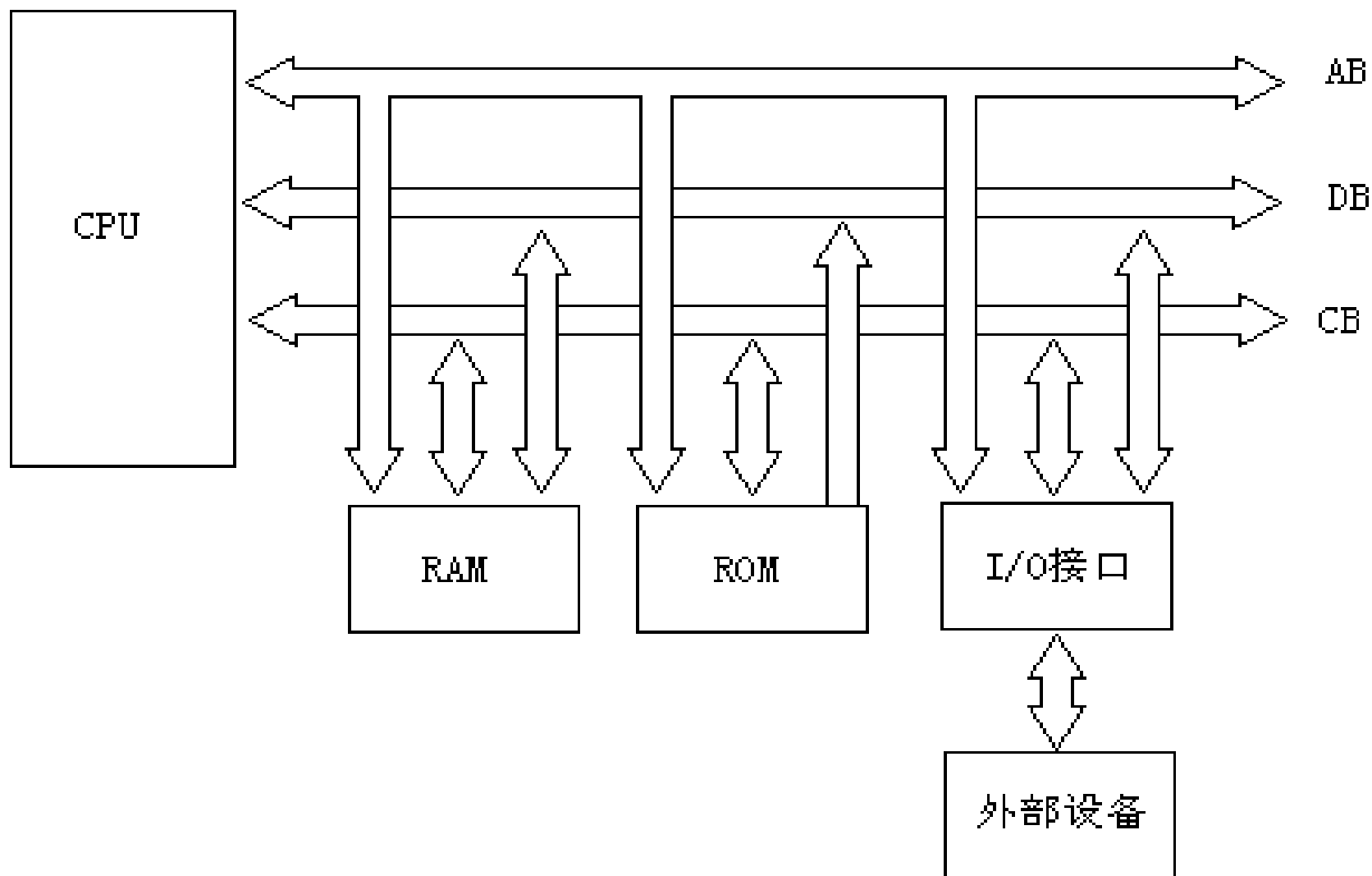


第三讲 ARM汇编程序设计

- 1.ARM编程模型
- 2.ARM指令格式
- 3.ARM指令寻址模式
- 4.ARM指令集
- 5.汇编伪指令与伪操作

1. ARM编程模型

程序的执行过程

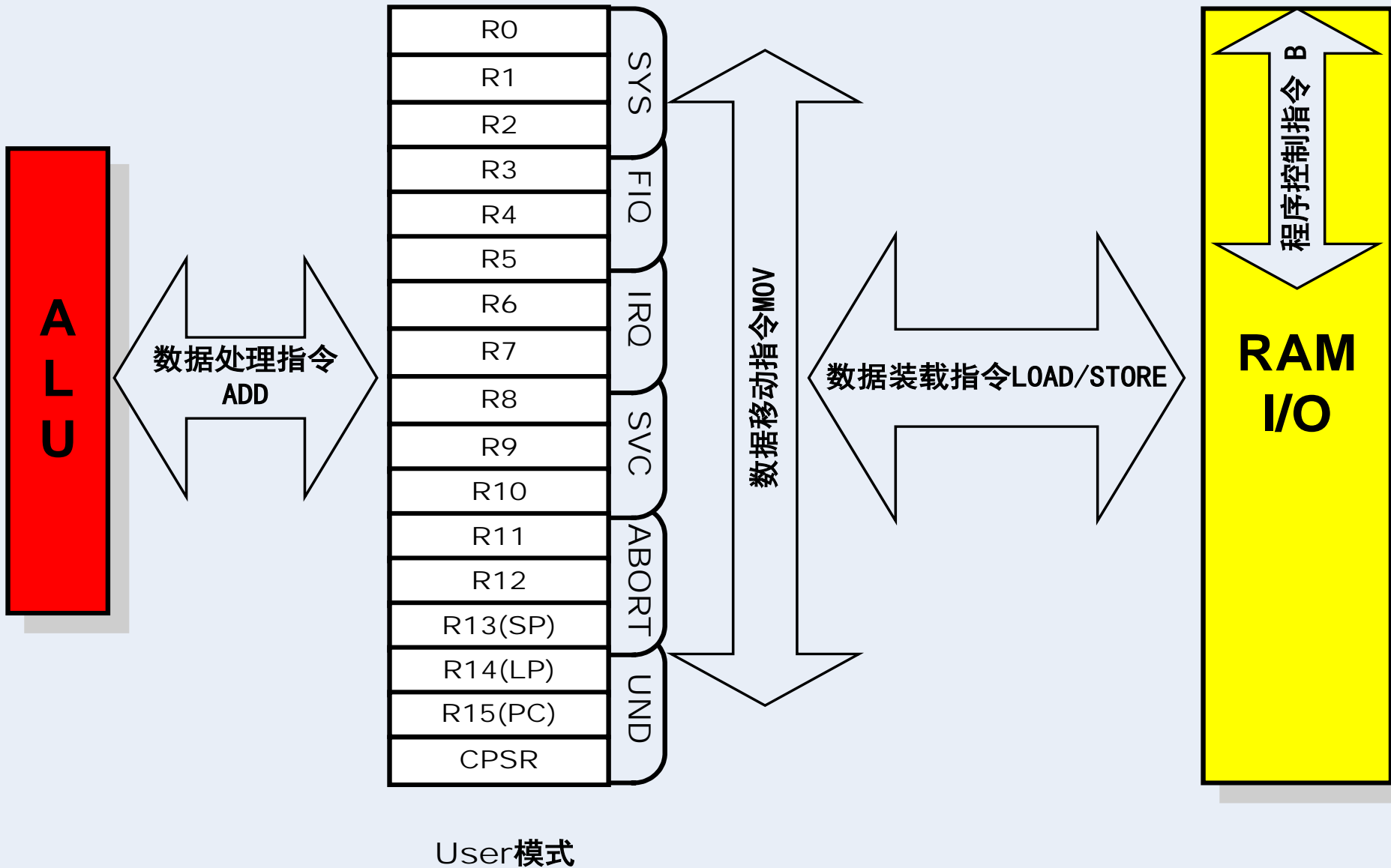


(1)执行一条指令的过程可分为取指、译码和执行等阶段。

(2)**取指**，首先，CPU进入取指阶段，通过控制总线（CB）发出读命令，并在地址总线上（AB）给出所取指令的地址，从存储器中取出的指令代码经过数据总线（DB）送到CPU的指令寄存器中

(3)然后对该**指令译码**

(4)**执行**：再转入执行阶段，在这期间，CPU执行指令指定的操作。



ARM编程模型

1.1 ARM处理器模式

- ARM微处理器支持**7种工作模式**：用户模式、系统模式、快速中断模式、外部中断模式、管理模式、中止模式、未定义指令模式。
- 除**用户模式**之外的其余6种称为非用户模式，或**特权模式**。
- 在特权模式中，除**系统模式**之外的其余5种称为**异常模式**。
- 处理器的各种工作模式由当前程序状态寄存器CPSR的低5位M[4:0]决定。
- **工作模式切换**：
 - (1) **发生异常**，处理器自动改变CPSR中M[4:0]的值，进入相应的工作模式；
 - (2) 处理器处于特权模式时，用指令**向CPSR的M[4:0]字段写入特定的值**，进入相应的工作模式。
- 用户模式时，不能改变工作模式，除非发生异常。

处理器模式

- 7种模式

处理器模式		说明	备注
用户 (usr)		正常程序工作模式	不能直接切换到其它模式
特权模式	系统 (sys)	用于支持操作系统的特权任务等	与用户模式类似，但具有可以直接切换到其它模式等特权
	异常模式	快中断 (fiq)	支持高速数据传输及通道处理 FIQ异常响应时进入此模式
		中断 (irq)	用于通用中断处理 IRQ异常响应时进入此模式
		管理 (svc)	操作系统保护代码 系统复位和软件中断响应时进入此模式
		中止 (abt)	用于支持虚拟内存和/或存储器保护 在ARM7TDMI 没有大用处
		未定义 (und)	支持硬件协处理器的软件仿真 未定义指令异常响应时进入此模式

处理器模式

- 特权模式

处理器模式			说明	备注
用户 (usr)			正常程序工作模式	不能直接切换到其它模式
特权模式	异常模式	系统 (sys)	<p>除用户模式外，其它模式均为特权模式。ARM内部寄存器和一些片内外设在硬件设计上只允许（或者可选为只允许）特权模式下访问。此外，特权模式可以自由的切换处理器模式，而用户模式不能直接切换到别的模式。</p>	
		快中断 (fiq)		
		中断 (irq)		
		管理 (svc)		
		中止 (abt)		
		未定义 (und)		

处理器模式

- 异常模式

处理器模式		说明	备注
用户 (usr)		正常程序工作模式	不能直接切换到其它模式
特权模式	系统 (sys)		<p>这五种模式称为异常模式。</p> <p>它们除了可以通过程序切换进入外，也可以由特定的异常进入。当特定的异常出现时，处理器进入相应的模式。每种异常模式都有一些独立的寄存器，以避免异常退出时用户模式的状态不可靠。</p>
	异常模式	快中断 (fiq)	
		中断 (irq)	
		管理 (svc)	
		中止 (abt)	
		未定义 (und)	

处理器模式

- 用户和系统模式

处理器模式		说明	备注	
	用户 (usr)	<p>这两种模式都不能由异常进入，而且它们使用完全相同的寄存器组。</p> <p>系统模式是特权模式，不受用户模式的限制。操作系统在该模式下访问用户模式的寄存器就比较方便，而且操作系统的一些特权任务可以使用这个模式访问一些受控的资源。</p>		
	系统 (sys)			
特权模式	异常模式		快中断 (fiq)	
			中断 (irq)	
			管理 (svc)	
			中止 (abt)	
		未定义 (und)		

ARM处理器7种工作模式

处理器模式	说明	备注	M[4:0]
用户 (usr)	正常程序执行模式	不能直接切换到其它模式	10000
快速中断 (fiq)	支持高速数据传输及通道处理	FIQ异常响应时进入此模式	10001
外部中断 (irq)	用于通用中断处理	IRQ异常响应时进入此模式	10010
管理 (svc)	操作系统保护模式	系统复位和软件中断响应时进入此模式	10011
中止 (abt)	用于支持虚拟内存和/或存储器保护	在ARM7TDMI没有大用处	10111
未定义 (und)	支持硬件协处理器的软件仿真	未定义指令异常响应时进入此模式	11011
系统 (sys)	运行操作系统的特权任务	与用户模式类似，但具有可以直接切换到其它模式等特权	11111

1.2 ARM内部寄存器

在ARM处理器内部共有37个用户可访问的寄存器，分别为31个通用32位寄存器和6个状态寄存器。

ARM处理器共有7种不同的处理器模式，每种模式都有一组相应的寄存器组，最多可以有18个活动的寄存器。

SPSR：普通模式和系统模式下是看不见SPSR这个寄存器的！只有当进入异常模式的时候，SPSR才会保存当前CPSR的状态，便于退出异常时恢复使用。

ARM状态各模式下的寄存器

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	中止	未定义	中断	快中断
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	CPSR	CPSR						
	SPSR	无		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

ARM状态各模式下可以访问的寄存器

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	中止	未定义	中断	快中断
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	CPSR	CPSR						
	SPSR	无		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

ARM状态各模式下的寄存器

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	中止	未定义	中断	快中断
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						
	<div>所有的37个寄存器，分成两大类：</div> <div><div>■31个通用32位寄存器；</div><div>■6个状态寄存器。</div></div>							
								R8_fiq
								R9_fiq
								R10_fiq
							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	CPSR	CPSR						
	SPSR	无		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

