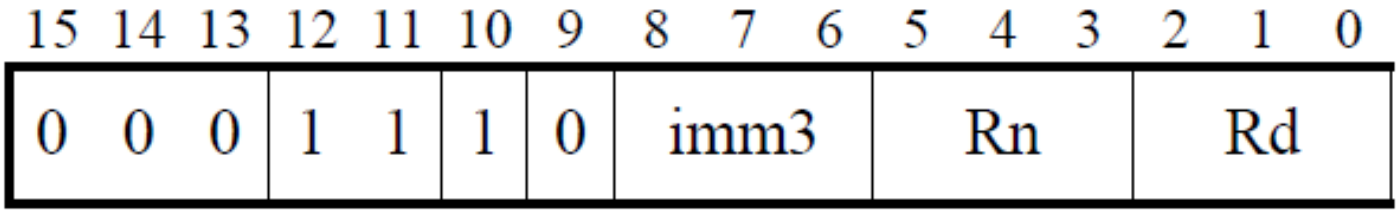
16-bit Thumb instruction encoding

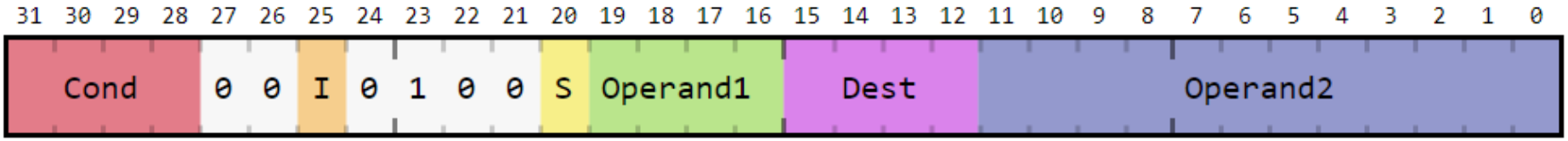


opcode	Instruction or instruction class
00xxxx	*Shift (immediate), add, subtract, move, and compare* on page A5-6
010000	*Data processing* on page A5-7
010001	*Special data instructions and branch and exchange* on page A5-8
01001x	Load from Literal Pool, see *LDR (literal)* on page A6-90
0101xx	*Load/store single data item* on page A5-9
011xxx	
100xxx	
10100x	Generate PC-relative address, see *ADR* on page A6-30
10101x	Generate SP-relative address, see *ADD (SP plus immediate)* on page A6-26
1011xx	*Miscellaneous 16-bit instructions* on page A5-10
11000x	Store multiple registers, see *STM / STMIA / STMEA* on page A6-218
11001x	Load multiple registers, see *LDM / LDMIA / LDMFD* on page A6-84
1101xx	*Conditional branch, and supervisor call* on page A5-12
11100x	Unconditional Branch, see *B* on page A6-40

ADDS <Rd>,<Rn>,#<imm3>

ADD<c> <Rd>,<Rn>,#<imm3>

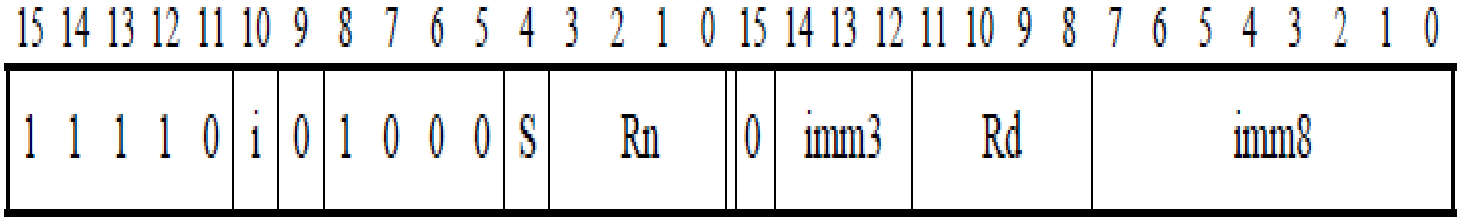




- Any instruction with bits 27 and 26 as 00 is data processing. The four-bit opcode field in bits 24–21 defines exactly which instruction this is: add, subtract, move, compare, and so on. 0100 is ADD.
- Bit 25 is the "immediate" bit. If it's 0, then operand 2 is a register. If it's set to 1, then operand 2 is an immediate value.
- Note that operand 2 is only 12 bits. That doesn't give a huge range of numbers: 0–4095, or a byte and a half. Not great when you're mostly working with 32-bit numbers and addresses.

- .N 表明此指令为16位指令
- .W 表面此指令为32位指令集

ADD{S}<c>.W <Rd>,<Rn>,#<const>



Thumb-2

“统一汇编语言（UAL）”语法机制

- 对于16 位指令和32 位指令均能实现的一些操作（常见于数据处理操作），有时虽然指令的实际操作数不同，或者对立即数的长度有不同的限制，但是汇编器允许开发者以相同的语法格式书写，并且由汇编器来决定是使用16 位指令，还是使用32 位指令。

- **ADD R0, R1 ; 使用传统的Thumb 语法**
- **ADD R0, R0, R1 ; UAL 语法允许的等值写法（R0=R0+R1**

- 有些操作既可以由16 位指令完成，也可以由32 位指令完成。
- 例如，R0=R0+1 这样的操作，16 位的与32 位的指令都提供了助记符为“ADD”的指令。在UAL 下，你可以让汇编器决定用哪个，也可以手工指定是用16 位的还是32 位的：
- ADDS R0, #1 ;汇编器将为了节省空间而使用16 位指令
- ADDS.N R0, #1 ;指定使用16 位指令（N＝Narrow）
- ADDS.W R0, #1 ;指定使用32 位指令（W=Wide）

3.2 ARM寻址方式

- 3.2.1 数据处理指令的操作数的寻址方式
- 3.2.2字及无符号字节的Load/Store指令的寻址方式
- 3.2.3 杂类Load/Store 指令的寻址方式
- 3.2.4 批量Load/Store 指令的寻址方式
- 3.2.5 协处理器Load/Store 指令的寻址方式

3.2.1 数据处理指令的操作数的寻址方式

- 立即数寻址
- 寄存器寻址
- 寄存器移位寻址

立即数

- 立即数会被表示为#immed_8r——常数表达式。
- <immediate>=immed_8循环右移(2*rotate_imm)
- 当立即数数值在0和0xFF范围时，令immed_8=<immediate>， rotate_imm=0，其他情况下，汇编编译器选择使rotate_imm数值最小的编码方式。

Rm——寄存器方式

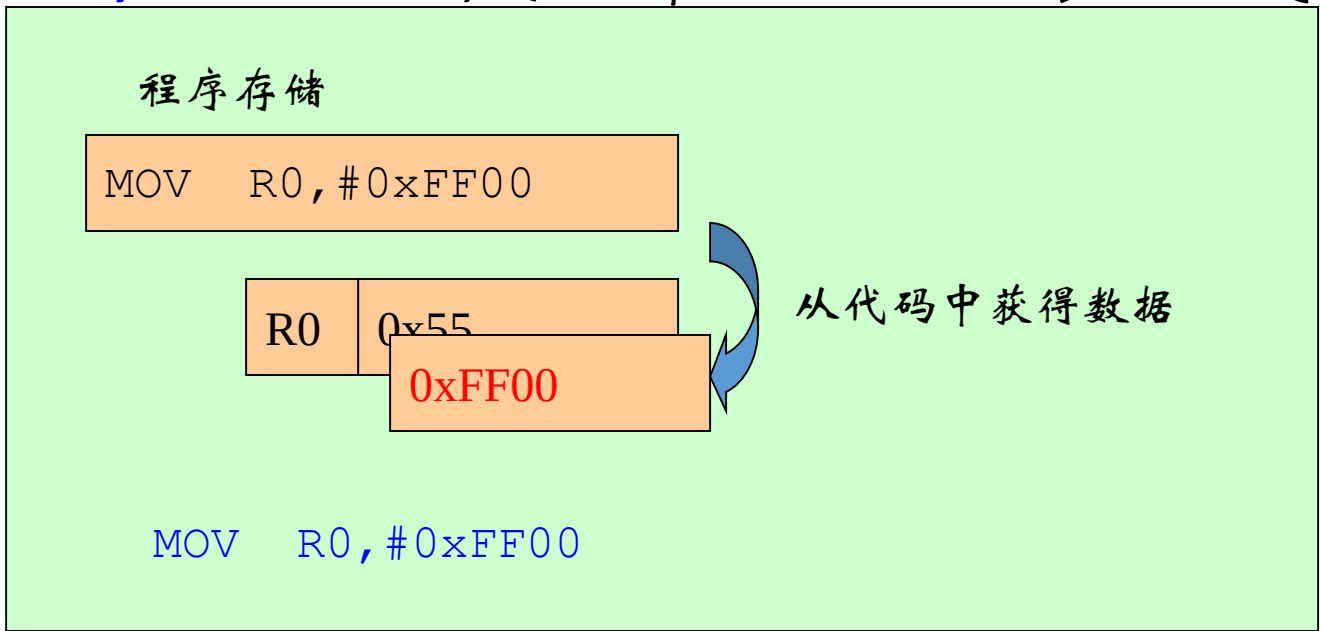
- 在寄存器方式下，操作数即为寄存器的数值。
- 例如：SUB R1,R1,R2 影响循环器进位值
- Rm,shift寄存器移位方式
  - 将寄存器的移位结果作为操作数（移位操作不消耗额外的时间），但Rm值保持不变

