嵌入式系统

北京邮电大学
计算机学院

戴志涛

Cortex M3/M4的指令集与程序设计

- Cortex M3/M4的指令集
- Cortex M3/M4的寄存器
- ARM汇编指令格式
- 操作数的寻址方式
- 汇编伪操作
- Cortex M3/M4常用指令
- IF-THEN 指令块
- ARM汇编伪指令

ARM CORTEX-M3/M4的指令集

北京郵電大學

ARM指令集

- 经典ARM处理器的指令集
  - 32位ARM指令集
  - 功能强大，性能优越
  - 与8位和16位处理器相比，程序的存储器空间更大
  - 功耗更高

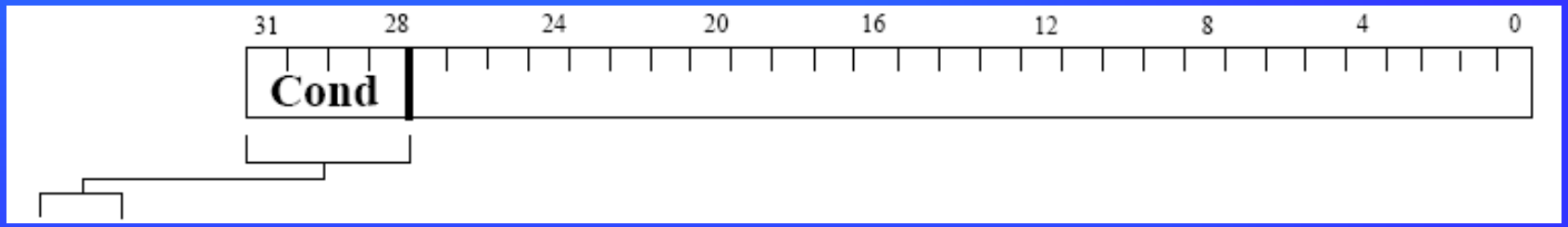
ARM指令集特点

- 32位等长指令字指令格式
- 所有ARM指令都使用4位条件编码决定指令是否执行
- Load/Store体系结构
  - 数据操作只会用到寄存器，不会直接访问内存
- 3操作数格式

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	r3	r1	r2

ARM指令的条件字段

- 当处理器工作在ARM状态时，几乎所有指令均为条件执行
  - 执行条件满足：指令被执行
  - 执行条件不满足：指令被忽略
  - 条件：指令中的条件字段
  - 状态：CPSR中的条件码



- 每一条ARM指令包含4位条件码，位于指令的最高4位[31:28]

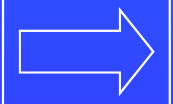
ARM指令中的条件字段

条件码	助记符后缀	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
1000	HI	C置位Z清零	无符号数大于
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

ARM指令中的条件字段

- EQ/NE: 等于/不等于(equal / not equal)
- HS/LO: 无符号数高于或等于/无符号数小于(higher or same/lower)
- HI/LS: 无符号数高于/无符号数低于或等于(higher/lower or same)
- GE/LT: 有符号数大于或等于/有符号数小于(greater or equal/less than)
- GT/LE: 有符号数大于/有符号数小于或等于(greater than/less or equal)
- MI/PL: 负/非负
- VS/VC: 溢出/不溢出(overflow set/overflow clear)
- CS/CC: 进位/无进位(carry set/carry clear)

```
if ( (a == b) && (c == d) )
{
        e++;
}
```



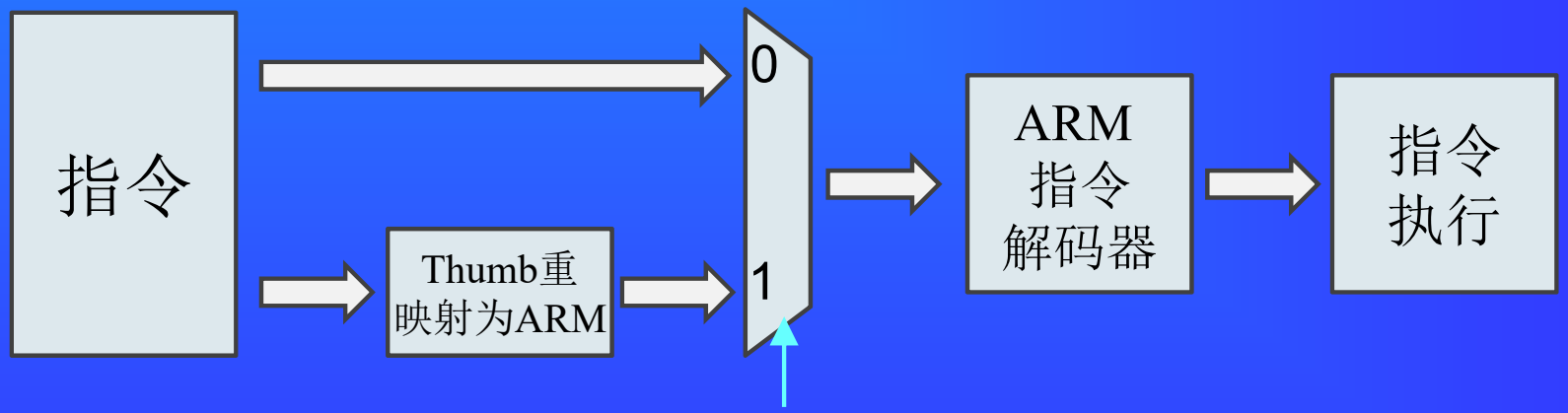
```
// r0, r1, r2, r3, r4: a, b, c, d, e
cmp    r0, r1
cmpeq  r2, r3
addeq  r4, r4, #1
```

ARM和Thumb指令集

- Thumb-1 指令集
  - 16位指令集
  - 1995年首先被应用于ARM7TDMI处理器
    - T: Thumb
    - D: 在片调试(Debug)功能，允许处理器响应调试请求暂停
    - M: 增强型乘法器，产生全64位结果
    - I: 嵌入式ICE硬件，提供片上断点和调试点支持
  - ARM指令集的功能子集
  - 与32位RISC指令集相比，代码密度大大提高
    - 代码的长度被降低了约30%
    - 性能降低约20%

ARM和Thumb指令集

- ARM搭配Thumb-1指令集
  - 同时兼备32位ARM（高性能）和16位 Thumb-1（高代码密度）的优势
  - 利用多路复用器，在两种状态之间切换：ARM状态（32位）和Thumb状态（16位）；需要切换开销



T bit:
0: 选择 ARM指令集
1: 选择 Thumb指令集

Thumb-2指令集

- Thumb-2指令集：
  - 原有16位Thumb-1指令集增加32位Thumb指令
  - 16位指令与32位指令并存
    - 当一个操作可以使用一条32位指令完成时就使用32位指令，加快运行速度
    - 当一次操作只需要一条16位指令完成时就使用16位指令，节约存储空间

Thumb-2指令集

- Thumb-2指令集：
  - 原有16位Thumb-1指令集增加32位Thumb指令
  - 16位指令与32位指令并存
  - 兼有ARM和Thumb指令集的优势
    - 与32位ARM指令集相比，代码长度减少约26%，性能不降低
    - 能在一个操作状态下（Thumb状态）处理所有的处理要求
      - 没有状态切换开销，节省执行时间和指令空间
      - 不需要分开ARM代码和Thumb代码的源文件，软件开发与维护更轻松
      - 便于优化效率和性能

Cortex-M4的指令集

- ARMv7E-M Thumb指令集（32位Thumb-2）
  - ARMv7-M架构+DSP扩展
- 只使用Thumb指令，始终在Thumb状态
  - 大多数指令16位长，有些32位
  - 大多数16位指令只能访问低寄存器（R0到R7）
  - 有些指令可以访问高寄存器（R8-R15）
- 指令半字对齐
- 程序计数器是奇数（lsb = 1）时表示Thumb状态
  - 不允许切换回ARM状态，跳转到偶数地址会产生异常
- 32位寻址空间
- 只有16位的跳转指令支持条件执行

ARM的三种指令集

- ARM指令集：
  - 指令全部32bits
  - 可以使用最少的指令完成功能，在相同频率下运行速度最快
  - 程序占用最多的程序空间
- Thumb-1指令集：
  - 指令全部16bits
  - 需要使用更多的指令完成功能，运行速度慢
  - 占用最少的程序空间
- Thumb-2指令集：
  - 16位指令与32位指令并存，兼有ARM和Thumb指令集的优势
  - 16位Thumb-1指令集的超集
  - 当一个操作可以使用一条32bits指令完成时就使用32bits的指令，加快运行速度
  - 当一个操作只需要一条16bits指令完成时就使用16bits的指令，节约存储空间

ARM CORTEX-M4 处理器的寄存器

Cortex-M4寄存器

- Load-Store架构
  - 所有运算均在寄存器上操作
  - 内存数据需首先加载到寄存器中，在处理器内部运算后再写回内存
- Cortex-M4的寄存器
  - 寄存器组（Register bank）
    - 16个32位寄存器
      - 其中13个为通用寄存器
  - 特殊功能寄存器

经典ARM寄存器组织

- ARM 有37个32位寄存器
  - 1个用作PC（program counter）
  - 1个用作CPSR（current program status register）
  - 5个用作SPSR（saved program status registers）
  - 30个通用寄存器
- 根据处理器的状态及工作模式被安排成不同的组
  - “逻辑”寄存器的名称：R0~R15、CPSR、SPSR
  - R0~R7：不分组寄存器
  - R8~R14：根据工作模式分组的寄存器
  - R15：程序计数器，不分组
  - CPSR：当前程序状态寄存器
  - SPSR：各工作模式下保留CPSR值的寄存器

经典ARM——不同工作模式下的存储器分组

寄存器类别	寄存器在汇编中的名称	各模式下实际访问的寄存器						
		用户	系统	管理员	中止	未定义	中断	快速中断
通用寄存器和程序计数器	R0(a1)	R0						
	R1(a2)	R1						
	R2(a3)	R2						
	R3(a4)	R3						
	R4(v1)	R4						
	R5(v2)	R5						
	R6(v3)	R6						
	R7(v4)	R7						
	R8(v5)	R8						R8_fiq *
	R9(SB,v6)	R9						R9_fiq *
	R10(SL,v7)	R10						R10_fiq *
	R11(FP,v8)	R11						R11_fiq *
	R12(IP)	R12						R12_fiq *
	R13(SP)	R13		R13_svc*	R13_abt *	R13_und *	R13_irq *	R13_fiq *
	R14(LR)	R14		R14_svc *	R14_abt *	R14_und *	R14_irq *	R14_fiq *
	R15(PC)	R15						
状态寄存器	R16(CPSR)	CPSR						
	SPSR	无		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

北京郵電大學

经典ARM寄存器组织

- ARM处理器工作在不同模式时使用的寄存器不同：37
  - 无论何种模式，R15均作为PC使用 1
  - CPSR为当前程序状态寄存器 1
  - R7~R0为公用的通用寄存器 8
  - R8~R12共2组 5*2=10
    - 标有“_fiq”的寄存器为快速中断模式专用，与其他模式的R8~R12使用不同的物理寄存器（“影子寄存器”）
  - R13、R14：
    - 用户模式和系统模式下分别为堆栈指针SP和链接寄存器LR 1+1
    - 其他模式下有专门的物理寄存器 5*2=10
  - 五个异常模式下可以看到各自的SPSR 5

北京郵電大學

Cortex-M3/M4寄存器组

通用寄存器

R0	通用寄存器
R1	通用寄存器
R2	通用寄存器
R3	通用寄存器
R4	通用寄存器
R5	通用寄存器
R6	通用寄存器
R7	通用寄存器

低寄存器：所有指令都能访问

R8	通用寄存器
R9	通用寄存器
R10	通用寄存器
R11	通用寄存器
R12	通用寄存器

高寄存器：　只有32位Thumb2指令和很少的16位Thumb指令能访问

R13(MSP)　R13(PSP)

- 主堆栈指针（MSP）：由OS内核、异常服务例程，以及所有需要特权访问的应用程序代码使用
- 进程堆栈指针（PSP）：用于常规应用程序代码

特殊功能寄存器

R14	链接寄存器（LR）
R15	程序计数器（PC）
xPSR	状态字寄存器（三合一）
PRIMASK	中断屏蔽寄存器
FAULTMASK	
BASEPRI	
CONTROL	控制寄存器

APSR　EPSR　IPSR

Application PSR　Execution PSR　Interrupt PSR

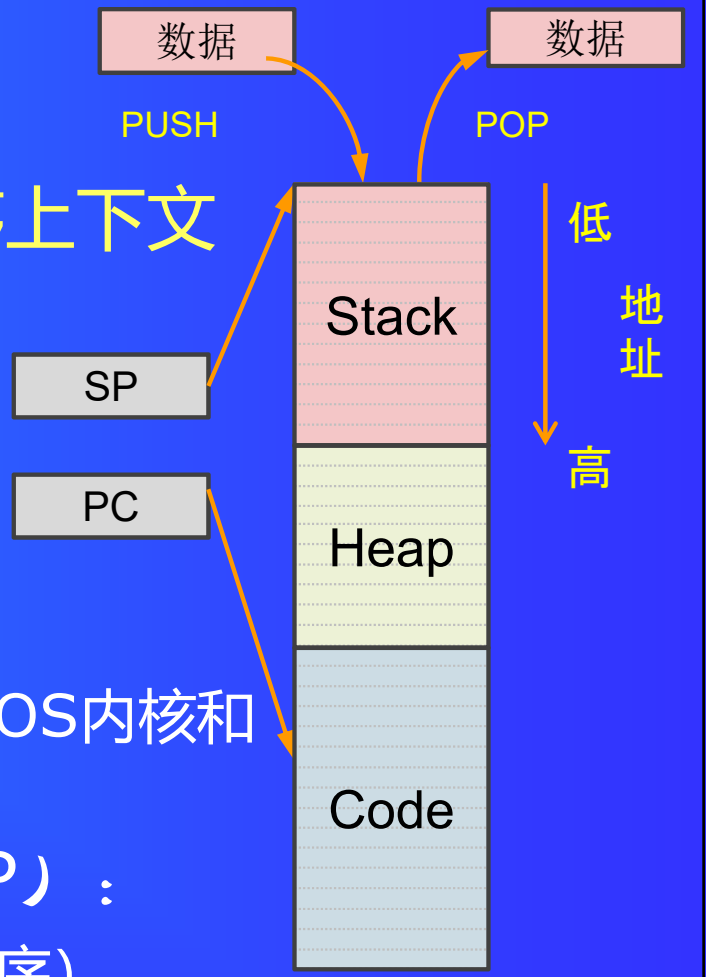
北京郵電大學

Cortex-M3/M4寄存器

- R0 – R12: 通用寄存器
  - 低寄存器（R0 – R7）：
    - 可作为16位或32位指令的操作数
    - 任何指令均可访问
    - 多数16位指令只能访问这些寄存器
  - 高寄存器（R8 – R12）：
    - 只能用作32位操作数
    - 32位指令可以访问，大多数16位指令不能访问

Cortex-M3/M4寄存器

- R13：堆栈指针（SP）
  - 记录当前堆栈地址
  - 在不同任务之间切换时用于保存程序上下文
  - 堆栈指针的最低两位永远是00
    - 堆栈总是4字节对齐的
  - Cortex-M4的两个SP：
    - 主堆栈指针MSP（Main SP）
      - 用于需要特权访问的应用程序，如OS内核和异常处理程序
    - 处理堆栈指针PSP（Process SP）：
      - 用于常规应用代码（非异常处理程序）