一般的通用寄存器

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	中止	未定义	中断	快中断
通用寄存器	R0(a1)	R0						
	R1(a2)	R1						
通用寄存器	R12(IP)	R12						
	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	CPSR	CPSR						
	SPSR	无		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

在汇编语言中寄存器R0~R13为保存数据或地址值的通用寄存器。它们是完全通用的寄存器，不会被体系结构作为特殊用途，并且可用于任何使用通用寄存器的指令。

一般的通用寄存器

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	中止	未定义	中断	快中断
通用寄存器	其中R0~R7为未分组的寄存器，也就是说对于任何处理器模式，这些寄存器都对应于相同的32位物理寄存器。							
	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						
	CPSR	CPSR						
状态寄存器	SPSR	无		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

一般的通用寄存器

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	中止	未定义	中断	快中断
	R0(a1)	R0						
	R1(a2)	R1						
	R2(a3)	R2						
	R3(a4)	R3						
	R4(v1)	R4						
		R5						
		R6						
		R7						
								R8_fiq
								R9_fiq
								R10_fiq
								R11_fiq
								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	CPSR	CPSR						
	SPSR	无		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

寄存器R8~R14为**分组寄存器**。它们所对应的物理寄存器取决于当前的处理器模式，几乎所有允许使用通用寄存器的指令都允许使用**分组寄存器**

一般的通用寄存器

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	中止	未定义	中断	快中断
通用寄存器和程序计数器	R0(a1)	R0						
	R1(a2)	<div>寄存器R8~R12有两个分组的物理寄存器。一个用于除FIQ模式之外的所有寄存器模式，另一个用于FIQ模式。这样在发生FIQ中断后，可以加速FIQ的处理速度。</div>						
	R2(a3)							
	R3(a4)							
	R4(v1)							
	R5(v2)							
	R6(v3)							
	R7(v4)							
	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	CPSR	CPSR						
	SPSR	无		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

一般的通用寄存器

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	中止	未定义	中断	快中断	
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)	寄存器R13、R14分别有6个分组的物理寄存器。一组用于用户和系统模式，其余5组分别用于5种异常模式。							
	R8(v5)								R8_fiq
	R9(SB,v6)								R9_fiq
	R10(SL,v7)								R10_fiq
	R11(FP,v8)								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	CPSR	CPSR							
	SPSR	无	SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq		

堆栈指针寄存器R13（SP）

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	中止	未定义	中断	快中断
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)	<div>寄存器R13常作为堆栈指针（SP）。在ARM指令集当中，没有以特殊方式使用R13的指令或其它功能，只是习惯上都这样使用。但是在Thumb指令集中存在使用R13的指令。</div>						
	R6(v3)							
	R7(v4)							
	R8(v5)							
	R9(SB,v6)							
	R10(SL,v7)							
	R11(FP,v8)							
	R12(IP)	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	CPSR	CPSR						
	SPSR	无	SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq	

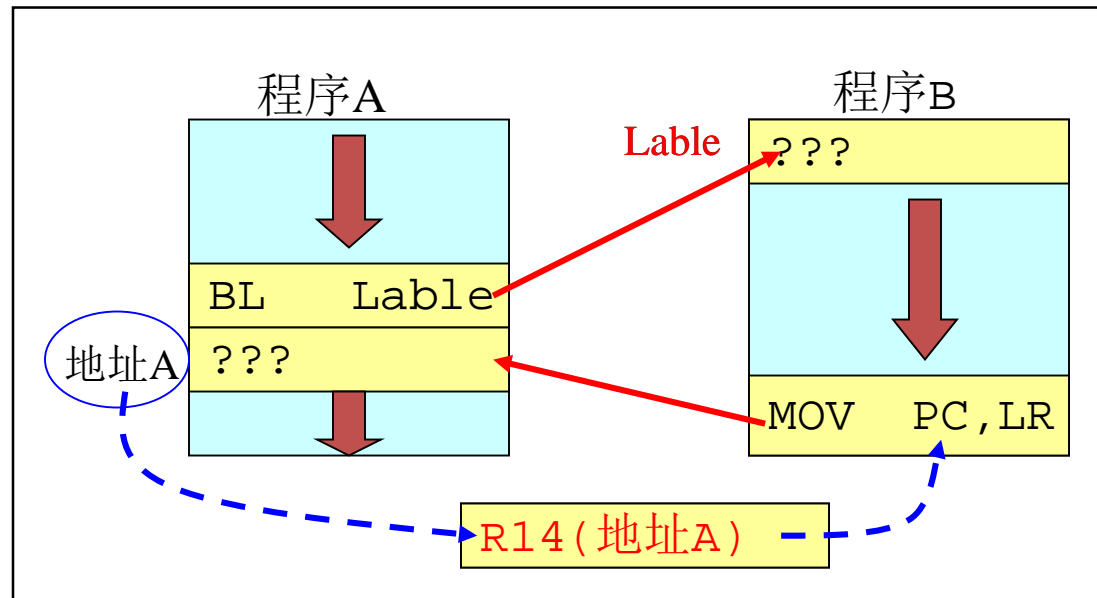
链接寄存器R14（LR）

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	中止	未定义	中断	快中断
通用寄存器和程序计数器	R0(a1)	R0						
	R1(a2)	R1						
	R2(a3)	R2						
	R3(a4)	<div> R14为链接寄存器（LR），在结构上有两个特殊功能： <ul style="list-style-type: none"> ■在每种模式下，模式自身的R14版本用于保存子程序返回地址； ■当发生异常时，将R14对应的异常模式版本设置为异常返回地址（有些异常有一个小的固定偏移量）。 </div>						
	R4(v1)							
	R5(v2)							
	R6(v3)							
	R7(v4)							
	R8(v5)							
	R9(SB,v6)							
	R10(SL,v7)							
	R11(FP,v8)							
	R12(IP)							
	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	CPSR	CPSR						
	SPSR	无		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

R14寄存器与子程序调用

操作流程

1. 程序A执行过程中调用程序B；
2. 程序跳转至标号Lable，执行程序B。同时硬件将“BL Lable”指令的下一条指令所在地址存入R14；
3. 程序B执行最后，将R14寄存器的内容放入PC，返回程序A；



R14寄存器与异常发生

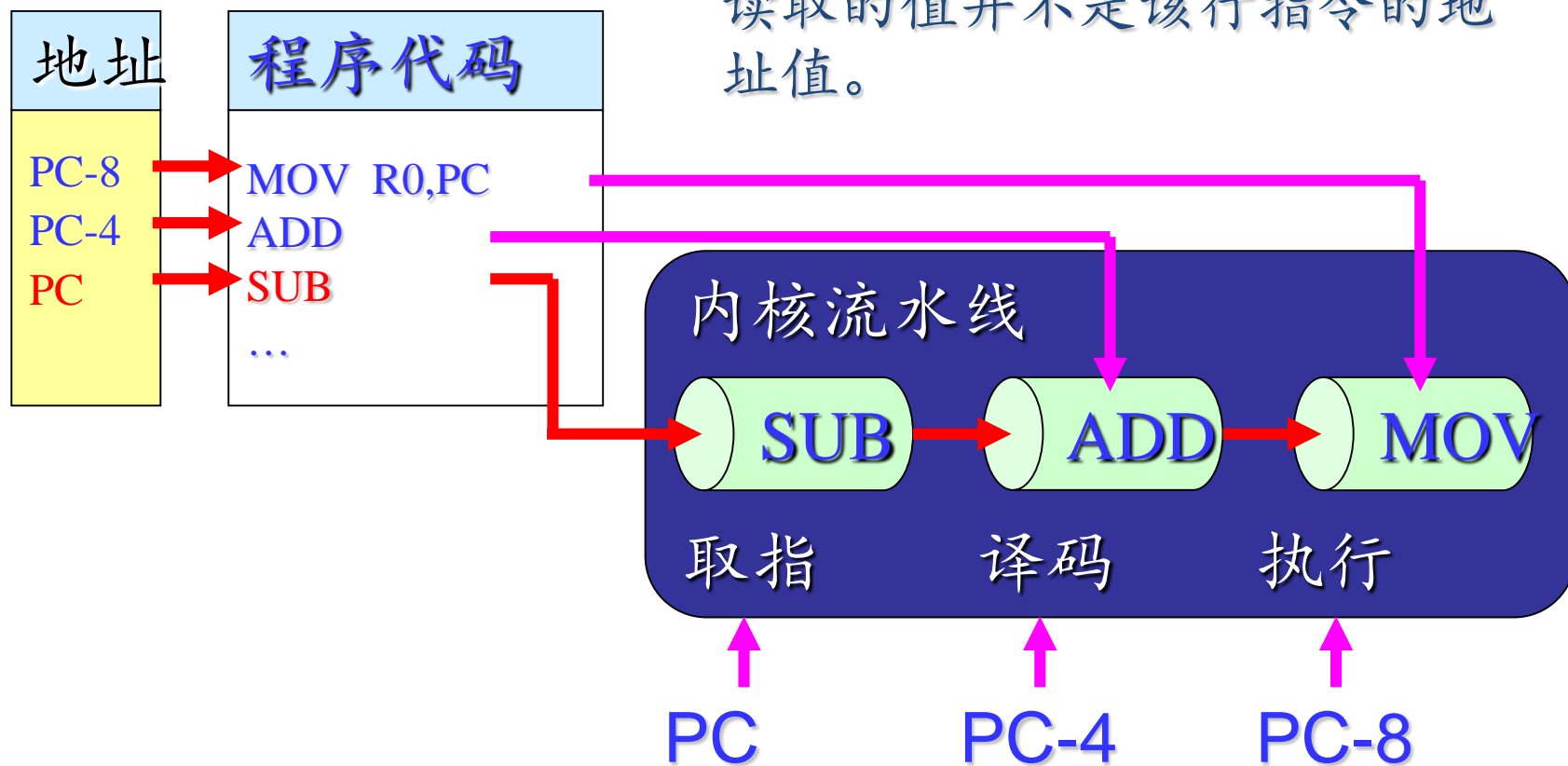
异常发生时，程序要跳转至异常服务程序，对返回地址的处理与子程序调用类似，都是由硬件完成的。区别在于有些异常有一个小常量的偏移。

程序计数器R15 (PC)

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	中止	未定义	中断	快中断
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)	<div> <p>寄存器R15为程序计数器 (PC)，它指向正在取指的地址。可以认为它是一个通用寄存器，但是对于它的使用有许多与指令相关的限制或特殊情况。如果R15使用的方式超出了这些限制，那么结果将是不可预测的。</p> </div>						
	R6(v3)							
	R7(v4)							
	R8(v5)							
	R9(SB,v6)							
	R10(SL,v7)							
	R11(FP,v8)							
	R12(IP)							
	R13(SP)							
	R14(LR)	R14		R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
	R15(PC)	R15						
状态寄存器	CPSR	CPSR						
	SPSR	无		SPSR_abt	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

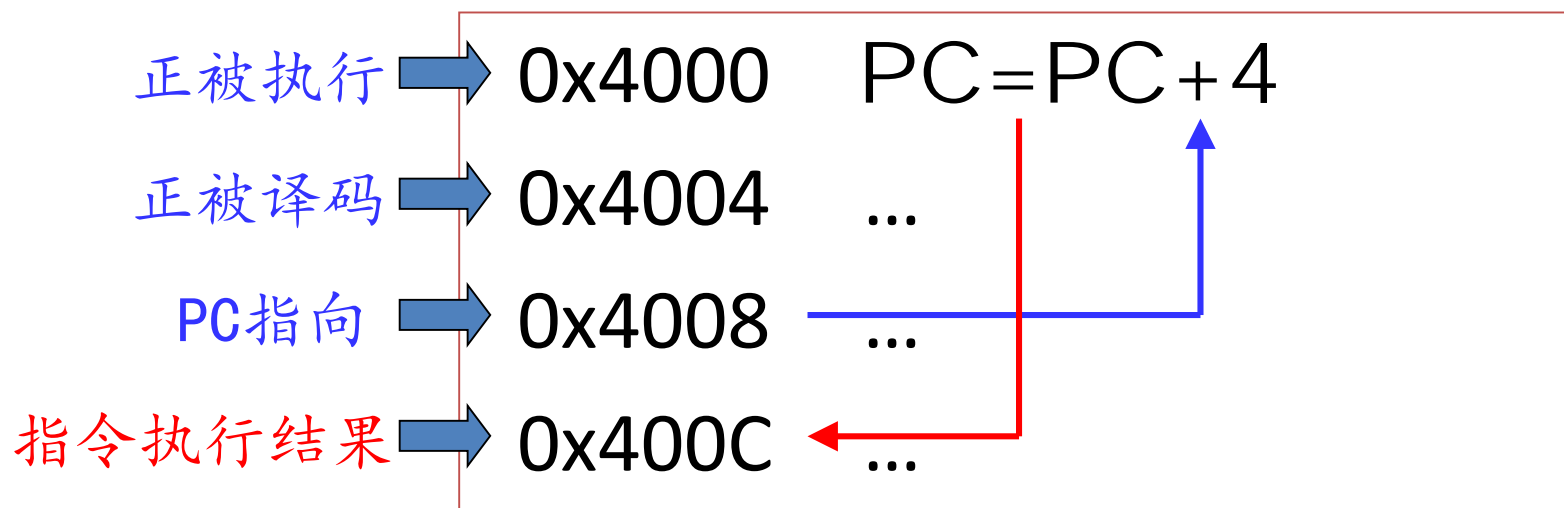
读R15的限制

注意：执行一条PC读取指令时，读取的值并不是该行指令的地址值。



正常操作时，从R15读取的值是处理器正在取指的地址，即当前正在执行指令的地址加上8个字节（两条ARM指令的长度）。由于ARM指令总是以字为单位，所以R15寄存器的最低两位总是为0。