操作码	说明	操作码	说明
ASR #n	算术右移n位	ROR #n	循环右移n位
LSL #n	逻辑左移n位	RRX	带扩展的循环右移1位
LSR #n	逻辑右移n位	Type Rs	Type为移位的一种类型，Rs为偏移量寄存器，低8位有效。

立即寻址

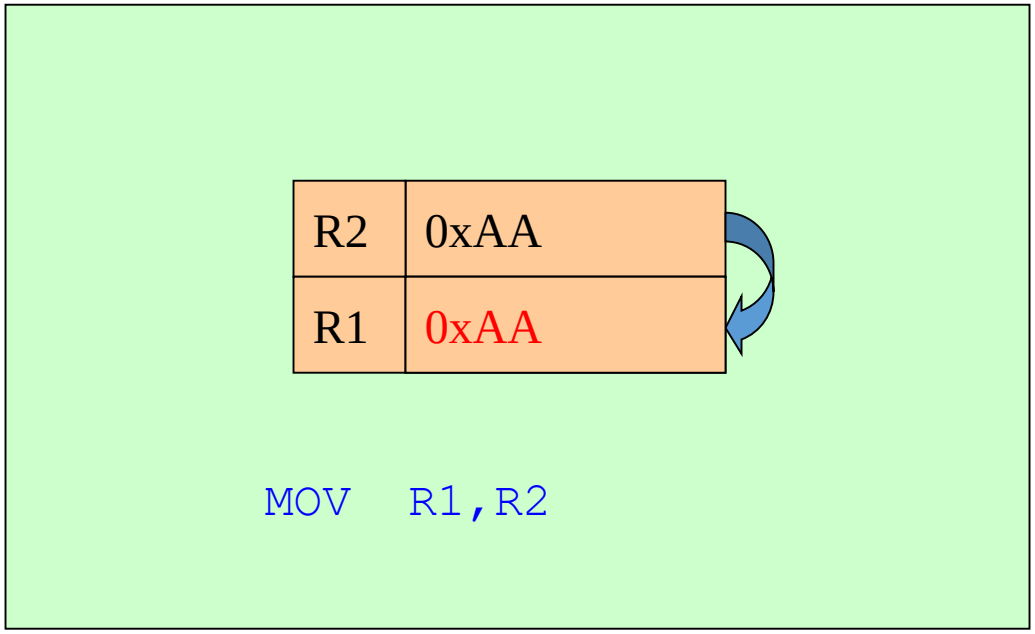
立即寻址指令中的操作码字段后面的地址码部分即是操作数本身，也就是说，数据就包含在指令当中，取出指令也就取出了可以立即使用的操作数(这样的数称为立即数)。**立即寻址指令举例如下：**

- **SUBS R0,R0,#1**;R0减1，结果放入R0，并且影响标志位
- **MOV R0,#0xFF000** ;将立即数0xFF000装入R0寄存器



寄存器寻址

- 操作数的值在寄存器中，指令中的地址码字段指出的是寄存器编号，指令执行时直接取出寄存器值来操作。
- 寄存器寻址指令举例如下：
- **MOV R1,R2** ;将R2的值存入R1
- **SUB R0,R1,R2** ;将R1的值减去R2的值，结果

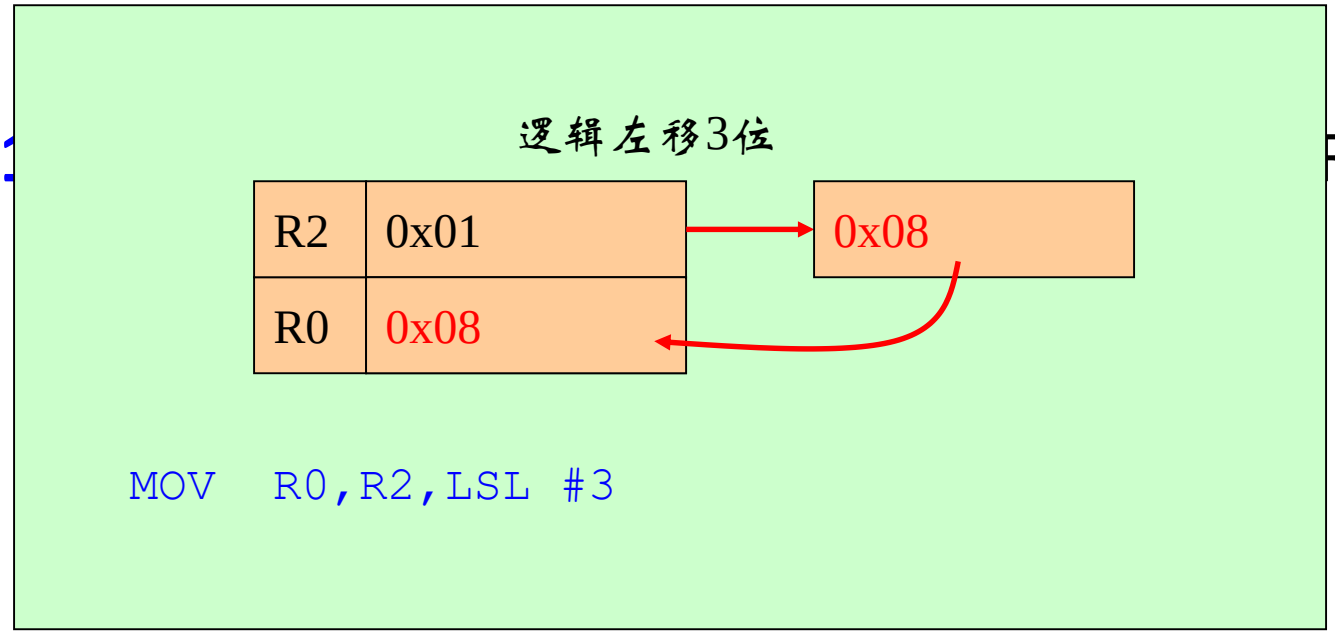


寄存器移位寻址

寄存器移位寻址是ARM指令集特有的寻址方式。当第2个操作数是寄存器移位方式时，第2个寄存器操作数在与第1个操作数结合之前，选择进行移位操作。**寄存器移位寻址指令举例如下：**

MOV R0,R2,LSL #3 ;R2的值左移3位，结果放入R0，

ANDS R1 R1相与



ARM指令寻址方式—字及无符号字节的Load/Store指令的寻址方式

- **LDR 语法**
- **LDR {<cond>}{B}{T}<Rd>,<address_mode>**
  - Address_mode 表示第二个操作数的内存地址
    1) [<Rn>,#+/-<offset_12>]立即数偏移寻址
    2) [<Rn>,+/-<Rm>]寄存器偏移寻址
    3) [<Rn>,+/-<Rm>,<shift>#<shift_imm>]带移位的寄存器偏移寻址
    4) [<Rn>,#+/-<offset_12>]!立即数前索引寻址
    5) [<Rn>,+/-<Rm>]！寄存器前索引寻址
    6) [<Rn>,+/-<Rm>,<shift>#<shift_imm>]!
    7) [<Rn>],#+/-<offset_12>立即数后索引寻址
    8) [<Rn>],+/-<Rm>
    9) [<Rn>],+/-<Rm>,<shift>#<shift_imm>