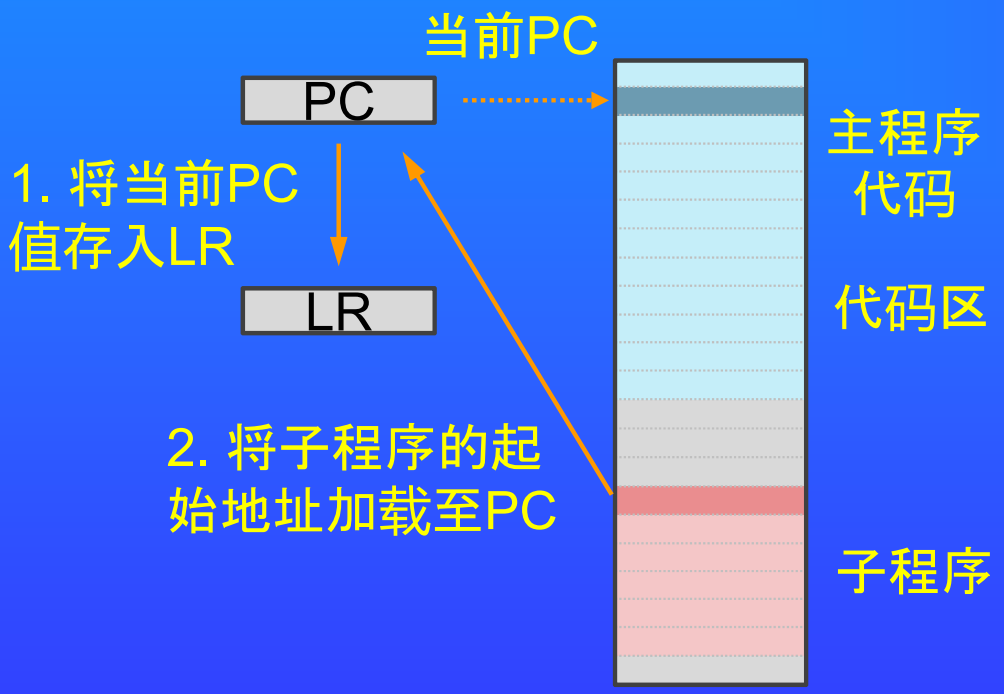


Cortex-M3/M4寄存器

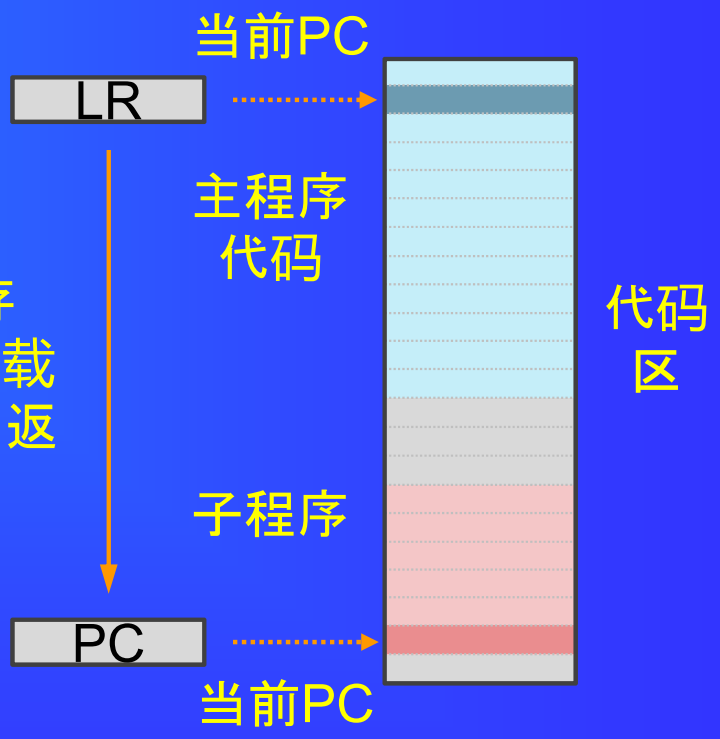
- R13：堆栈指针（SP）
  - Cortex-M4的两个SP：
    - 主堆栈指针MSP（Main SP）
      - 用于需要特权访问的应用程序，如OS内核和异常处理程序
    - 处理堆栈指针PSP（Process SP）：
      - 用于常规应用代码（非异常处理程序）
  - 堆栈的选择
    - 在处理模式下，只能使用主堆栈
    - 在线程模式下，可以使用主堆栈，也可以使用进程堆栈
      - 由 CONTROL 寄存器控制
    - 上电默认：MSP

Cortex-M3/M4寄存器

- R14：链接寄存器（LR，Link Register）
  - 跳转及链接指令（B&L）调用子程序时保存返回地址
  - 函数执行完毕返回时，LR被加载至程序计数器（PC）



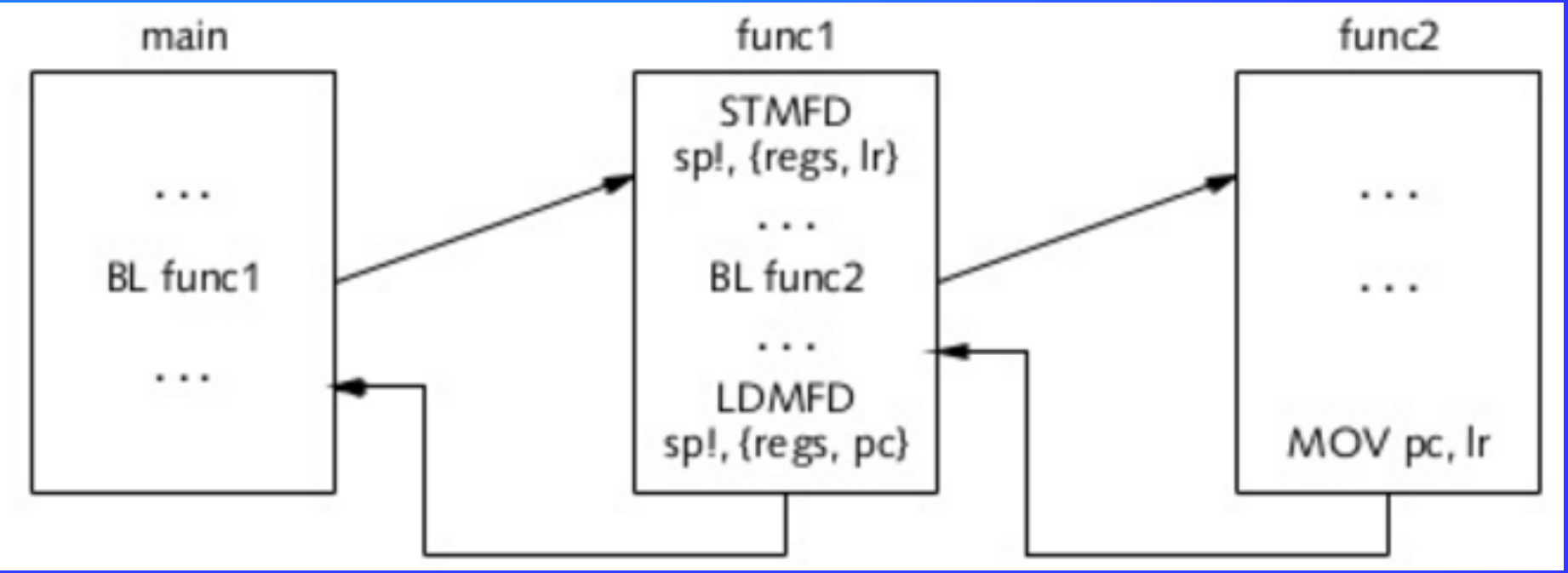
调用子程序



从子程序返回到主程序

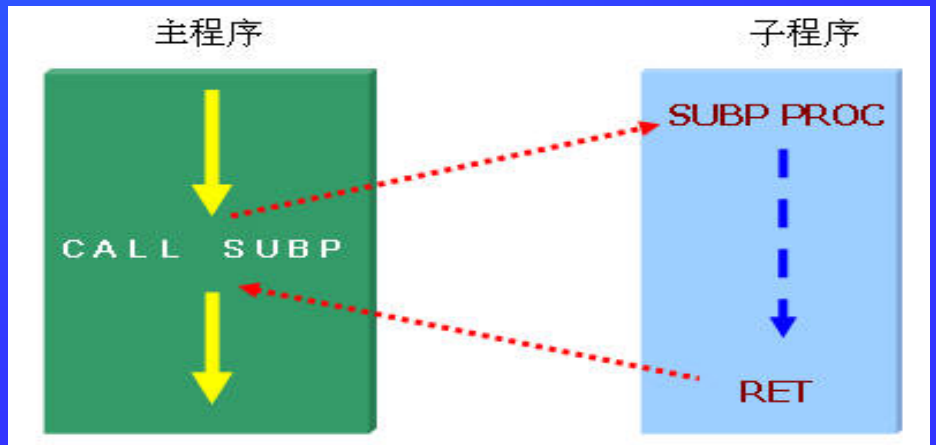
R14：子程序链接寄存器（LR）

- R14寄存器的嵌套使用
  - 在尾函数（不调用任何函数的函数）中，不需要保存LR
  - 非尾函数需要保存LR至堆栈
    - 否则当发生嵌套调用时，会发生R14寄存器冲突



Cortex-M3/M4寄存器

- R15：程序计数器（PC）
  - 记录当前指令的地址
  - 当处理器处于Thumb状态：
    - 所有指令必须半字对齐
    - PC值由R15的bits[31:1]决定，bits[0]无意义
  - 非分支指令：
    - 读PC时返回当前指令地址+4
  - 分支指令：
    - 将PC变更为一个特定地址，并将当前PC值存入LR寄存器
      - 例：函数调用指令



ARM汇编指令格式

Syntax of source lines in assembly language

- 汇编语法与汇编器有关:
  - Keil ARM MDK ARM汇编器（armasm)
  - GNU工具链
    - 伪指令、标号和注释等语法可能稍有不同
    - /* 中间为注释 */
    - @后面为注释

Syntax of source lines in assembly language

- ARM汇编器汇编源程序行的一般格式:

  {symbol} {instruction|directive|pseudo-instruction} {;comment}

  - 三部分均可省略
- symbol:
  - 必须从第一列开始
  - 在指令和伪指令中：标号（label）
    - 地址的符号表示
  - 在某些伪操作（directive）中：变量名或常数名
- Instructions and pseudo-instructions:
  - make up the code a processor uses to perform tasks

ARM汇编器语法

- ARM汇编器（armasm）汇编指令语法:

  label

  mnemonic operand1, operand2，… ; Comments

  - Label ： 地址位置参照
  - Mnemonic：指令助记符
  - Operand1，Operand2，……：操作数
    - 可以有较少的操作数
    - 第一个操作数一般是目的操作数（<Rd>）
    - 其他的操作数是源操作数（<Rn>、<Rm>）
  - Comments：注释，在“；”之后，对程序不产生影响

Syntax of source lines in assembly language

- 实例
  - MOVS R3, #0x11 ;为寄存器R3赋值0x11
  - ADDS <Rd>, <Rn>, <Rm> ;寄存器加：<Rd> = <Rn> + <Rm>
  - AND <Rdn>, <Rm> ;按位与：<Rdn> = <Rdn> & <Rm>
  - CMP <Rn>, <Rm> ;比较
    - 根据<Rn> - <Rm>的计算结果设置条件标志

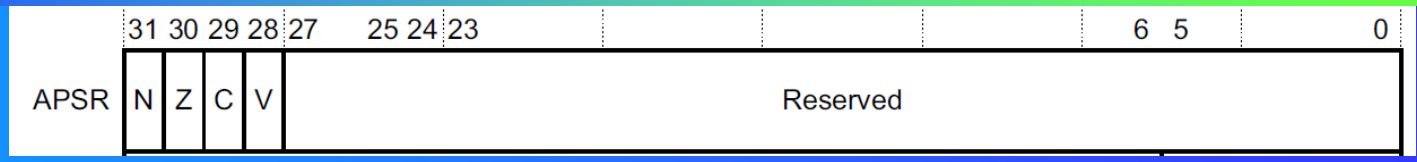
基本指令格式

三地址指令格式

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

- <>：必备项
- {}：可选项
- opcode：指令助记符
- cond：执行条件
- S：是否更新APSR寄存器
- Rd：目标寄存器
- Rn：第1源操作数的寄存器
- operand2：第2源操作数

指令后缀

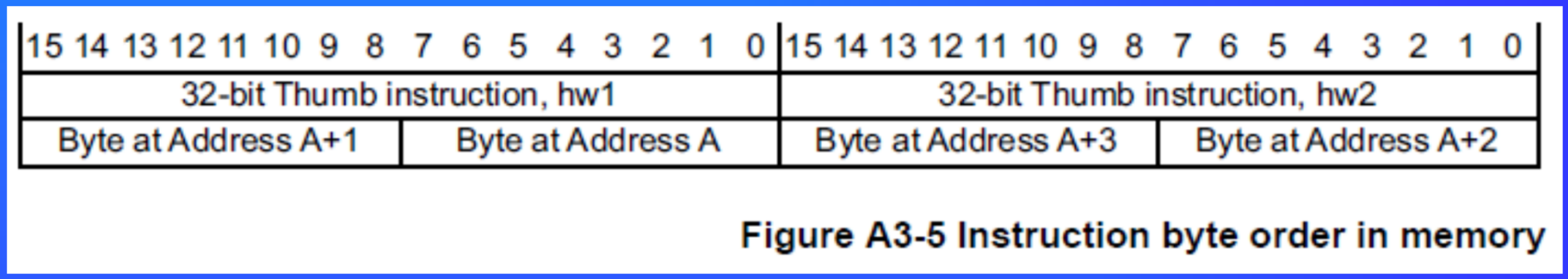


- S后缀
  - 更新APSR中的条件码（标志位）
  - ADDS R0, R1 ;根据加法的结果更新APSR中的标志
- EQ、NE、LT、GT等15种条件执行后缀
  - 条件执行
    - 分支指令（B）：可以自由使用
    - 其它指令：在IF - THEN指令块中才可以加后缀
  - EQ=Euqal
  - NE=Not Equal
  - LT= Less Than
  - GT= Greater Than
  - ……
  - BEQ <Label>；仅当满足EQ条件时转移

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

Thumb指令集编码

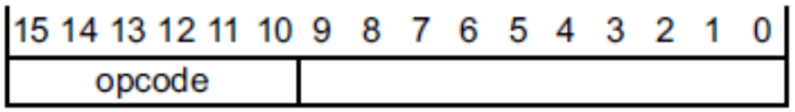
- Thumb指令流：由半字对齐（halfword-aligned）的半字（halfwords）序列构成
- 指令流中的每条Thumb指令均为16或32位：
  - 单一16位半字指令
  - 由两个连续半字hw1和hw2组成的32位指令
    - hw1在低地址



15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
32-bit Thumb instruction, hw1	32-bit Thumb instruction, hw2
Byte at Address A+1 / Byte at Address A	Byte at Address A+3 / Byte at Address A+2

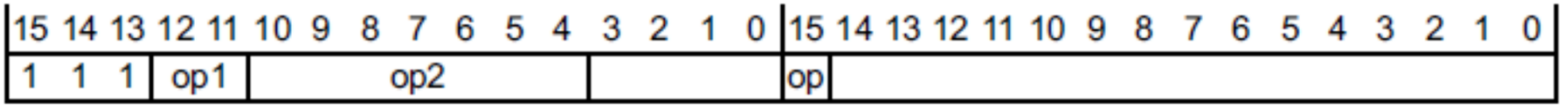
Figure A3-5 Instruction byte order in memory

The encoding of 16-bit Thumb instructions is:



15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
opcode

The encoding of 32-bit Thumb instructions is:

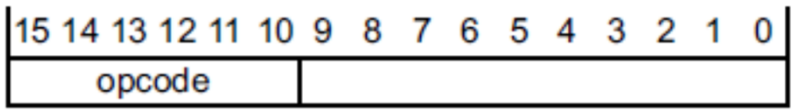


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 1 1 / op1 / op2	op

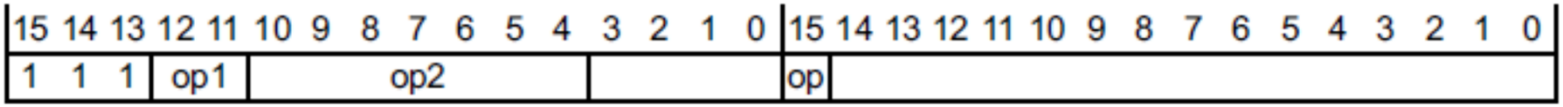
Thumb指令集编码

- Thumb指令流：由半字对齐（halfword-aligned）的半字（halfwords）序列构成
- 指令流中的每条Thumb指令均为16或32位：
  - 单一16位半字指令
  - 由两个连续半字hw1和hw2组成的32位指令
    - hw1在低地址
- 判断指令长度：指令译码后的半字的[15:11]位编码
  - 0b11101，0b11110，0b11111：32位指令的前半字hw1
  - 其他：16位指令

The encoding of 16-bit Thumb instructions is:



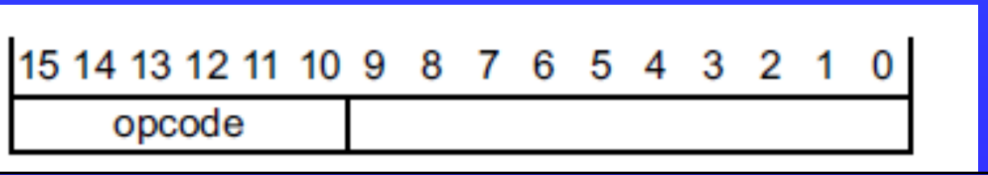
The encoding of 32-bit Thumb instructions is:



16-bit Thumb instruction encoding

opcode	Instruction or instruction class
00xxxx	Shift (immediate), add, subtract, move, and compare
010000	Data processing
010001	Special data instructions and branch and exchange
01001x	Load from Literal Pool, see LDR (literal)
0101xx 011xxx 100xxx	Load/store single data item
10100x	Generate PC-relative address, see ADR
10101x	Generate SP-relative address, see ADD (SP plus immediate)
1011xx	Miscellaneous 16-bit instructions
11000x	Store multiple registers, see STM, STMIA, STMEA
11001x	Load multiple registers, see LDM, LDMIA, LDMFD
1101xx	Conditional branch, and Supervisor Call
11100x	Unconditional Branch, see B