假设CPU正在运行以下的程序，并正在执0x4000处的指令（它的作用是将PC值加4后写入PC），**请问指令执行后PC值是什么？**

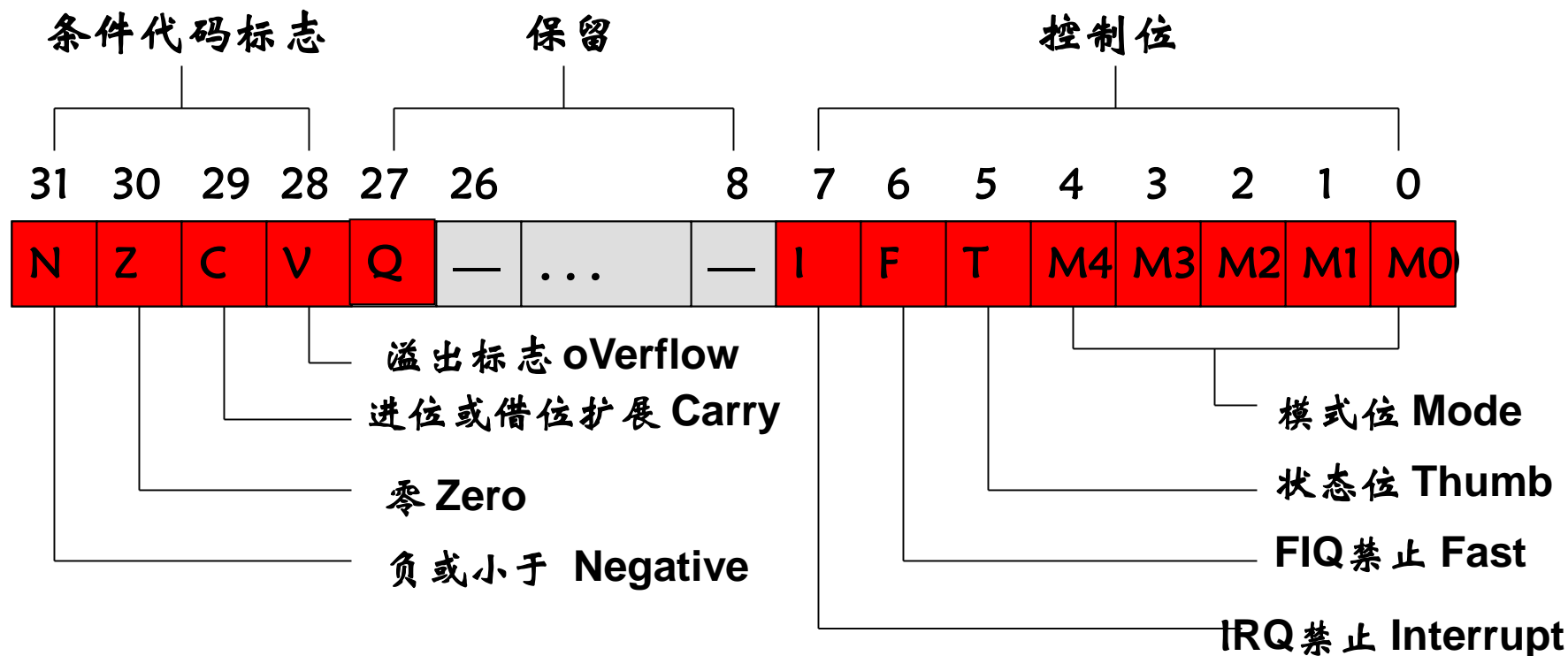


程序状态寄存器CPSR

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	中止	未定义	中断	快中断
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)	<div>寄存器CPSR为当前程序状态寄存器，在异常模式中，另外一个寄存器“保存程序状态寄存器（SPSR）”可以被访问。 每种异常都有自己的SPSR，在进入异常时它保存CPSR的当前值，异常退出时可通过它恢复CPSR。</div>						
	R9(SB,v6)							
	R10(SL,v7)							
	R11(FP,v8)							
	R12(IP)							
	R13(SP)							
	R14(LR)							
	R15(PC)	R15						
状态寄存器	CPSR	CPSR						
	SPSR	无	SPSR_abt	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq	

1.4 程序状态寄存器

- **CPSR**: 当前程序状态寄存器，可以在任何工作模式下被访问。
- **SPSR**: 程序状态保存寄存器，只有在异常模式下，才能被访问；各异常模式拥有自己的SPSR。发生异常时，SPSR保存CPSR的值，格式同CPSR。
- **状态标志**: 5个，N符号位，Z零标志，C进位，V溢出位，Q DSP运算溢出位。
- **控制标志**: 4个，I中断允许，F快速中断允许，T状态选择，M[4:0] 处理器工作模式



条件标志位

标志位	含 义
N	当两个补码表示的带符号数运算时，N=1 表示运算的结果为负数；N=0 表示运算的结果为正数或零；
Z	Z=1 表示运算的结果为零；Z=0表示运算的结果不为零；
C	有3种情况会改变C的值： 加法运算（包括比较指令CMN）：当运算结果产生了进位时（无符号数上溢出），C=1，否则C=0。 减法运算（包括比较指令CMP）：当运算时产生了借位（无符号数下溢出），C=0，否则C=1。 对于包含移位操作的非加/减运算指令，C为移出值的最后一位。
V	对于加/减法运算指令，当操作数和运算结果为二进制的补码表示的带符号数时，V=1表示符号位溢出。
Q	在ARM v5及以上版本的E系列处理器中，用Q标志位指示增强的DSP运算指令是否发生了溢出。

控制位

标志位	含义		
I	I=1，表示禁止IRQ中断；否则，表示允许IRQ中断		
F	F=1，表示禁止FIQ中断；否则，表示允许FIQ中断		
T	对于ARM v4以上版本的T系列处理器，T=0，表示执行ARM指令，否则，表示执行Thumb指令； 对于ARM v5以上版本的非T系列处理器，T=0，表示指令ARM指令，否则，表示强制下一条执行的指令产生未定义指令中断。		
M[4:0]	M[4:0]	处理器工作模式	可访问的寄存器
	10000	用户模式	PC, R0~R14, CPSR
	10001	快速中断模式	PC, R0~R7, R8_fiq~R14_fiq,CPSR,SPSR_fiq
	10010	外部中断模式	PC, R0~R12, R13_irq~R14_irq,CPSR,SPSR_irq
	10011	管理模式	PC, R0~R12, R13_svc~R14_svc,CPSR,SPSR_svc
	10111	中止模式	PC, R0~R12, R13_abt~R14_abt,CPSR,SPSR_abt
	11011	未定义指令模式	PC, R0~R12, R13_und~R14_und,CPSR,SPSR_und
	11111	系统模式	PC, R0~R14, CPSR

1.5 异常

- **异常中断**：处理器由于外部或内部的原因，停止执行当前任务，转而处理特定的事件，处理完后返回原程序，继续执行。
- 当异常发生时，处理器首先自动保存当前状态，即返回地址存入寄存器R14，当前寄存器CPSR存入SPSR中，接着进入相应的工作模式，并执行特定地址的指令。
- 如果同时发生两个或更多异常，那么将按照固定的顺序来处理异常，详见“异常优先级”部分。
- ARM共有7种类型的异常，不同类型的异常将导致处理器进入不同的工作模式，并执行不同特定地址的指令。
- **注**：异常总是在ARM状态中进行处理。当处理器处于Thumb状态时发生了异常，在异常向量地址装入PC时，会自动切换到ARM状态。