1. [<Rn>,#+/-<offset_12>]

- 地址计算方法：

  address =Rn +/- offset_12

- 适用范围

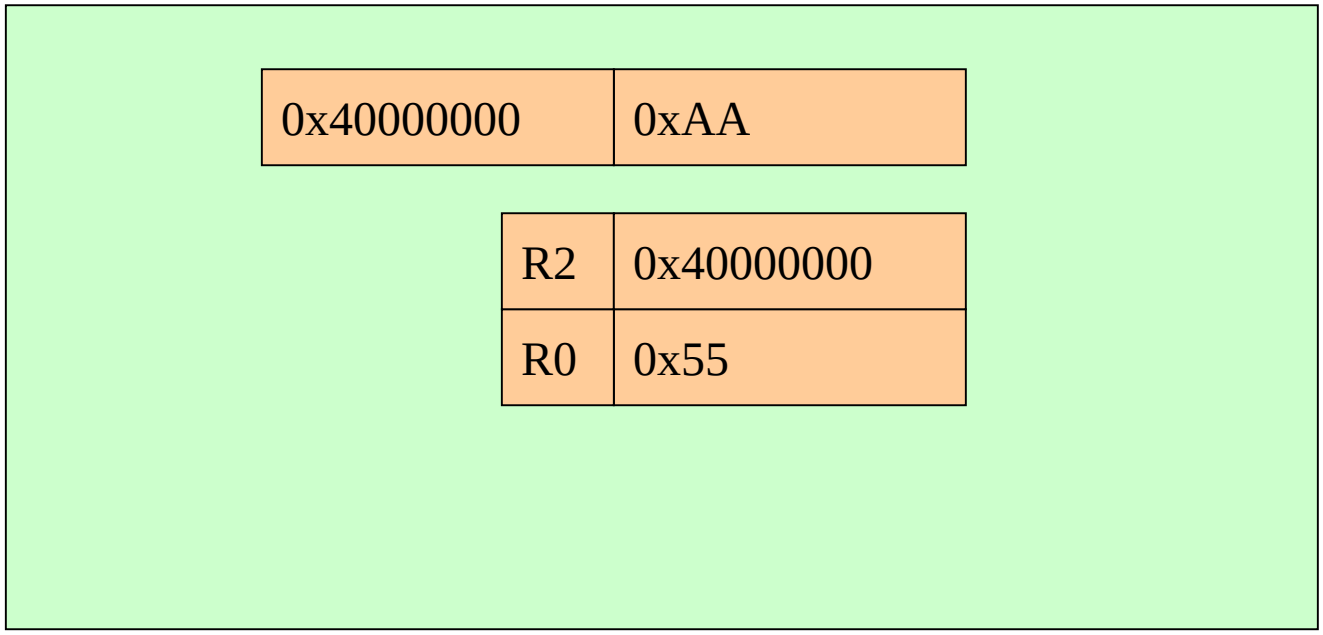
适合访问结构化数据的数据成员，当地址偏移量为0，则访问的是Rn指向的内存单元

+/-由指令编码中的U位决定
B=1，无符号字节数据，B=0字数据
L=1，指令是Load，L=0，指令是Store

寄存器间接寻址

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

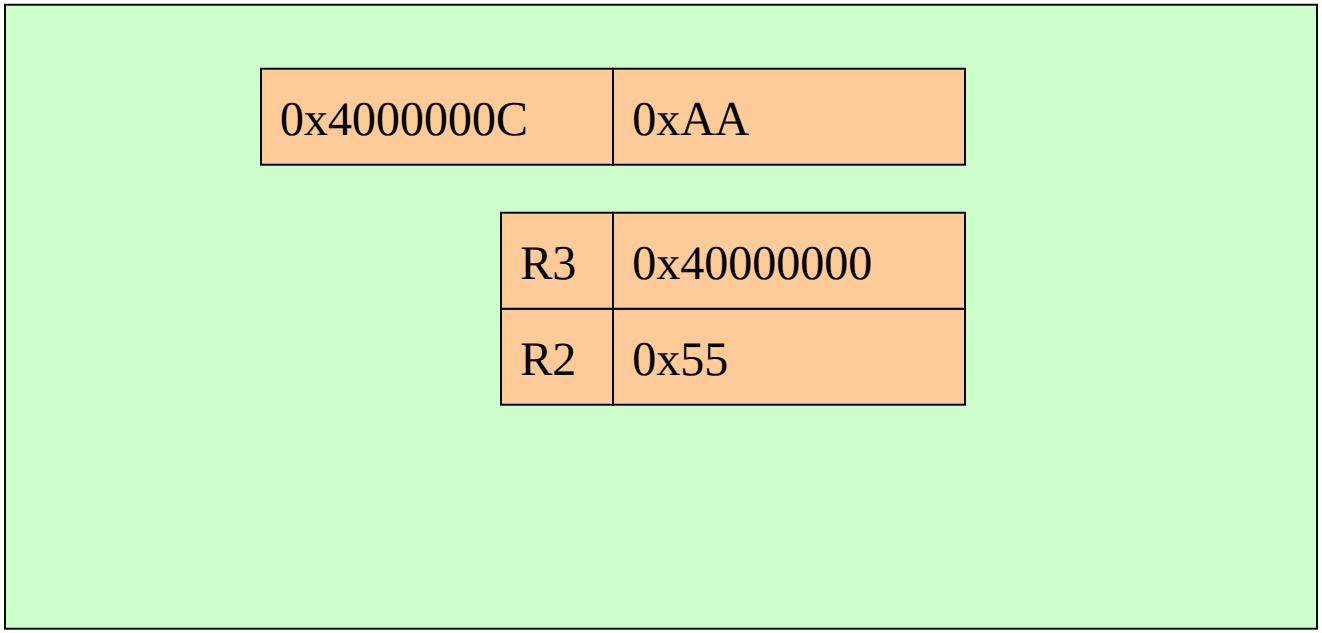
```
LDR     R1,[R2]    ;将R2指向的存储单元的数据读出
                   ;保存在R1中
SWP     R1,R1,[R2]         ;将寄存器R1的值和R2指定的存储
                           ;单元的内容交换
```



基址变址寻址

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

```
LDR     R2,[R3,#0x0C]  ;读取R3+0x0C地址上的存储单元
                       ;的内容，放入R2
STR     R1,[R0,#-4]!   ;先R0=R0-4，然后把R1的值寄存
                       ;到保存到R0指定的存储单元
```



2. [<Rn>,+/-<Rm>]

- 地址计算方法

  address= Rn +/-Rm

- 适用范围

  适用访问字节数组中的数据成员。 **R15**作基址**Rn**时，内存基址为当前指令地址加**8**字节偏移量，**R15**用作索引寄存器**Rm**时，会产生不可预知结果

- 例子：

LDR R0，[R1，R2]；

LDR R0，[R1，-R2]；

3.[<Rn>,+/-<Rm>,<shift>#<shift_imm>]

- 地址计算方法

  address= Rn +/-index （index的值需查算法）

- 适用范围

  当数组中的数据成员长度大于等于1个字节，可高效地访问数组中的数据成员；

R15作基址Rn时，内存基址为当前指令地址加8字节偏移量，R15用作索引寄存器Rm时，会产生不可预知结果

- 例子：

LDR R0，[R1，R2，LSR #2]；

4. [<Rn>, #+/-<offset_12>]!

- 地址计算方法

  address= Rn +/-offset_12

- 适用范围

  Rn,Rm是同一个寄存器，会产生不可预知的结果

  R15作基址Rn时，会产生不可预知结果

- 例子：

LDR R0，[R1，#4]！；

5. [<Rn>,+/-<Rm>]!

- 地址计算方法

  address= Rn +/-Rm

- 适用范围

  Rn,Rm是同一个寄存器，会产生不可预知的结果

  R15作基址Rn或Rm时，会产生不可预知结果

- 例子：

LDR R0，[R1，R2]！；

6. [<Rn>,+/-<Rm>,<shift>#<shift_imm>]!