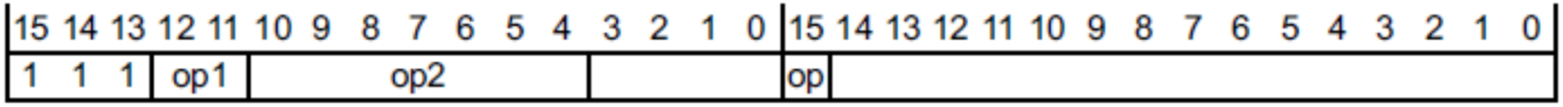
北京郵電大學



32-bit Thumb instruction encoding

op1	op2	op	Instruction class
01	00xx0xx	x	Load Multiple and Store
01	00xx1xx	x	Load/store dual or exclusive, table
01	01xxxxx	x	Data processing (shifted register)
01	1xxxxxx	x	Coprocessor instructions
10	x0xxxxx	0	Data processing (modified immediate)
10	x1xxxxx	0	Data processing (plain binary immediate)
10	xxxxxxx	1	Branches and miscellaneous control
11	000xxx0	x	Store single data item
11	00xx001	x	Load byte, memory hints
11	00xx011	x	Load halfword, memory hints
11	00xx101	x	Load word
11	00xx111	x	UNDEFINED
11	010xxxx	x	Data processing (register)
11	0110xxx	x	Multiply, multiply accumulate, and absolute difference
11	0111xxx	x	Long multiply, long multiply accumulate, and divide
11	1xxxxxx	x	Coprocessor instructions

The encoding of 32-bit Thumb instructions is:



统一汇编语言书写语法（UAL）

- 允许开发者以相同的语法格式书写，由汇编器决定使用16位指令还是32位指令
  - ADD R0, R1 ; R0=R0+R1，传统Thumb语法
  - ADD R0, R0, R1 ; UAL语法的等价写法

- UAL：可以由汇编器/手工指定用16位/32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）
  - ADDS R0, #1 ;汇编器将为节省空间而使用16位指令
    - 无后缀时，汇编器会尽量选择更短的指令

统一汇编语言书写语法（UAL）

- 统一汇编语言语法：指令是否更新标志位取决于S后缀
  - AND　　R0，　R1　　　　;传统Thumb语法
    - 有些指令默认更新APSR
  - ANDS　R0，　R0，R1　　;UAL语法：有S后缀才更新

- 32位Thumb-2指令可以按半字对齐
  - 0x1000：　LDR r0，[r1]　;一条16位指令
  - 0x1002：　RBIT.W　r0
    - ;一条32位位反转指令，跨越字边界

指令集汇总

指令类型	指令
移动	MOV
内存读/写	LDR, LDRB, LDRH, LDRSH, LDRSB, LDM, STR, STRB, STRH, STM
加法、减法、乘法	ADD, ADDS, ADCS, ADR, SUB, SUBS, SBCS, RSBS, MULS
比较	CMP, CMN
逻辑运算	ANDS, EORS, ORRS, BICS, MVNS, TST
移位和循环移位	LSLS, LSRS, ASRS, RORS
堆栈操作	PUSH, POP
条件跳转	IT, B, BL, B{cond}, BX, BLX
扩展	SXTH, SXTB, UXTH, UXTB
保留	REV, REV16, REVSH
处理器状态	SVC, CPSID, CPSIE, BKPT
空操作	NOP
Hint	SEV, WFE, WFI

操作数的寻址方式

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

ARM汇编指令中的操作数符号

- #：立即数前缀
  - 二进制数、十进制数或十六进制数
  - 以十六进制表示的立即数：#0x
  - 以二进制表示的立即数：#0b
  - 以十进制表示的立即数：#0d，或#（缺省）

立即数的表示

- 立即数的汇编表示：#imm
  - immediate fields are unsigned unless otherwise stated in the instruction description
- 32位立即数的机器表示：用12位编码表示一个32位立即数
  - 8位数值+4位移位位数
- 每个立即数由一个8位常数循环右移偶数位得到
  - 循环右移的位数为4位二进制数的两倍

    <immediate>=immed_8 循环右移 (2*rotate_imm) 位
  - <immediate>：立即数
  - immed_8：8位常数
  - rotate_imm：4位的循环右移值
- 只有能够通过上述构造方法得到的才是合法的立即数
  - 例：不合法的立即数
    - 0x101，0x102，0xFF1

立即数的表示

- 每个立即数由一个8位常数循环右移偶数位得到
  - 循环右移的位数为4位二进制数的两倍

<immediate>=immed_8 循环右移 (2*rotate_imm) 位

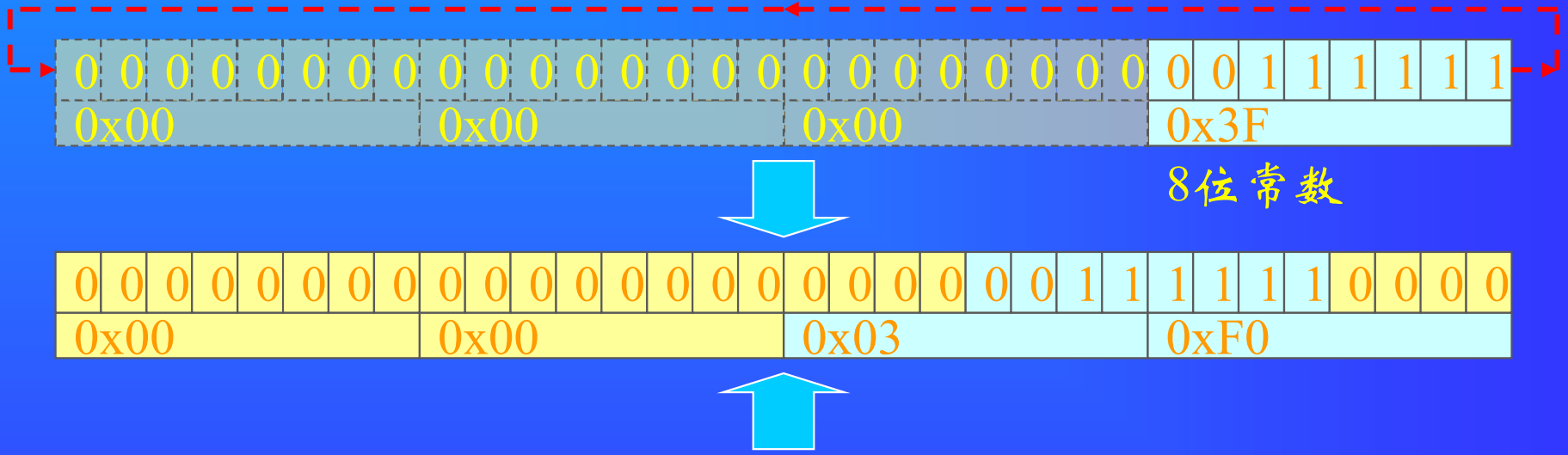
  - <immediate>：立即数
  - immed_8：8位常数
  - rotate_imm：4位的循环右移值
- 完全自由构造立即数会存在不确定性
  - 一个合法的立即数可能有多种编码方式
  - 汇编器约定：
    - 立即数数值在0~0xFF范围时，令immed_8=<immediate>， rotate_imm=0
    - 其他情况下，选择使rotate_imm数值最小的编码

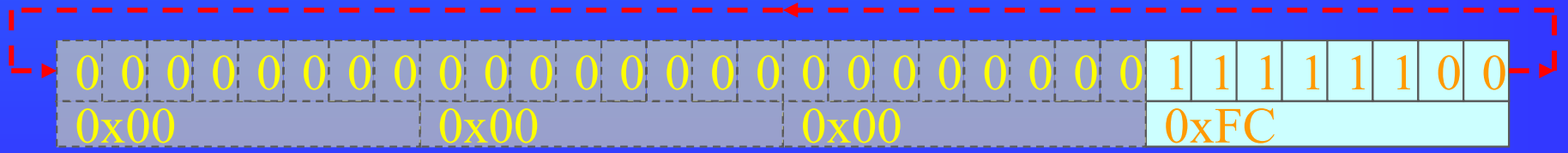
立即数的表示

- 例：常数0x3F0的构造方式
- 011 1111 0000

移位数值最小

0x3F循环右移28位(0xE*2)



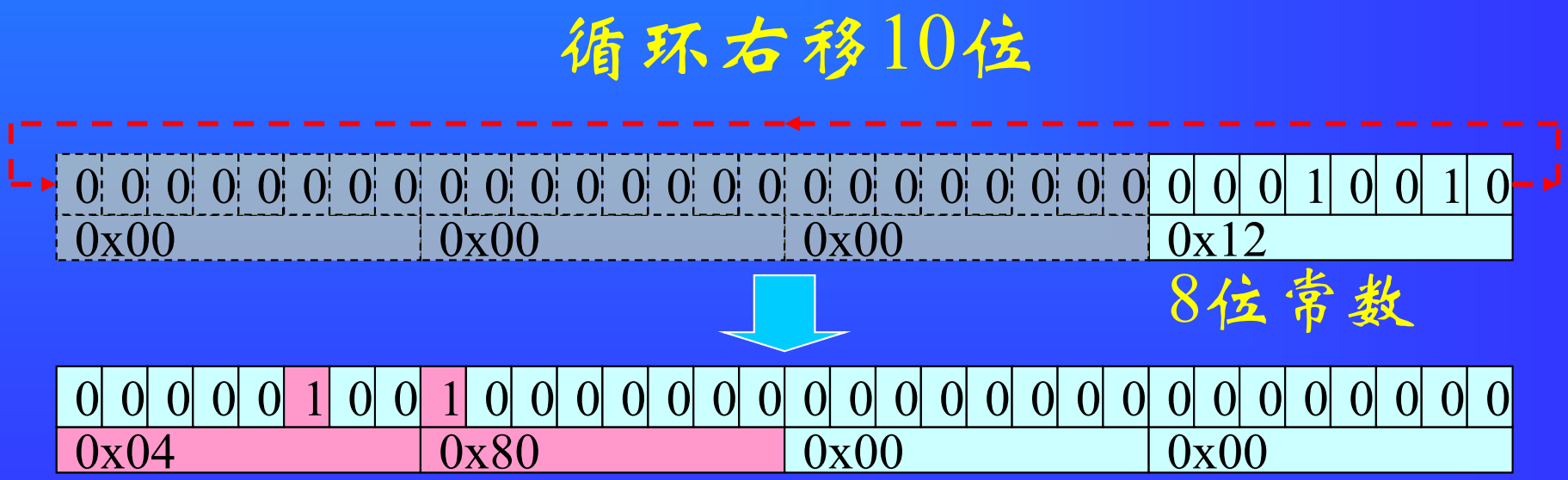


0xFC循环右移30位（0xF*2）

课堂练习：立即数的表示

- 立即数 0x480 0000 生成方式

- 0480 0000
- 0000 0100 1000 0000 .....

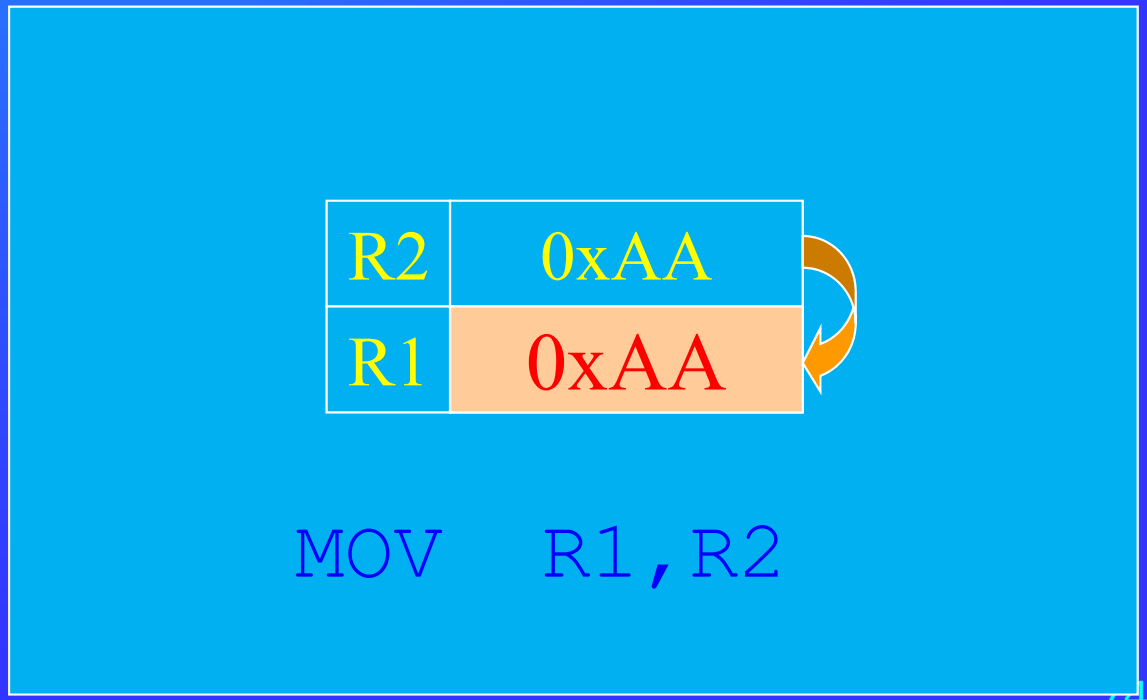


立即寻址

- 指令中的地址码字段即是操作数本身
- 例:
  - SUBS R0, R0,#1 ;R0减1，结果放入R0，并更新标志位
  - MOV R0, #0xFF00 ;将立即数0xFF00装入R0寄存器

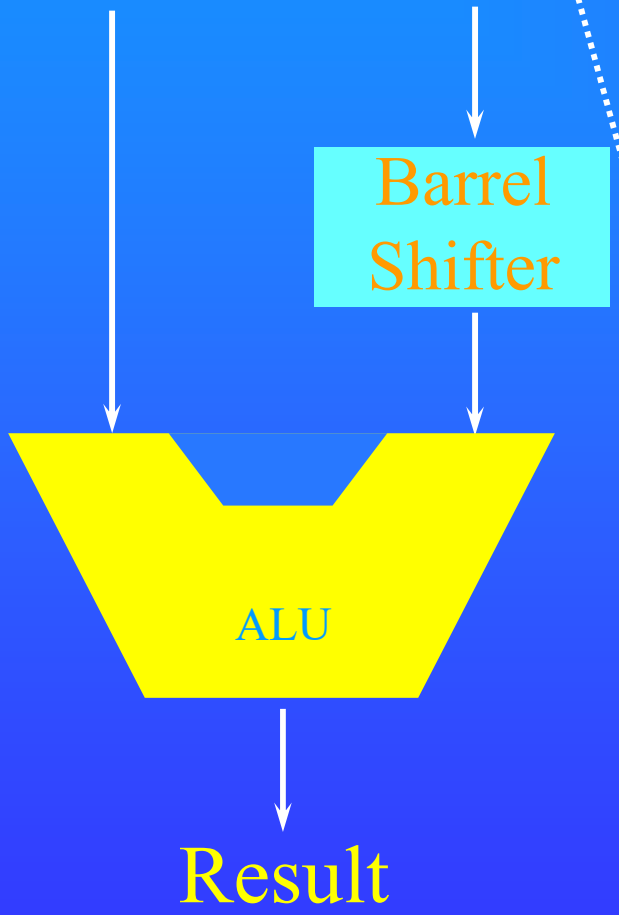
寄存器寻址

- 操作数在寄存器中，指令中的地址码字段给出寄存器编号
- 例
  - MOV R1, R2 ;将R2的值存入R1
  - SUB R0, R1, R2 ;将R1的值减去R2的值，结果保存到R0



立即数和寄存器移位结构

第二操作数可以是：



- 寄存器，支持可选的移位操作
  - 给出移位值：
    - 5位无符号整数
    - 另一个寄存器的最低字节给出
- 立即数
  - 8位数值（取值范围0-255）循环右移偶数位

