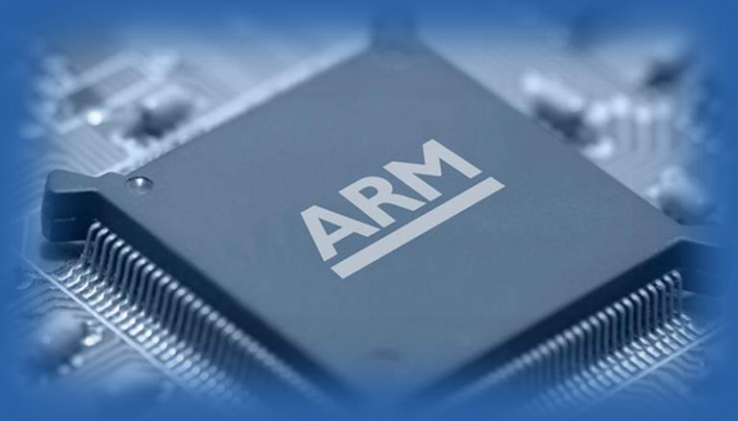
仪器科学与光电工程学院
School of Instrumentation Science and Opto-electronics Engineering

微机原理与接口技术

第六讲 ARM指令系统




```
bl   __isoc99_scanf
ldr r2, [sp, #12]
ldr r3, [sp, #8]
cmp r2, r3
bgt .L7
ldrlt   r1, [sp, #4]
ldrge   r1, [sp, #4]
sublt   r1, r1, #1
strlt   r1, [sp, #4]
```

1. 简介

ARM处理器是(RISC)原理设计的，指令集和相关译码机制较为简单。

不同的ARM体系架构，指令集也不同

ARM处理器是（RISC)原理设计的，指令集和相关译码机制较为简单。不同的ARM体系架构，指令集也不同

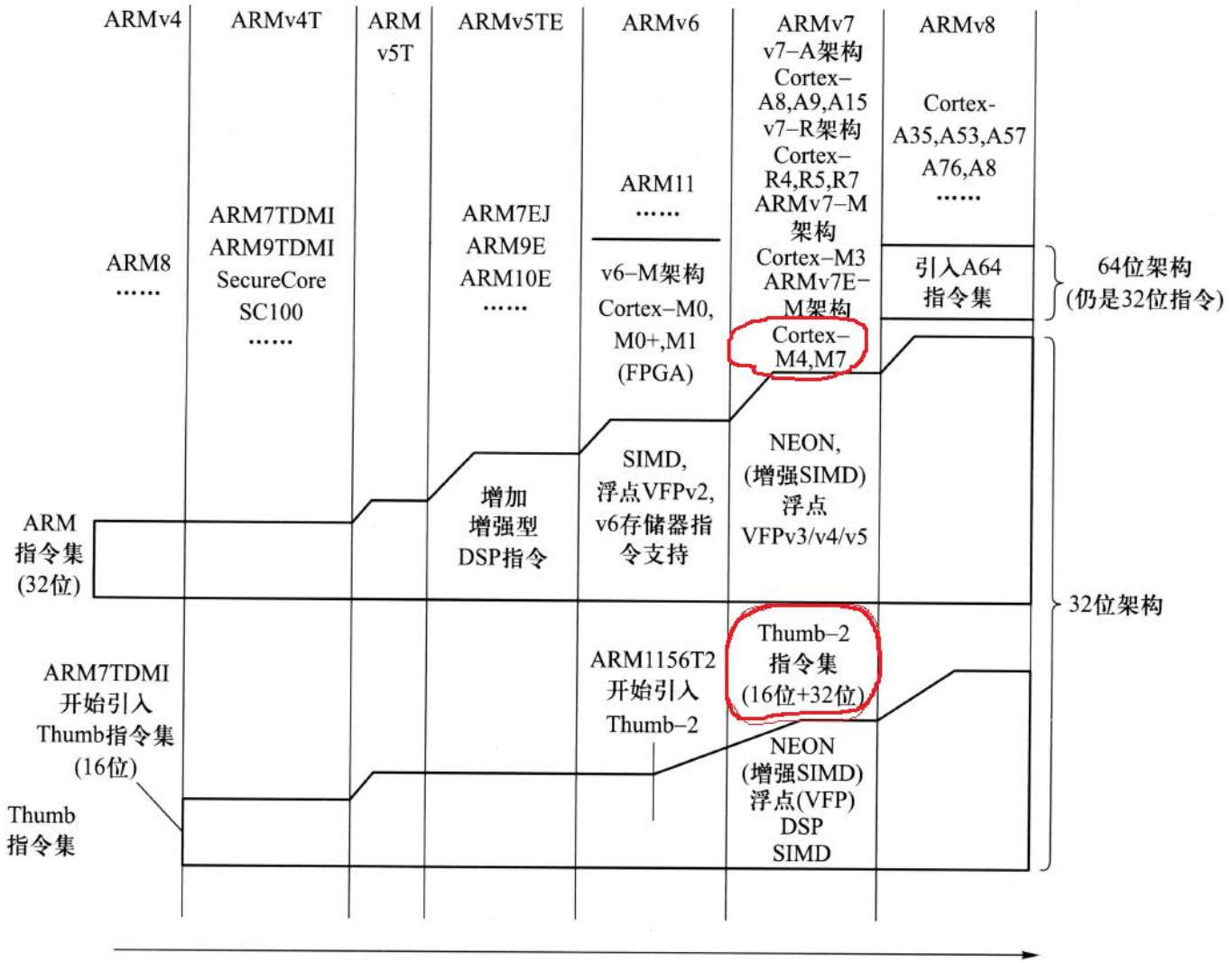


图 3–1–1　ARM 架构演进图

3.1.1 ARM 指令系统简介

体系架构与指令系统

ARM 开发工具的UAL（*Unified Assembler Language*）统一了ARM 指令集和Thumb 指令集的语法，在书写Thumb-2 指令时不需要分析这条指令是32 位还是16 位指令，汇编器会按照最简原则自动完成汇编，但在统一汇编语言UAL 中可以利用后缀".N” 和“.W” 来指定指令长度。

```
ADD     R1,  R2                 ; 自动汇编成 16 位代码
ADD     R1,  R2,  #0x8000       ; 有大于 8 位数范围立即数的指令汇编成 32 位代码
ADD     R1,  R1,  R2            ; 自动汇编成 16 位代码
```

3.1.2 指令格式

* 语法格式

<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APSR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；

3.1.2 指令格式

* ARM指令集——条件码

ARM指令的基本格式如下：

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

条件执行指令必须位于if-then指令模块中。

• 指令条件码表

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

3.1.2 指令格式

* ARM指令集——条件执行

Example 3-1: absolute value shows the use of a conditional instruction to find the absolute value of a number. R0 = abs(R1).

Example 3-1: absolute value

```
MOVS    R0, R1          ; R0 = R1, setting flags.
IT      MI              ; Skipping next instruction if value 0 or
                        ; positive.
RSBMI   R0, R0, #0      ; If negative, R0 = -R0.
```

Example 3-2: compare and update value shows the use of conditional instructions to update the value of R4 if the signed values R0 is greater than R1 and R2 is greater than R3.

Example 3-2: compare and update value

```
CMP     R0, R1          ; Compare R0 and R1, setting flags.
ITT     GT              ; Skip next two instructions unless GT condition
                        ; holds.
CMPGT   R2, R3          ; If 'greater than', compare R2 and R3, setting
                        ; flags.
MOVGT   R4, R5          ; If still 'greater than', do R4 = R5.
```

3.1.2 指令格式

* ARM指令集——第2个操作数

ARM指令的基本格式如下：

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

灵活的使用第2个操作数“operand2”能够提高代码效率。它有如下的形式：

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

3.1.2 指令格式

Thumb-2指令可以支持immed_8、immed_12和immed_16等不同格式的立即数。immed_8格式是一个8位范围的立即数格式，immed_12格式立即数是由一个8位立即数按照循环码移位获得的数值，immed_16是一个16位范围的立即数格式。

以immed_12为例：

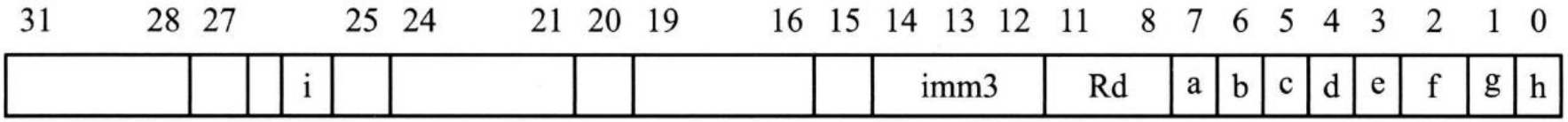


图 3-1-3　第二操作数为 immed_12 格式立即数的指令编码格式（Thumb-2 指令集）

a、b、c、……、h是8位立即数的具体位(immed_8)，而i、3位imm3和a组成了循环右移的位数，移位位数的范围是8~31。

3.1.2 指令格式

表 3-1-2　Thumb-2 指令集中 immed_12 格式立即数的编码表

循环码 i：imm3：a	生成的立即数 （二进制格式，a~h 为任一二进制数值）	说明
0000x	00000000 00000000 00000000 abcdefgh	无移位
0001x	00000000 abcdefgh 00000000 abcdefgh	
0010x	abcdefgh 00000000 abcdefgh 00000000	
0011x	abcdefgh abcdefgh abcdefgh abcdefgh	
01000	1bcdefgh 00000000 00000000 00000000	向右移 8 位
01001	01bcdefg h0000000 00000000 00000000	向右移 9 位
01010	001bcdef gh000000 00000000 00000000	向右移 10 位
01011	0001bcde fgh00000 00000000 00000000	向右移 11 位
（n）	8 位二进制常数继续向右移位	向右移 n 位
11101	00000000 00000000 000001bc defgh000	向右移 29 位
11110	00000000 00000000 0000001b cdefgh00	向右移 30 位
11111	00000000 00000000 00000001 bcdefgh0	向右移 31 位

3.1.2 指令格式

合法的立即数：0xFF 、0xl0l 、0x104 、0xFFFF 、0x10001 、0x20002 、0xlF001F 、0x32003200

不合法的立即数：0x10002 、0x2E001E、0xF000000F

3.1.2 指令格式

* ARM指令集——第2个操作数

▪Rm——寄存器方式

在寄存器方式下，操作数即为寄存器的数值。

例如：

SUB　　R1,R1,R2

3.1.2 指令格式

* ARM指令集——第2个操作数

▪Rm,shift——寄存器移位方式

将寄存器的移位结果作为操作数，但Rm值保持不变，移位方法如下：

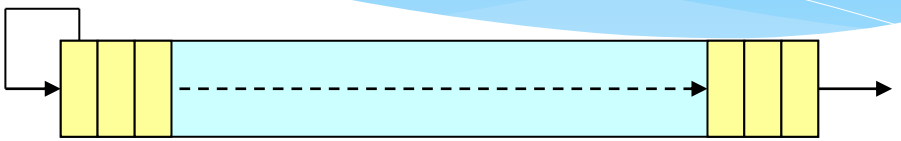
操作码	说明	操作码	说明
LSL #n	逻辑左移n位	ROR #n	循环右移n位
LSR #n	逻辑右移n位	RRX	带扩展的循环右移1位
ASR #n	算术右移n位	Type Rs	Type为移位的一种类型，Rs为偏移量寄存器，低8位有效。

ARM指令集——第2个操作数

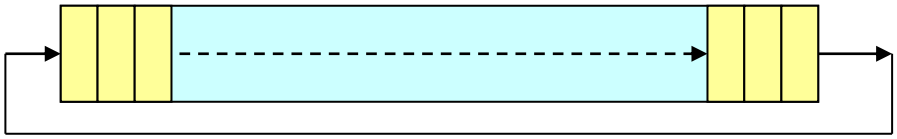
LSL移位操作：

LSR移位操作：

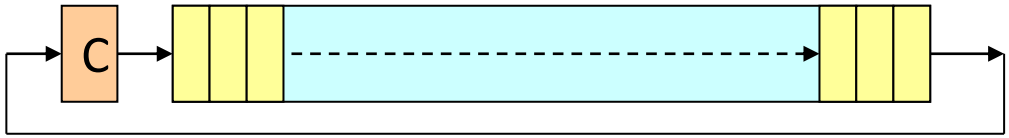
ASR移位操作：



ROR移位操作：

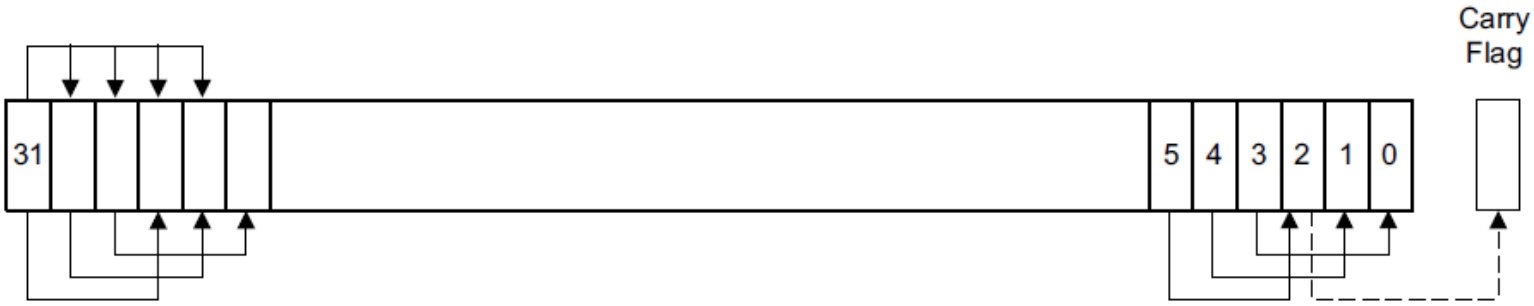


RRX移位操作：



操作码	说明	操作码	说明
LSL #n	逻辑左移n位	ROR #n	循环右移n位
LSR #n	逻辑右移n位	RRX	带扩展的循环右移1位
ASR #n	算术右移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。