- 地址计算方法

  address= Rn +/-index

- 适用范围

  Rn,Rm是同一个寄存器，会产生不可预知的结果

  R15作基址Rn或Rm时，会产生不可预知结果

- 例子：

LDR R0，[R1，R2，LSL #2]！；

7. [<Rn>],#+/-<offset_12>

- 地址计算方法

  address=Rn, 当满足条件时Rn= Rn +/-offset_12

- 适用范围

  事后访问（**post_index**）

  R15作基址Rn或Rm时，会产生不可预知结果

- 例子：

LDR R0，[R1]， #4；

8. [<Rn>],+/-<Rm>

- 地址计算方法

  address=Rn, 当满足条件时Rn= Rn +/-Rm

- 适用范围

  R15作基址Rn或Rm时，会产生不可预知结果

  Rn,Rm是同一个寄存器时，会产生不可预知的结果

- 例子：

LDR R0，[R1]， R2

9. [<Rn>],+/-<Rm>,<shift>#<shift_imm>

- 地址计算方法

  address=Rn, 当满足条件时Rn= Rn +/-index

- 适用范围

  R15作基址Rn或Rm时，会产生不可预知结果

  Rn,Rm是同一个寄存器时，会产生不可预知的结果

- 例子：

LDR R0，[R1]， R2，LSL #2；

杂类Load/Store 指令的寻址方式

- **LDR|STR{<cond>} H|SH|SB|D <Rd>,<addressing_mode>**
- **特点：操作数为半字、带符号字节、双字等**
- **6种格式**
  - [<Rn>,#+/-<offset_8>]
  - [<Rn>,+/-Rm]
  - [<Rn>,#+/-<offset_8>]!
  - [<Rn>,+/-Rm]!
  - [<Rn>],#+/-<offset_8>
  - [<Rn>],+/-Rm

[<Rn>,#+/-<offset_8>]

Offset_8被编码成高4位immedH和低4位immedL
B=1 无符号字节数据， B=0
L=1， load操作，L=1，Store操作
S=1，数据为带符号数，S=0，数据无符号数
S=0，H=0， 无符号字节数据
S=1，L=0，带符号数的Store，尚未实现

- 地址计算
  offset_8=(immedH<<4)OR immedL
  address= Rn+/-offset_8
- 适用范围
  R15用作基址寄存器Rn时，内存基地址为当前指令地址加8字节偏移量
- 例子

LDRSB R0,[R1,#3]

[<Rn>,+/-Rm]

- 地址计算

  address= Rn+/-Rm

- 适用范围

  R15用作基址寄存器Rn时，内存基地址为当前指令地址加8字节偏移量；当R15作索引寄存器Rm时，会产生不可预知的结果。

- 例子

  STRH R0,[R1,R2]

批量Load/Store 指令的寻址方式

- 一般语法格式

DM|STM{<cond>}<addressing_mode><Rn>{!},<registers>{^}

- **Addressing_mode有以下4种方式**
  - IA（Increment After）事后递增方式
  - IB（Increment Before）事先递增方式
  - DA（Decrement After）事后递减方式
  - DB （Decrement Before）事先递减方式

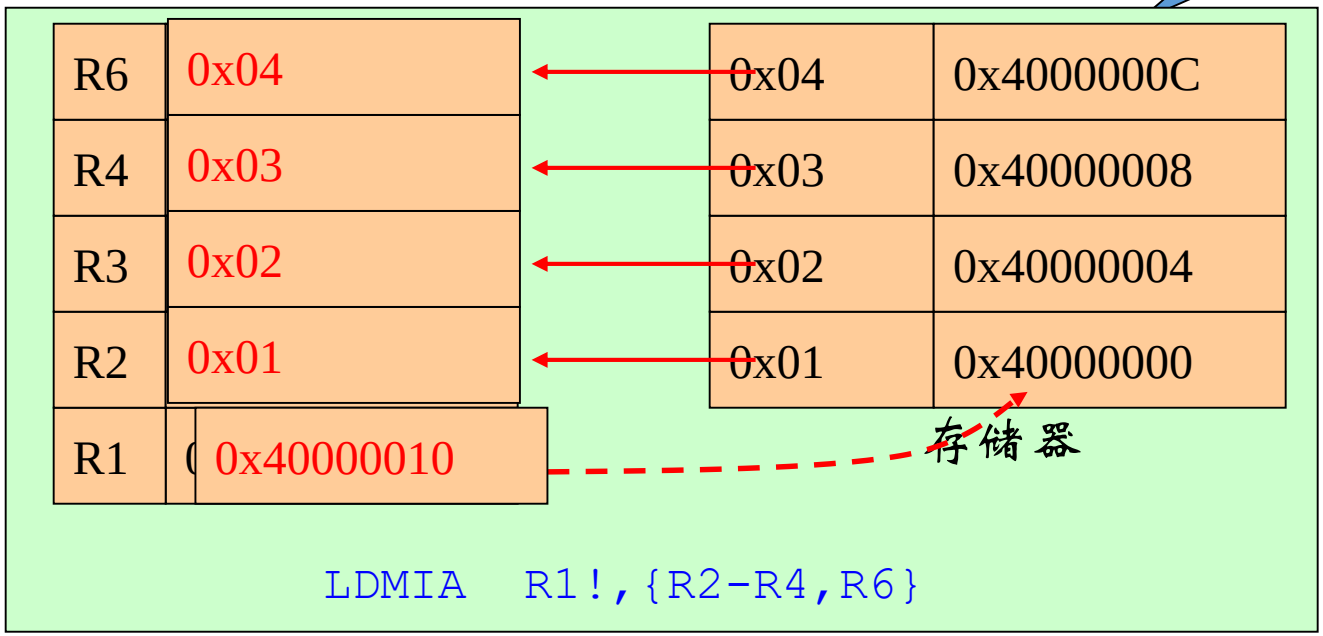
多寄存器寻址

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

多寄存器寻址指令举例如下：

```
LDMIA   R1!,{R2-R7,R12} ;将R1指向的单元中的数据读出到
                        ;R2～R7、R12中(R1自动加1)
STMIA   R0!,{R2-R7,R12} ;将寄存器R2～R7、R12的值保
                        ;存到R0指向的存储单元中
                        ;(R0自动加1)
```