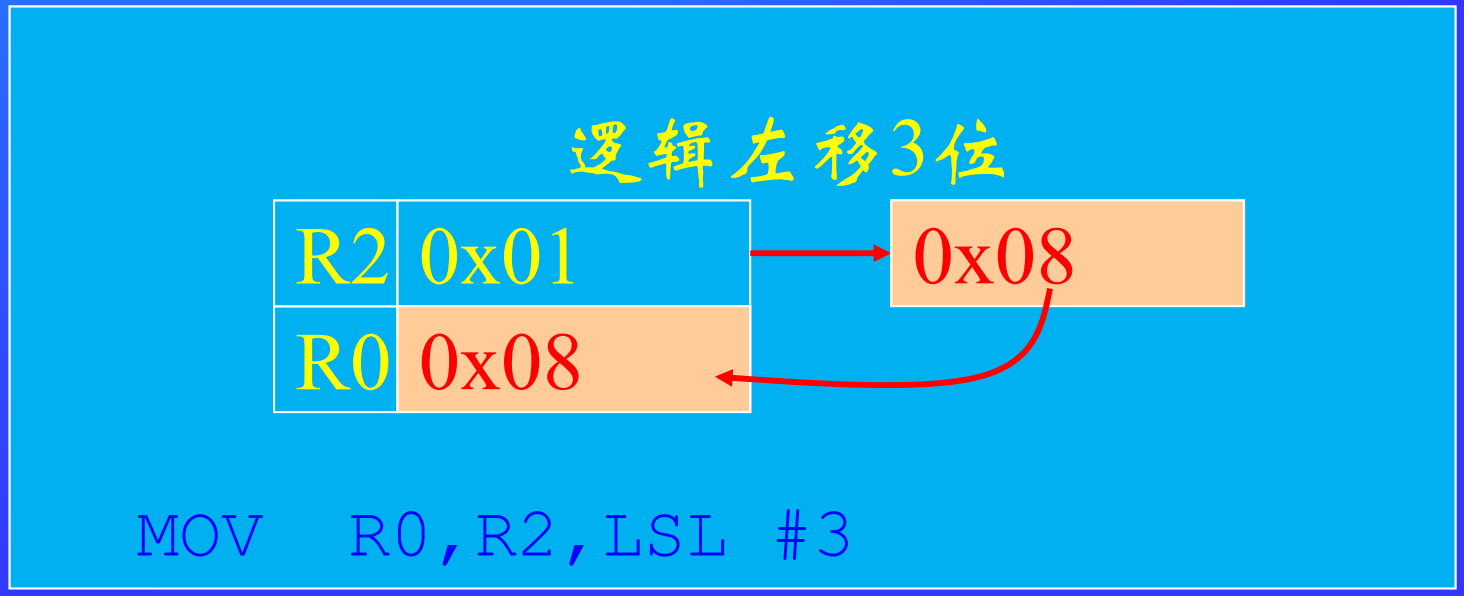
Rm,shift——寄存器移位寻址

- 将寄存器移位的结果作为操作数，但Rm值保持不变
  - 移位操作不消耗额外的时间

操作码	说明	
ASR #n	算术右移n位	Arithmetic Shift Right
LSL #n	逻辑左移n位	Logical Shift Left
LSR #n	逻辑右移n位	Logical Shift Right
ROR #n	循环右移n位	Rotate Right
RRX	带扩展的循环右移1位	Rotate Right with Extend
Type Rs	Type为移位类型 Rs寄存器低8位保存移位的位数	

Rm,shift——寄存器移位寻址

- 仅第2操作数可以使用
- 第2寄存器操作数进行移位操作后参与运算
- 例:
  - MOV R0, R2, LSL #3 ;R2左移3位，结果放入R0，
    ;R0=R2×8
  - ANDS R1, R1, R2, LSL R3 ;R2左移R3位，然后和R1相与，
    ;结果放入R1



访存（Load/Store）指令的寻址方式

- 访存指令
  - Load指令：从内存单元读取数据放入寄存器
  - Store指令：将寄存器中的数据保存到内存单元
- Load/Store指令的地址构成
  - 基址寄存器：任意一个通用寄存器R0-R7、SP或PC
  - 地址偏移量：
    - 无符号常数
    - 通用寄存器
    - 通用寄存器及一个移位常数

访存（Load/Store）指令的寻址方式

- 访存地址计算：
  - 基址偏移量：操作数地址为基址寄存器的值加上或减去一个偏移量
    - 偏移寻址Offset addressing
    - [<Rn>,<offset>]
- 自动变址：把有效地址写回到基址寄存器！
  - 基址寄存器的值和偏移量做加减运算，生成操作数地址访存；用生成的操作数地址更新基址寄存器
    - 前变址寻址Pre-indexed addressing
    - [<Rn>,<offset>]!

访存（Load/Store）指令的寻址方式

- 访存地址计算：
  - 基址偏移量：偏移寻址Offset addressing
    - [<Rn>,<offset>]
- 自动变址：把有效地址写回到基址寄存器！
  - 基址寄存器的值和偏移量做加减运算，生成操作数地址访存；用生成的操作数地址更新基址寄存器
    - 前变址寻址Pre-indexed addressing
    - [<Rn>,<offset>]!
  - 将基址寄存器的值作为操作数地址访存；基址寄存器中的值和地址偏移量做加减运算，生成新的操作数地址更新基址寄存器
    - 后变址寻址Post-indexed addressing
    - [<Rn>],<offset>

[ ]:取地址

访存（Load/Store）指令的寻址方式

- 偏移寻址Offset addressing [<Rn>,<offset>]
- 前变址寻址Pre-indexed addressing [<Rn>,<offset>]!
- 后变址寻址Post-indexed addressing [<Rn>],<offset>

  - <Rn>：base register，基址寄存器
  - <offset> can be:
    - immediate 立即变址
      - such as <imm8> or <imm12>
    - register 变址寄存器
    - scaled register（shifted index register）
      - 缩放寄存器/移位变址寄存器
      - such as <Rm>, LSL #<shift>

访存（Load/Store）指令的寻址方式组合

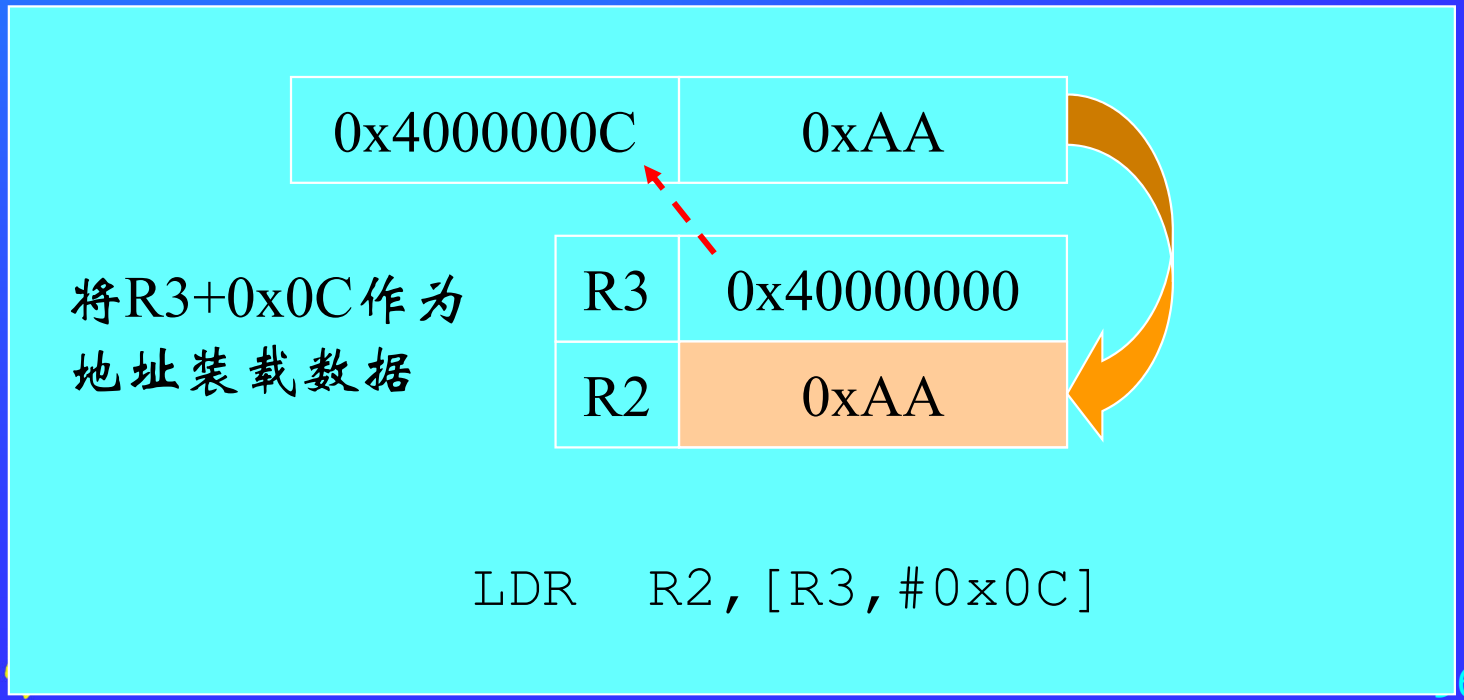
- LDR
  - Load Register (register)
  - 语法　LDR {<cond>} <Rd>,<address_mode>
  - Address_mode：第二个操作数的内存地址
    1) [<Rn>,#+/-<offset_12>]　立即变址偏移寻址
    2) [<Rn>,#+/-<offset_12>]!　立即数前变址寻址
    3) [<Rn>,+/-<Rm>]　寄存器偏移寻址
    4) [<Rn>,+/-<Rm>,<shift>#<shift_imm>]
       带移位的寄存器偏移寻址
    5) [<Rn>,+/-<Rm>]!　寄存器前变址寻址
    6) [<Rn>,+/-<Rm>,<shift>#<shift_imm>]!
       带移位的寄存器前变址寻址
    7) [<Rn>],#+/-<offset_12>　立即数后变址寻址
    8) [<Rn>],+/-<Rm>　寄存器后变址寻址
    9) [<Rn>],+/-<Rm>,<shift>#<shift_imm>
       带移位的寄存器后变址寻址

访存（Load/Store）指令的寻址方式

- [<Rn>,#+/-<offset_12>]
  - 立即变址偏移寻址
  - 地址计算方法:
    - 访存地址为寄存器值与常数偏移值之和
    - address =Rn +/- offset_12
  - 适用范围
    - 访问结构化数据的数据成员，使用一个基址寄存器访问位于同一区域的多个存储单元

访存（Load/Store）指令的寻址方式

- [<Rn>,#+/-<offset_12>]
  - 立即变址偏移寻址
  - 地址计算方法:
    - address =Rn +/- offset_12
  - 例:
    - LDR R2, [R3, #0x0C] ;R3+0x0C地址的内容送入R2
    - LDR R0, [R1，#4]　　;R0←[R1+4]



访存（Load/Store）指令的寻址方式

- [<Rn>,#+/-<offset_12>]
  - 立即变址偏移寻址
  - 地址计算方法：
    - address =Rn +/- offset_12
  - 地址偏移量为0时，访问Rn指向的内存单元
    - 等价于寄存器间接寻址
    - 例： LDR R1, [R2]

      ;R2指向的存储单元的数据读出保存在R1中

访存（Load/Store）指令的寻址方式

- [<Rn>, #+/-<offset_12>]!
  - 立即数前变址寻址
  - 地址计算方法
    - address= Rn +/-offset_12
    - 自动变址：实现基址寄存器自动修改，让程序追踪一个数据表
  - 例:
    - LDR R0，[R1，#4]！ ;R0←[R1＋4]，R1←R1＋4
      - 功能等效于"寄存器间接取数指令＋数据处理指令"，但不消耗额外的时间
    - STR R1, [R0, #-4]！ ;R0=R0-4，把R1的值保存到;R0指定的存储单元

访存（Load/Store）指令的寻址方式

- [<Rn>,+/-<Rm>]
  - 寄存器偏移寻址
  - 地址计算方法
    - address= Rn +/- Rm
  - 适用范围
    - 访问字节数组中的数据成员
  - 例:
    - LDR R0，[R1，R2]　;R0←[R1＋R2]
    - LDR R0，[R1，-R2]

访存（Load/Store）指令的寻址方式

- [<Rn>,+/-<Rm>]!
  - 寄存器前变址（Register pre-indexed）
  - 地址计算方法
    - address= Rn +/-Rm
  - 例:
    - LDR R0，[R1，R2]!

访存（Load/Store）指令的寻址方式

- [<Rn>,+/-<Rm>,<shift>#<shift_imm>]
  - 带移位的寄存器偏移寻址
  - 地址计算方法
    - address= Rn +/-index
  - 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
  - 例:
    - LDR R0，[R1，R2，LSL #2]

访存（Load/Store）指令的寻址方式

- [<Rn>,+/-<Rm>,<shift>#<shift_imm>]!
  - 寄存器移位前变址 (Scaled Register pre-indexed)
  - 地址计算方法
    - address= Rn +/-index
  - 例:
    - LDR R0，[R1，R2，LSL #2]！；
      - ; R1=R1+R2<<2, R0=memory(new r1)

访存（Load/Store）指令的寻址方式

- [<Rn>],#+/-<offset_12>
  - 立即数后变址（Immediate post-indexed）
  - 地址计算方法
    - address=Rn，当满足条件时Rn=Rn+/-offset_12
    - 基址寄存器不加偏移作为访存地址使用，完成操作后再加上偏移量送入基址寄存器
  - 例：
    - LDR R0, [R1], #4 ；R0←[R1]，R1←R1+4
    - 等效于“寄存器间接寻址取数指令+改变地址的数据处理指令”

访存（Load/Store）指令的寻址方式

- [<Rn>],+/-<Rm>
  - 寄存器后变址（Register post-indexed）
  - 地址计算方法
    - address=Rn，当满足条件时Rn= Rn +/- Rm
    - base address（base register）+Another register，Updates base register after data transfer
  - 例：
    - LDR R0，[R1]， R2；
      - ；R0=memory(old r1), r1=r1+r2

访存（Load/Store）指令的寻址方式

- [<Rn>],+/-<Rm>,<shift>#<shift_imm>
  - 带移位的寄存器后变址寻址
  - 地址计算方法
    - address=Rn, 当满足条件时Rn= Rn +/- index
  - 例:
    - LDR R0, [R1], R2, LSL #2；

寻址方式对比

- LDR R0, [R1, #4] ;R0←[R1+4]
- LDR R0, [R1, #4]! ;R0←[R1+4], R1←R1+4
- LDR R0, [R1], #4 ;R0←[R1], R1←R1+4
- LDR R0, [R1, R2] ;R0←[R1+R2]
- LDR R2, [R0, R1]! ;R2←[R0+R1], R0←R0+R1
- LDR R2, [R0, R1 LSL #2]! ;R0←R0+R1<<2, R2←[new R0]
- LDR R2, [R0], R1 ;R2←[old R0], R0←R0+R1

相对寻址

- 基址寻址的变种，相对于程序计数器PC
  - 由程序计数器PC提供基地址，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
- 例：子程序调用

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

  - 偏移量由汇编器自动形成

汇编伪操作

汇编伪操作

- 汇编伪操作（directive）
  - 作用：为汇编器提供信息，指导汇编器完成汇编工作
    - 或者对汇编过程起作用，或者对最终输出有影响
    - 汇编器可见，CPU不可见
      - 不会被汇编器汇编为机器指令
      - 在汇编生成的可执行程序的二进制代码中不可见
  - 与特定的汇编器相关
    - ARMASM
    - GNU

汇编伪操作

- 符号定义（Symbol Definition）伪操作
  - 用于定义ARM汇编程序中的变量、对变量赋值以及定义寄存器的别名
  - name EQU expr
    - 将symbol定义为expr
    - abc EQU 2 ;Assigns the value 2 to the ;symbol abc
    - xyz EQU label+8 ;Assigns the address ;(label+8) to the symbol xyz

汇编伪操作

- 数据定义（Data Definition）伪操作
  - {label} DCB expr{,expr}...
    - 定义一串字节常数，分配存储空间并初始化
    - Expr可以是：
      - 数值表达式，取值为-128至255的整数
      - 引号内的字符串，字符串内的字符将被加载至连续字节存储单元
    - = is a synonym for DCB

```
HELLO_TEXT
   DCB "Hello\n",0
```

汇编伪操作

- 数据定义（Data Definition）伪操作
  - {label} DCD{U} expr{,expr}
    - 在内存中分配1个或多个字的存储空间
      - DCD：存储器按4字节边界对齐
        - 必要时在第一个字之前插入最多3个填充字节
      - DCDU：存储器不要求对齐
    - 定义存储器的初始值
    - expr可以是：
      - 数值表达式
      - 相对PC的偏移值
    - & is a synonym for DCD

```
MY_NUMBER
    DCD 0x12345678
```

汇编伪操作

- 汇编控制（Assembly Control）伪操作
- 信息报告（Reporting）伪操作
- 其他（Miscellaneous）伪操作
  - CODE16
    - 告诉汇编器后面的指令序列为16位Thumb指令
  - CODE32
    - 告诉汇编器后面的指令序列为32位ARM指令
  - END
    - 告诉汇编器已经到了源程序结尾

CORTEX M3/M4常用指令

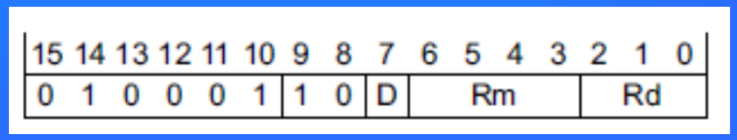
Cortex M3指令集

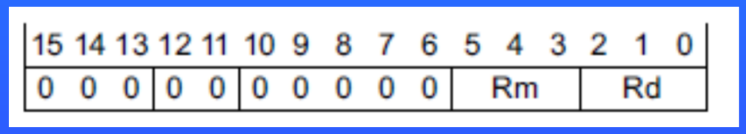
数据传送类指令

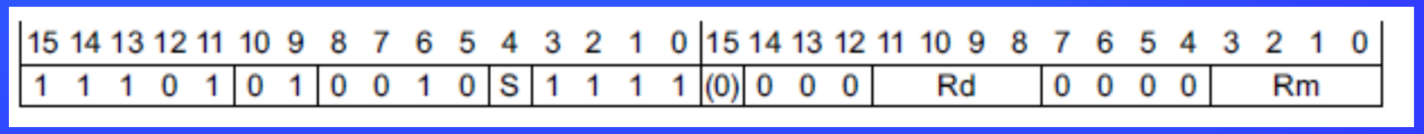
- 两个寄存器间传送数据
- 寄存器与存储器间传送数据
- 通用寄存器与特殊功能寄存器间传送数据
- 加载一个立即数到寄存器