图示进入异常过程

2. 用户程序运行时发生

IRQ中断在系统模式或用户模式下运行
 确定程序，假定当前处理

■ 将状态寄存器的内容存入允许

IRQ模式的SPSR寄存器

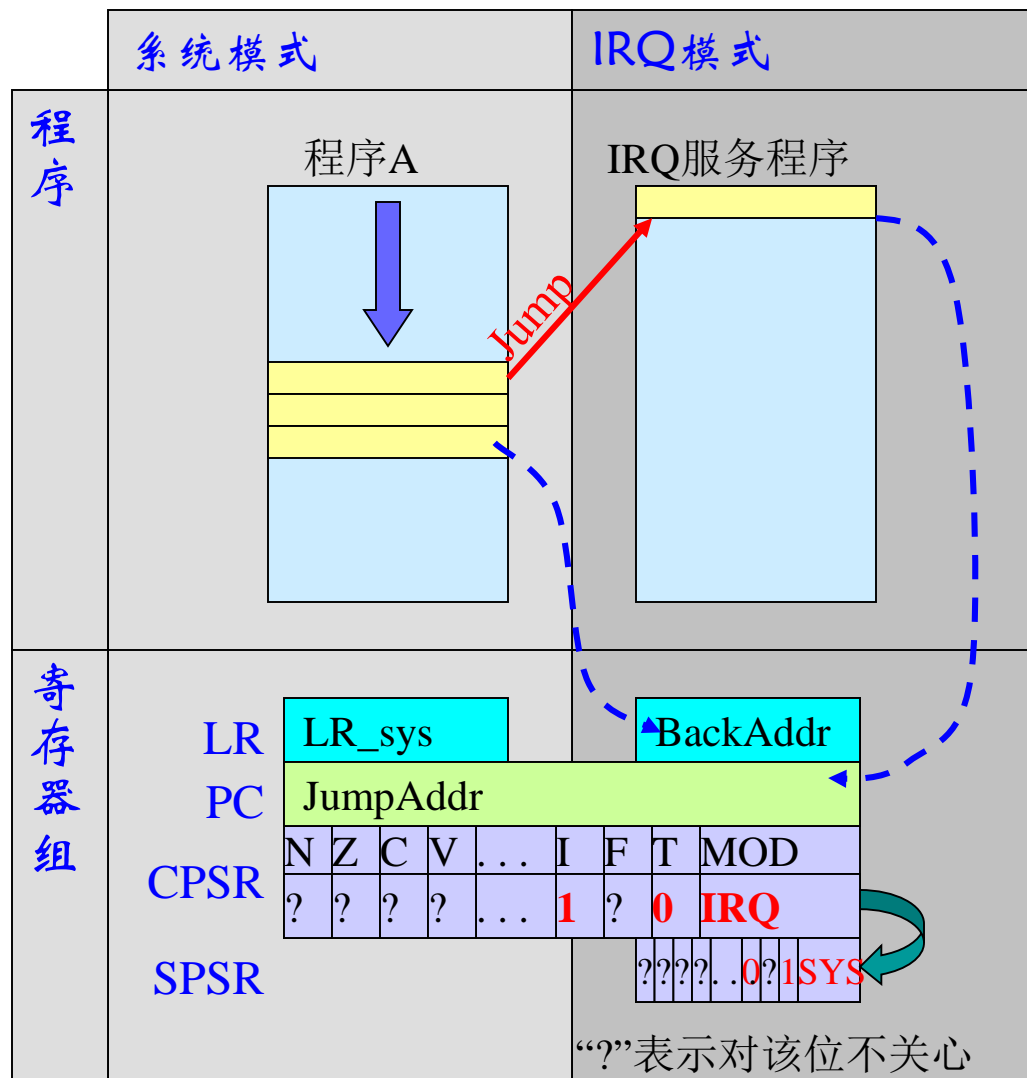
■ 置位I位（禁止IRQ中断）

■ 清零T位（进入ARM状态）

■ 设置MOD位，切换处理器
 模式至IRQ模式

■ 将下一条指令的地址存入
 IRQ模式的LR寄存器

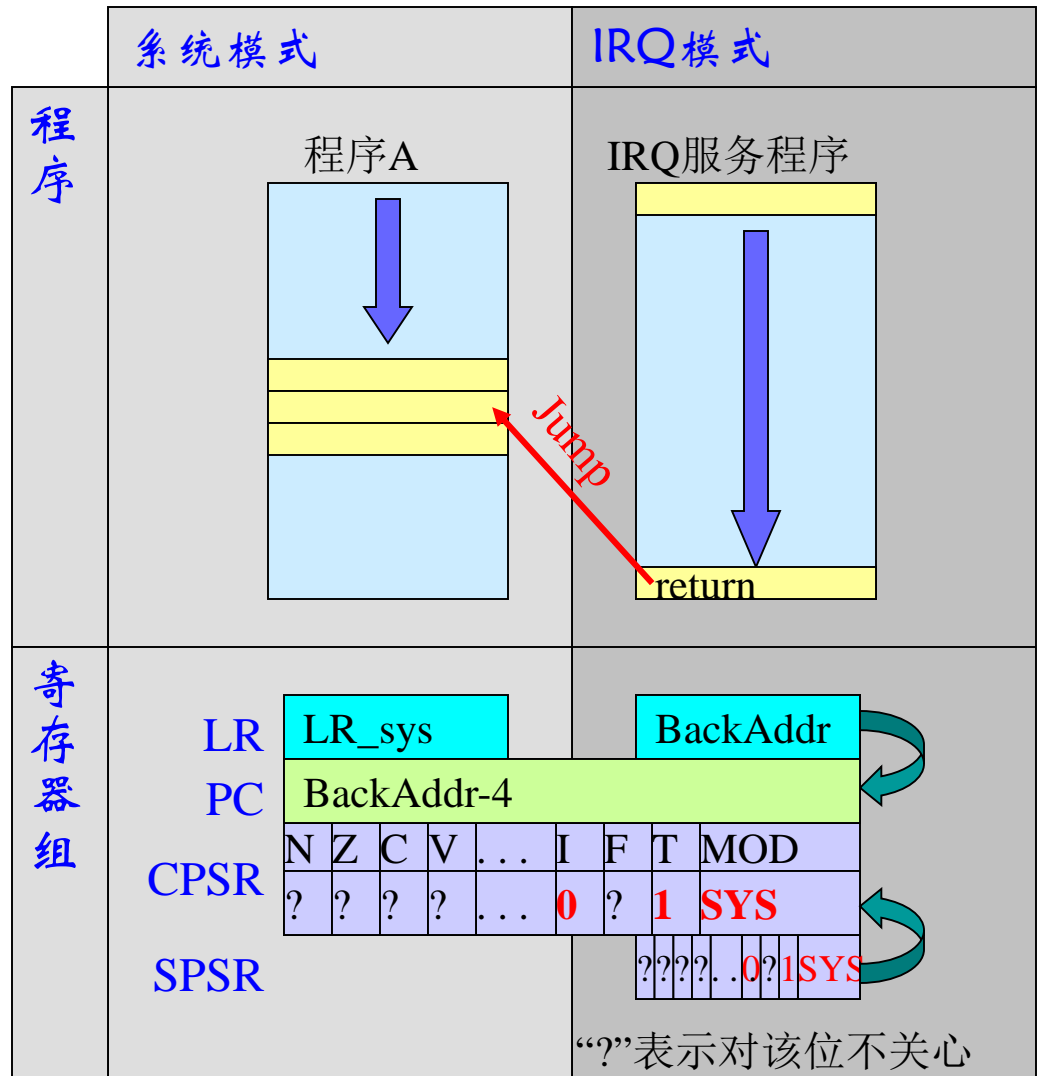
■ 将跳转地址存入PC，实
 现跳转



图示退出异常过程

在异常处理结束后，异常处理程序完成以下动作：


- 将SPSR寄存器的值复制回CPSR寄存器；
- 将LR寄存的值减去一个常量后复制到PC寄存器，跳转到被中断的用户程序。



7种类型异常中断

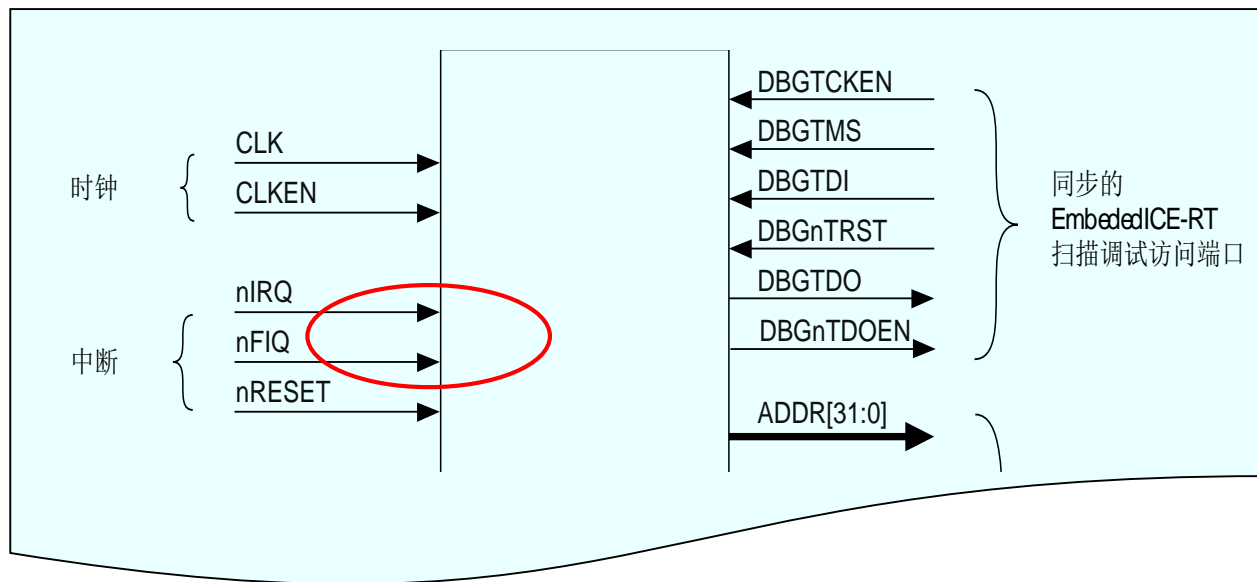
- **复位**：复位异常时，处理器立即停止当前程序，进入禁止中断的管理模式，并从地址0x00000000处开始执行。
- **未定义指令**：当前指令未定义时，便产生未定义指令中断。
- **软件中断**：用户模式下使用指令SWI时，处理器便产生软件中断，进入管理模式，以调用特权操作。
- **指令预取中止**：预取指令的地址不存在，或不允许当前指令访问时，存储器会向处理器发出中止信号；预取指令被执行时才会产生该类异常。
- **数据访问中止**：数据访问指令的地址不存在，或不允许当前指令访问时，产生数据中止异常。
- **外部中断请求**：外部中断请求引脚有效，且CPSR中的I位为0时，产生IRQ异常。
- **快速中断请求**：快速中断请求引脚有效，且CPSR中的F位为0时，产生FIQ异常。

异常中断的优先级

异常类型	优先级	
复位	1（最高优先级）	 优先级降低
数据中止	2	
FIQ	3	
IRQ	4	
预取中止	5	
未定义指令	6	
SWI	6（最低优先级）	

1.5.1 FIQ和IRQ中断

FIQ和IRQ中断请求由ARM内核的nFIQ和nIRQ信号线输入，当信号线上出现有效触发电平时，产生相应的FIQ或IRQ中断，ARM核进入相应的异常模式。



分别执行下面的指令从中断返回：

SUBS PC,R14_fiq,#4; → FIQ模式返回

SUBS PC,R14_irq,#4; → IRQ模式返回

快速中断请求

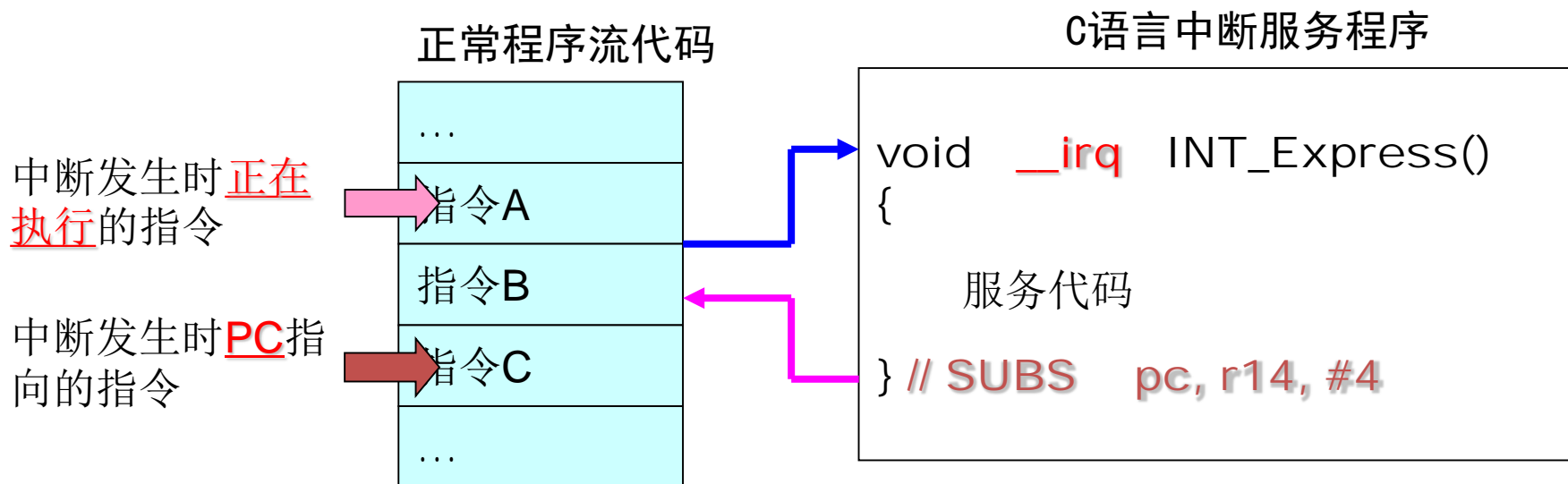
快速中断请求(FIQ)适用于对一个突发事件的快速响应，这得益于在ARM状态中，快中断模式有8个专用的寄存器用来满足寄存器保护的需要（这可以加速上下文切换的速度）。

ARM状态各模式下可以访问的寄存器

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	中止	未定义	中断	快中断
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	CPSR	CPSR						
	SPSR	无		SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq	