使用多寄存器寻址指令时，寄存器子集的顺序是按由小到大的顺序排列，连续的寄存器可用“—”连接；否则用“，”分隔书写。



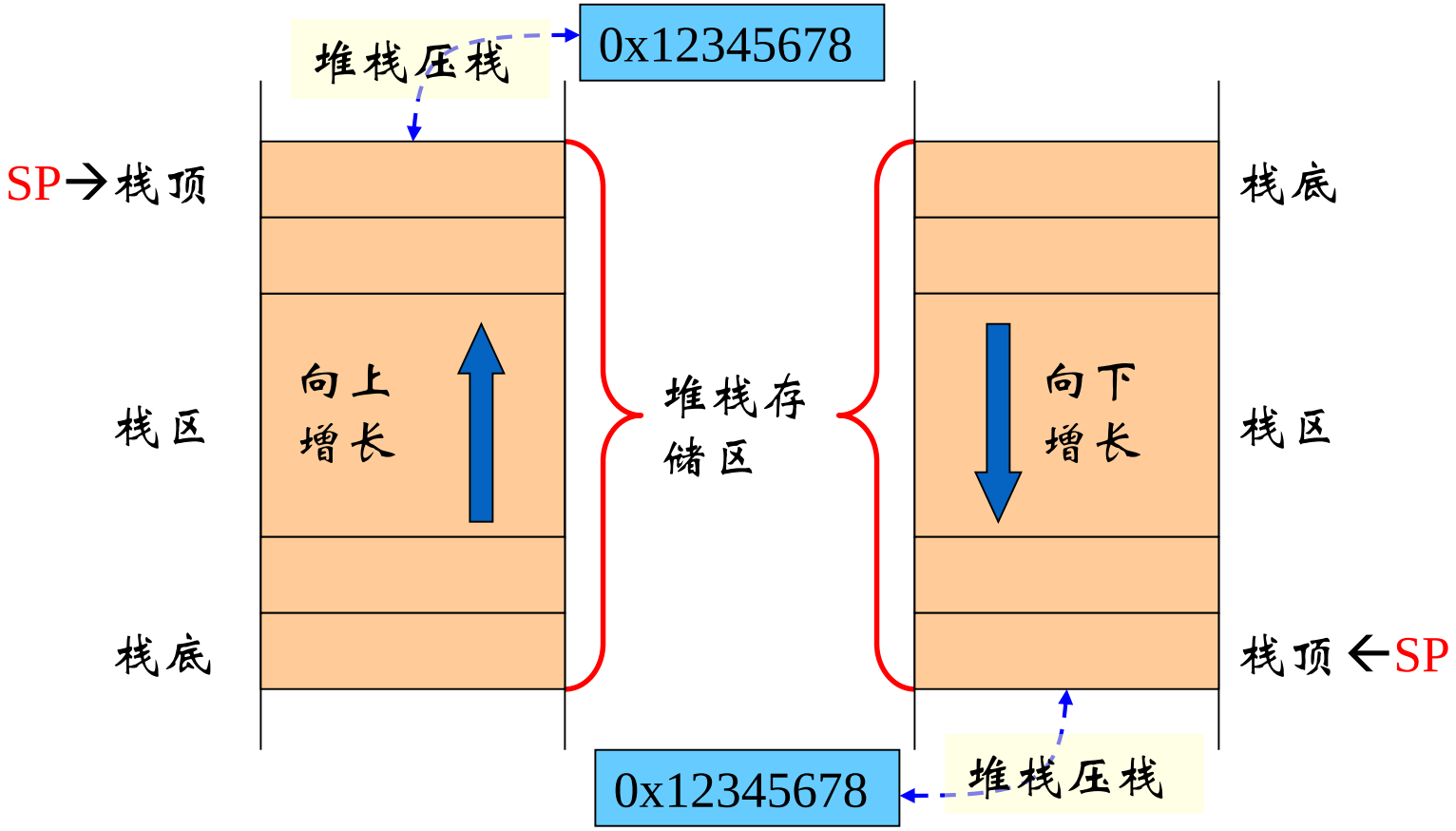
相对寻址

- 相对寻址是基址寻址的一种变通。由程序计数器PC提供基准地址，指令中的地址码字段作为偏移量，两者相加后得到的地址即为操作数的有效地址。
- **相对寻址指令举例如下：**
- **BL　SUBRl** ；调用到SUBRl子程序
- . . .
- **SUBR1…**
- **MOV PC,R14** ；返回

堆栈寻址

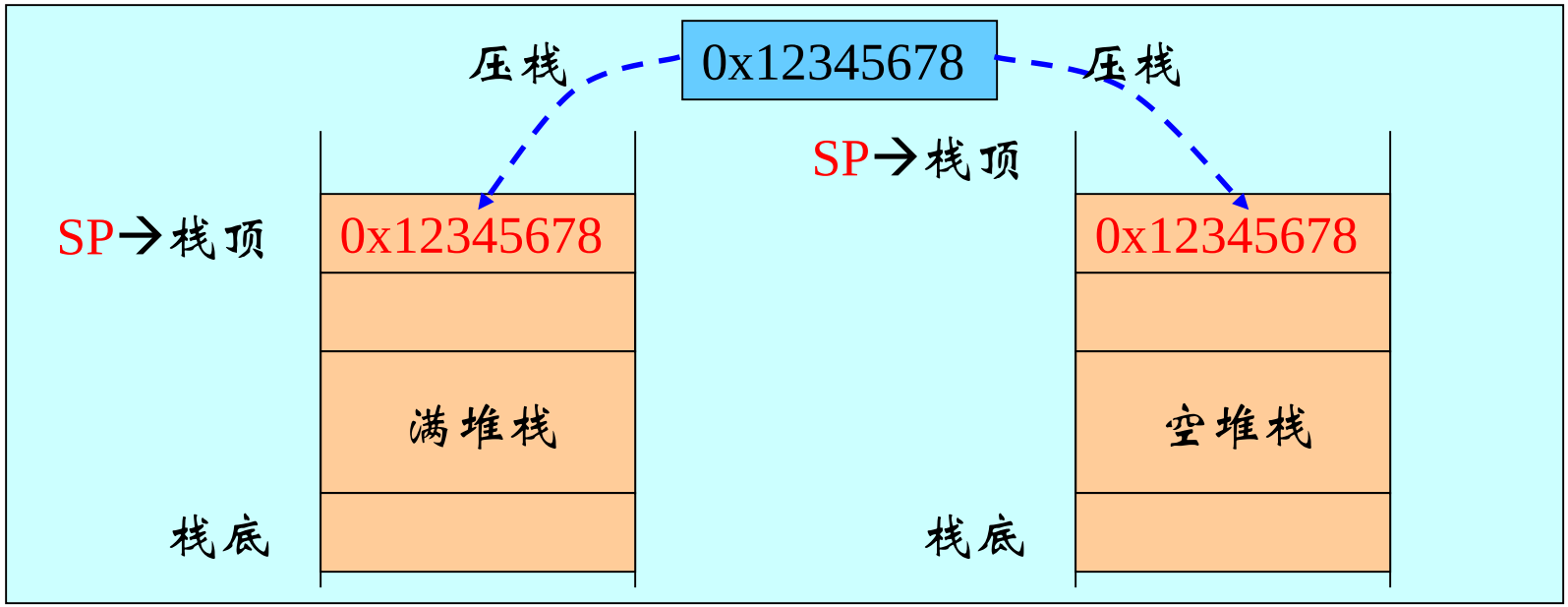
- 堆栈是一个按特定顺序进行存取的存储区，操作顺序为“后进先出”。堆栈寻址是隐含的，它使用一个专门的寄存器(堆栈指针)指向一块存储区域(堆栈)，指针所指向的存储单元即是堆栈的栈顶。**存储器堆栈可分为两种：**
  - 向上生长：向高地址方向生长，称为递增堆栈
  - 向下生长：向低地址方向生长，称为递减堆栈

堆栈寻址


堆栈压栈
0x12345678
SP→栈顶
栈区
栈底
向上
增长
堆栈存
储区
向下
增长
栈底
栈区
栈顶←SP
0x12345678
堆栈压栈

堆栈寻址

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



堆栈寻址

- 所以可以组合出四种类型的堆栈方式：
  - **满递增**：堆栈向上增长，堆栈指针指向内含有效数据项的最高地址。指令如LDMFA、STMFA等；
  - **空递增**：堆栈向上增长，堆栈指针指向堆栈上的第一个空位置。指令如LDMEA、STMEA等；
  - **满递减**：堆栈向下增长，堆栈指针指向内含有效数据项的最低地址。指令如LDMFD、STMFD等；
  - **空递减**：堆栈向下增长，堆栈指针向堆栈下的第一个空位置。指令如LDMED、STMED等。

指令集（Instruction Set Architecture, ISA）

- **数据传送类指令**
  - 两个寄存器间传送数据
  - 寄存器与存储器间传送数据
  - 寄存器与特殊功能寄存器间传送数据
  - 把一个立即数加载到寄存器

寄存器到寄存器传送

- **MOV 指令、MVN指令**
  MOV R8, R3; R8 = R3
  MVN R8, R3; R8 = -R3

存储器到寄存器传送

- **LDRx 指令、LDMxy指令；**

LDRx 指令的x可以是B（byte）、H（half word）、D（Double word）或者省略（word）

示例	功能描述
LDRB Rd, [Rn, #offset]	**从地址Rn+offset处读取一个字节送到Rd**
LDRH Rd, [Rn, #offset]	**从地址Rn+offset处读取一个半字送到Rd**
LDR Rd, [Rn, #offset]	**从地址Rn+offset处读取一个字送到Rd**
LDRD Rd1, Rd2, [Rn, #offset]	**从地址Rn+offset处读取一个双字(64位整数)送到Rd1（低32位）和Rd2（高32位）中。**

LDR-链表操作

- 链表的元素包括2个字，第一个字包含一个字节数据，第2个字包含指向下一个链表元素的指针。执行前R0指向链表头，R1放要搜索的数据；执行后R0指向第一个匹配的元素。

```
llsearch
CMP R0,#0;
LDRNEB R2,[R0];
CMPNE R1,R2;
LDRNE R0,[R0,#4];
BNE llsearch;
MOV PC,LR;
```

LDR指令的应用——简单的串比较

- 执行前R0指向第一个串，R1指向第二个串；执行后R0保存比较结果。

LDR指令的应用——长跳转

通过直接向PC寄存器中读取字数据，程序可以实现4GB的长跳转。

```
ADD LR， PC， #4；将子程序function的返回地址设为当前指          ;令地址后12字节处，即return_here处
LDR PC， [PC，#-4]；从下一条指令（DCD）中取跳转的目标          ;地址，即function
DCD  function;
return_here;
```