寄存器到寄存器传送

- Move (register) 指令
  - 复制源寄存器的值至目标寄存器
  - 可以基于数据值更新标志位





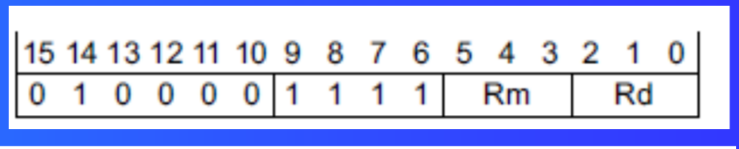


寄存器到寄存器传送

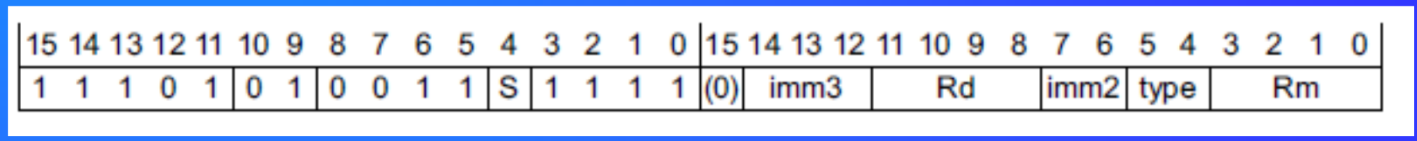
- Move (register) 指令
  - Assembler syntax
    - MOV{S}<c><q> <Rd>, <Rm>
      - S：更新标志位
      - <c>：执行条件
      - <q>：.N/.W
      - <Rd>: The destination register
      - <Rm>: The source register
  - 例:
    - MOV R8, R3；R8 = R3

寄存器到寄存器传送

- MVN (register)指令
  - 位取反（Bitwise NOT）传送，将某寄存器的值按位取反后送入目标寄存器
  - 可以基于数据值更新标志位



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	1	1	Rm			Rd		

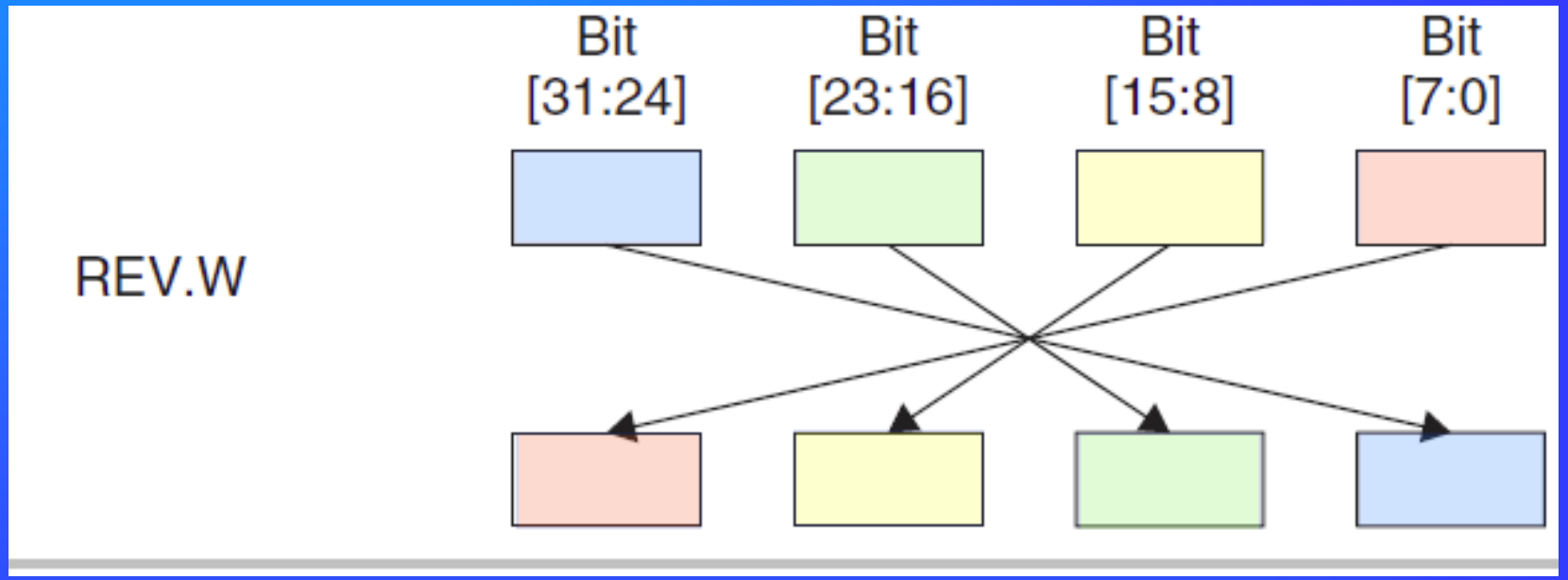


15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	1	0	0	1	1	S	1	1	1	1	(0)	imm3			Rd				imm2		type		Rm			

  - Assembler syntax
    - MVN{S}<c><q> <Rd>, <Rm> {, <shift>}
    - <shift>：Specifies the shift to apply to the value read from <Rm>
  - 例:
    - MVN R8, R3 ; R8 = ~R3

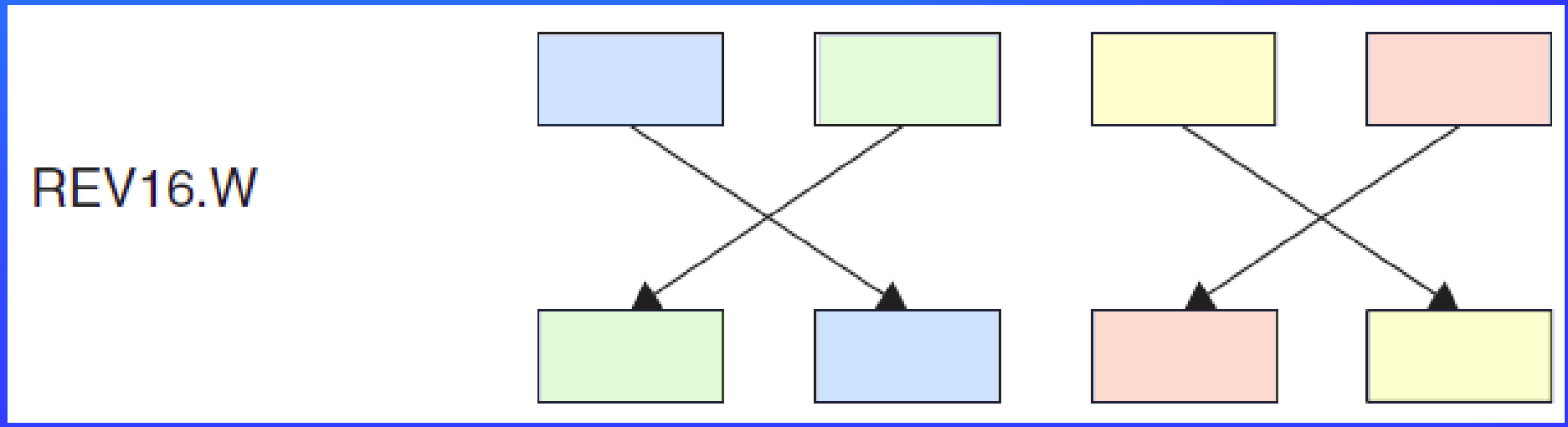
寄存器到寄存器传送

- 字内字节交换（Byte-Reverse Word）指令
  - REV.W Rd, Rn; 在字中反转字节序



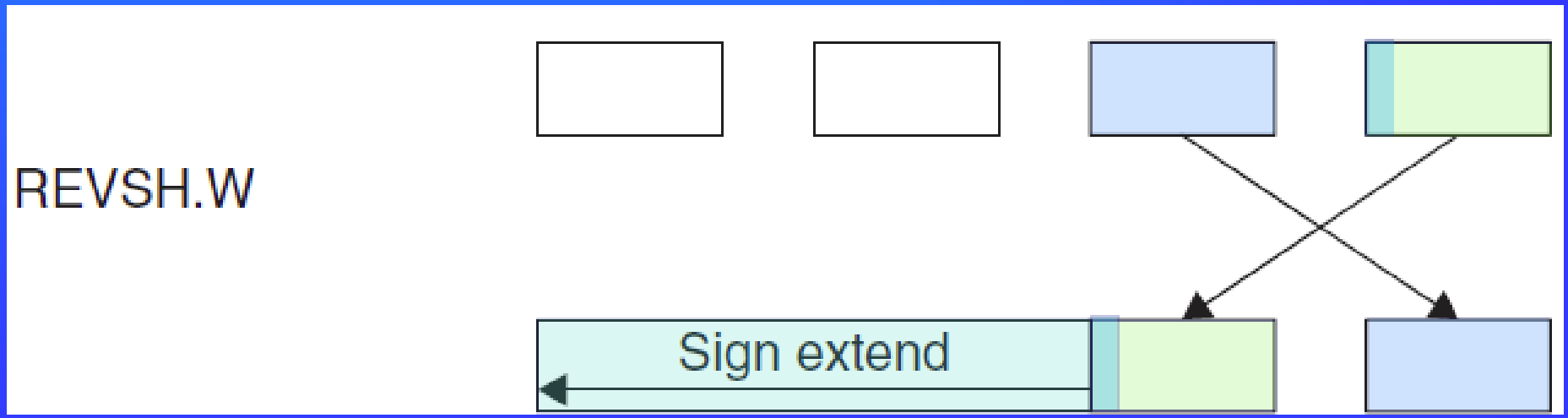
寄存器到寄存器传送

- 半字内字交换（Byte-Reverse Packed Halfword）指令
  - 交换一个32位寄存器中每个16位半字内的两个字节数据
  - REV16.W Rd, Rn; 在高低半字中反转字节序



寄存器到寄存器传送

- 带符号数半字内字节交换指令
  - Byte-Reverse Signed Halfword
  - 交换一个32位寄存器中的低16位半字内的两个字节数据，并符号扩展至32位
  - REVSH.W Rd, Rn; 在低半字中反转字节序，并符号扩展

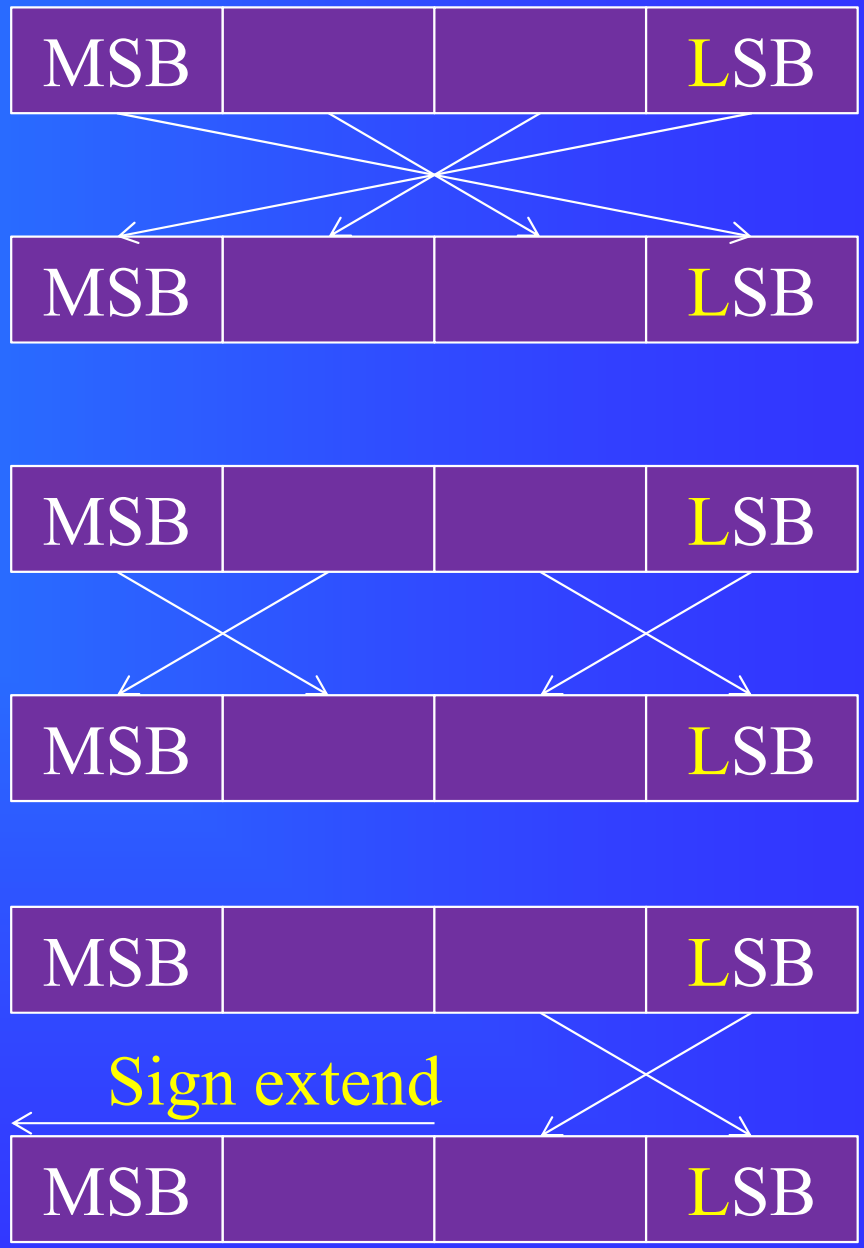


寄存器到寄存器传送

- RBIT（Reverse Bits）指令
  - reverses the bit order in a 32-bit register
  - RBIT<c><q> <Rd>, <Rm>

寄存器到寄存器传送

- 字节序反转指令REV——反转一个字中所有的字节
  - REV <Rd>,<Rm>
- REV16 ——一个字中的两个半字内的字节反转（交换）
  - REV16 <Rd>,<Rm>
- REVSH ——低半字的两个字节交换，然后扩展符号位
  - REVSH <Rd>,<Rm>
- RBIT（Reverse Bits）指令
  - reverses the bit order in a 32-bit register
  - RBIT<c><q> <Rd>, <Rm>



寄存器与存储器间传送数据

- load和 store指令可以处理字（32位）、双字（64位）、半字（16位）、字节（8位）和多字（n*32位）数据

- LDR: 从内存中读入数据到寄存器
  - LDR <Rt>, 源地址

- STR: 把寄存器数据写入内存
  - STR <Rt>, 目的地址

存储器到寄存器传送

- LDRx 指令
  - x:
    - B（byte）
    - H（half word）
    - D（Double word）
    - 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）

存储器到寄存器传送

- 读入一个字节或半字时：高位放什么?
- LDRB/LDRH
  - Load Register Byte/Halfword
  - 执行流程:
    - 基于基址寄存器和偏移值计算访存地址
    - 从内存单元读出一个字节/半字
    - 零扩展（zero-extends） 至32位字
      - 例：0x80零扩展为32位：0x0000_0080
    - 写入寄存器

存储器到寄存器传送

- 读入一个字节、半字或双字时：高位放什么?
- LDRSB/LDRSH
  - Load Register Signed Byte/Halfword
  - 执行流程:
    - 基于基址寄存器和偏移值计算访存地址
    - 从内存单元读出一个字节/半字
    - 符号扩展（sign-extends） 至32位字
      - 例：0x80符号扩展为32位：
        - 0xFFFF_FF80 = -128
    - 写入寄存器

	带符号	无符号
字节	LDRSB	LDRB
半字	LDRSH	LDRH

寄存器到存储器传送

- STRx 指令
  - 写内存时直接把双字、半字或字节写入内存，不考虑有符号无符号的区别

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）组成的双字存储到地址Rn+offset处

寄存器与存储器间批量传送

- LDMxy/STMxy指令
  - Load Multiple/Store Multiple
  - 一次传送多个数据
  - X：
    - I：Increment
    - D：Decrement
  - Y：自增或自减的时机选择在每次访问的前后
    - A：After
    - B：Before
  - xy组合：IA、IB、DA、DB

批量Load/Store指令的寻址方式

- LDM|STM{<cond>}<addressing_mode><Rn>{!},<registers>{^}
  - LDMIA/STMIA:
    - Increment After：先传送，后地址加4
  - LDMIB/STMIB:
    - Increment Before：先地址加4，后传送
  - LDMDA/STMDA:
    - Decrement After：先传送，后地址减4
  - LDMDB/STMDB:
    - Decrement Before：先地址减4，后传送

寄存器与存储器间批量传送

示例	功能描述
LDMIA Rd!, {寄存器列表}	从基址Rd处读取多个字并依次送到寄存器列表中的寄存器中，每读一个字后Rd自增一次。16位指令
LDMIA.W Rd!, {寄存器列表}	从基址Rd处读取多个字并依次送到寄存器列表中的寄存器中。每读一个字后Rd自增一次。32位指令
STMIA Rd!, {寄存器列表}	将寄存器列表中各寄存器的值依次存储到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}	将寄存器列表中各寄存器的值依次存储到Rd给出的地址。每存一个字后Rd自增一次。32位指令
LDMDB.W Rd!, {寄存器列表}	从基址Rd处读取多个字并依次送到寄存器列表中的寄存器中。每读一个字之前Rd自减一次。32位指令
STMDB.W Rd!, {寄存器列表}	存储多个字到Rd处。每存一个字之前Rd自减一次。32位指令