与其它模式相比，它有更多的自有寄存器，可以减少程序中堆栈的操作，提高处理速度。

IRQ与FIQ的返回地址处理



1.5.2 中止

中止发生在对存储器的访问不能完成时。如试图访问一个保留地址或未分配区域的地址，ARM处理器将产生中止异常。

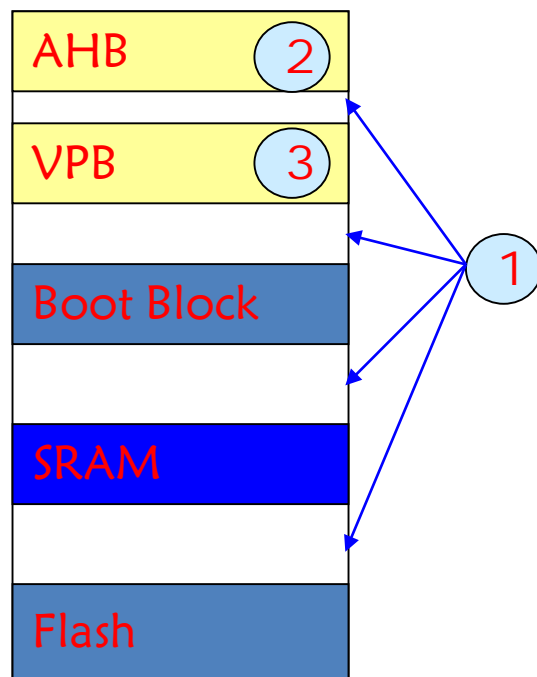
中止包含两种类型：

- 预取中止 发生在指令预取过程中
- 数据中止 发生在对数据访问时

执行下面的指令从中止返回：

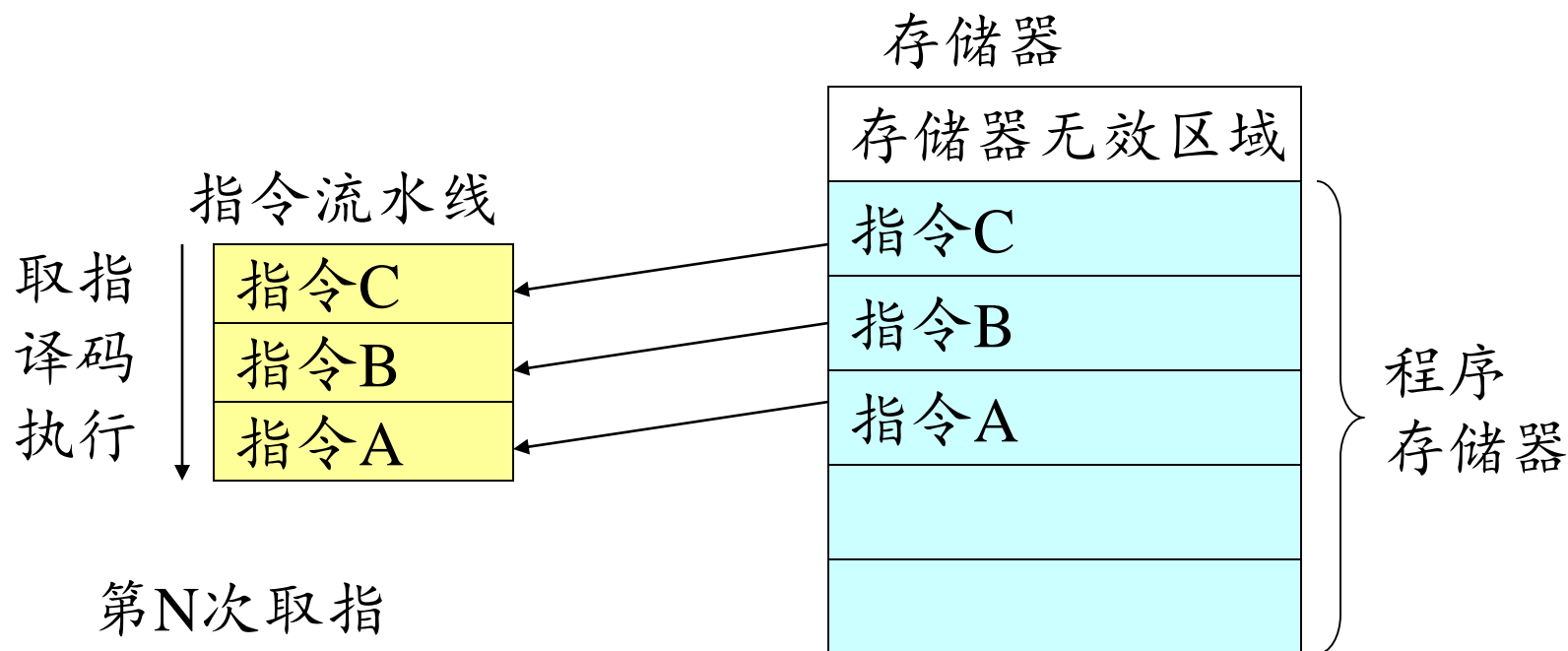
SUBS PC,R14_abt,#4; → 指令预取中止

SUBS PC,R14_abt,#8; → 数据预取中止

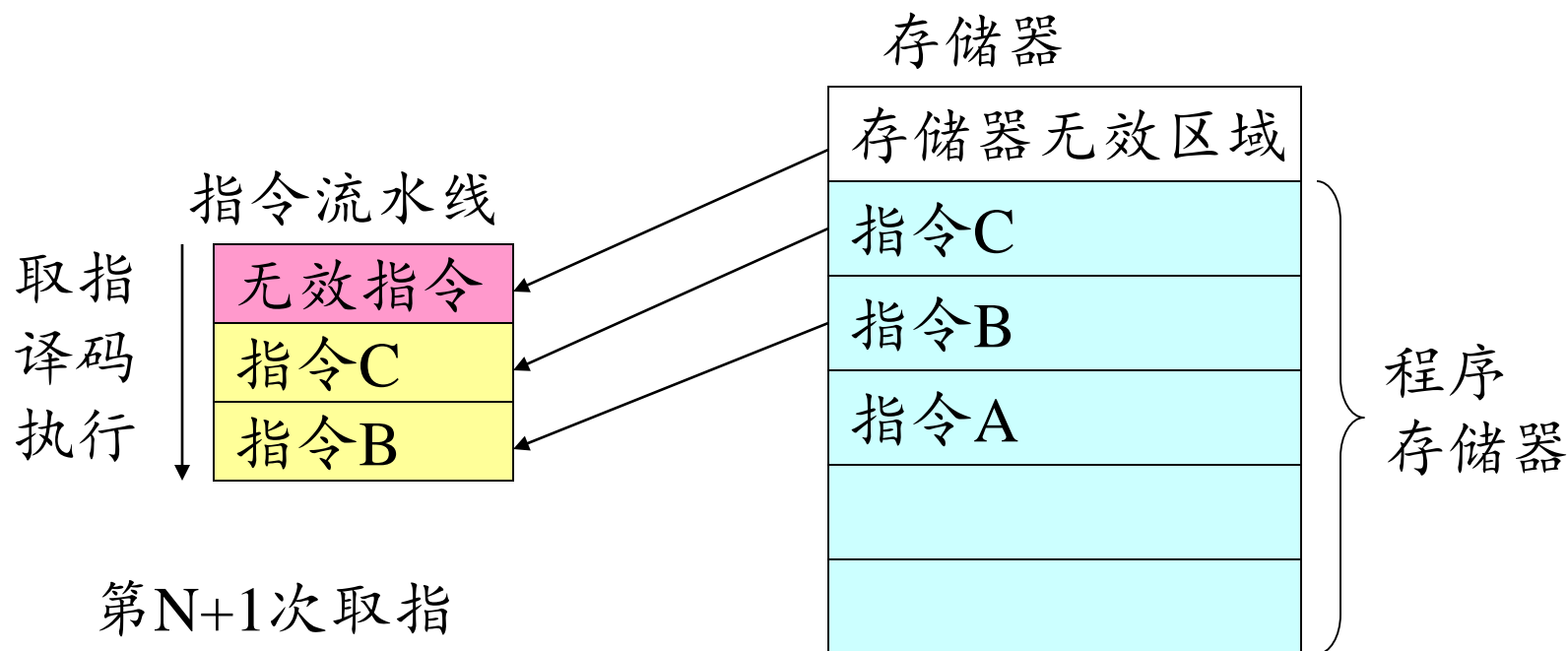


地址空间

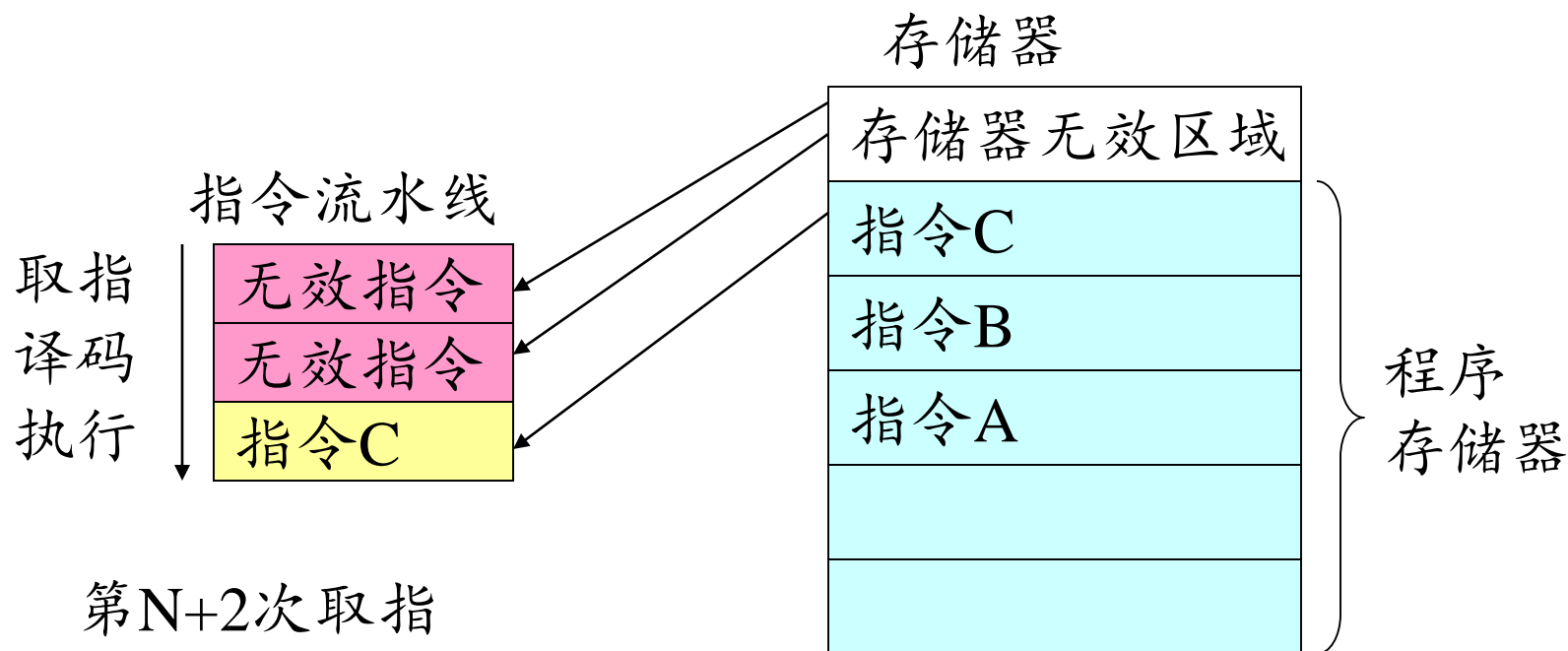
当发生预取中止时，ARM7TDMI内核将预取的指令标记为无效，但在指令到达流水线的执行阶段时才进入异常。如果指令在流水线中因为发生分支而没有被执行，中止将不会发生。



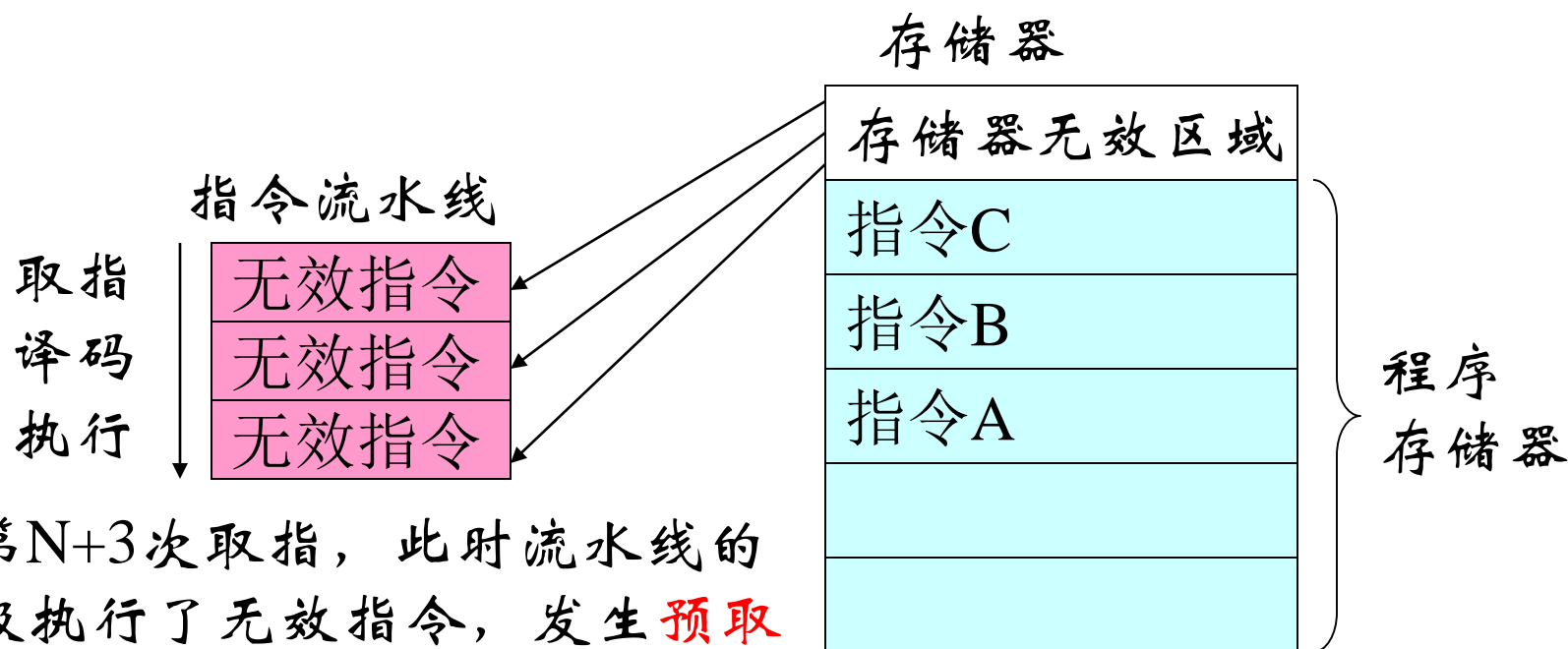
当发生预取中止时，ARM7TDMI内核将预取的指令标记为无效，但在指令到达流水线的执行阶段时才进入异常。如果指令在流水线中因为发生分支而没有被执行，中止将不会发生。



当发生预取中止时，ARM7TDMI内核将预取的指令标记为无效，但在指令到达流水线的执行阶段时才进入异常。如果指令在流水线中因为发生分支而没有被执行，中止将不会发生。



当发生预取中止时，ARM7TDMI内核将预取的指令标记为无效，但在指令到达流水线的执行阶段时才进入异常。如果指令在流水线中因为发生分支而没有被执行，中止将不会发生。



第N+3次取指，此时流水线的执行级执行了无效指令，发生**预取指中止**。如果指令C为跳转指令则可以避免预取指中止的发生。

1.5.3 软中断

使用软件中断(SWI)指令可以进入管理模式，通常用于请求一个特定的管理函数。SWI处理程序通过执行下面的指令返回：

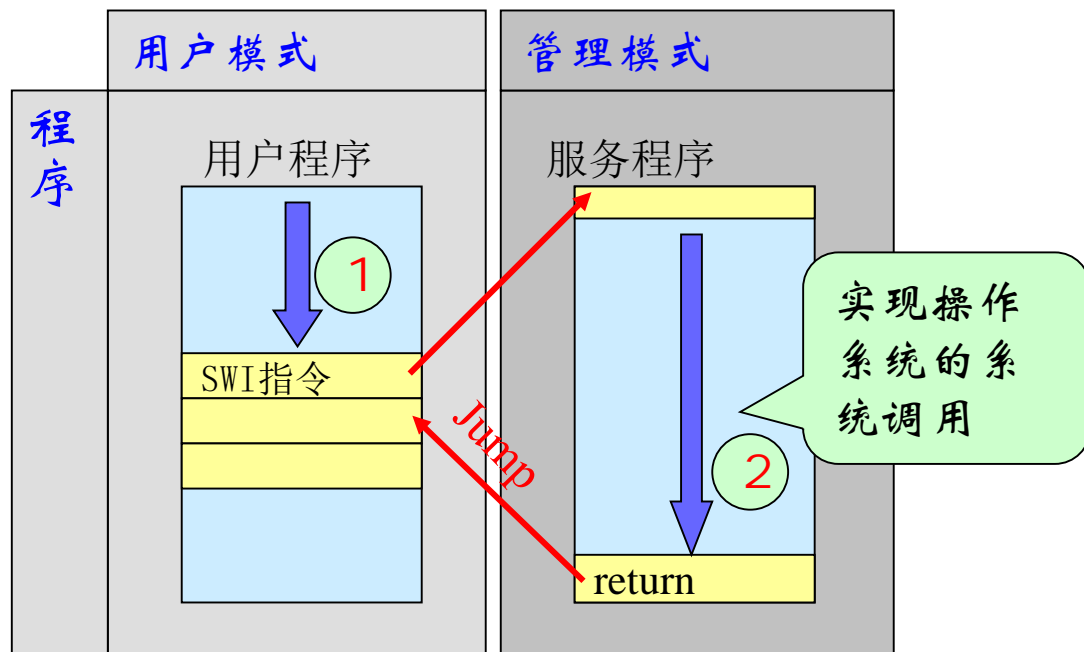
```
MOVS      PC,R14_svc
```

这个动作恢复了PC和CPSR并返回到SWI之后的指令。SWI处理程序读取操作码以提取SWI函数编号。

软中断指令

1. 执行用户程序中的软件中断指令，产生软件中断异常，通常用于系统调用；

2. 执行异常处理程序，并返回。



1.5.4 未定义的指令

当ARM7TDMI处理器遇到一条自己和系统内任何协处理器都无法处理的指令时，ARM7TDMI内核执行未定义指令陷阱。软件可使用这一机制通过模拟未定义的协处理器指令来扩展ARM指令集。