寄存器到存储器

- **STRx 指令、STMxy指令**

示例	功能描述
STRB Rd, [Rn, #offset]	把**Rd**中的低字节存储到地址**Rn+offset**处
STRH Rd, [Rn, #offset]	把**Rd**中的低半字存储到地址**Rn+offset**处
STR Rd, [Rn, #offset]	把**Rd**中的低字存储到地址**Rn+offset**处
STRD Rd1, Rd2, [Rn, #offset]	把**Rd1**（低**32**位）和**Rd2**（高**32**位）表达的双字存储到地址**Rn+offset**处

- **LDR.W R0,[R1, #20]! ;预索引**

上面语句的意思是先把地址**R1+20**处的值加载到**R0**，然后，**R1=R1+ 20**

- **STR.W R0, [R1], #-12 ;** 后索引

把**R0**的值存储到地址**R1**处。完毕后，**R1=R1+(-12)**

注意，**[R1]**后面是没有“！”的。在后索引中，基址寄存器是无条件被更新的

批量传送

- LDMxy指令和STMxy指令可以一次传送更多的数据。
- X可以为要I或D，I表示自增(Increment)，D表示自减（Decrement）。
- Y可以为A或B，表示自增或自减的时机是在每次访问前(Before)还是访问后(After)。

示例	功能描述
LDMIA Rd!, {寄存器列表}	从**Rd**处读取多个字，并依次送到寄存器列表中的寄存器。每读一个字后**Rd**自增一次，**16**位指令
LDMIA.W Rd!, {寄存器列表}	从**Rd**处读取多个字，并依次送到寄存器列表中的寄存器。每读一个字后**Rd**自增一次
STMIA Rd!, {寄存器列表}	依次存储寄存器列表中各寄存器的值到**Rd**给出的地址。每存一个字后**Rd**自增一次，**16**位指令
STMIA.W Rd!, {寄存器列表}	依次存储寄存器列表中各寄存器的值到**Rd**给出的地址。每存一个字后**Rd**自增一次
LDMDB.W Rd!, {寄存器列表}	从**Rd**处读取多个字，并依次送到寄存器列表中的寄存器。每读一个字前**Rd**自减一次
STMDB.W Rd!, {寄存器列表}	存储多个字到**Rd**处。每存一个字前**Rd**自减一次

- 这里需要特别注意！的含义，它表示要自增(Increment)或自减（Decrement）基址寄存器Rd的值，时机是在每次访问前(Before)或访问后(After)。比如：
- 假设 R8=0x8000，则

STMIA.W R8!, {R0-R3} ; R8值变为0x8010

STMIA.W R8, {R0-R3} ; R8值不变

批量指令的应用——简单块复制

```
loop
LDMIA R12！，{R0-R11}；从源数据区读取48个字
STMIA R13！，{R0-R11}；将48个字保存到目标区
CMP R12， R14；是否到达源数据尾
BLO loop;
```

批量指令的应用——子程序进入和退出的数据保护和恢复（ARM指令）

```
function
STMFD R13!, {R4-R12,R14};
…
Insert the function body here
..
LDMFD R13!, {R4-R12,R14};
```

堆栈操作

- **Push some or all of registers (R0-R7, LR) to stack**
  - PUSH {<registers>}
  - **Decrements** SP by 4 bytes for each register saved
  - Pushing LR saves return address
  - PUSH {r1, r2, LR}

- **Pop some or all of registers (R0-R7, PC) from stack**
  - POP {<registers>}
  - **Increments** SP by 4 bytes for each register restored
  - If PC is popped, then execution will branch to new PC value after this POP instruction (e.g. return address)
  - POP {r5, r6, r7}

数据处理指令

- **基本的加、减法运算有四条指令，分别是ADD、SUB、ADC、SBC**

ADD Rd,Rn, Rm ;
Rd = Rn+Rm
ADD Rd,Rm ;
Rd += Rm
ADD Rd,#imm ;
Rd += imm

ADC Rd,Rn, Rm ;
Rd = Rn+Rm+C
ADC Rd,Rm ;
Rd += Rm+C
ADC Rd,#imm ;
Rd += imm+C

SUB Rd,Rn ;
SBC.W Rd,Rn, #imm12 ;

堆栈指针加操作

- **Add SP and immediate value**
  - ADD <Rd>,SP,#<imm8>
  - ADD SP,SP,#<imm7>

- **Add SP value to register**
  - ADD <Rdm>, SP, <Rdm>
  - ADD SP,<Rm>

比较

- **Compare - subtracts second value from first, discards result, updates APSR**
  - CMP <Rn>,#<imm8>
  - CMP <Rn>,<Rm>

- **Compare negative - adds two values, updates APSR, discards result**
  - CMN <Rn>,<Rm>

乘、除法指令

- 包括 MUL、UDIV/SDIV 等。
- MUL Rd,Rm ; Rd *= Rm
- MUL.W Rd,Rn, Rm ; Rd = Rn*Rm
- UDIV Rd,Rn, Rm ; Rd = Rn/Rm （无符号除法）
- SDIV Rd,Rn, Rm ; Rd = Rn/Rm （带符号除法）

- 还能进行32位乘32位的乘法运算（结果为64位）：
- SMULL RL, RH, Rm, Rn ;[RH:RL]= Rm*Rn，带符号的64位乘法
- SMLAL RL, RH, Rm, Rn ;[RH:RL]+= Rm*Rn，带符号的64位乘法
- UMULL RL, RH, Rm, Rn ;[RH:RL]= Rm*Rn，无符号的64位乘法
- UMLAL RL, RH, Rm, Rn ;[RH:RL]+= Rm*Rn，无符号的64位乘法

逻辑运算相关的指令

- ;按位与
- **AND Rd, Rn ; Rd &= Rn**
- **AND.W Rd, Rn, #imm12 ; Rd = Rn & imm12**
- **AND.W Rd, Rm, Rn ; Rd = Rm & Rn**

按位或

- **ORR Rd, Rn ; Rd |= Rn**
- **ORR.W Rd, Rn, #imm12 ; Rd = Rn | imm12**
- **ORR.W Rd, Rm, Rn ; Rd = Rm | Rn**

按位清零

- **BIC Rd, Rn ; Rd &= ~Rn**
- **BIC.W Rd, Rn, #imm12 ; Rd = Rn & ~imm12**
- **BIC.W Rd, Rm, Rn ; Rd = Rm & ~Rn**

- **如何清0第n位？**

按位或反

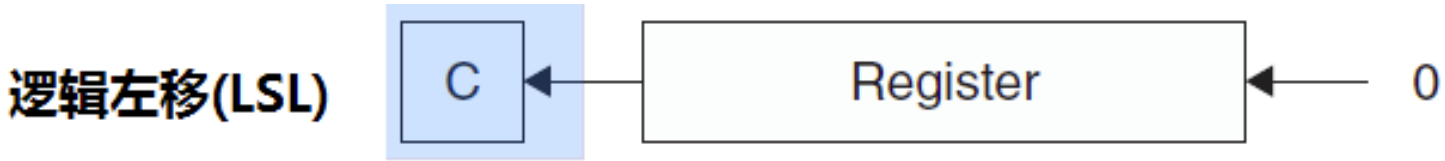
- **ORN.W Rd, Rn, #imm12 ; Rd = Rn | ~imm12**
- **ORN.W Rd, Rm, Rn ; Rd = Rm | ~Rn**

按位异或

- EOR Rd, Rn ; Rd ^= Rn
- EOR.W Rd, Rn, #imm12 ; Rd = Rn ^ imm12
- EOR.W Rd, Rm, Rn ; Rd = Rm ^ Rn

逻辑左移

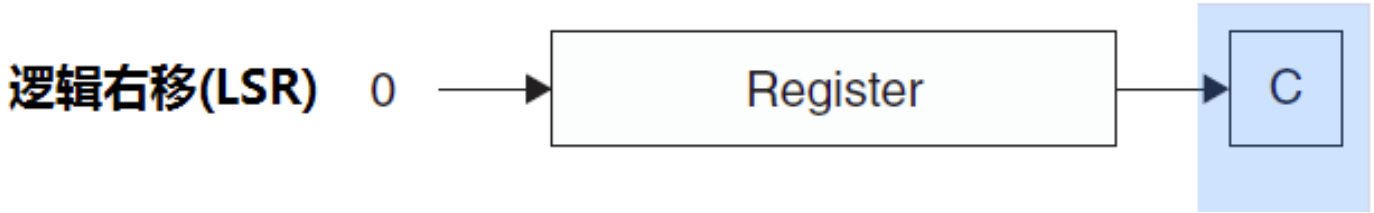
- LSL Rd, Rn, #imm5 ; Rd = Rn<<imm5
- LSL Rd, Rn ; Rd <<= Rn
- LSL.W Rd, Rm, Rn ; Rd = Rm<<Rn