！：自增(Increment)或自减（Decrement）后修改基址寄存器Rd的值

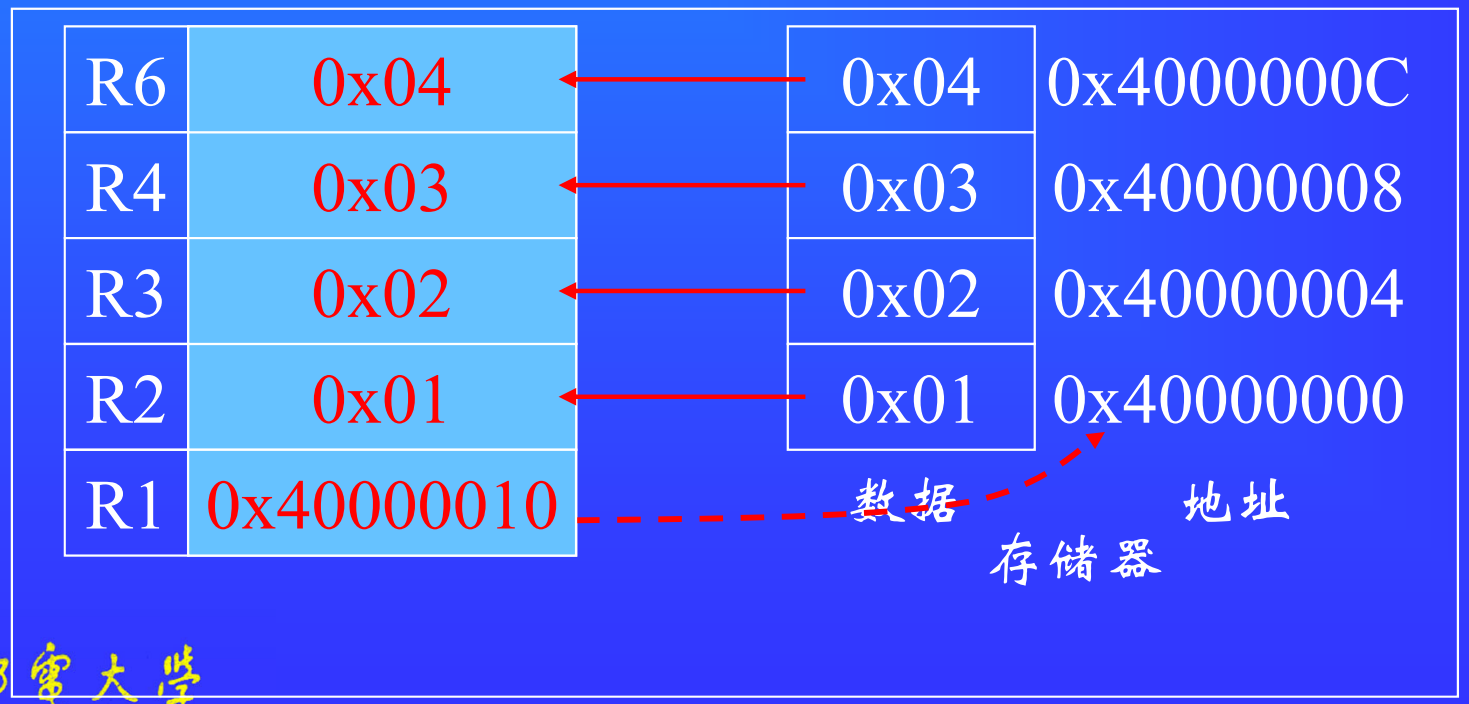
- 例：设R8=0x8000，则

STMIA.W R8!, {R0-R3}；R8值变为0x8010

STMIA.W R8, {R0-R3} ；R8值不变

块拷贝（多寄存器）寻址

- 一条指令可以传送多个寄存器（最多16个）的值
- 例：LDMIA R1, {R0, R2, R5}
  - 将R1所指向的连续3个存储单元中的内容分别送到寄存器R0、R2、R5中
  - ; R0←[R1], R2←[R1+4], R5←[R1+8]
  - 传送的数据项总是32位的字，基址R1需字对齐
- 例：**LDR R1!,{R2-R4,R6}**

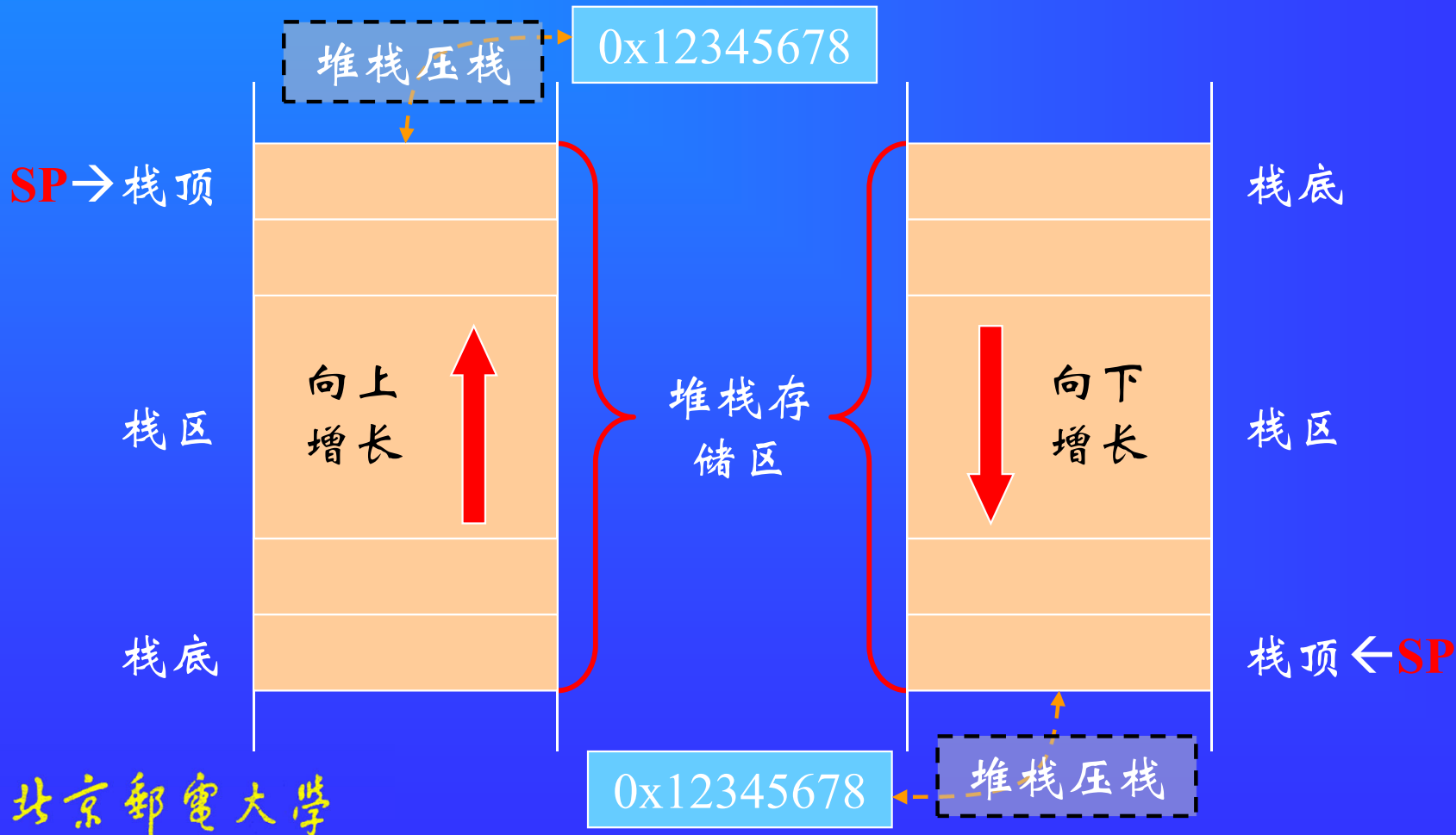


块拷贝（多寄存器）寻址

- 寄存器子集的顺序是按由小到大的顺序排列
- 连续编号寄存器可用“-”连接；不连续编号寄存器用“，”分隔
- 例：
  - 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4）
  - 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4）

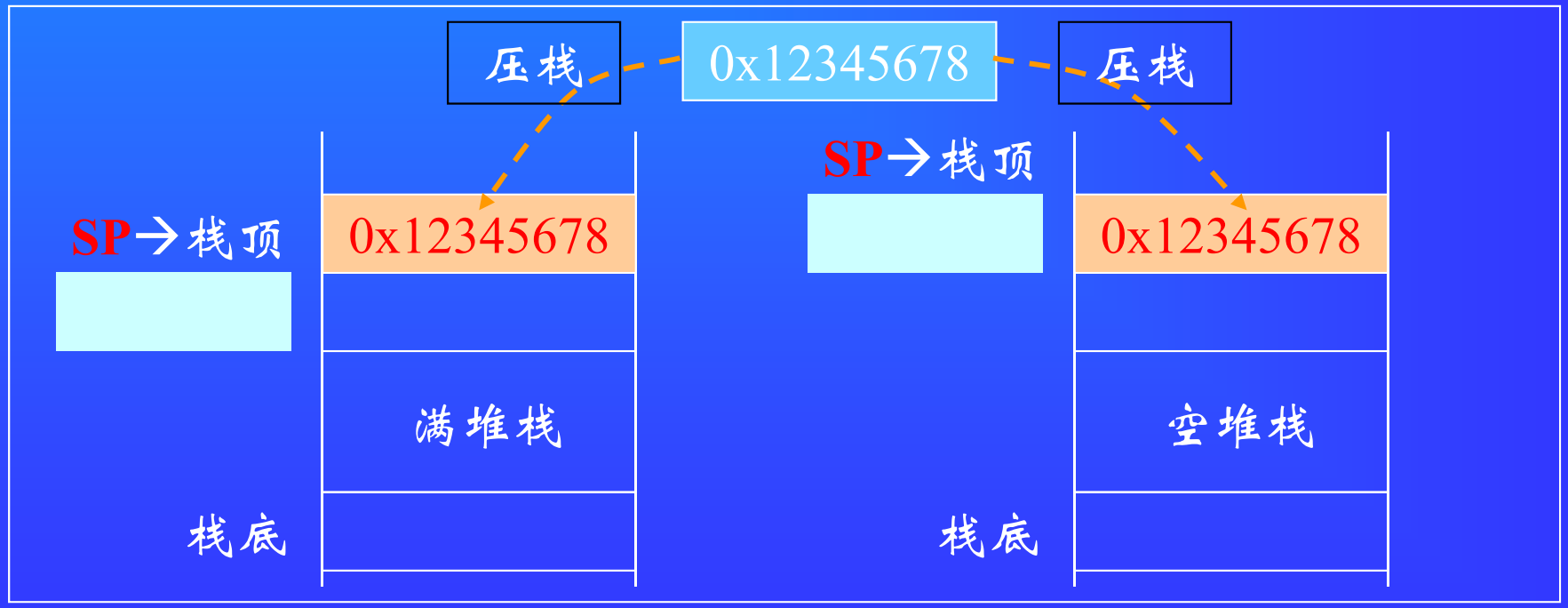
堆栈寻址

- 先进后出，隐含使用堆栈指针寄存器指向存储器区域
- 两种类型：
  - 向上生长（递增，Ascending）堆栈：地址向高地址方向生长
  - 向下生长（递减，Decending）堆栈：地址向低地址方向生长



堆栈寻址

- 两种方式
  - 满堆栈（Full Stack）：SP指向最后压入堆栈的有效数据单元
  - 空堆栈（Empty stack）：SP指向下一个入栈位置（空单元）



经典ARM处理器的堆栈寻址

- 使用LDM/STM指令加FD、ED、FA、EA后缀实现入栈/出栈操作
- 支持四种类型的堆栈工作方式：
  - 满递增堆栈FA（Full Ascending）：堆栈指针指向最后压入的数据单元，且由低地址向高地址生长
  - 满递减堆栈FD（Full Descending）：堆栈指针指向最后压入的数据单元，且由高地址向低地址生长
  - 空递增堆栈EA（Empty Ascending）：堆栈指针指向下一个将要放入数据的空单元，且由低地址向高地址生长
  - 空递减堆栈ED（Empty Descending）：堆栈指针指向下一个将要放入数据的空单元，且由高地址向低地址生长
- 例：
  - STMFD sp!, {r4-r7, lr} ；将 r4~r7、lr入栈，满递减堆栈
  - LDMFD sp!, {r4-r7, pc} ；数据出栈，放入r4~r7、pc寄器

ARMv7-M堆栈寻址

- 增加PUSH/POP指令
  - 满递增堆栈FA（Full Ascending）：
    - 堆栈指针指向最后压入的数据单元，且由低地址向高地址生长

  - PUSH<c><q> <registers> ; Standard syntax
  - STMDB<c><q> SP!, <registers> ; Equivalent STM syntax

  - POP<c><q> <registers> ; Standard syntax
  - LDMIA<c><q> SP!, <registers> ; Equivalent LDM syntax

堆栈操作

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

- POP {<registers>}
  - Pop some or all of registers (R0-R7, PC) from stack
  - **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)
  - e.g. POP {r5, r6, r7}

批量指令的应用——简单块复制

```
loop
LDMIA R12!, {R0-R11}    ;从源数据区读取12个字
STMIA R13!, {R0-R11}    ;将12个字保存到目标区
CMP R12, R14            ;是否到达源数据尾?
BLO loop
```

Cortex M3/M4指令集

- 数据处理类指令
  - 算术运算
  - 逻辑运算

算术运算指令

- 基本的加、减法运算指令
  - ADD：加法
    - ADD Rd, Rn, Rm ; Rd = Rn+Rm
    - ADD Rd, Rm ; Rd += Rm
    - ADD Rd, #imm ; Rd += imm
    - ADDS Rd, Rn, Rm ; 影响条件标志
    - ADDS Rd, Rn, #imm
    - ADD Rdm, SP, Rdm ; 堆栈指针加寄存器操作
    - ADD SP, Rm
    - ADD Rd, SP, #imm ;堆栈指针加立即数操作
    - ADD SP, SP, #imm

算术运算指令

- 基本的加、减法运算指令
  - ADD：加法
  - ADC：带进位加
    - ADC Rd, Rn, Rm ; Rd = Rn+Rm+C
    - ADC Rd, Rm ; Rd += Rm+C
    - ADC Rd, #imm ; Rd += imm+C
    - ADCS Rd, Rm ; 影响条件标志
  - SUB：减法
    - SUB Rd, Rn
    - SUBS Rd, Rn, Rm ;两个寄存器减，影响条件标志
    - SUBS Rd, Rn, #imm;寄存器减去一个立即数
    - SUBS Rd, #imm
    - SUB SP, SP, #imm ；从SP中减去一个立即数

算术运算指令

- 基本的加、减法运算指令
  - ADD：加法
  - ADC：带进位加
  - SUB：减法
  - SBC：带借位的寄存器减
    - SBC.W Rd, Rn, #imm
  - RSB：反向减法
    - RSB.W Rd, Rn, #imm ; Rd = #imm-Rn
    - RSB.W Rd, Rn, Rm ; Rd = Rm – Rn

算术运算指令

- CMP指令：比较（Compare）
  - subtracts second value from first, discards result, updates APSR
  - CMP Rn, #imm
  - CMP Rn, Rm
- CMN指令：取负数比较（Compare negative）
  - **adds** two values, updates APSR, discards result
  - CMN <Rn>,<Rm>

算术运算指令

- 乘、除法指令
  - MUL：Multiply multiplies two register values
    - Assembler syntax：
      - MUL{S}<c><q> {<Rd>,} <Rn>, <Rm>
    - The least significant 32 bits of the result are written to the destination register
      - 结果的高32位被丢弃
      - These 32 bits do not depend on whether signed or unsigned calculations are performed
    - It can optionally update the condition flags based on the result
    - 例：
      - MUL Rd, Rm ; Rd *= Rm
      - MUL.W Rd, Rn, Rm ; Rd = Rn*Rm

算术运算指令

- 32位乘32位乘法运算（结果为64位）：
  - SMULL：带符号数64位乘法
    - Signed Multiply Long
    - multiplies two 32-bit signed values to produce a 64-bit result
    - SMULL RL, RH, Rm, Rn ;[RH:RL]= Rm*Rn
  - UMULL：无符号数64位乘法
    - UMULL RL, RH, Rm, Rn ;[RH:RL]= Rm*Rn

算术运算指令

- 乘、除法指令
  - UDIV：Unsigned Divide（无符号数除法）
    - Assembler syntax
      - UDIV<c><q> {<Rd>,} <Rn>, <Rm>
    - divides a 32-bit unsigned integer register value by a 32-bit unsigned integer register value
    - writes the result to the destination register
    - The condition flags are not affected
    - 例：
      - UDIV Rd, Rn, Rm ; Rd = Rn/Rm

算术运算指令

- 乘、除法指令
  - UDIV：Unsigned Divide
  - SDIV：Signed Divide（带符号数除法）
    - Assembler syntax
      - SDIV<c><q> {<Rd>,} <Rn>, <Rm>
    - divides a 32-bit signed integer register value by a 32-bit signed integer register value
    - writes the result to the destination register
    - The condition flags are not affected
    - 例
      - SDIV Rd, Rn, Rm ; Rd = Rn/Rm