* 算数右移3位



R0=0x800000A5,

3.1.2 指令格式

* ARM指令集——第2个操作数

  - Rm,shift——寄存器移位方式

  例如：

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

  SUB　　R1,R1,R2,LSR R3　;R1=R1-(R2/2^{R3})

指令的编码

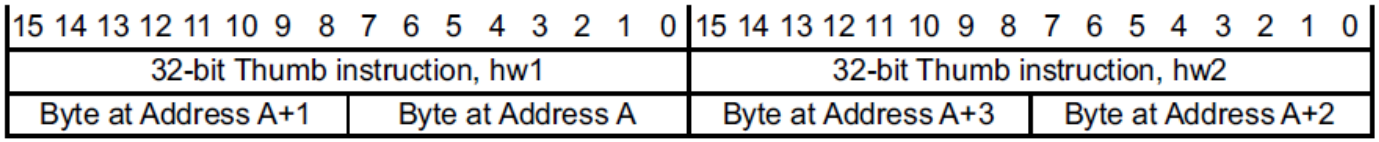
- 每一条指令是16位长的半字或由2个半字组成的32位指令。所有指令是16位对称方式存储的。
- 如果一个半字的高5位[15:11]是如下的数值，那么此半字就是一个32位指令的前半字。否则，此半字就是1条16位指令。

  0b11101

  0b11110

  0b11111

- 32位指令由hw1和hw2两个半字组成，hw1位于低地址。下图所示的指令编码图，给出了其组成的4个字节的存放顺序。



32位指令在内存中存放的字节顺序

3.2 指令的寻址方式

* 寻址方式：如何根据指令中给出的地址信息寻找操作数的物理地址。
* 操作数可能在：
  * 指令中直接给出：立即数
  * 寄存器
  * 内存单元

3.2 指令的寻址方式

7种基本寻址方式

* 1.立即寻址；
* 2.寄存器直接寻址；
* 3.寄存器移位寻址；
* 4.寄存器间接寻址；
* 5.基址变址寻址；
* 6.多寄存器寻址；
* 7.堆栈寻址；