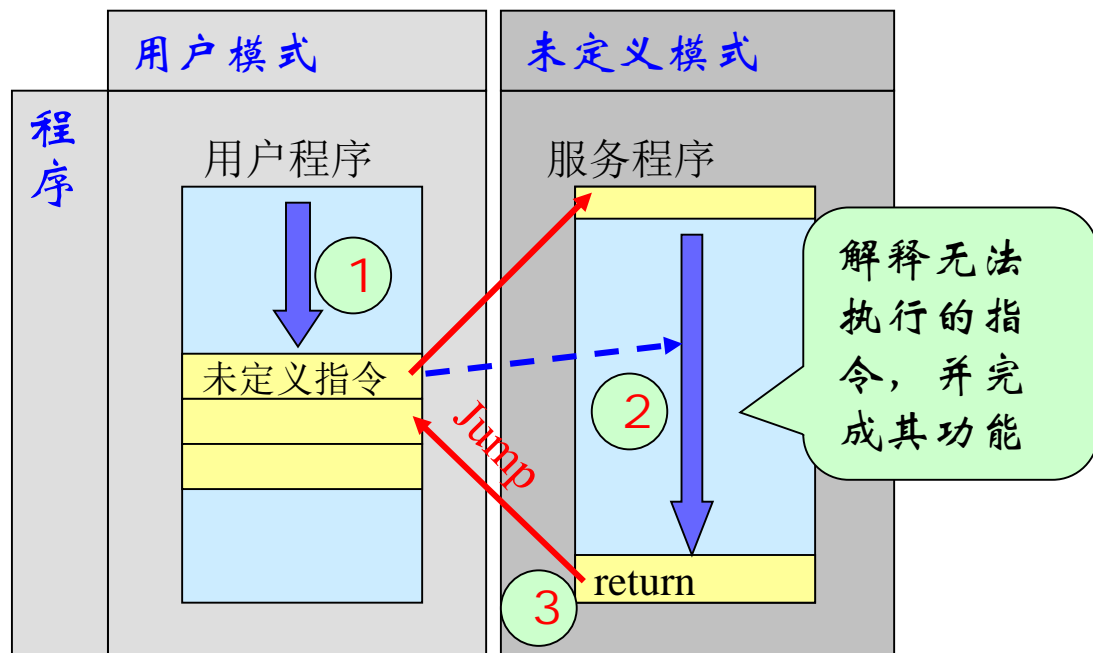
在模拟处理了未定义的指令后，陷阱程序执行下面的指令：

```
MOVS      PC,R14_und
```

这个动作恢复了PC和CPSR并返回到未定义指令之后的指令。

未定义的指令

1. 执行用户程序时遇到无法执行的指令，产生未定义指令异常；
2. 在服务程序中取出无法执行指令，完成其功能；
3. 异常处理程序执行结束后返回。



1.5.5 复位

- 当nRESET信号由低变为高电平时，ARM处理器执行下列操作：
 1. 强制CPSR中的M[4:0]变为b10011（管理模式）；
 2. 置位CPSR中的I和F位；
 3. 清零CPSR中的T位；
 4. 强制PC从地址0x00开始对下一条指令进行取指；
 5. 返回到ARM状态并恢复执行。

1.5.6 异常向量

地址	异常类型	进入时的模式	进入时I的状态	进入时F的状态
0x0000 0000	复位	管理	禁止	禁止
0x0000 0004	未定义指令	未定义	I	F
0x0000 0008	软件中断 (SWI)	管理	禁止	F
0x0000 000C	预取中止(指令)	中止	I	F
0x0000 0010	数据中止	中止	I	F
0x0000 0014	保留	保留	—	—
0x0000 0018	IRQ	外部中断	禁止	F
0x0000 001C	FIQ	快速中断	禁止	禁止

2.ARM 指令格式

2.1 ARM指令格式

ARM指令的基本格式如下：

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

其中<>号内的项是必须的， {}号内的项是可选的。各项的说明如下：

opcode：指令助记符；如ADD表示算术加操作指令

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，为寄存器；

operand2：第2个操作数；（常称为shifter_operand）

例如，指令 **ADDEQS** **R1, R2, #5**

条件域<cond>

- 几乎所有的ARM指令都可以根据当前程序状态寄存器CPSR中标志位的值，有条件地执行。ARM指令的条件域<cond>有16种类型。

cond	CPSR中标志位	含义
EQ 0000	Z置位	相等
NE 0001	Z清零	不相等
CS 0010	C置位	无符号数大于或等于
CC 0011	C清零	无符号数小于
MI 0100	N置位	负数
PL 0101	N清零	正数或零
VS 0110	V置位	溢出
VC 0111	V清零	未溢出
HI 1000	C置位Z清零	无符号数大于
LS 1001	C清零Z置位	无符号数小于或等于
GE 1010	N等于V	带符号数大于或等于
LT 1011	N不等于V	带符号数小于
GT 1100	Z清零且（N等于V）	带符号数大于
LE 1101	Z置位或（N不等于V）	带符号数小于或等于
AL 1110	忽略	无条件执行

1111

依版本不同，定义不同

2.2 第2个操作数

ARM指令的基本格式如下：

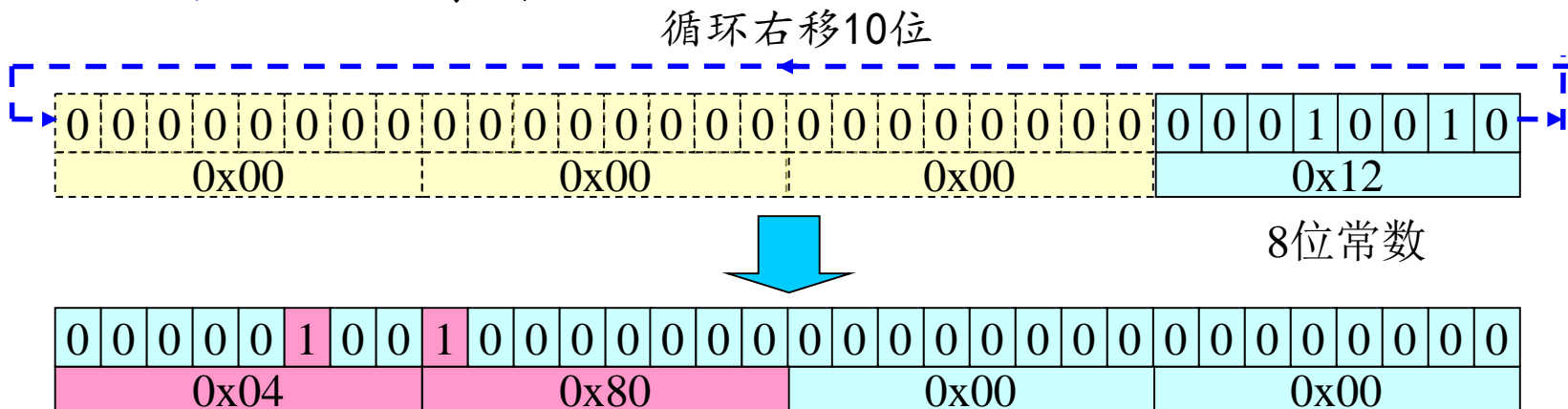
<code><opcode> {<cond>} {S} <Rd> , <Rn> [, <operand2>]</code>
--

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_8r——常数表达式；
- Rm——寄存器方式；
- Rm, shift——寄存器移位方式；

■ #immed_8r——常数表达式

该常数必须对应8位位图，即一个8位的常数通过循环右移偶数位得到。



例如：

ADD R1, R2, #0x0F

■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
```

■ 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

例如：

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

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

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。

2.3 ARM指令编码格式

各种ARM指令被变成32位固定长度：

例如，指令

ADDEQS R0,R1,R2;

该指令的编码格式为：

31~28	27~25	24~21	20	19~16	15~12	11~0
cond		opcode	S	Rn	Rd	op2
0000	001	0100	1	0001	0000	000000000010

(ARMV5参考手册)

[illegible]

3. ARM寻址方式

3.1 立即数寻址

3.2 寄存器寻址

3.3 寄存器移位寻址

3.4 寄存器间接寻址

3.5 基址变址寻址

3.6 相对寻址

3.7 多寄存器寻址

3.8 块拷贝寻址

3.9 堆栈寻址

3.10 简单的ARM程序

3.1 立即数寻址

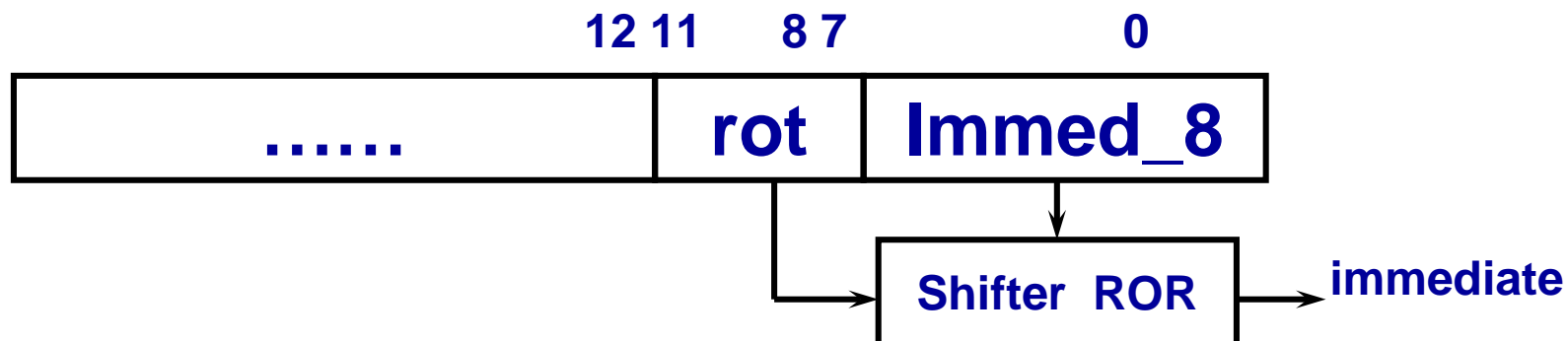
- 在立即数寻址中，操作数本身直接在指令中给出，取出指令也就获得了操作数，这个操作数也称为立即数。

- 例：

ADD	R0, R1, #5;	$R0 = R1 + 5$
MOV	R0, #0x55;	$R0 = 0x55$

- 其中：操作数5，0x55就是立即数，立即数在指令中要以“#”为前缀，后面跟实际数值。
 - 十六进制数，#后加0x或&，如#0x3f, #&3f.
 - 二进制数，#后加0b, 如#0b1011
 - 十进制数，#后加0d或缺省, 如#0d678, #789
- 如何构造32位立即数？

在指令格式中，第二个操作数有12位：



因此有效立即数immediate可以表示成：

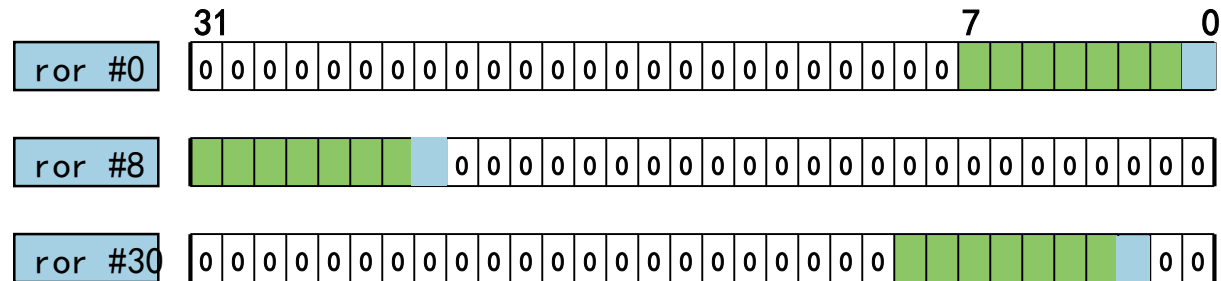
<immediate>=immed_8 循环右移 (2×rot)

4 bit 移位值 (0-15) 乘以2，得到一个范围在0-30，步长为 2的移位值。

因此，将ARM中的立即数称为**8位位图**。

记住一条准则：“**最后8位移动偶数位**”得到立即数。

例：



下列命令中，汇编器把立即数转换为移位操作：