算术运算指令的应用

- 64位数据运算
  - ADDS R0, R0, R2 ；低32位相加，设置APSR的C标志
  - ADC R1，R1，R3 ；高32位的带进位相加
  - SUBS R0，R0，R2; 低32位相减，设置APSR的C标志
  - SBC R1，R1，R3 ；高32位的带借位相减
  - CMP R1，R3　　；比较高32位
  - CMPEQ R0，R2　；如果高32位相等，则比较低32位

逻辑运算指令

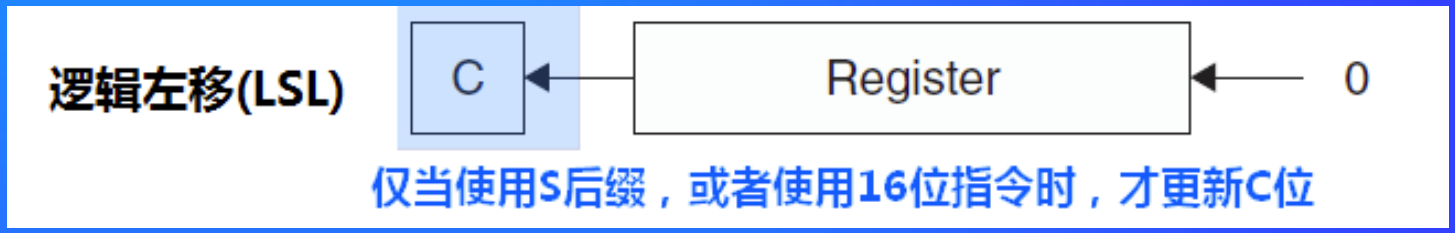
- 按位与
  - AND.W Rd, Rm, Rn ; Rd = Rm & Rn
  - AND.W Rd, Rn, #imm12 ; Rd = Rn & imm12
  - AND Rd, Rn ; Rd &= Rn
- 按位或
  - ORR.W Rd, Rm, Rn ; Rd = Rm | Rn
  - ORR.W Rd, Rn, #imm12 ; Rd = Rn | imm12
  - ORR Rd, Rn ; Rd |= Rn
- 位测试
  - 两个寄存器按位相与，不保存结果，影响条件标志
  - TST <Rn>, <Rm>

逻辑运算指令

- 按位清零（Bit Clear)
  - BIC.W Rd, Rn, #imm12 ; Rd = Rn & ~imm12
    - BIC R0,R0,#0xFF000000 ; Clear top 8 bits
  - BIC.W Rd, Rm, Rn ; Rd = Rm & ~Rn
  - BIC Rd, Rn ; Rd &= ~Rn
- 按位或反（Logical OR NOT)
  - ORN.W Rd, Rm, Rn ; Rd = Rm | ~Rn
  - ORN.W Rd, Rn, #imm12 ; Rd = Rn | ~imm12
- 按位异或
  - EOR.W Rd, Rm, Rn ; Rd = Rm ^ Rn
  - EOR.W Rd, Rn, #imm12 ; Rd = Rn ^ imm12
  - EOR Rd, Rn ; Rd ^= Rn

移位指令

- 逻辑左移（Logical Shift Left）
  - shifts a register value left by a variable number of bits
  - shifting in zeros
  - writes the result to the destination register



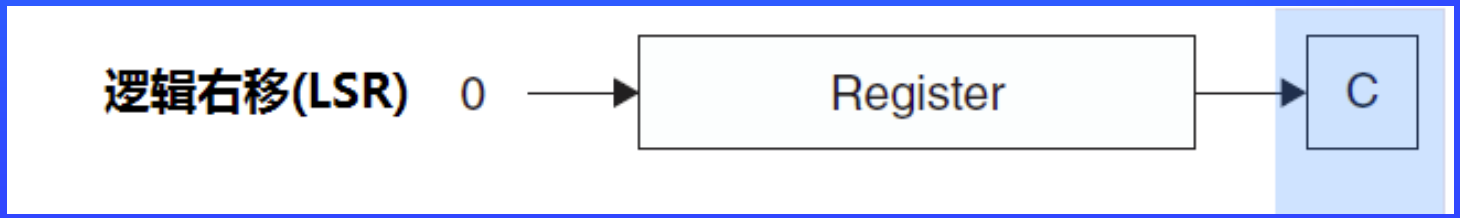
  - LSL.W Rd, Rm, Rn ; Rd = Rm<<Rn<7:0>
    - The variable number of bits is read from the bottom byte of a register
  - LSL Rd, Rn, #imm5 ; Rd = Rn<<imm5
  - LSL Rd, Rn ; Rd <<= Rn<7:0>

MOV R0, R2, LSL #3

LSL：寄存器移位寻址

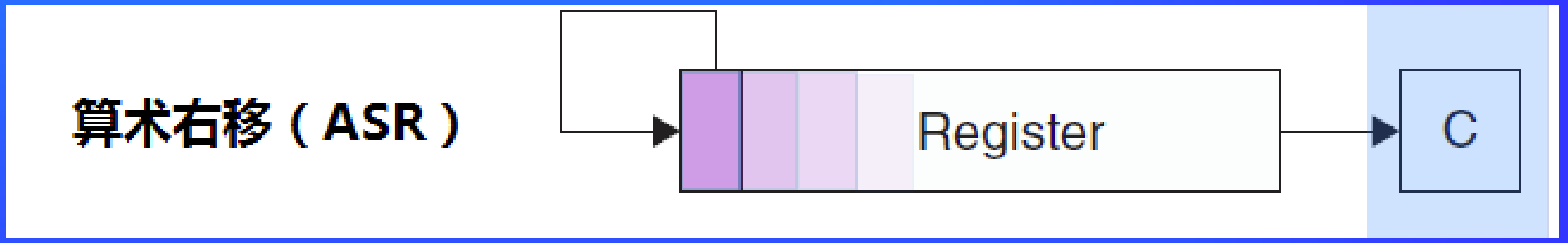
移位指令

- 逻辑右移
  - LSR.W Rd, Rm, Rn ; Rd = Rm>>Rn<7:0>
  - LSR Rd, Rn, #imm5 ; Rd = Rn>>imm5
  - LSR Rd, Rn ; Rd >>= Rn<7:0>



移位指令

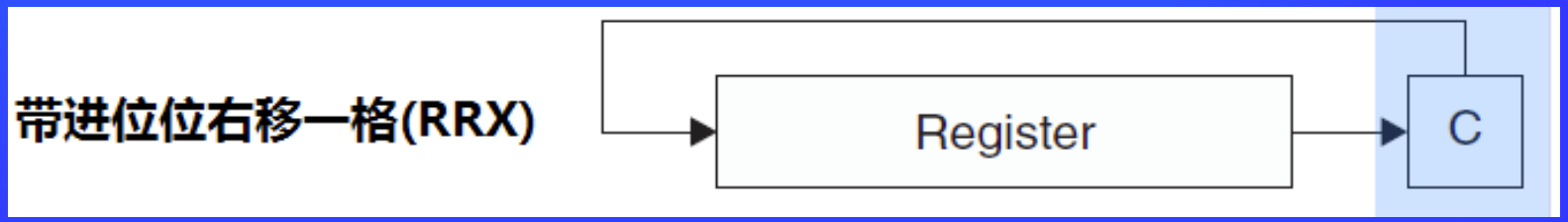
- 算术右移（Arithmetic Shift Right）
  - ASR.W Rd, Rm, Rn ; Rd = Rm>>Rn<7:0>
  - ASR Rd, Rn, #imm5 ; Rd = Rn>> imm5
  - ASR Rd, Rn ; Rd =>> Rn<7:0>



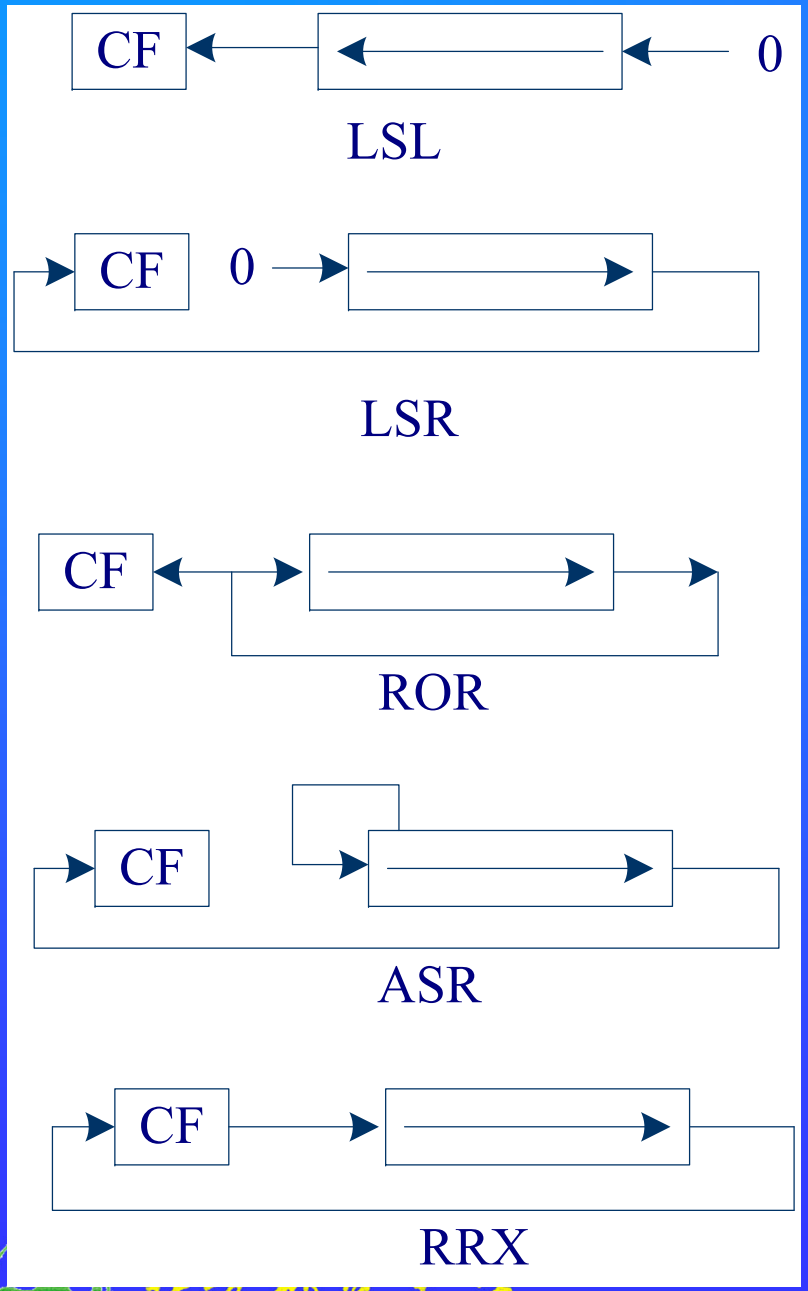
移位指令

- 循环右移（Rotate Right）
  - ROR.W Rd, Rm, Rn ;
  - ROR Rd, Rn ;

- 扩展循环右移（Rotate Right with Extend）
  - RRX: provides the value of the contents of a register shifted right by one place, with the carry flag shifted into bit[31].



移位指令



- 思考：为什么没有
  - ASL（算术左移）？
    - ASL=LSL
  - ROL（循环左移）？
    - ROR 32-n

符号扩展指令

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

算术逻辑运算指令的应用

- 将R2的高8位数据传送到R3的低8位中

MOV R0，R2，LSR#24

ORR R3，R0，R3，LSL #8

分支指令

- Branch：B
  - B{cond} label ; 跳转到Label处对应的地址
  - B Label ; Branch Unconditional to Label
  - BLE ng ; Conditionally branch to label ng
- Branch indirect：BX（Branch With exchange）
  - BX{cond} Rm ;跳转到由寄存器reg给出的地址
    - Rm的bit[0]必须是1，但在创建跳转地址时会把bit[0]置为0

分支指令

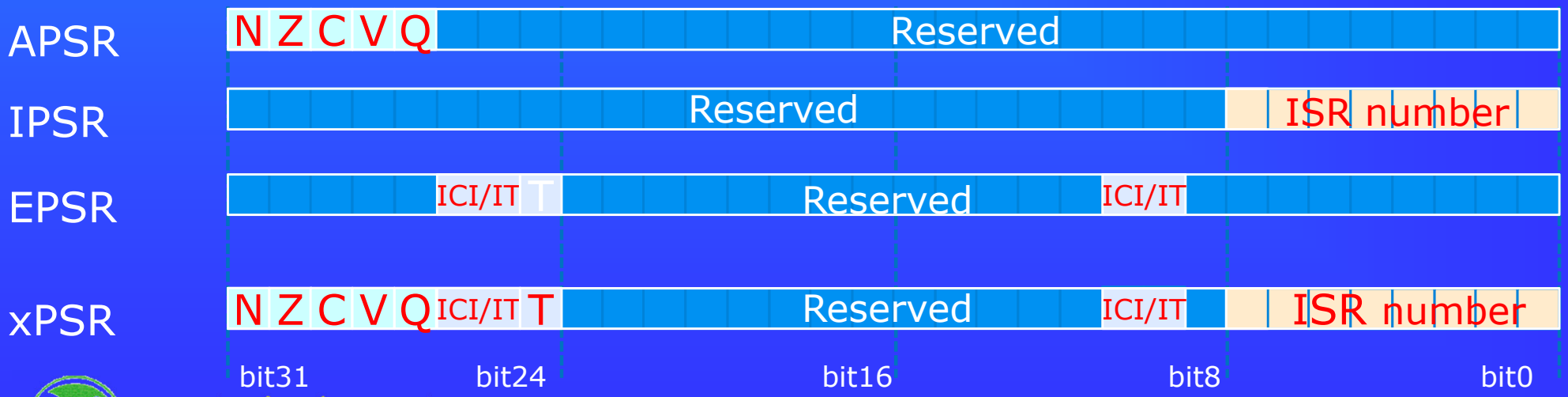
- Branch：B
- Branch indirect：BX (Branch With exchange)
  - BX{cond} Rm ；跳转到由寄存器reg给出的地址
    - Rm的bit[0]必须是1，但在创建跳转地址时会把bit[0]置为0
- branch with link：BL
  - BL{cond} label ；跳转到Label对应的地址，并且把转移前的
    ；下条指令地址保存到LR
    ；调用<label>处的子程序
  - BX LR ；从子程序返回
- branch indirect with link：BLX
  - BLX{cond} Rm ;跳转到由寄存器Rm给出的地址，并把转移
    ;前的下条指令地址保存到LR
    ;调用地址在寄存器Rm中的子程序

转移条件?