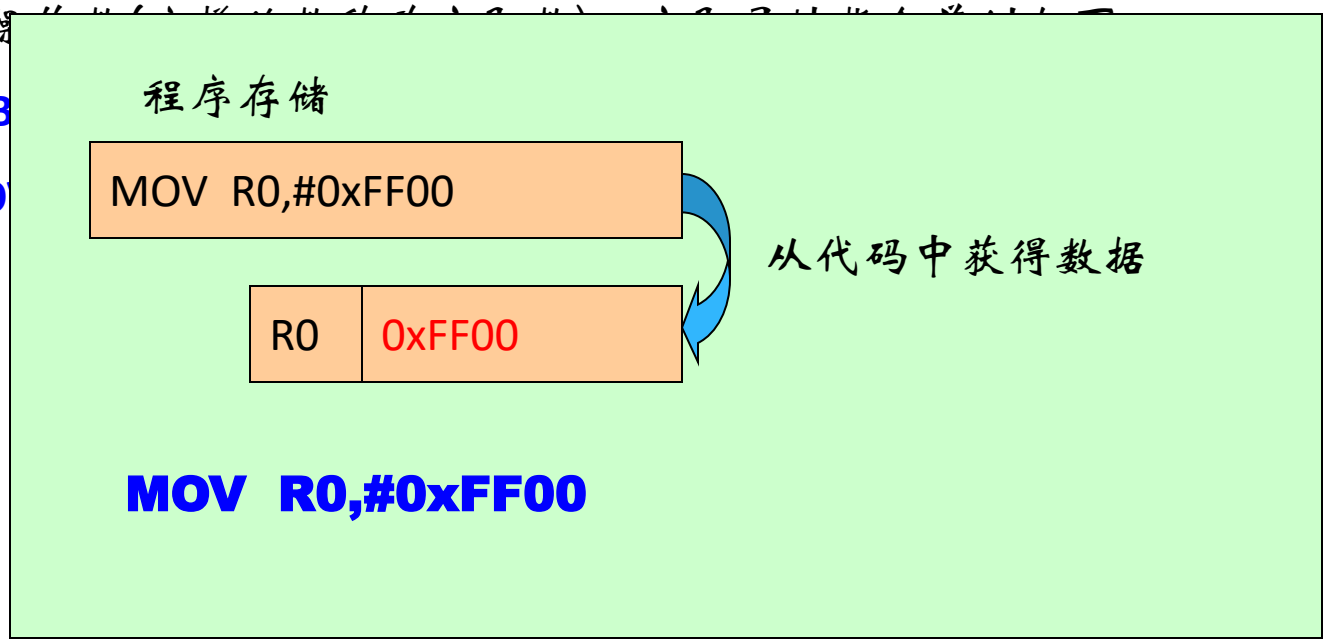
3.2 ARM处理器寻址方式

* 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

SUB

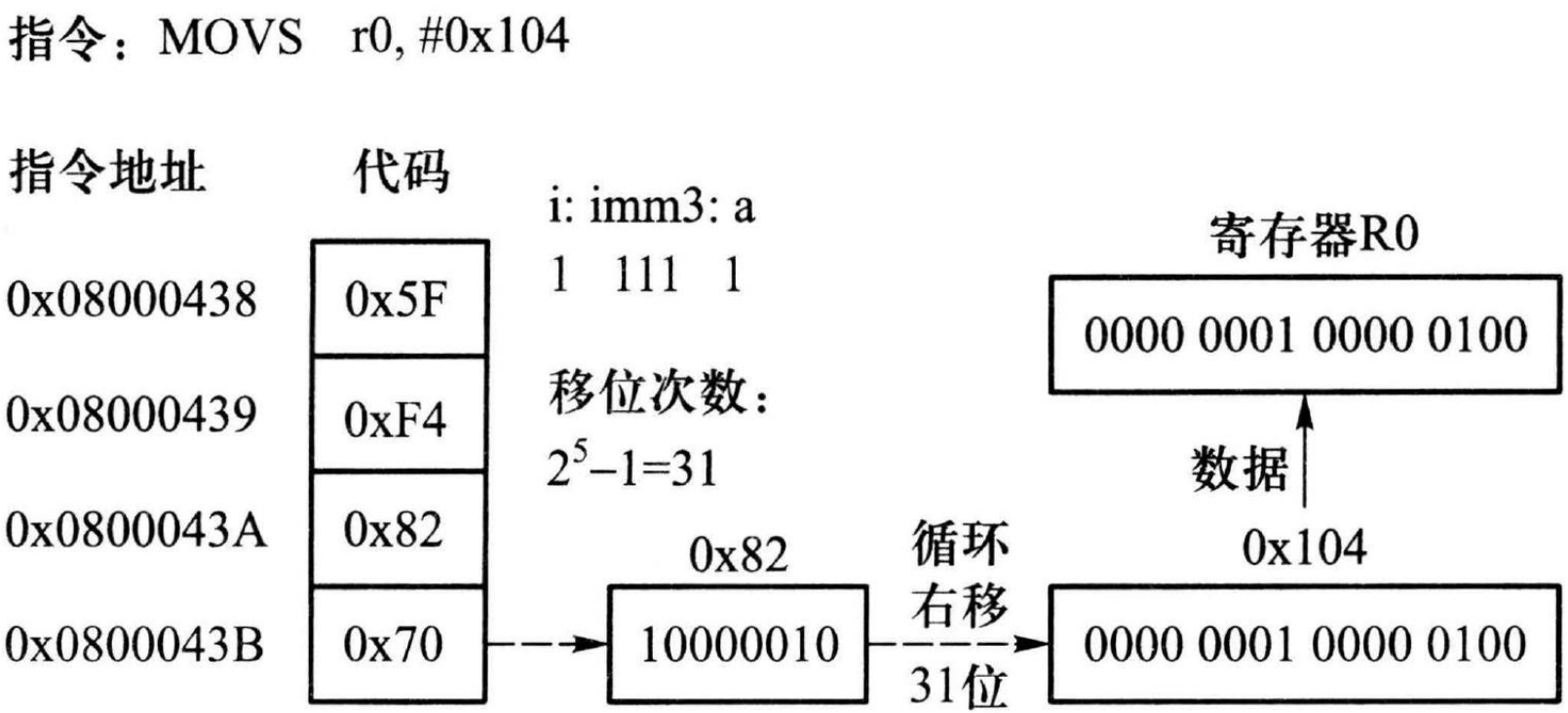
MO



3.2 ARM处理器寻址方式

* **立即寻址**

MOVS R0, #0x104 的十六进制编码为： 0xF45F7082



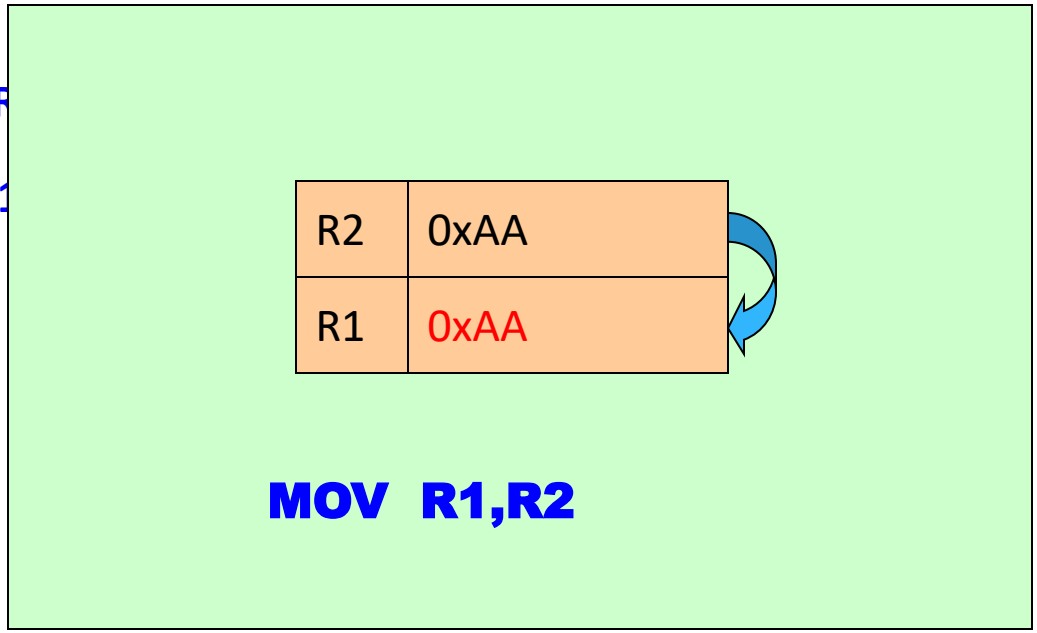
3.2 ARM处理器寻址方式

* **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：

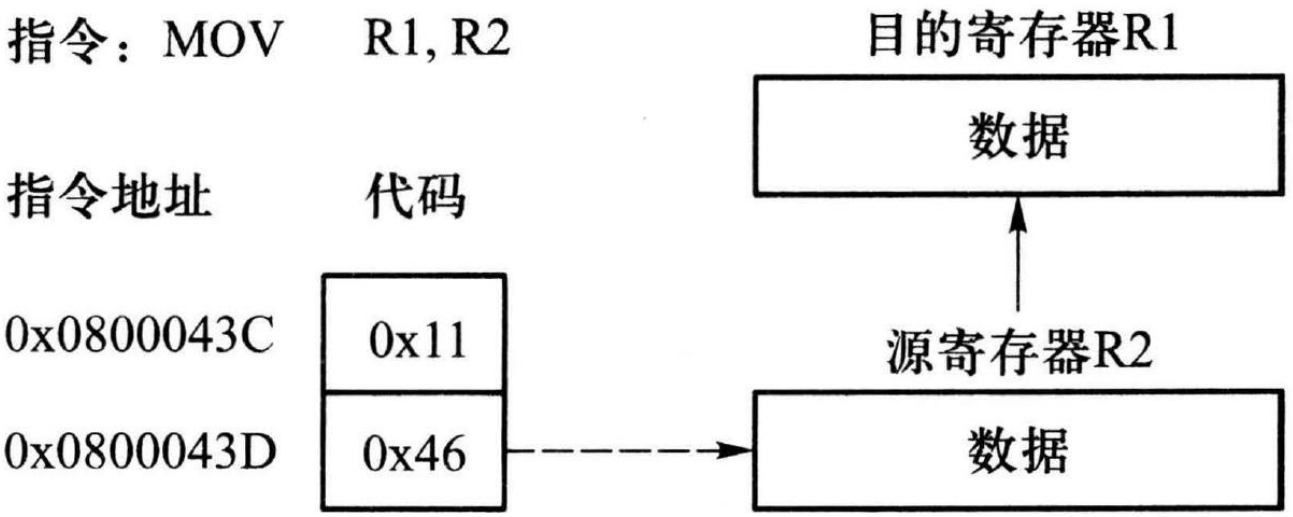
MOV R1,R

SUB R0,R1



3.2 ARM处理器寻址方式

* 寄存器寻址



3.2 ARM处理器寻址方式

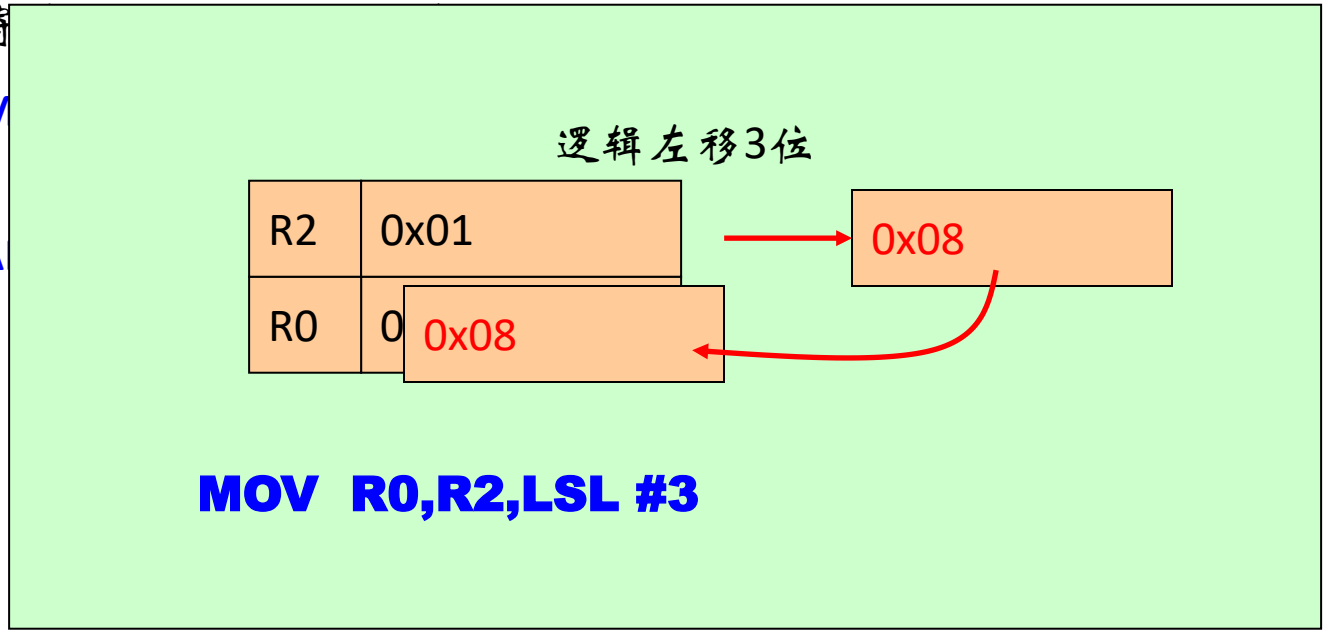
* **寄存器移位寻址**

ARM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

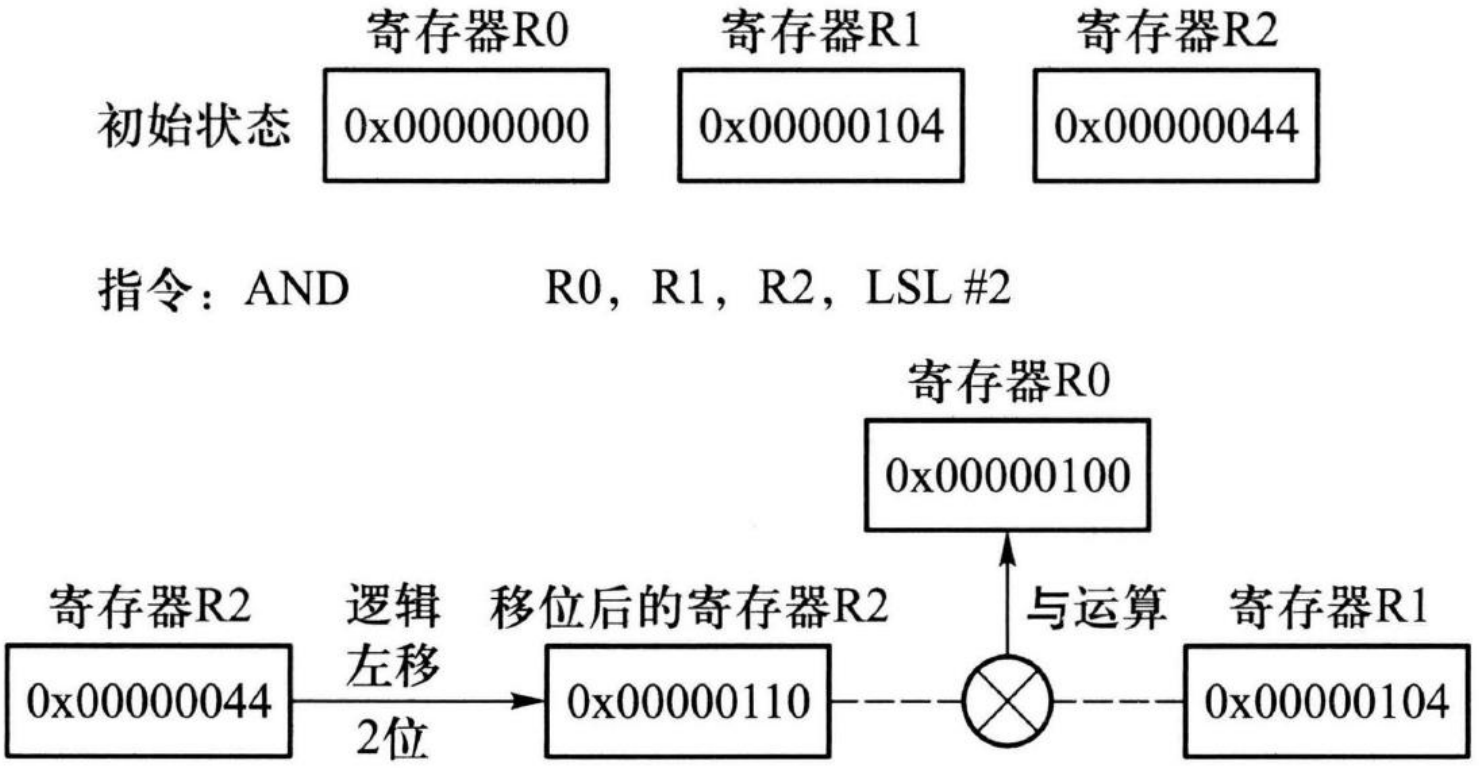
M

A



3.2 ARM处理器寻址方式

* **寄存器移位寻址**

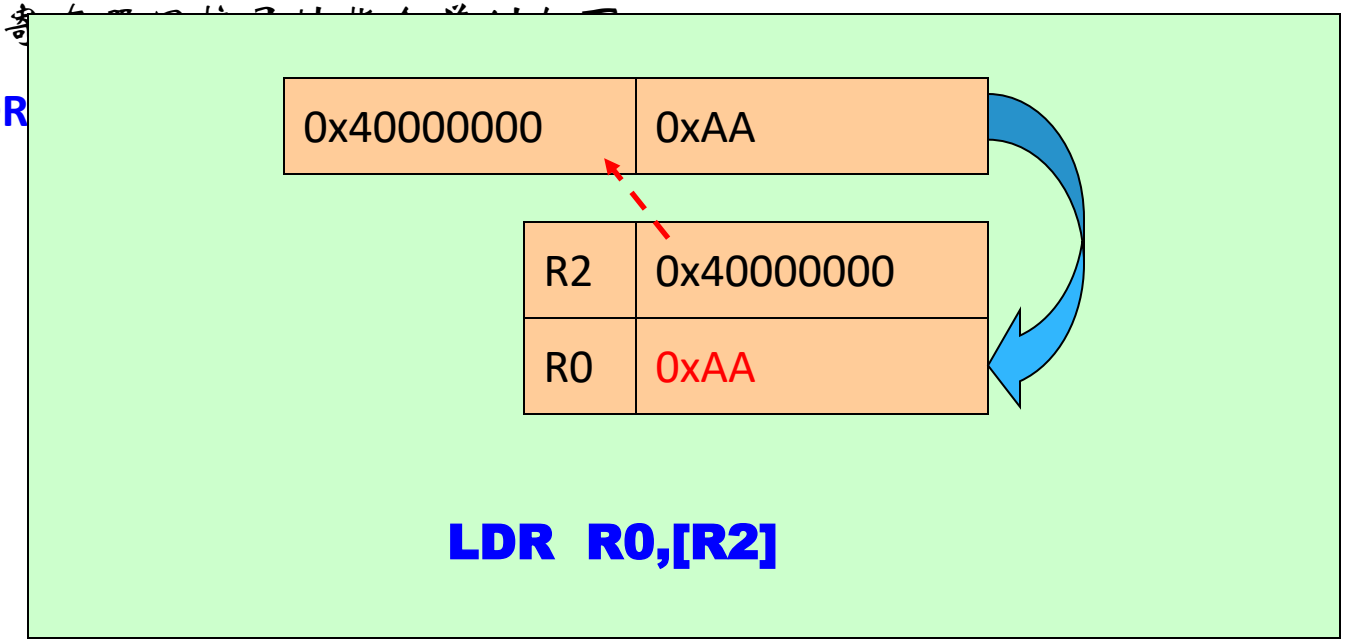


3.2 ARM处理器寻址方式

* **寄存器间接寻址**

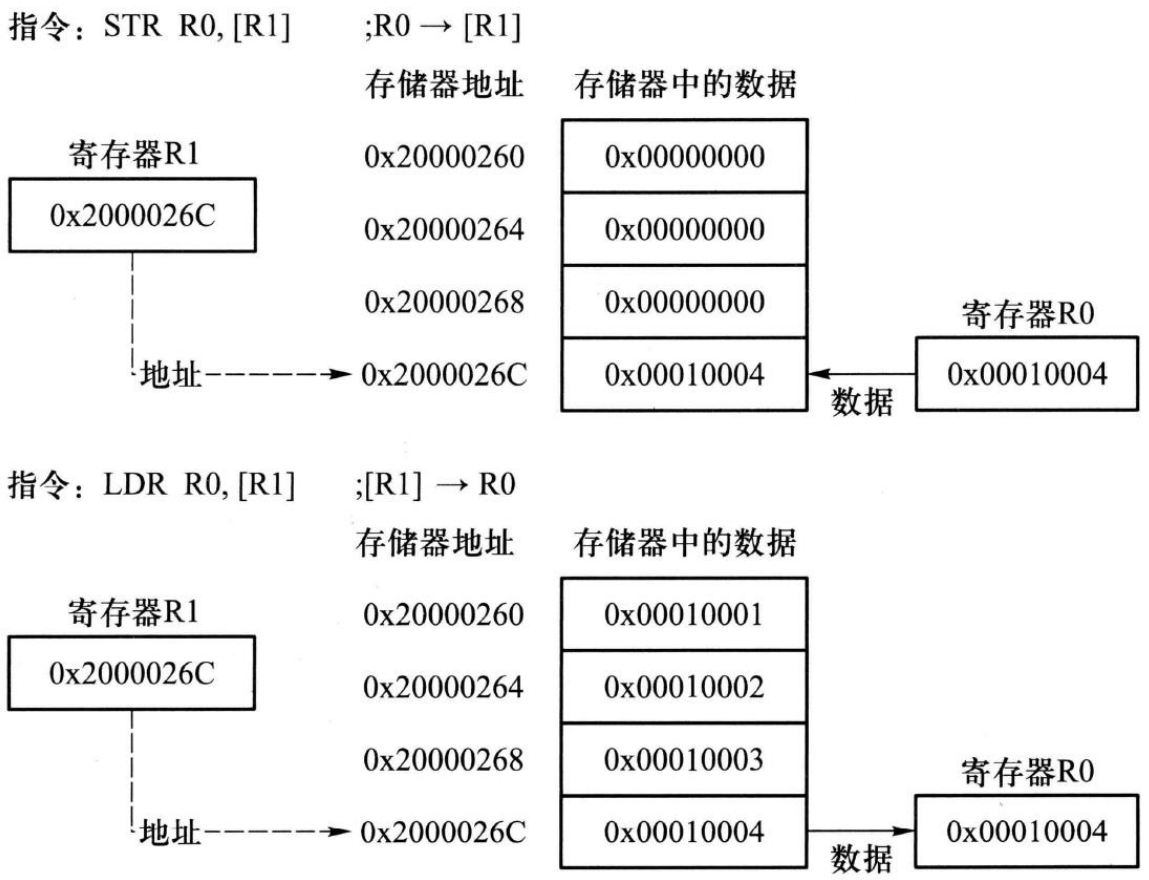
指令中的地址码给出的是一个通用寄存器的编号，所需的操作数保存在寄存器指定地址的存储单元中，即寄存器为操作数的地址指针。寄