```
MOV R0,#4096           ; uses 0x40 ror 26
ADD R1,R2,#0xFF0000    ; uses 0xFF ror 16
```


带有立即数的MOV 指令的二进制编码为：

```
MOV R0, #0xF200          ; E3A00CF2.    0xF200 =0xF2循环右移 (2*C)
MOV R1, #0x110000        ; E3A01811.    0x110000 =0x11循环右移 (2*8)
MOV R4, #0x12800         ; E3A04B4A.    0x12800 =0x4A循环右移 (2*B)
```

只有能够通过此构造方法得到的才是合法的立即数。

- 合法立即数：
 - 0xFF; 0x104 (其8位图为0x41) ; 0xFF0; 0xFF00
- 非法立即数：
 - 0x101; 0x102; 0xFF1

深入理解：一个合法的立即数可能有多种编码方法，将使某些指令的执行产生不同的结果。 如

0x3F0  $\left\{ \begin{array}{l} \text{immed_8}=0x3F, \text{ rot}=0xE, \text{ 对}3F\text{左移}4\text{位} \\ \text{immed_8}=0xFC, \text{ rot}=0xF, \text{ 对}FC\text{左移}2\text{位} \end{array} \right.$

ARM汇编编译器生成立即数的规则为：

- 当立即数数值在0到0xFF范围时，令immed_8=<immediate>, rot=0。
- 其它情况下，汇编编译器选择使rot数值最小的编码方式。

3.2 寄存器寻址

- 在寄存器寻址方式下，寄存器的值即为操作数。ARM指令普遍采用此种寻址方式。

• 例：

ADD R0, R1, R2; $R0 = R1 + R2$

MOV R0, R1 ; $R0 = R1$

3.3 寄存器移位寻址

- 寄存器移位寻址的操作数由寄存器的数值做相应移位而得到。
- 移位的方式在指令中以助记符的形式给出，而移位的位数可用立即数或寄存器寻址方式表示。
- 例：

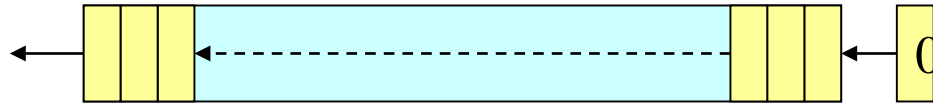
ADD R0, R1, R2, ROR #5; R0=R1+R2循环右移5位

MOV R0, R1, LSL R3; R0=R1逻辑左移R3位

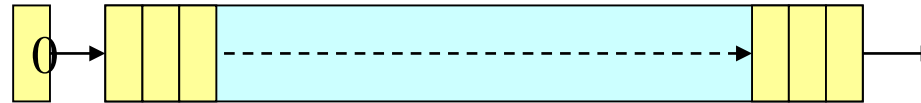
- 移位操作在ARM指令集中不作为单独的指令使用，ARM指令集共有5种位移操作。

ARM指令集的5种位移操作

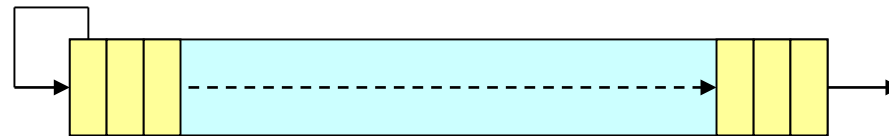
LSL逻辑左移 : $Rx, LSL \langle op1 \rangle$



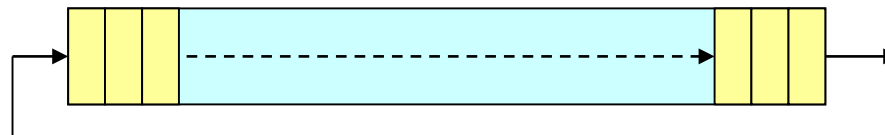
LSR逻辑右移 : $Rx, LSR \langle op1 \rangle$



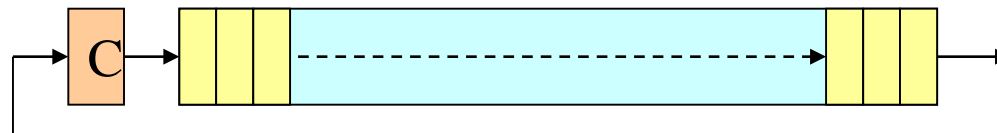
ASR算术右移 : $Rx, ASR \langle op1 \rangle$



ROR循环右移 : $Rx, ROR \langle op1 \rangle$



RRX带扩展的循环右移: Rx, RRX



3.4 寄存器间接寻址

- 寄存器中的值为操作数的物理地址, 而实际的操作数存放在存储器中。

- 例:

```
STR    R0, [R1]    ;    [R1]=R0
LDR    R0, [R1]    ;    R0=[R1]
```

3.5 基址变址寻址

- 将寄存器（称为基址寄存器）的值与指令中给出的偏移地址量相加，所得结果作为操作数的物理地址。

• 例：

LDR R0, [R1, #5];	R0=[R1+5]
LDR R0, [R1,R2];	R0=[R1+R2]

3.6 相对寻址

- 相对寻址同基址变址寻址相似，区别只是将程序计数器PC作为基址寄存器，指令中的标记作为地址偏移量。
- 作业：请分析相对寻址与基址变址寻址的区别

• 例：

```
BEQ  process1
```

```
.....
```

```
process1
```

```
.....
```

3.7 多寄存器寻址

- 在多寄存器寻址方式中，一条指令可实现一组寄存器值的传送。
- 连续的寄存器间用“—”连接，否则用“，”分隔。

- 例：

```
LDMIA  R0, {R1-R5} ; R1=[R0]  
                        ; R2=[R0+4]  
                        ; R3=[R0+8]  
                        ; R4=[R0+12]  
                        ; R5=[R0+16]
```

- 指令中IA表示在执行完一次Load操作后，R0自增4。该指令将以R0为起始地址的5个字节数据分别装入R1，R2，R3，R4，R5中。

3.8 块拷贝寻址

- 块拷贝寻址可实现连续地址数据从存储器的某一位置拷贝到另一位置。

- 例：

LDMIA R0, {R1-R5};

STMIA R1, {R1-R5};

- 第一条指令从以R0的值为起始地址的存储单元中取出5个字的数据，第二条指令将取出的数据存入以R1的值为起始地址的存储单元中。
- 实际上是多寄存器寻址的组合。

3.9 堆栈寻址

- 堆栈寻址用于数据栈与寄存器组之间批量数据传输。
- 当数据写入和读出内存的顺序不同时，使用堆栈寻址可以很好的解决这问题。

- 例：

```
STMFD    R13!, {R0,R1,R2,R3,R4};  
LDMFD    R13!, {R0,R1,R2,R3,R4}
```

- 第一条指令，将R0—R4中的数据压入堆栈，R13为堆栈指针；
- 第二条指令，将数据出栈，恢复R0—R4原先的值。

3.10 简单的ARM程序-1

;文件名: **TEST1.S** (所有标号顶格写,而指令和伪指令**不能**顶格写?)

;功能: 使用“;”进行注释

;说明: 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
        BL     ADD_SUB ;调用子程序
LOOP   ADD_SUB ;跳转到LOOP
ADD_SUB
        ADDS   R0, R0, R1 ;R0 = R0 + R1
        MOV    ;子程序返回
        END    ;文件结束
```

标号顶格写

实际代码段

简单的ARM程序-2

;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
```

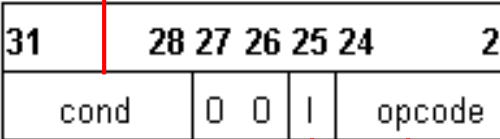
4. ARM指令集

- 4.1 基本数据处理指令（16种主要助记符）
- 4.2 存储器访问指令（4种主要助记符）
- 4.2 跳转指令（4种主要助记符）
- 4.3 Load/Store指令（16种主要助记符）
- 4.4 程序状态寄存器指令（2种主要助记符）
- 4.5 协处理器指令（5种主要助记符）
- 4.6 软件中断指令（2种主要助记符）

4.1 数据处理指令

数据处理指令编码

指令执行的条件码



1用于区别立即数（1为1）和寄存器移位（1为0）

opcode数据处理指令操作码

opcode操作码功能表

操作码	指令助记符	说明
0000	AND	逻辑与操作指令
0001	EOR	逻辑异或操作指令
0010	SUB	减法运算指令
0011	RSB	逆向减法指令
0100	ADD	加法运算指令
0101	ADC	带进位加法
0110	SBC	带进位减法指令
0111	RSC	带进位逆向减法指令
1000	TST	位测试指令
1001	TEQ	相等测试指令
1010	CMP	比较指令
1011	CMN	负数比较指令
1100	ORR	逻辑或操作指令
1101	MOV	数据传送
1110	BIC	位清除指令
1111	MVN	数据非传送

第一操作数

0

寄存器

指令中的

4.1.1 数据处理指令1

1. MOV 数据传送指令

- 格式: $\text{MOV}\{<\text{cond}>\}\{S\} \quad <\text{Rd}>, <\text{op1}>;$
- 功能: $\text{Rd} = \text{op1}$

op1可以是寄存器、被移位的寄存器或立即数。

- 例：

```
MOV    R0 , # 5           ; R0=5
```

```
MOV    R0,R1           ;R0=R1
```

MOV R0,R1,LSL#5 ;R0=R1左移5位

4.1.2 数据处理指令2

2. MVN 数据取反传送指令

- 格式: $\text{MVN}\{<\text{cond}>\}\{S\} \quad <\text{Rd}>, <\text{op1}>;$
- 功能: 将op1表示的值传送到目的寄存器Rd中, 但该值在传送前被按位取反, 即 $\text{Rd} = \neg \text{op1}$;

op1可以是寄存器、被移位的寄存器或立即数。

- 例:

`MVN R0, #0 ; R0=-1`

4.1.3 数据处理指令3

3. ADD 加法指令

- 格式: $\text{ADD}\{<\text{cond}>\}\{S\} \quad <\text{Rd}>, <\text{Rn}>, <\text{op2}>;$
- 功能: $\text{Rd} = \text{Rn} + \text{op2}$

op2可以是寄存器，被移位的寄存器或立即数。

- 例:

ADD R0, R1, #5 ; R0=R1+5

ADD R0, R1, R2 ; R0=R1+R2

ADD R0, R1, R2, LSL #5 ; R0=R1+R2左移5位

4.1.4 数据处理指令4

4. ADC 带进位加法指令

- 格式: $\text{ADC}\{\text{<cond>}\}\{\text{S}\} \quad \text{<Rd>}, \text{<Rn>}, \text{<op2>};$
- 功能: $\text{Rd} = \text{Rn} + \text{op2} + \text{carry}$

op2可以是寄存器、被移位的寄存器或立即数；carry为进位标志值。该指令用于实现超过32位的数的加法。

- 例: (64位加法的实现)

第一个64位操作数存放在寄存器R2, R3中;

第二个64位操作数存放在寄存器R4, R5中;

64位结果存放在R0, R1中。