Cortex-M3/M4寄存器组——特殊寄存器

- xPSR：组合程序状态寄存器（combined Program Status Register），由三个程序状态寄存器组成
  - 应用PSR（APSR）：包含前一条指令执行后的条件标志
  - 中断PSR（IPSR）：包含当前ISR的异常编号
  - 执行PSR（EPSR）：包含Thumb状态位

三个分离寄存器可以作为一个组合寄存器（PSR）访问



条件码标志（APSR）

- 条件码标志：指令执行结果的特征
  - N（负或小于）标志：带符号数（补码）
    - 负数：N=1
    - 正数或0：N=0
  - Z（零）标志：
    - 0：Z=1
    - 非0：Z=0
  - V（溢出）标志
    - 加法或减法指令（带符号数）
      - 有溢出：V=1
      - 无溢出：V=0
    - 非加法/减法指令：
      - V标志不变

条件码标志（APSR）

- 条件码标志：指令执行结果的特征
  - C（进位、借位、扩展）标志：
    - 加法指令以及比较指令CMN
      - 有进位：C=1
      - 无进位：C=0
    - 减法指令以及比较指令CMP
      - 有借位：C=0
      - 无借位：C=1
    - 移位操作指令
      - C标志值为移出的最后一位的值
    - 其他非加法/减法指令
      - C标志不变

条件码

cond	Mnemonic extension	Meaning, integer arithmetic	Meaning, floating-point arithmetic[a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS (HS)	Carry set (unsigned higher or same)	Greater than, equal, or unordered	C == 1
0011	CC (LO)	Carry clear(unsigned lower)	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL)[d]	Always (unconditional)	Always (unconditional)	Any

分支指令

- Instruction Branch range

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	寄存器可以保存任何值
BLX{cond} Rm	寄存器可以保存任何值

分支指令

- **Examples**
  - B loopA ; Branch to loopA
  - BLE ng ; Conditionally branch to label ng
  - B.W target ; Branch to target within 16MB range
  - BEQ target ; Conditionally branch to target
  - BEQ.W target ; Conditionally branch to target within 1MB
  - BL funC ; Branch with link (Call) to function funC,
    ; return address stored in LR
  - BX LR ; Return from function call
  - BXNE R0 ; Conditionally branch to address stored in R0
  - BLX R0 ; Branch with link and exchange (Call) to a
    ; address stored in R0

分支指令

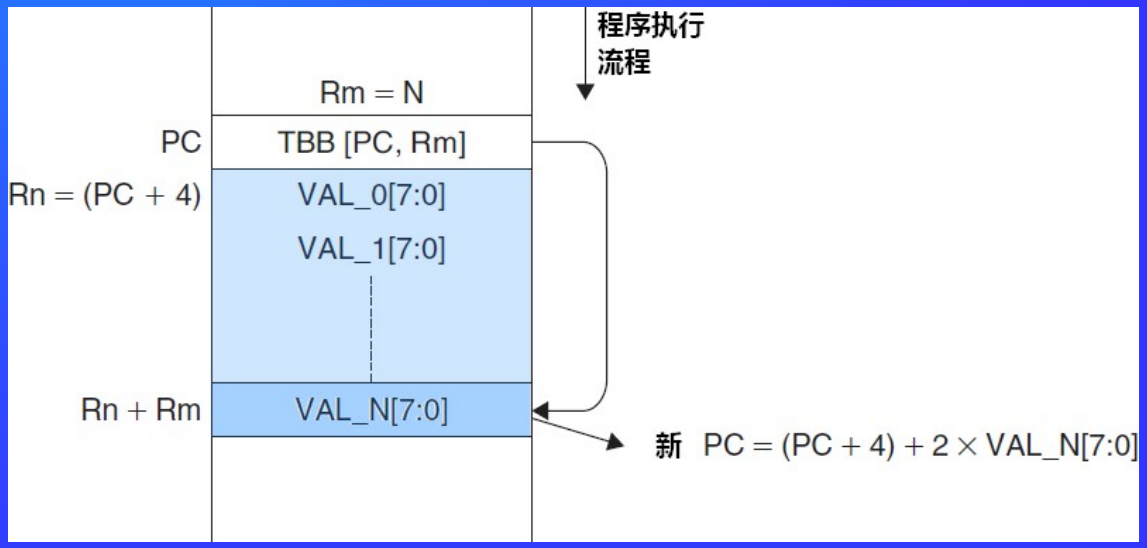
- TBB/TBH指令
  - Table Branch Byte/Table Branch Halfword
  - 实现C语言中的switch case结构，从一个字节/半字数组表中查找转移地址
    - TBB<c><q> [<Rn>, <Rm>]
    - TBH<c><q> [<Rn>, <Rm>, LSL #1]
  - causes a PC-relative forward branch using a table of single byte/halfword offsets
    - A base register <Rn> provides a pointer to the table
    - A second register <Rm> supplies an index into the table

      Rn：指向跳转表（数组）基址

      Rm：数组中元素的下标
    - The branch length is twice the value of the byte/halfword returned from the table

分支指令

- TBB/TBH指令
  - Assembler syntax
    - TBB<c><q> [<Rn>, <Rm>]
    - <Rn>：可以是PC（当前地址+4）。此时跳转表紧跟在当前指令之后
    - TBB.W [PC, Rm]　　; PC+= (Rn+Rm)*2



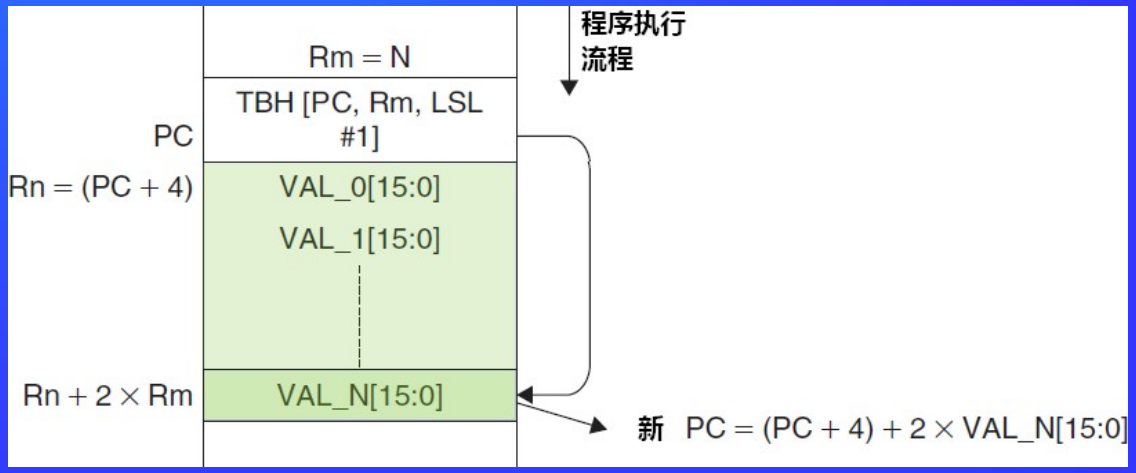
Rn：指向跳转表（数组）基址
Rm：数组中元素的下标

分支指令

TBB/TBH指令

- Assembler syntax
  - TBH<c><q> [<Rn>, <Rm>, LSL #1]
  - <Rn>：The base register，contains the address of the table of branch lengths
  - TBH [PC, Rm，LSL #1]

; PC+= (Rn+Rm*2)*2

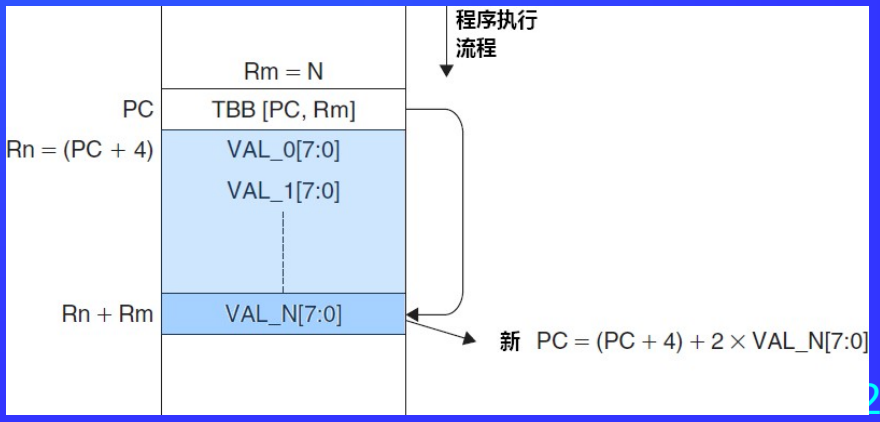


Rn：指向跳转表（数组）基址
Rm：数组中元素的下标

分支指令

- TBB/TBH指令
  - TBB<c><q> [<Rn>, <Rm>]
  - TBH<c><q> [<Rn>, <Rm>, LSL #1]
- 跳转范围
  - 偏移量是无符号数
    - 只能作前向跳转
  - 最大偏移地址：
    - TBB: $2*2^8=512$字节
    - TBH: $2*2^{16}=128$k字节

Rn：指向跳转表（数组）基址
数组中元素的下标： Rn+2*Rm



分支指令

例:

```
TBB.W [pc, r0]          ; 执行此指令时，PC的值正好等于branchtable
branchtable
DCB ((dest0 – branchtable)/2)
DCB ((dest1 – branchtable)/2)
DCB ((dest2 – branchtable)/2)
DCB ((dest3 – branchtable)/2)
dest0
... ; r0 = 0 时执行
dest1
... ; r0 = 1 时执行
dest2
... ; r0 = 2 时执行
dest3
... ; r0 = 3 时执行
```

特殊寄存器指令

- 特殊寄存器至通用寄存器
  - MRS <Rd>, <spec_reg>
- 通用寄存器至特殊寄存器
  - MSR <spec_reg>, <Rd>
- 改变处理器状态——修改PRIMASK寄存器
  - CPSIE - Interrupt enable
  - CPSID - Interrupt disable

Special register	Contents
APSR	The flags from previous instructions.
IAPSR	A composite of IPSR and APSR.
EAPSR	A composite of EPSR and APSR.
XPSR	A composite of all three PSR registers.
IPSR	The Interrupt status register.
EPSR	The execution status register.[b]
IEPSR	A composite of IPSR and EPSR.
MSP	The Main Stack pointer.
PSP	The Process Stack pointer.
PRIMASK	Register to mask out configurable exceptions.[c]
CONTROL	The CONTROL register, see *The special-purpose CONTROL register* on page B1-215.

特殊寄存器指令

- 用MRS和MSR指令访问PRIMASK、FAULTMASK和BASEPRI寄存器

仅在特权等级才能访问

  - MRS r0, BASEPRI
    - ; Read BASEPRI register into R0
  - MRS r0, PRIMASK
    - ; Read PRIMASK register into R0
  - MRS r0, FAULTMASK
    - ; Read FAULTMASK register into R0
  - MSR BASEPRI, r0
    - ; Write R0 into BASEPRI register
  - MSR PRIMASK, r0
    - ; Write R0 into PRIMASK register
  - MSR FAULTMASK, r0
    - ; Write R0 into FAULTMASK register

特殊寄存器指令

- 用MRS和MSR指令访问控制寄存器:
  - MRS r0, CONTROL
    - ; Read CONTROL register into R0
  - MSR CONTROL, r0
    - ; Write R0 into CONTROL register
  - CONTRL[0] 位只在特权状态可写

Other Instructions

- No Operation - does nothing!
  - NOP
- Breakpoint - causes hard fault or debug halt - used to implement software breakpoints
  - BKPT #<imm8>
- Wait for interrupt - Pause program, enter low-power state until a WFI wake-up event occurs (e.g. an interrupt)
  - WFI
- Supervisor call generates SVC exception (#11), same as software interrupt
  - SVC #<imm>

IF-THEN 指令块

If-Then (IT) instruction block

- IT指令块包括：
  - 一条IT指令，包含执行条件
  - 跟着1至4条条件执行指令
    - 各条指令的条件或者全部相同，或者某些条件正好相反

```
CMP R0, R1              ; 比较R0和R1
ITTEE  EQ          ; 如果相等, 则Then-则Then-否则Else-否则Else
ADDEQ R3, R4, R5   ; 相等时加法EQ ;
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
```

If-Then (IT) instruction block

- Syntax: IT{x{y{z}}} cond
  - cond：IT块中第一条指令的执行条件
  - 执行开关
    - x：IT块中第二条指令的执行开关
    - y：IT块中第三条指令的执行开关
    - z：IT块中第四条指令的执行开关
    - 执行开关可以是：
      - T——Then. 满足条件时执行
      - E——Else. 不满足条件时执行

```
CMP R0, R1              ; 比较R0和R1
ITTEE  EQ       ; 如果相等, 则Then-则Then-否则Else-否则Else
ADDEQ R3, R4, R5   ; 相等时加法
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
```

If-Then (IT) instruction block

- IT的使用形式:
  - 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
  - <x>, <y>,<z>的取值: “T”或者“E”

If-Then (IT) instruction block

- 在If-Then块中的指令必须加上条件后缀
  - T：对应条件成立时执行的语句
    - T对应的指令必须使用和IT指令中相同的条件
  - E：对应条件不成立时执行的语句
    - E对应的指令必须使用和IT指令中相反的条件
  - 对T和E的顺序没有要求
  - IT带一个"T"，还可以最多再带 3 个"T"或者"E"
    - 仅一条条件执行指令：IT
    - 两条条件执行指令：ITT, ITE
    - 三条条件执行指令：ITTT, ITTE, ITET, ITEE
    - 四条条件执行指令：ITTTT, ITTTE, ITTET, ITTEE, ITETT, ITETE, ITEET, ITEEE

```
ITTEE  EQ
ADDEQ R3, R4,
ASREQ  R3, R3, #1
ADDNE R3, R6, R7
ASRNE R3, R3, #1
```

IF-THEN 指令块

例：

```
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          ; 如果相等, 则Then-则Then-否则Else-否则Else
ADDEQ R3, R4, R5   ; 相等时加法EQ ;
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
```

IF-THEN 指令块

- 例：将R0中的一位16进制数转成ASCII码
  - （ ‘0’ - ‘9’ ， ‘A’ - ‘F’ ）

  - CMP R0，#9
  - ITE GT
  - ADDGT R1，R0，#55；转换成A-F
  - ADDLE R1，R0，#48；转换成0-9

IF-THEN 指令块

- IT指令的优势
  - 允许指令条件执行
  - 不再预取不满足条件的指令
  - 取代条件转移指令，避免在执行分支转移时清空流水线和重新取指的开销

```
CMP R0, R1                ; 比较R0和R1
ITTEE  EQ          ; 如果相等, 则Then-则Then-否则Else-否则Else
ADDEQ R3, R4, R5   ; 相等时加法EQ ;
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
```

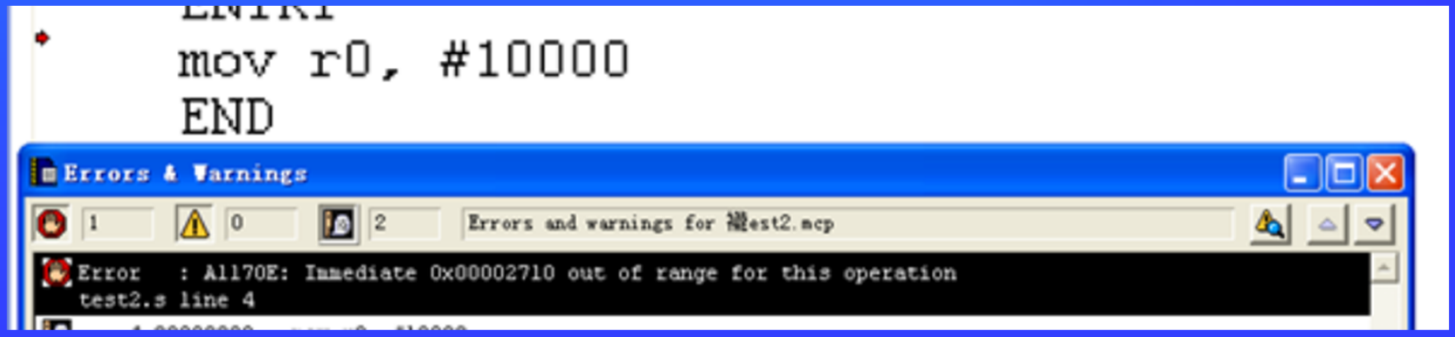
ARM汇编伪指令

伪指令

- ARM汇编伪指令（pseudo-instruction）
  - 为了编程方便而定义，不是ARM指令集中的指令
  - 由汇编器用等效的机器指令或机器指令组合取代
    - 仅在汇编过程中起作用
    - 与特定的汇编器相关
  - 与指令（instruction）的区别
    - 指令有对应的操作码
    - 伪指令在指令集中没有对应的机器编码，由汇编器把一条伪指令翻译为一条或多条机器指令
  - 与汇编伪操作（directive）的区别
    - 会生成机器指令
  - 编程时可以像机器指令一样使用

回顾：立即寻址

- 指令中的地址码字段即是操作数本身
- 例：
  - MOV R0,#0xFF000 ;将立即数0xFF000装入R0寄存器
  - 每个立即数由一个8位常数循环右移偶数位得到
    - 循环右移的位数为4位二进制数的两倍
    - 只有能够通过上述构造方法得到的才是合法的立即数
    - mov r0, #10000 ；汇编时出错:立即数超出范围



LDR伪指令

- LDR R0, =0x100
  - 伪指令LDR的参数有=，无#
  - 汇编器：mov r0, #0x100
    - 常数0x100能够被12bit表示
- LDR R0, =0x12345678 ; Set R0 to 0x12345678
  - 汇编源程序时，LDR伪指令被汇编器器替换成一条适当的机器指令和数据定义伪操作
  - 汇编器将常量放入文字池，并使用一条相对寻址的LDR机器指令从文字池读出常量

```
LDR R0, [PC, #偏移量]

……

DCD 0x12345678    ；定义literal pool（文字池）
```

    - 偏移量：文字池的位置相对于程序计数器的偏移值（通常由汇编器计算）

LDR伪指令

- ldr r0, __main
  - ；LDR指令，将__main位置中的值装入r0中
  - 汇编后，__main将被一个相对于PC的表达式取代
  - LDR R0, [PC, #offset]
- ldr r0, =__main
  - LDR伪指令，取得标号 __main的绝对地址

ADR伪指令

- ADR <Rd>,<label>
  - 地址读入寄存器
  - PC值加立即数，结果写入寄存器
  - 汇编器试图产生一条ADD或SUB指令，基于PC值装入地址
    - ADD Rd, PC, #offset

```
ADR R0, DataTable ；取得标号DataTable的地址到R0
……
ALIGN
DataTable
DCD 0, 245, 132, ……
```

课堂练习

1. 为程序堆栈指针PSP赋新值0x20006000:

```
LDR  R0, =0x20006000
MSR  PSP, R0
```

2. 已知R0等于0x12345678，执行下列指令:

```
REV   R1, R0
REVH  R2, R0
```

R1=? 0x78563412

R2=? 0x34127856

课堂练习

3. R1=0xB4E10C23，执行:

RBIT.W R0, R1

R0=? 0xC430872D

RBIT:以比特为单位将一个字反向

1011_0100_1110_0001_0000_1100_0010_0011
1100_0100_0011_0000_1000_0111_0010_1101

本章重点

- 了解Cortex M3/M4处理器内核的常用机器指令的功能和工作机制
- 能够借助指令表完成基本汇编语言程序设计
- 理解计算机系统工作的核心机制

本章结束