LDR



3.2 ARM处理器寻址方式

* **寄存器间接寻址**



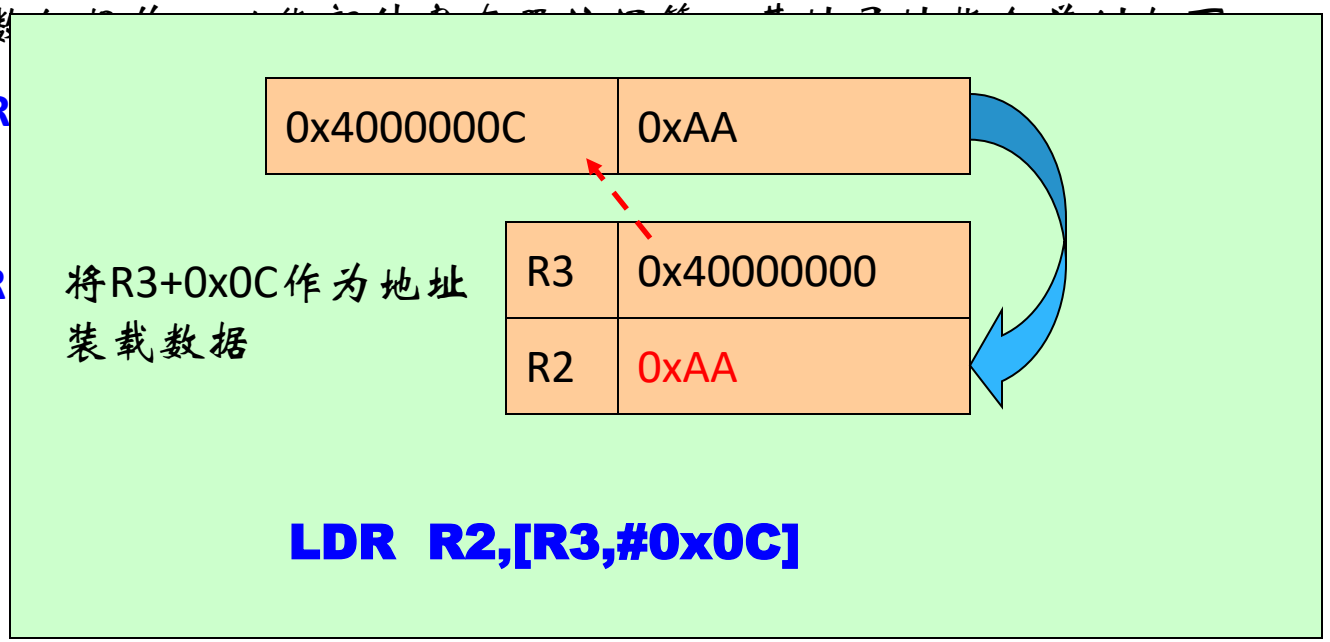
3.2 ARM处理器寻址方式

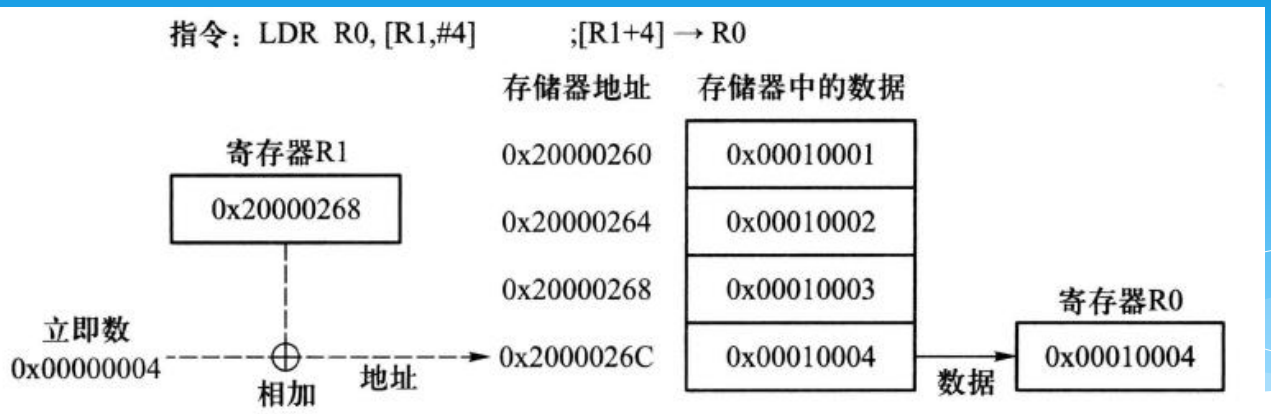
* **寻址方式分类——基址变址寻址**

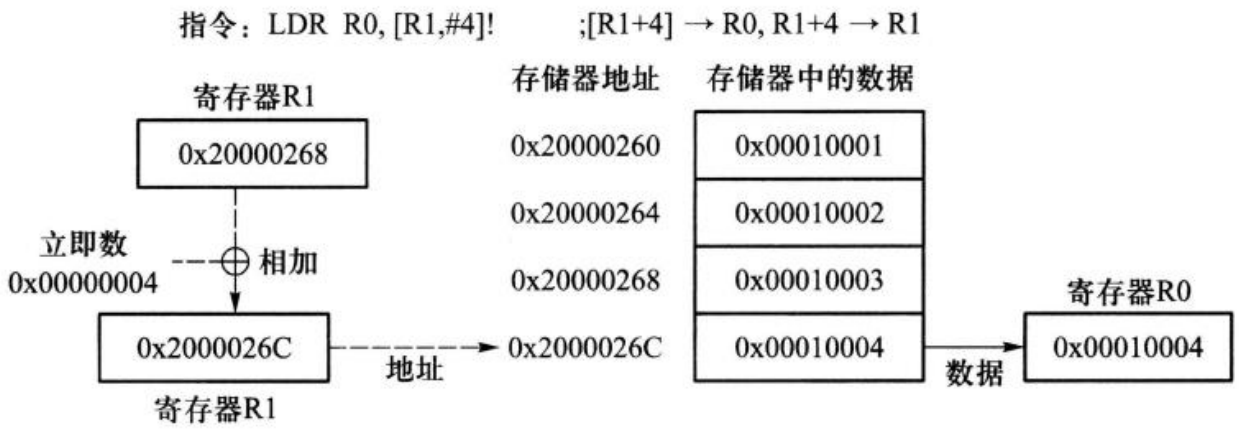
将基址寄存器的内容与指令中给出的偏移量相加，形成操作数的有效地址。基址寻址用于访问基址附近的存储单元，常用于查表

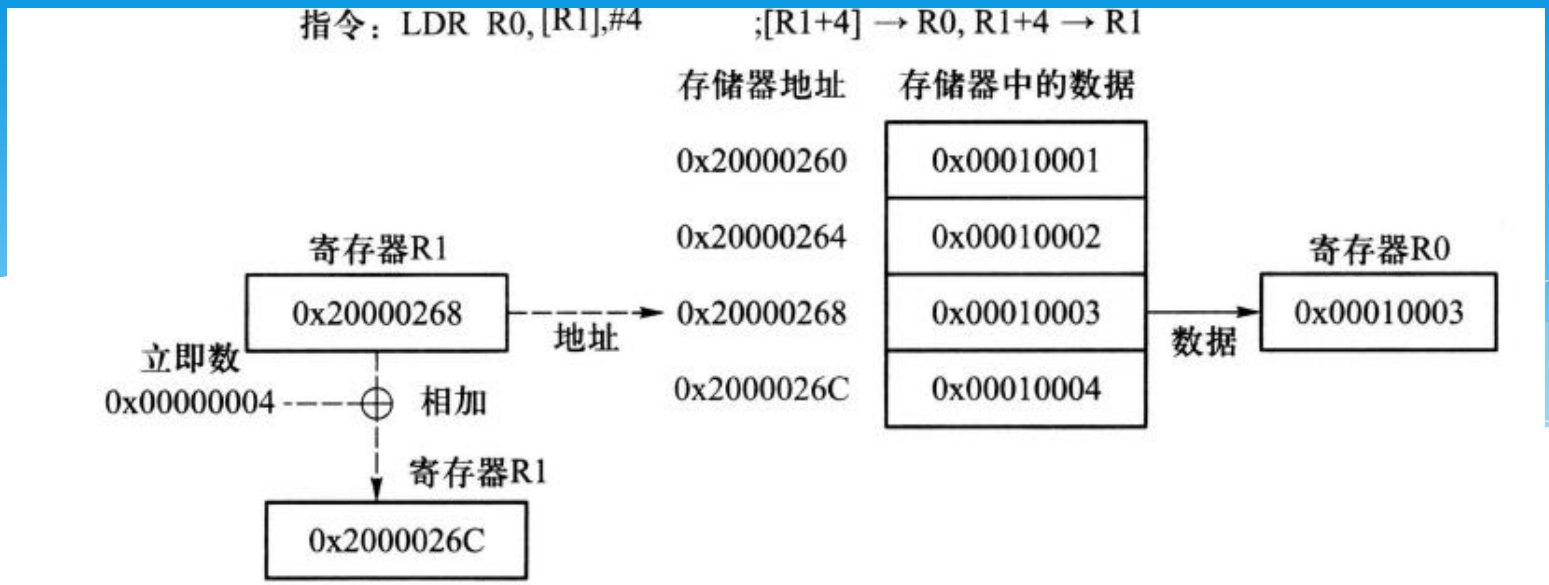
LDR

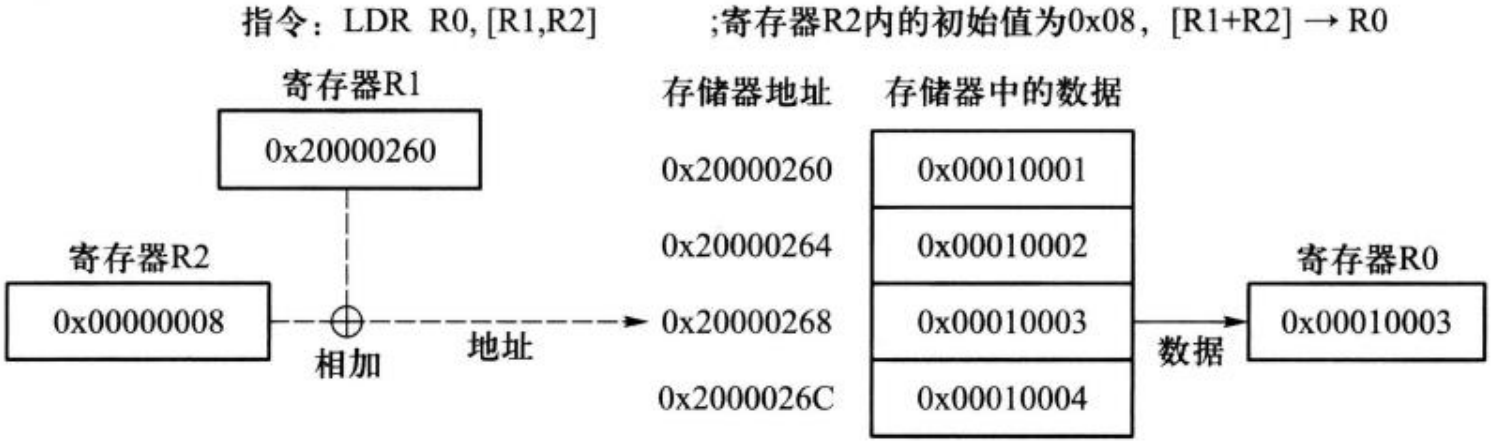
STR




指令：LDR R0, [R1,#4]
;[R1+4] → R0
存储器地址
存储器中的数据
寄存器R1
0x20000268
0x20000260
0x00010001
0x20000264
0x00010002
0x20000268
0x00010003
立即数
0x00000004
相加
地址
0x2000026C
0x00010004
数据
寄存器R0
0x00010004


指令：LDR R0, [R1,#4]!
;[R1+4] → R0, R1+4 → R1
存储器地址
存储器中的数据
寄存器R1
0x20000268
0x20000260
0x00010001
0x20000264
0x00010002
立即数
0x00000004
相加
0x20000268
0x00010003
寄存器R0
0x2000026C
地址
0x2000026C
0x00010004
数据
0x00010004
寄存器R1