仅当使用S后缀，或者使用16位指令时，才更新C位

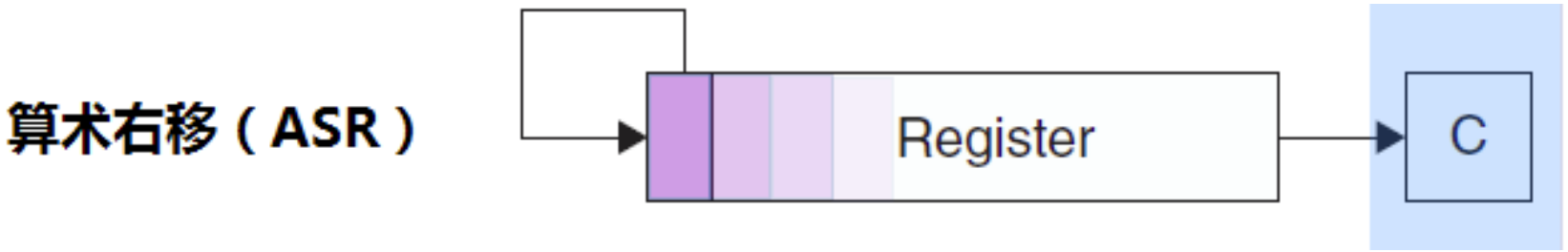
逻辑右移

- LSR Rd, Rn, #imm5 ; Rd = Rn>>imm5
- LSR Rd, Rn ; Rd >>= Rn
- LSR.W Rd, Rm, Rn ; Rd = Rm>>Rn



算术右移

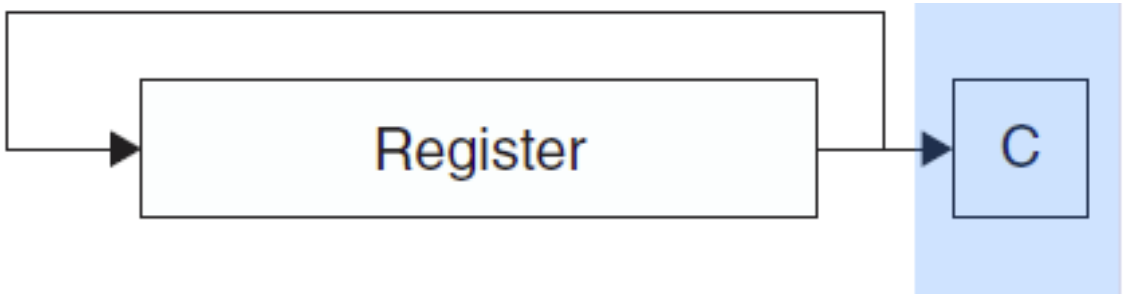
- ASR Rd, Rn, #imm5 ; Rd = Rn>> imm5
- ASR Rd, Rn ; Rd =>> Rn
- ASR.W Rd, Rm, Rn ; Rd = Rm>>Rn



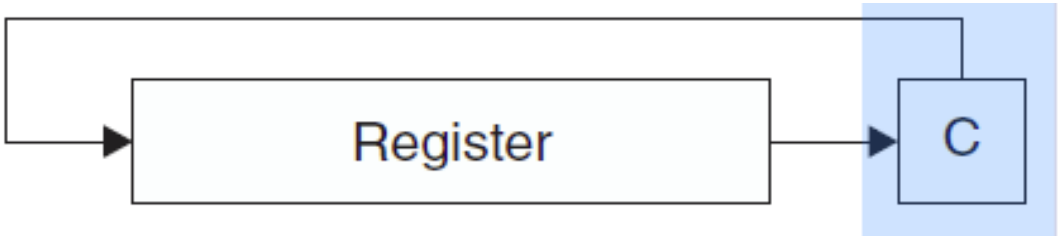
循环右移

- ROR Rd, Rn ;
- ROR.W Rd, Rm, Rn ;

圆圈右移(ROR)



带进位位右移一格(RRX)

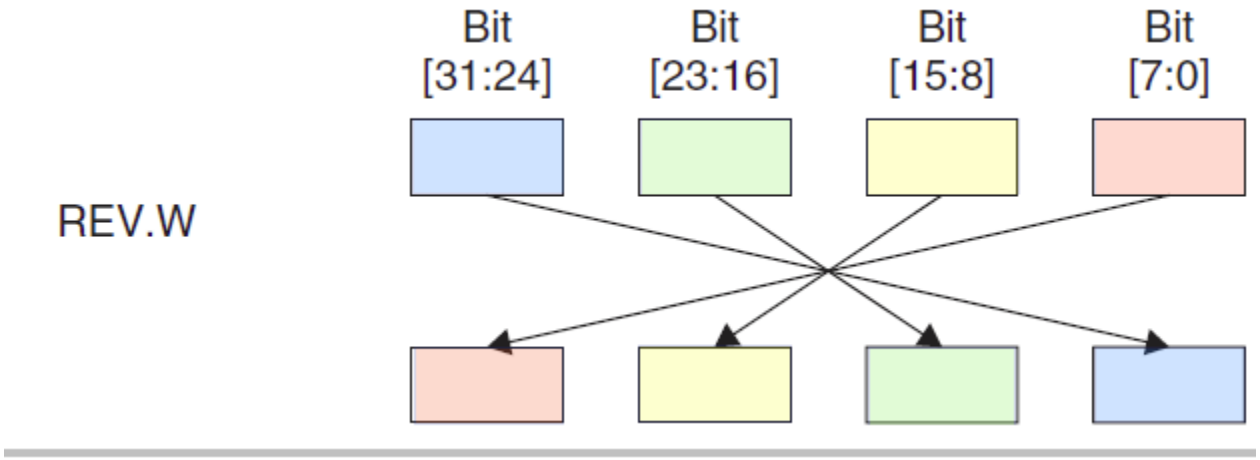


符号扩展指令

- **SXTB Rd, Rm ; Rd = Rm的带符号扩展，把带符号字节整数扩展到32位**
- **SXTH Rd, Rm ; Rd = Rm的带符号扩展，把带符号半字整数扩展到32位**