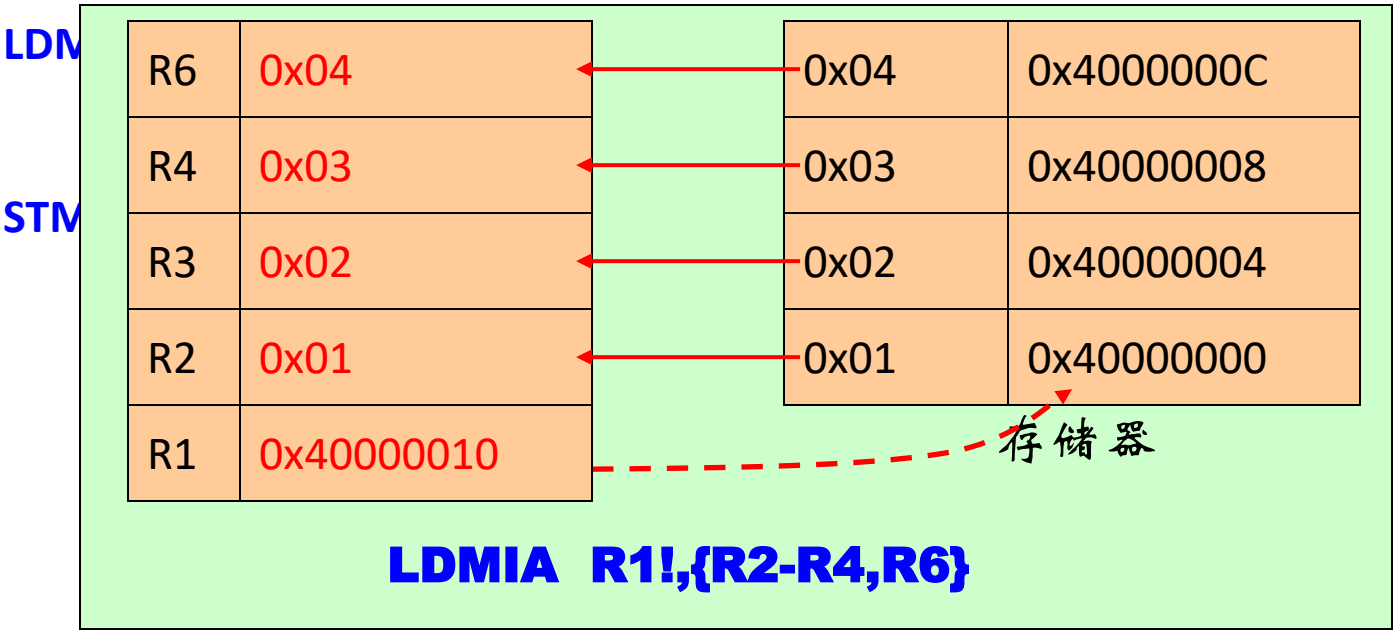

指令：LDR R0, [R1],#4
;[R1+4] → R0, R1+4 → R1
存储器地址
存储器中的数据
0x20000260
0x00010001
0x20000264
0x00010002
0x20000268
0x00010003
0x2000026C
0x00010004
寄存器R1
0x20000268
地址
立即数
0x00000004
相加
寄存器R1
0x2000026C
寄存器R0
0x00010003
数据


指令：LDR R0, [R1,R2]
;寄存器R2内的初始值为0x08，[R1+R2] → R0
寄存器R1
0x20000260
寄存器R2
0x00000008
相加
地址
存储器地址
存储器中的数据
0x20000260
0x00010001
0x20000264
0x00010002
0x20000268
0x00010003
0x2000026C
0x00010004
寄存器R0
0x00010003
数据

3.2 ARM处理器寻址方式

* 多寄存器寻址

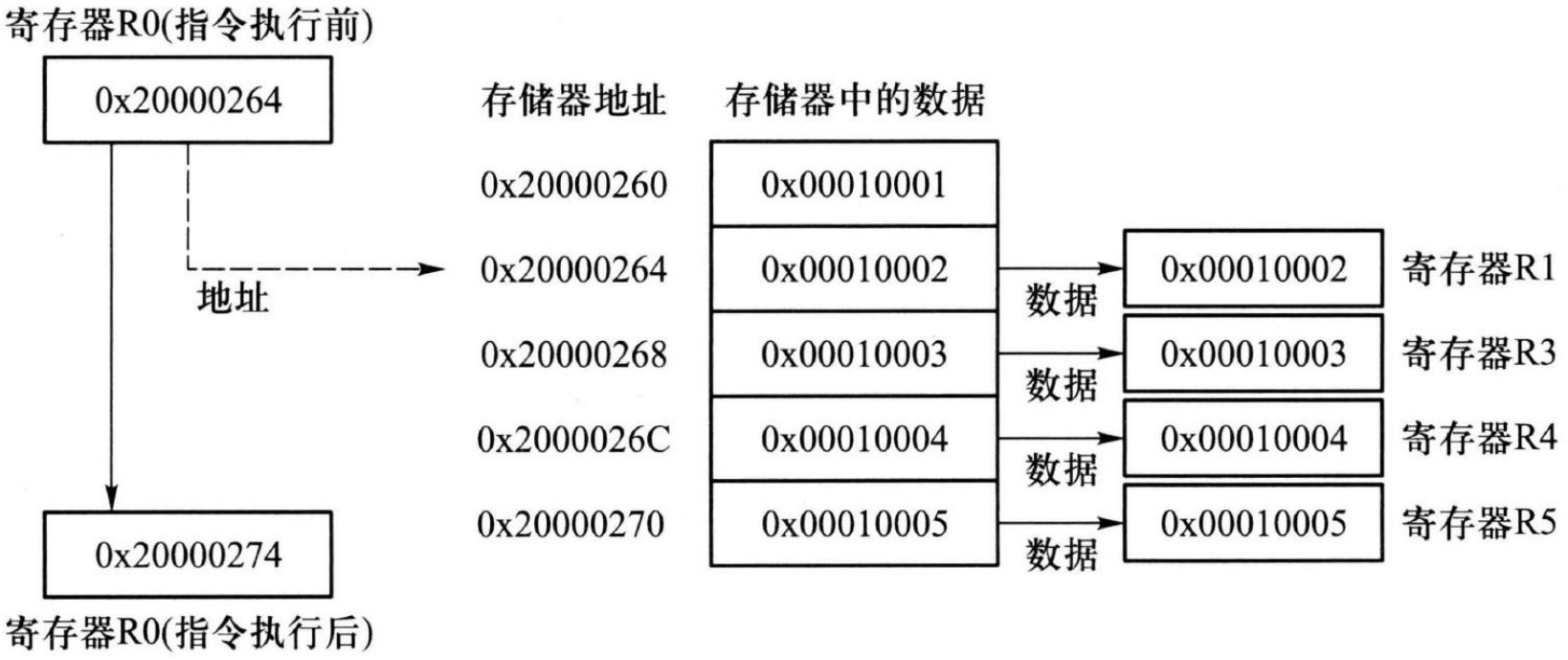
一次可传送几个寄存器值，允许一条指令传送16个寄存器的任何子集或所有寄存器。多寄存器寻址指令举例如下：



3.2 ARM处理器寻址方式

* **多寄存器寻址**

指令：LDMIA　R0!, {R1, R3-R5}　;[R0]→R1, [R0+4]→R3, [R0+8]→R4, [R0+12]→R5



3.1 ARM处理器寻址方式

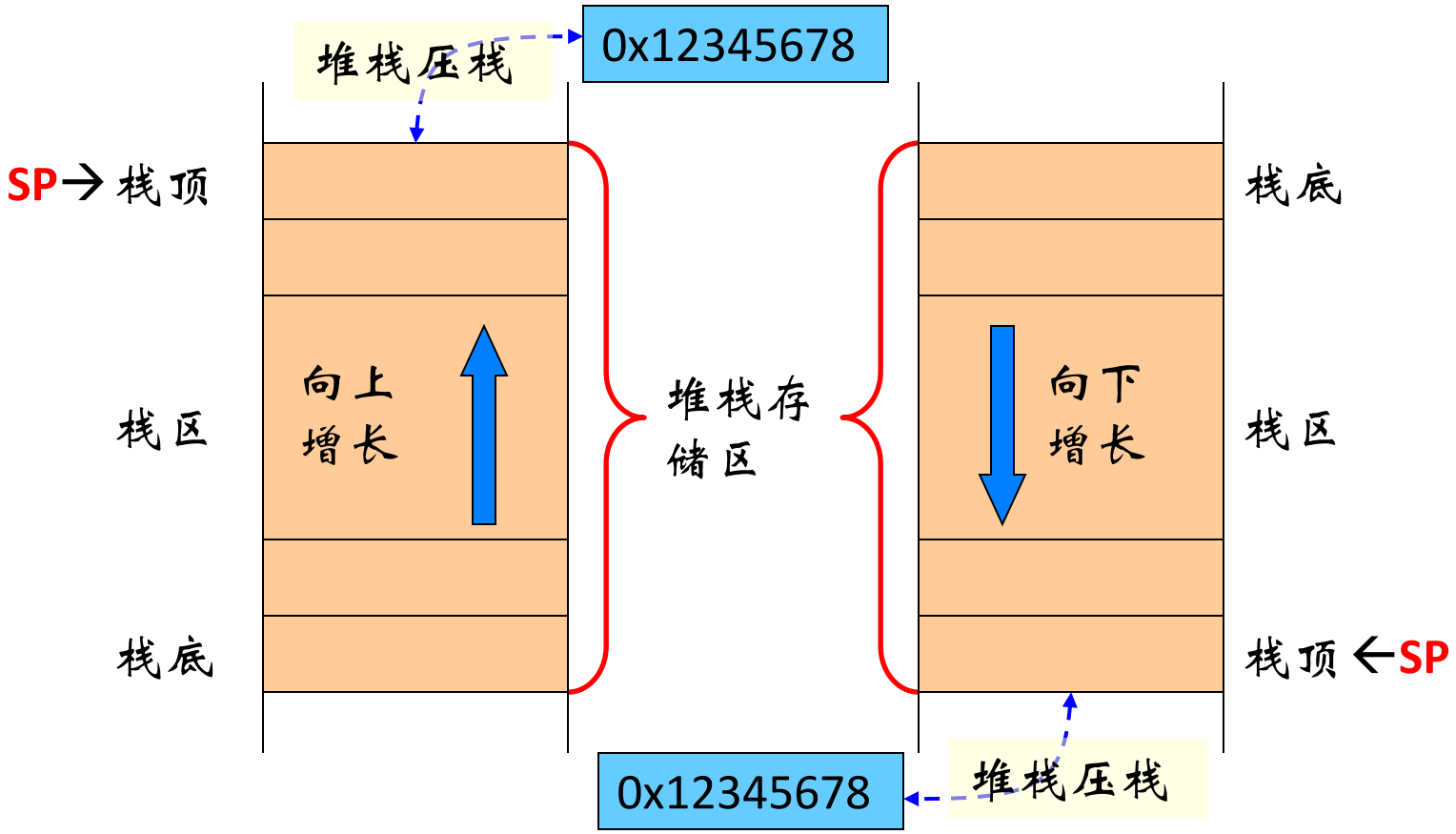
* 堆栈寻址

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

3.2 ARM处理器寻址方式

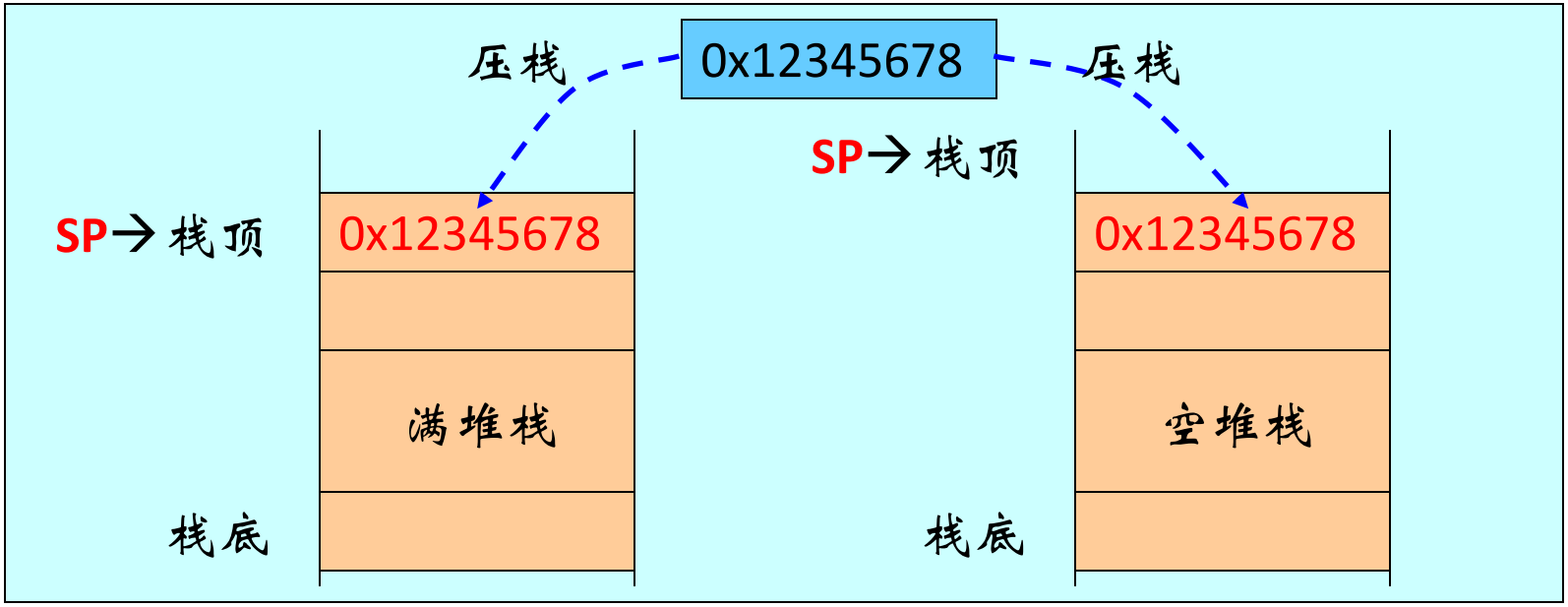
* **堆栈寻址**



3.2 ARM处理器寻址方式

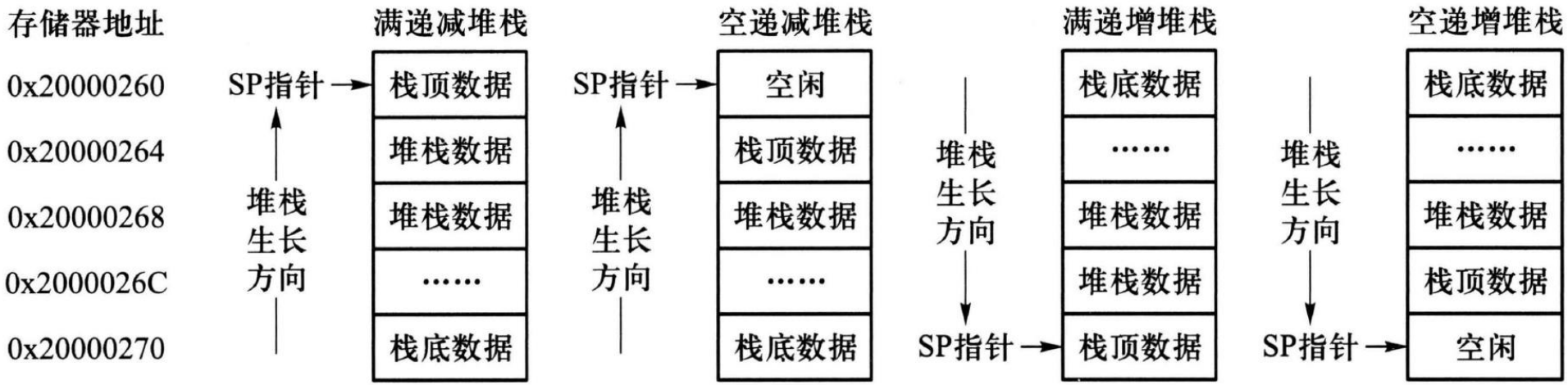
* **堆栈寻址**

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。



3.2 ARM处理器寻址方式

* 堆栈寻址



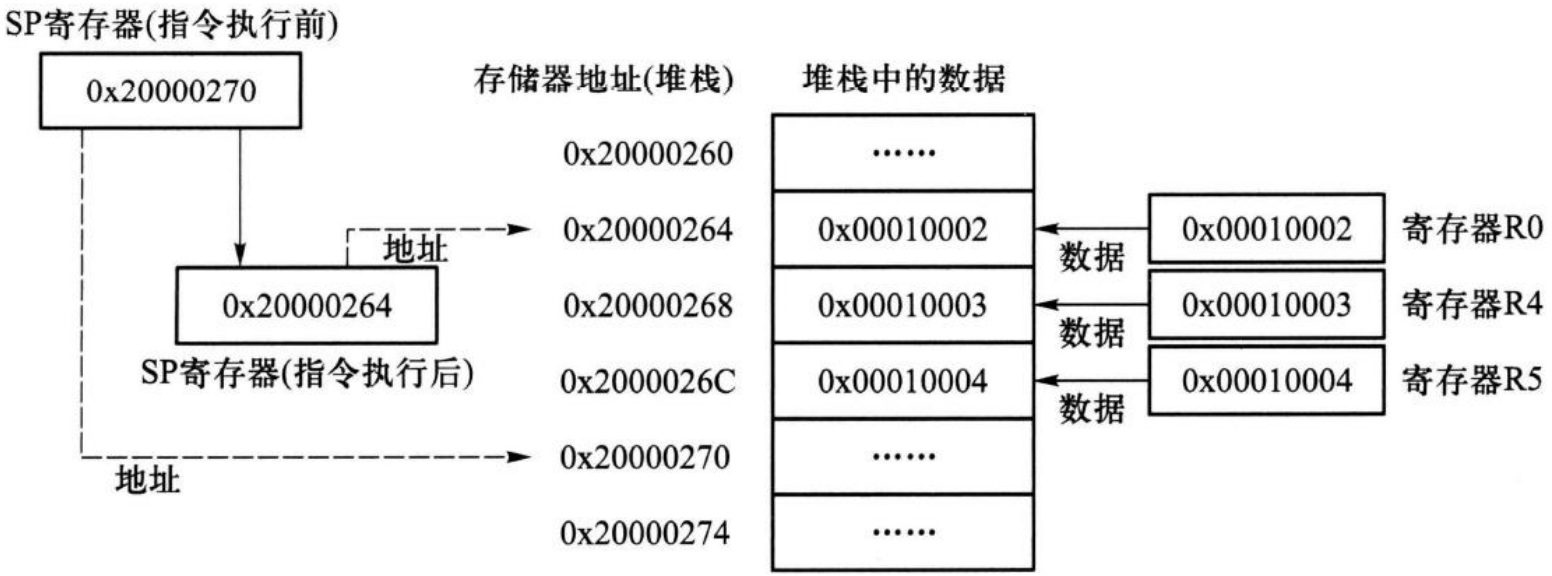
3.2 ARM处理器寻址方式

* **堆栈寻址**

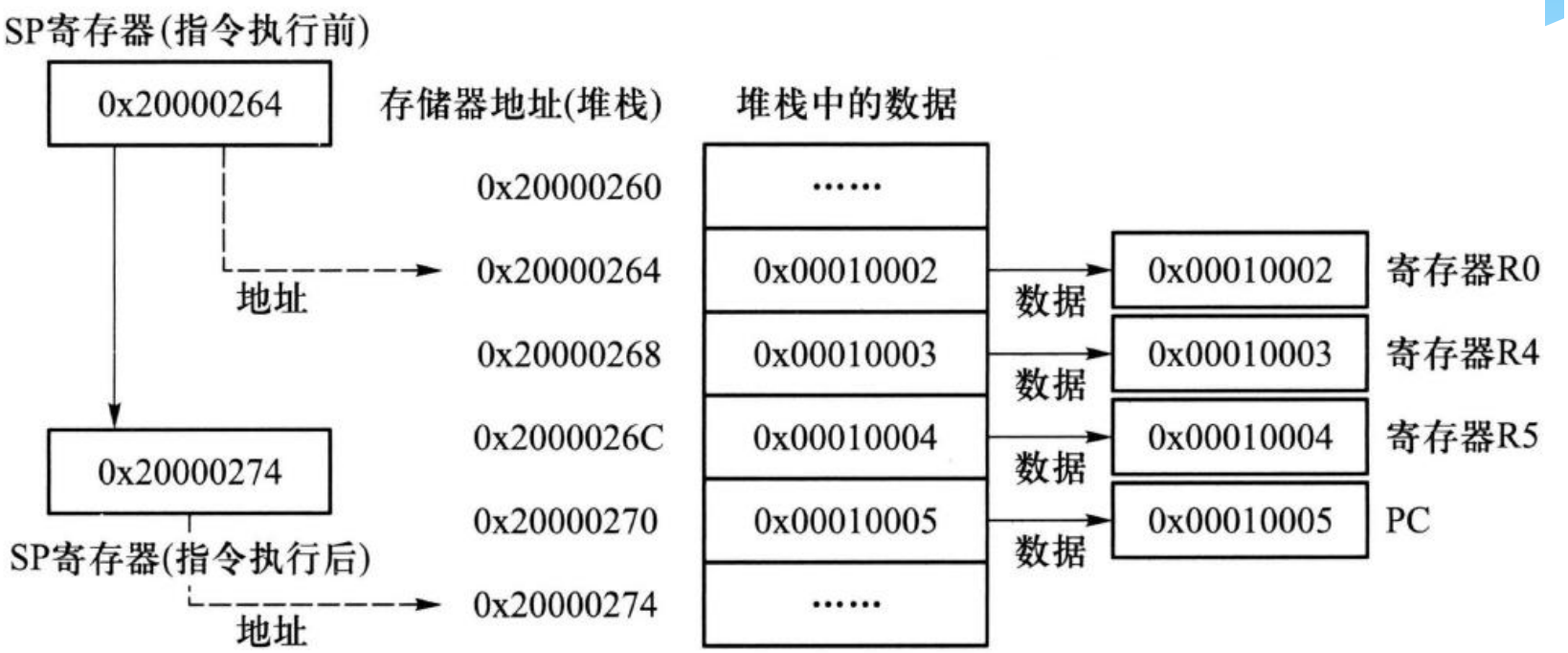
堆栈方式：

- 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LDMIA、STMDB、PUSH/POP等；

指令：PUSH {R0, R4, R5} ;R5→[SP–4], R4→[SP–8], R0→[SP–12], SP–12→SP



指令：POP {R0, R4-R5, PC} ;[SP]→R0, [SP+4]→R4, [SP+8]→R5, [SP+12]→PC, SP+16→SP



3.2 ARM处理器寻址方式

* 堆栈寻址

```
POP     {R0, R4-R5, PC}       ; 出栈操作，寄存器列表中不能有 SP 寄存器
                              ; R0 ←[ SP ], R4 ←[ SP+4 ], R5 ←[ SP+8 ], PC ←[ SP+12 ], SP ← SP+16
LDMIA   SP!, {R0, R4-R5, PC}  ; 同前一 POP 指令等效
PUSH        {R0, R4, R5}      ; 入栈操作，寄存器列表中不能有 PC 和 SP 寄存器
                              ;[ SP-4 ]← R5,[ SP-8 ]← R4,[ SP-12 ]← R0, SP ← SP-12
STMDB   SP!, {R0, R4, R5}     ; 同前一 PUSH 指令等效
```

简单的ARM程序

```
;文件名：TEST1.S
;功能：实现两个寄存器相加
;说明：使用“；”进行注释，调试
        AREA    init,CODE,READONLY   ;声明代码段Example1
        Reset_Handler proc
;implemet code here.
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
```

标号顶格写

实际代码段

声明文件结束

4. ARM指令集

ARM指令种类

- 存储器访问指令
- 数据处理指令
- 乘法指令
- 移位和循环指令
- 符号扩展指令
- 字节调序指令
- 位域处理指令
- 比较和测试指令
- 子程序调用与无条件转移指令
- 饱和运算指令

ARM存储器访问指令——（1）单寄存器存取

LDR/STR指令用于对内存变量的访问、内存缓冲区数据的访问、查表、外围部件的控制操作等。

若使用LDR指令加载数据到PC寄存器，则实现程序跳转功能，这样也就实现了程序散转。

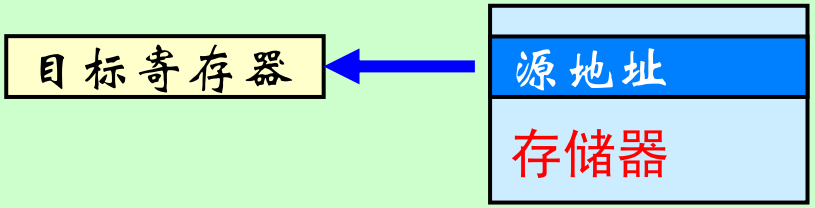
所有单寄存器加载/存储指令可以实现字、（无符号/有符号）半字、（无符号/有符号）字节的操作。

表 3-3-3　单寄存器加载和存储指令

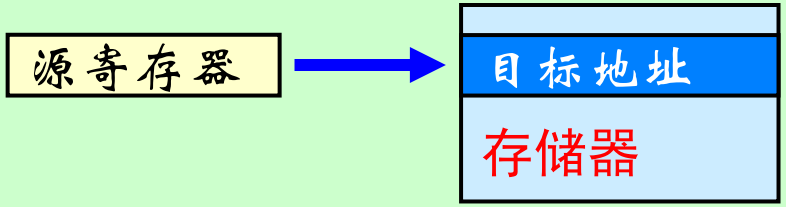
指令	功能描述
LDRB Rd, [Rn, #offset]	从地址 Rn+offset 处读取一个字节到 Rd
LDRH Rd, [Rn, #offset]	从地址 Rn+offset 处读取一个半字到 Rd
LDR Rd, [Rn, #offset]	从地址 Rn+offset 处读取一个字到 Rd
LDRD Rd1, Rd2, [Rn, #offset]	从地址 Rn+offset 处读取一个双字(64 位整数)到 Rd1(低 32 位)和 Rd2(高 32 位)中
STRB Rd, [Rn, #offset]	把 Rd 中的低字节存储到地址 Rn+offset 处
STRH Rd, [Rn, #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

ARM存储器访问指令——（1）单寄存器存取

装载指令：LDR　　目标寄存器，源地址



存储指令：STR　　源寄存器，目标地址



ARM存储器访问指令——（1）单寄存器存取

装载指令：LDRx

存储指令：STRx

LDR/STR指令搭配不同的后缀实现不同方式的单寄存器存取操作：

字/半字(H)/字节(B)

双字（D）控制

无/有符号(S)控制

• ARM存储器访问指令——（2）地址形式

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]

• ARM存储器访问指令——（3）自动索引

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

零偏移： LDR Rd,[Rn]

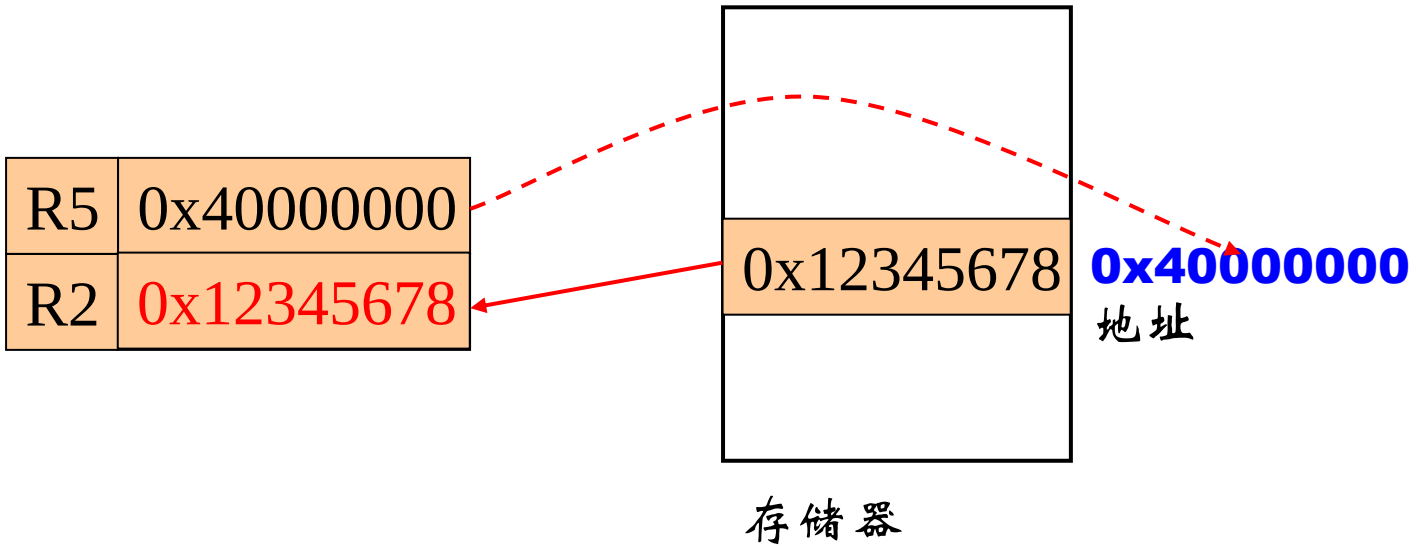
前索引偏移： LDR Rd,[Rn,#0x04]!