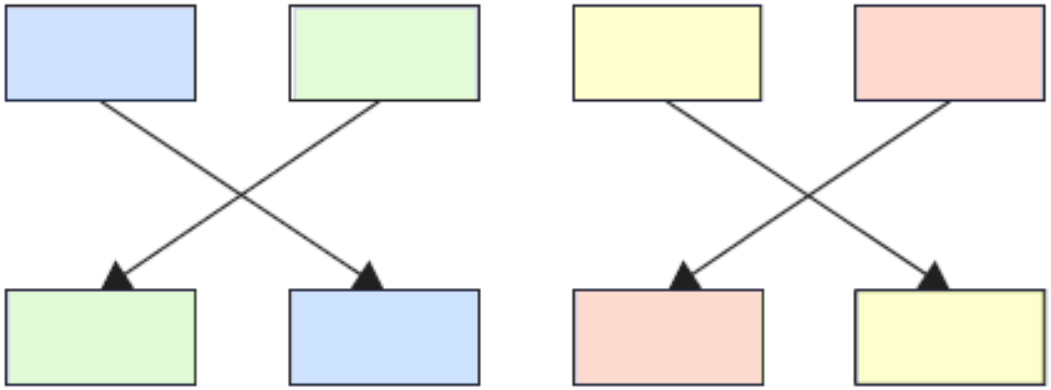
字节序反转指令

- **REV.W Rd, Rn; 在字中反转字节序**



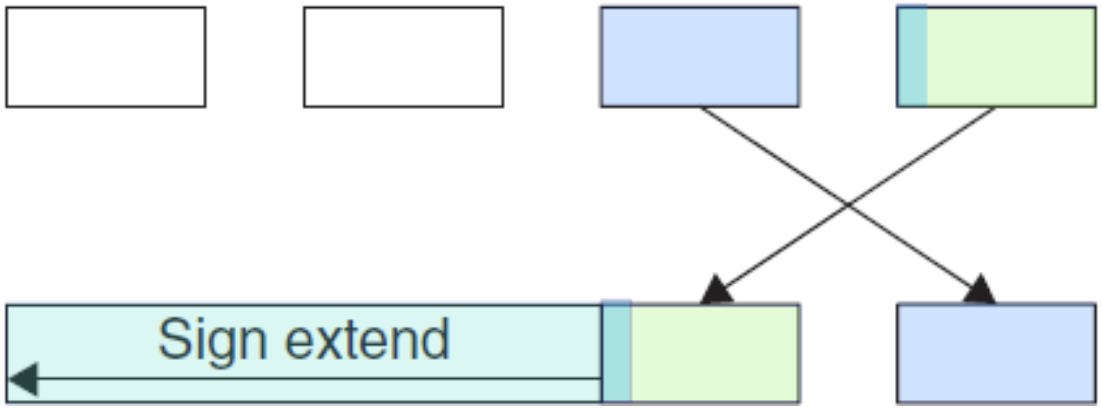
- **REV16.W Rd, Rn; 在高低半字中反转字节序**

REV16.W



- **REVSH.W; 在低半字中反转字节序，并做带符号扩展**

REVSH.W



算术逻辑运算指令的应用——64位数据运算

ADDS R0,R0,R2 ;低32位相加，同时设CPSR的C标志位

ADC R1，R1，R3；高32位的带位相加

SUBS R0，R0，R2；低32位相加，同时设CPSR的C标志位

SBC R1，R1，R3；高32位的带位相减

CMP R1，R3；比较高32位

CMPEQ R0，R2；如果高32位相等，比较低32位

算术逻辑运算指令的应用——位操作指令

- R2中的高8位数据传送到R3的低8位中

```
MOV R0，R2，LSR#24          ；
BIC R3，#0xff
ORR R3，R0，R3  ；
```

无条件跳转指令

- **B Label ;跳转到Label处对应的地址, 无条件跳转指令**
- **BX reg ;跳转到由寄存器reg给出的地址, 无条件跳转指令。Rm的bit[0]必须是1，但跳转地址在创建时会把bit[0]置为0。**
- **BL Label ;跳转到Label对应的地址，并且把跳转前的下条指令地址保存到LR**
- **BLX reg ;跳转到由寄存器reg给出的地址，并根据REG的LSB切换处理器状态，还要把转移前的下条指令地址保存到LR。**

- 例子：
- B loopA； 无条件跳转到loopA的位置
- BLE ng; LE条件跳转到标号ng
- B.W target ;在16MB内跳到target

操作数	跳转范围
B label	-16MB~+16 MB
B{cond} label(IT块外)	-1MB~+1 MB
B{cond} label(IT块内)	-16MB~+16 MB
BL{cond} label	-16MB~+16 MB
BX{cond} Rm	寄存器可以表示任何值

IT块

- **IT的使用形式如下：**
  - IT <cond> ;围起1条指令的IF-THEN块
  - IT<x> <cond> ;围起2条指令的IF-THEN块
  - IT<x><y> <cond> ;围起3条指令的IF-THEN块
  - IT<x><y><z> <cond>;围起4条指令的IF-THEN块
- **其中<x>, <y>,<z>的取值可以是“T”或者“E”。**

IF-THEN 指令块

要实现如下的功能：

```
if (R0==R1)
{
R3 = R4 + R5;
R3 = R3 / 2;
}
else
{
R3 = R6 + R7;
R3 = R3 / 2;
}
```

```
CMP R0, R1 ; 比较R0和R1
ITTEE    EQ ; 如果 R0 == R1, Then-Then-Else-Else
ADDEQ R3, R4, R5 ; 相等时加法EQ ;
如果R0 == R1,Then-Then-  Else-Else
ASREQ R3, R3, #1 ; 相等时算术右移
ADDNE R3, R6, R7 ; 不等时加法
ASRNE R3, R3, #1 ; 不等时算术右移
```

- 例子：将R0 16进制数转成ASCII码的（‘0’-‘9’）或（‘A’-’ ‘F’）
- CMP R0，#9
- ITE GT；以下2条指令是本IT块内指令
- ADDGT R1，R0，#55；转换成A-F
- ADDLE R1，R0，#48；转换成0-9

- ITTEE EQ
- MOVEQ R0,R1； EQ满足R0=R1
- ADDEQ R2,R2,#10;EQ满足，r2+=10
- ANDNE R3,R3,#1;NE条件满足，R3&=1

使用注意事项：分支指令和修改PC值得指令必须放在IT块外或IT块最后一条。IT块里每条指令必须制定带条件码后缀，必须与IT指令相同或相反。

课堂测试

- 用减奇数次数的方法，求一个数的近似平方根，这个平方根是一个整数。如求17的平 方根，可以用17相继减去奇数1、3、5、7、…，当结果为负数时停止，即：17-1-3-5-7-9＜0 可以看出，17 在减去 5 次奇数后结果变为负数，可以近似认为 17 的平方根在 4 与 5 之间， 计算 NUM 的平方根，如果 NUM=17，则 ANS 中保存结果 4。

Barrier隔离指令

指令名	功能描述
DMB	数据存储器隔离。DMB指令保证： 仅当所有在它前面的存储器访问操作都执行完毕后，才提交(commit)在它后面的存储器访问操作。
DSB	数据同步隔离。比DMB严格： 仅当所有在它前面的存储器访问操作都执行完毕后，才执行在它后面的指令（亦即任何指令都要等待存储器访问操作——译者注）
ISB	指令同步隔离。最严格：它会清洗流水线，以保证所有它前面的指令都执行完毕之后，才执行它后面的指令。

其他一些有用的指令

- RBIT指令
- RBIT是按位反转的，相当于把32位整数的二进制表示法水平旋转180度。其格式为：
- RBIT.W Rd, Rn

课堂习题讲解

- 设在 A、A+1, A+2分别存放着 3 个数。若 3 个数都不是 0，则求出三个数的和并 存放在 S 单元中；若其中有一个数为 0，则返回。请编写此程序。（A, S为存储地址）

课堂习题讲解

- 已定义了两个整数变量 A 和 B,试编写程序完成下列功能：若两个数种有一个是奇数，则将奇数存入 A 中，偶数存入 B 中；

课堂习题讲解

- 已定义了两个整数变量 A 和 B,试编写程序完成下列功能：若两个数种有一个是奇数，则将奇数存入 A 中，偶数存入 B 中；

```
FUNC
MOV R1,#A;
MOV R2,#B;
LDR R3, [R1];
LDR R4, [R2];
ADD R3, R3, #1;
ADD R4, R4, #1;
CMP R3,#0;
STMNE R3, [R1];
STMEQ R3, [R2];
CMP R4,#0;
STMNE R4, [R1, #1];
STMEQ R4, [R2,#1];
 MOV PC LR
```

课堂习题讲解

- 在首地址为 DATA 的字数组中，存放了 8个16位无符号数，试编写一个程序，求 出它们的平均值放在 R1 寄存器中；并求出数组中有多少个数小于此平均值，将结果放在 BX 寄存器中.

课堂习题讲解

- 在首地址为 DATA 的字数组中，存放了 8个16位无符号数，试编写一个程序，求 出它们的平均值放在 R1 寄存器中；并求出数组中有多少个数小于此平均值，将结果放在 BX 地址中

```
Average
MOV R0,#DATA;
MOV R1,#0;
MOV R2,#0;
LOOP: LDRH R3, [R0], #2;
ADD R1,R3;
ADD R2,#1;
CMP R2,#8;
BNE LOOP
LRR R1, #3;
MOV PC LR
```

课堂习题讲解

- 在首地址为 DATA 的字数组中，存放了 8个16位无符号数，试编写一个程序，求 出它们的平均值放在 R1 寄存器中；并求出数组中有多少个数小于此平均值，将结果放在 BX 寄存器中.

```
Average_CMP
BL Average;   >>>平均值存在R2中
MOV R0,#DATA;
MOV R4,#0;
MOV R5,#0;
LOOP1: LDRH R3, [R0], #1;
CMP R3, R1;
ADDLO R4,#1;
ADD R5,#1;
CMP R5,#8;
BNE LOOP1
STR R5,=BX
MOV PC LR
```