后索引偏移： LDR Rd,[Rn],#0x04

• ARM存储器访问指令——单寄存器转载应用

应用示例1：

LDR　R2,[R5]　;将R5指向地址的字数据存入R2



• ARM存储器访问指令——单寄存器保存应用

应用示例2：

STR　R1,[R2,#0x04] ；将R1的数据存储到R0+0x04地址

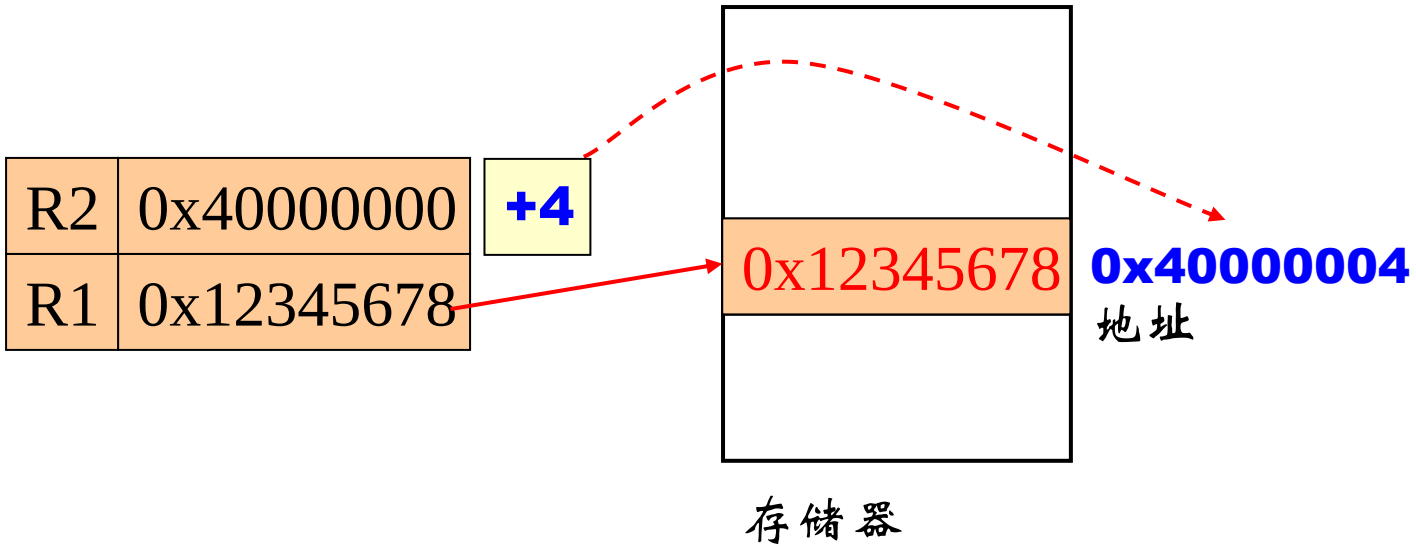


表 3-3-3　单寄存器加载和存储指令

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

ARM存储器访问指令——（4）多寄存器存取

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。

LDM为加载多个寄存器；

STM为存储多个寄存器。

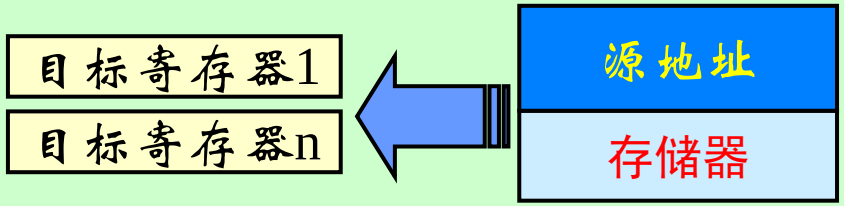
允许一条指令传送16个（R0～R15）寄存器的任何子集或所有寄存器。它们主要用于现场保护、数据复制、常数传递等。

表 3-3-3　单寄存器加载和存储指令

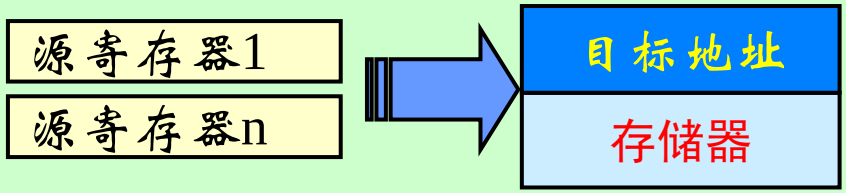
指令	功能描述
LDRB Rd, [Rn, #offset]	从地址 Rn+offset 处读取一个字节到 Rd
LDRH Rd, [Rn, #offset]	从地址 Rn+offset 处读取一个半字到 Rd
LDR Rd, [Rn, #offset]	从地址 Rn+offset 处读取一个字到 Rd
LDRD Rd1, Rd2, [Rn, #offset]	从地址 Rn+offset 处读取一个双字(64 位整数)到 Rd1(低 32 位)和 Rd2(高 32 位)中
STRB Rd, [Rn, #offset]	把 Rd 中的低字节存储到地址 Rn+offset 处
STRH Rd, [Rn, #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

ARM存储器访问指令——（4）多寄存器存取

装载指令：LDM　源地址,目标寄存器列表



存储指令：STM　目标地址,源寄存器列表



ARM存储器访问指令——（4）多寄存器存取

搭配不同后缀实现不同地址增长方式：

装载指令：**LDMx**

存储指令：**STMx**

IA：每次传送后地址加4

DB：每次传送前地址减4

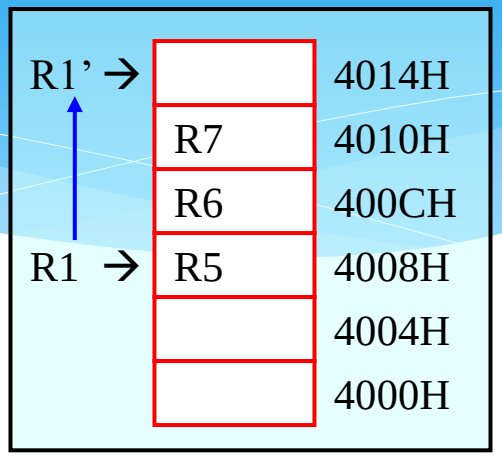
压栈：**PUSH**

出栈：**POP**

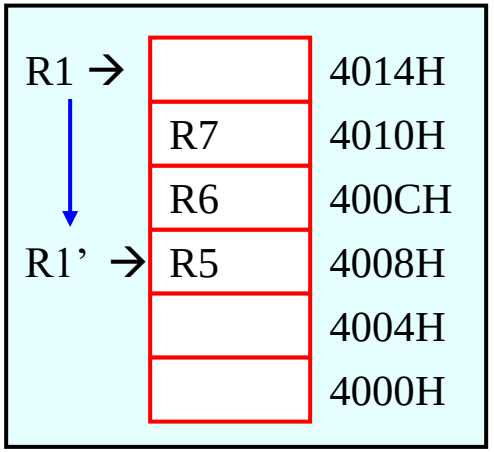
ARM存储器访问指令——（4）多寄存器存取

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**DB** R1!,{R5-R7}

• ARM存储器访问指令——（4）多寄存器存取

应用示例3：

LDMIA R1!,{R2-R4,R6}

将R1指向的内存数据读取到R0-R4和R6寄存器中

R6	0x??
R4	0x??
R3	0x??
R2	0x??
R1	0x40000000

0x04	0x4000000C
0x03	0x40000008
0x02	0x40000004
0x01	0x40000000

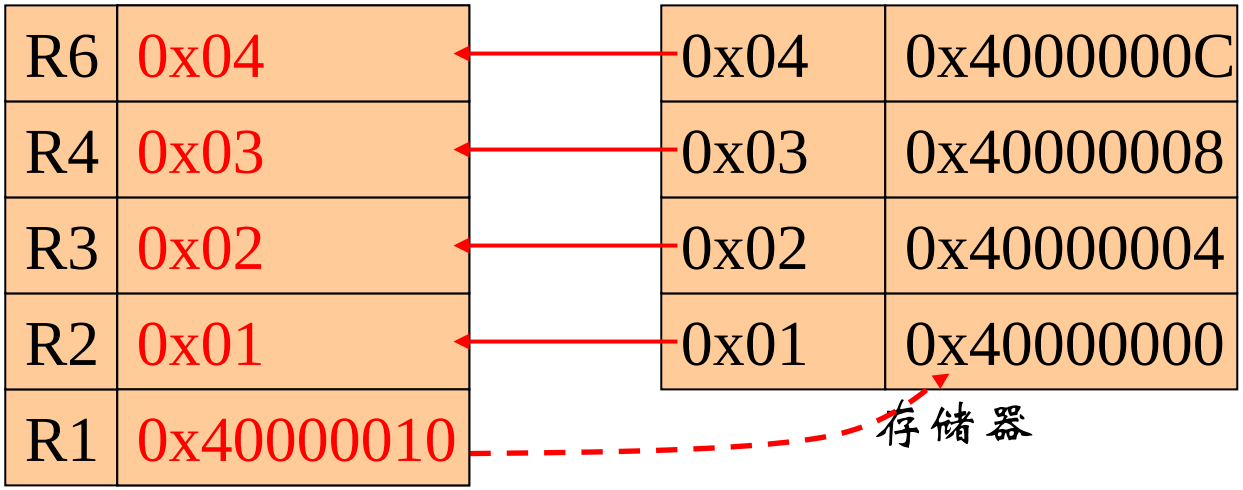
存储器

• ARM存储器访问指令——（4）多寄存器存取

应用示例3：

LDMIA R1!,{R2-R4,R6}

将R1指向的内存数据读取到R0-R4和R6寄存器中



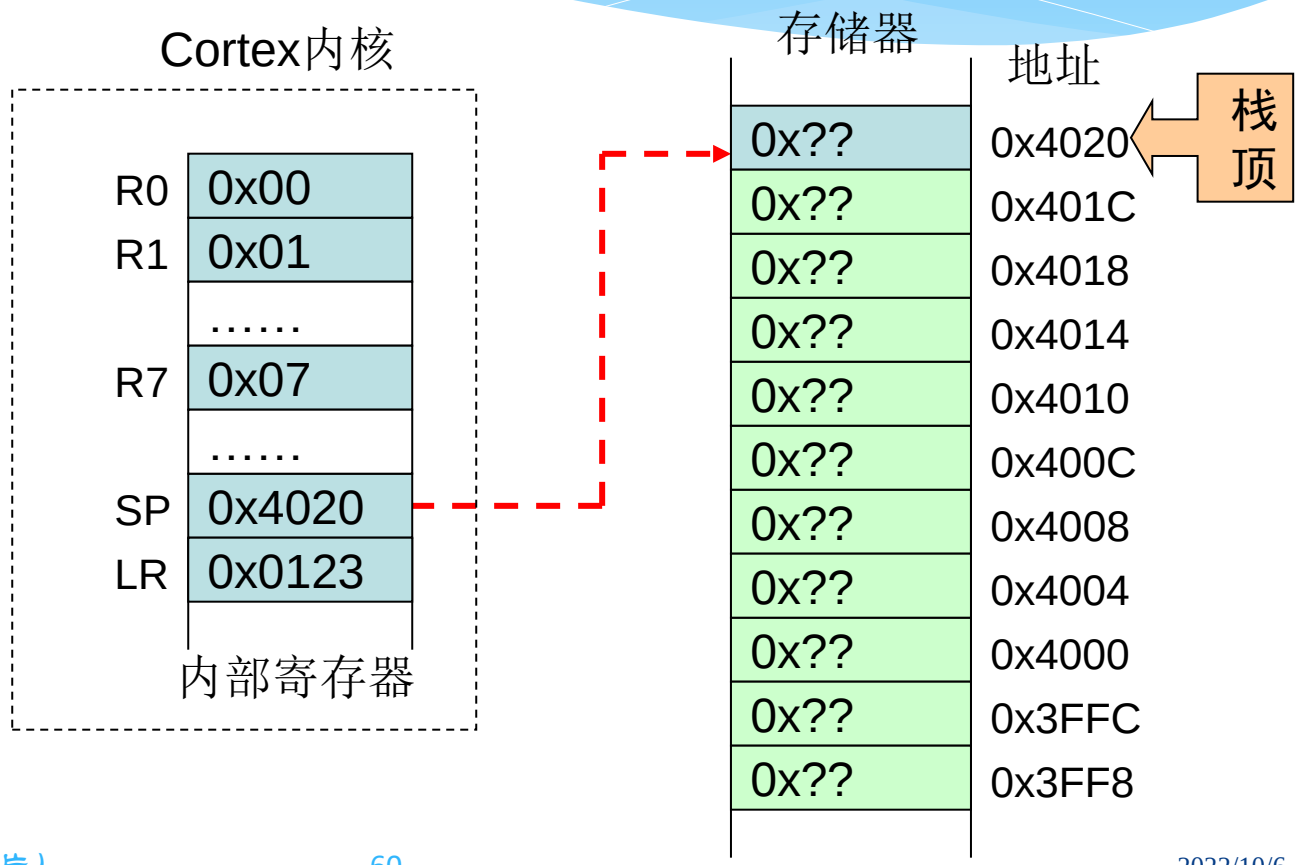
• ARM存储器访问指令——满递减压栈操作

应用示例4：

PUSH {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；



• ARM存储器访问指令——满递减压栈操作

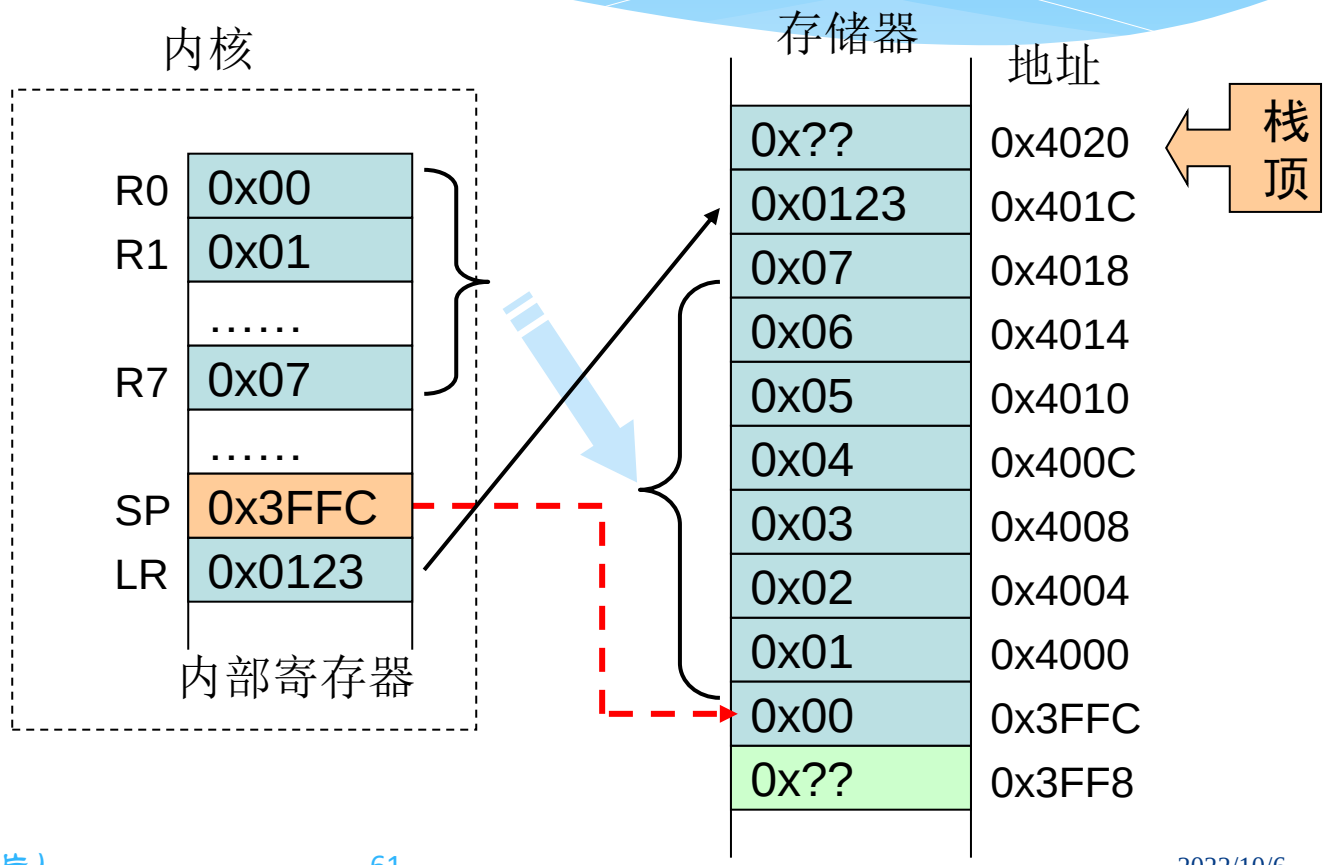
应用示例4：

POP {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



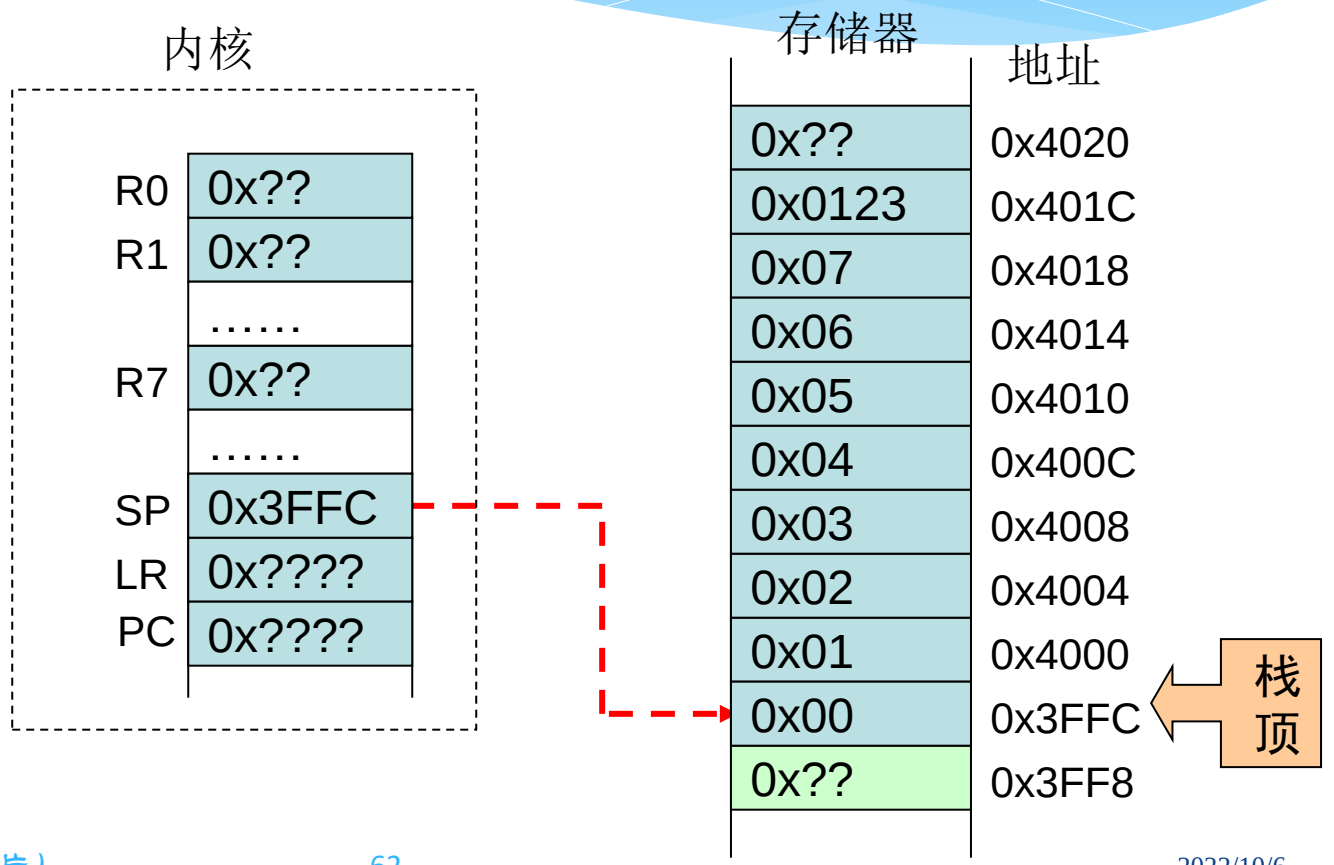
• ARM存储器访问指令——满递减出栈操作

应用示例5：

POP {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；



• ARM存储器访问指令——满递减出栈操作

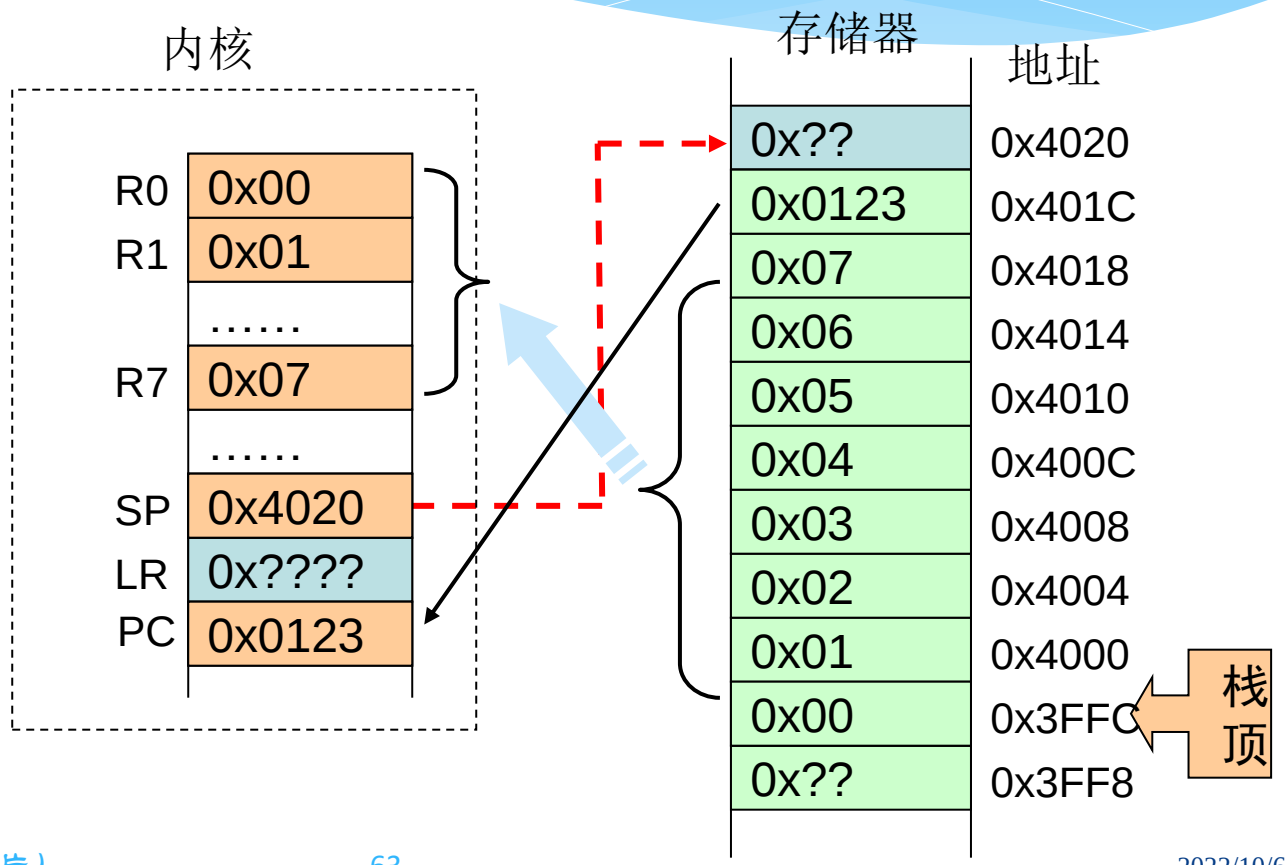
应用示例5：

POP {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



STM/LDM与PUSH/POP区别

（1）STM/LDM 指令能对任意的地址空间进行操作，而 PUSH/POP 指令只能对堆栈空间进行操作；

（2）STM/LDM 指令的生长方式可以支持向上和向下两种方式，而 PUSH/POP 指令只能支持向下生长；

（3）当两对指令的操作数都为 SP 时，STM/LDM 指令可以选择是否回写修改 SP 值，而 PUSH/POP 指令会自动修改 SP 值。

* **STMDB SP!,{R3-R7,LR}**

* **PUSH {R3-R7,LR}**