```
ADDS    R0, R2, R4;    低32位相加,  
                        ;    S表示结果影响条件标志位的值  
ADC     R1, R3, R5;    高32位相加
```


4.1.5 数据处理指令5

5. SUB 减法指令

- 格式: $\text{SUB}\{\langle\text{cond}\rangle\}\{\text{S}\}\ \langle\text{Rd}\rangle, \langle\text{Rn}\rangle, \langle\text{op2}\rangle;$
- 功能: $\text{Rd} = \text{Rn} - \text{op2}$

op2可以是寄存器、被移位的寄存器或立即数。

- 例:

`SUB R0, R1, # 5 ; R0=R1-5`

`SUB R0, R1, R2 ; R0=R1-R2`

`SUB R0, R1, R2, LSL # 5 ; R0=R1-R2左移5位`

4.1.6 数据处理指令6

6. RSB 反向减法指令

- 格式: $\text{RSB}\{<\text{cond}>\}\{S\} \quad <\text{Rd}>, <\text{Rn}>, <\text{op2}>;$
- 功能: 同SUB指令, 但倒换了两操作数的前后位置, 即 $\text{Rd} = \text{op2} - \text{Rn}$ 。

- 例:

RSB R0, R1, #5 ; R0=5-R1

RSB R0, R1, R2 ; R0=R2-R1

RSB R0, R1, R2, LSL #5 ; R0=R2左移5位-R1

4.1.7 数据处理指令7

7. SBC 带借位减法指令

- 格式：SBC{<cond>} {S} <Rd>, <Rn>, <op2>;

- 功能：Rd=Rn-op2-!carry

op2可以是寄存器、被移位的寄存器或立即数。

SUB和SBC生成进位标志的方式不同于常规，如果需要借位则清除进位标志，所以指令要对进位标志进行一个非操作。

- 例（64位减法的实现）：

第一个64位操作数存放在寄存器R2，R3中；

第二个64位操作数存放在寄存器R4，R5中；

64位结果存放在R0，R1中。

SUBS	R0, R2, R4;	低32位相减,
		S表示结果影响条件标志位的值
SBC	R1, R3, R5;	高32位相减

4.1.8 数据处理指令8

8. RSC 带借位的反向减法指令

- 格式: $\text{RSC}\{\text{<cond>}\}\{\text{S}\} \quad \text{<Rd>, <Rn>, <op2>;}$
- 功能: 同SBC指令, 但倒换了两操作数的前后位置, 即 $\text{Rd} = \text{op2} - \text{Rn} - !\text{carry}$ 。
- 例:

前提条件与SBC例子相同, 操作数1-操作数2的实现语句需改为:

SUBS	R0, R2, R4;	低32位相减,
		S表示结果影响寄存器CPSR的值
RSC	R1, R5, R3;	高32位相减

4.1.9 数据处理指令9

9. AND 逻辑与指令

- 格式: $\text{AND}\{\langle\text{cond}\rangle\}\{\text{S}\} \quad \langle\text{Rd}\rangle, \langle\text{Rn}\rangle, \langle\text{op2}\rangle;$
- 功能: $\text{Rd} = \text{Rn} \text{ AND } \text{op2}$

op2可以是寄存器，被移位的寄存器或立即数。一般用于清除Rn的特定几位。

- 例:

AND R0, R0, #5;

保持R0的第0位和第2位，其余位清0

4.1.10 数据处理指令10

10. ORR 逻辑或指令

- 格式: $\text{ORR} \{<\text{cond}>\} \{S\} \quad <Rd>, <Rn>, <op2>;$
- 功能: $Rd = Rn \text{ OR } op2$

$op2$ 可以是寄存器、被移位的寄存器或立即数。一般用于设置 Rn 的特定几位。

- 例:

$\text{ORR} \quad R0, R0, \#5;$ $R0$ 的第0位和第2位设置为1, 其余位不变

4.1.11 数据处理指令11

11. EOR 逻辑异或指令

- 格式: $\text{EOR}\{\langle\text{cond}\rangle\}\{S\} \quad \langle\text{Rd}\rangle, \langle\text{Rn}\rangle, \langle\text{op2}\rangle;$
- 功能: $\text{Rd} = \text{Rn} \text{ EOR } \text{op2}$

op2可以是寄存器、被移位的寄存器或立即数。一般用于将Rn的特定几位取反。

- 例:

$\text{EOR} \quad \text{R0}, \text{R0}, \#5;$ R0的第0位和第2位取反, 其余位不变

4.1.12 数据处理指令12

12. BIC 位清除指令

- 格式: $\text{BIC}\{\text{<cond>}\}\{\text{S}\} \quad \text{<Rd>}, \text{<Rn>}, \text{<op2>};$
- 功能: $\text{Rd} = \text{Rn} \text{ AND } (!\text{op2})$

用于清除寄存器Rn中的某些位，并把结果存放到目的寄存器Rd中。

操作数op2是一个32位掩码（mask），如果在掩码中设置了某一位，则清除Rn中的这一位；未设置的掩码位指示Rn中此位保持不变。其中，op2可以是寄存器、被移位的寄存器或立即数。

- 例:

`BIC R0, R0, #5;`

R0中第0位和第2位清0，其余位不变

4.1.13 数据处理指令13

13. CMP 比较指令

- 格式: $\text{CMP}\{\text{<cond>}\} \quad \text{<Rn>}, \text{<op1>};$
- 功能: $\text{Rn} - \text{op1}$

该指令进行一次减法运算，但不存储结果，根据结果更新CPSR中条件标志位的值。

该指令不需要显式地指定S后缀来更改状态标志。其中，操作数op1为寄存器或立即数。

- 例:

`CMP R0, #5;`

`ADDGT R0, R0, #5;`

计算 $\text{R0} - 5$ ，根据结果设置条件标志位
如果 $\text{R0} > 5$ ，则执行ADDGT指令

4.1.14 数据处理指令14

14. CMN 反值比较指令

- 格式: $\text{CMN}\{\text{<cond>}\} \quad \text{<Rn>}, \text{<op1>;}$
- 功能: 同CMP指令, 但寄存器Rn的值是和op1取负的值进行比较。
- 例:
`CMN R0, #5` ; 把R0与-5进行比较

4.1.15 数据处理指令15

15. TST 位测试指令

- 格式: $\text{TST}\{<\text{cond}>\} \quad <\text{Rn}>, <\text{op1}>;$
- 功能: $\text{Rn} \text{ AND } \text{op1}$

根据结果更新CPSR中条件标志位的值，但不存储结果。
用于检查寄存器Rn是否设置了op1中相应的位。

- 例:

`TST R0, #5;` 测试R0中第0位和第2位是否为1

4.1.16 数据处理指令16

16. TEQ 相等测试指令

- 格式: $\text{TEQ}\{\text{<cond>}\} \quad \text{<Rn>}, \text{<op1>};$
- 功能: Rn EOR op1

将寄存器Rn的值和操作数op1所表示的值按位作逻辑异或操作，根据结果更新CPSR中条件标志位的值，但不存储结果。

用于检查寄存器Rn的值是否和op1所表示的值相等。

- 例:

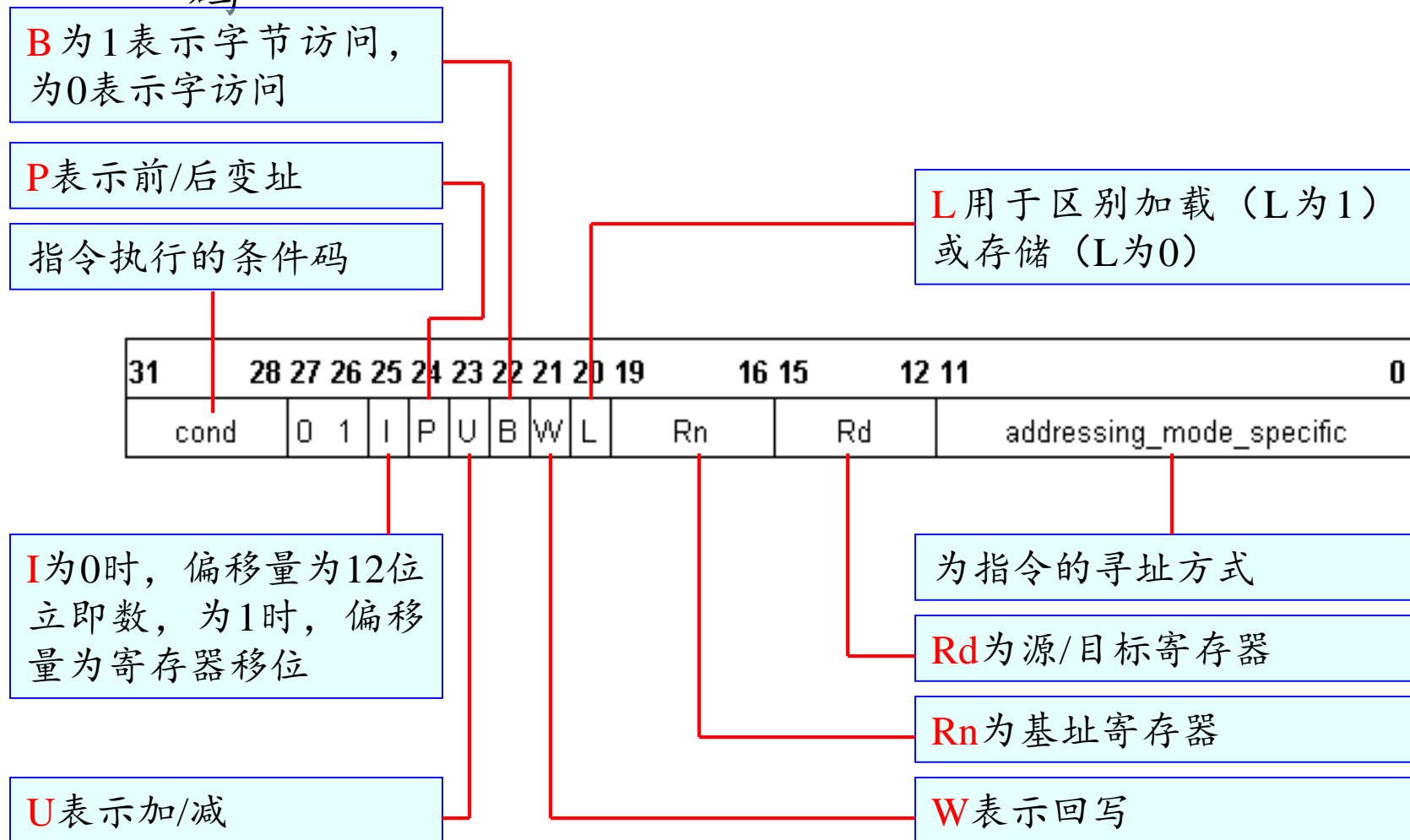
$\text{TEQ} \quad \text{R0}, \#5 \quad ; \text{判断R0的值是否和5相等}$

4.2 存储器访问指令

- 用于寄存器和内存间数据的传送。
 - Load用于把内存中的数据装载到寄存器中。
 - Store用于把寄存器中的数据存入内存。
- 该集合的指令使用频繁，在指令集中**最为重要**，因为**其他指令只能操作寄存器**，当数据存放在内存中时，必须先把数据从内存装载到寄存器，执行完后再把寄存器中的数据存储到内存中。
- 存储器访问指令分为**3类**：
 - (1) 单一数据传送指令（LDR和STR等）
 - (2) 多数据传送指令（LDM和STM）
 - (3) 数据交换指令（SWP和SWPB）

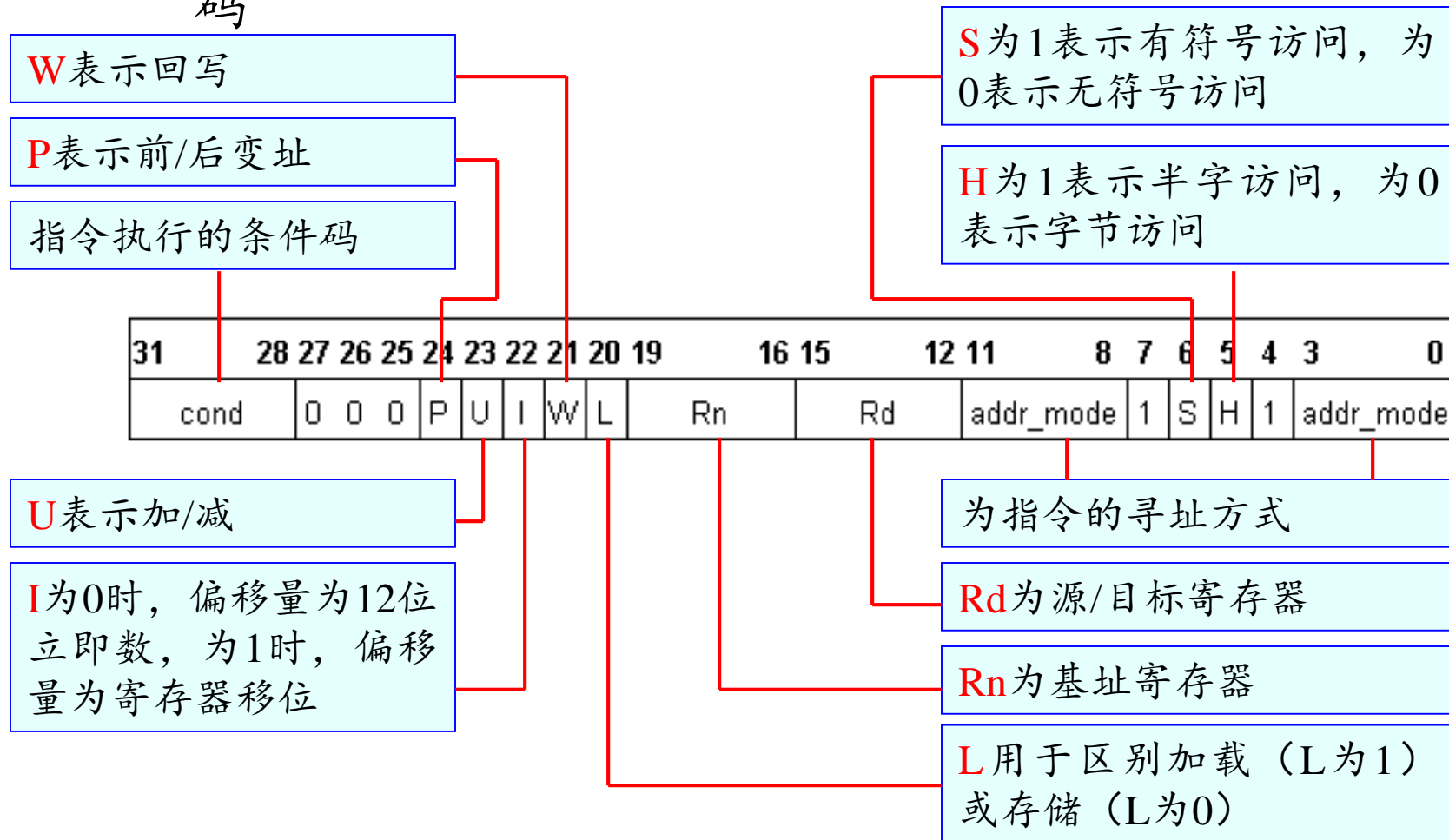
单寄存器存取指令编码

•LDR和STR——字和无符号字节加载/存储指令编码



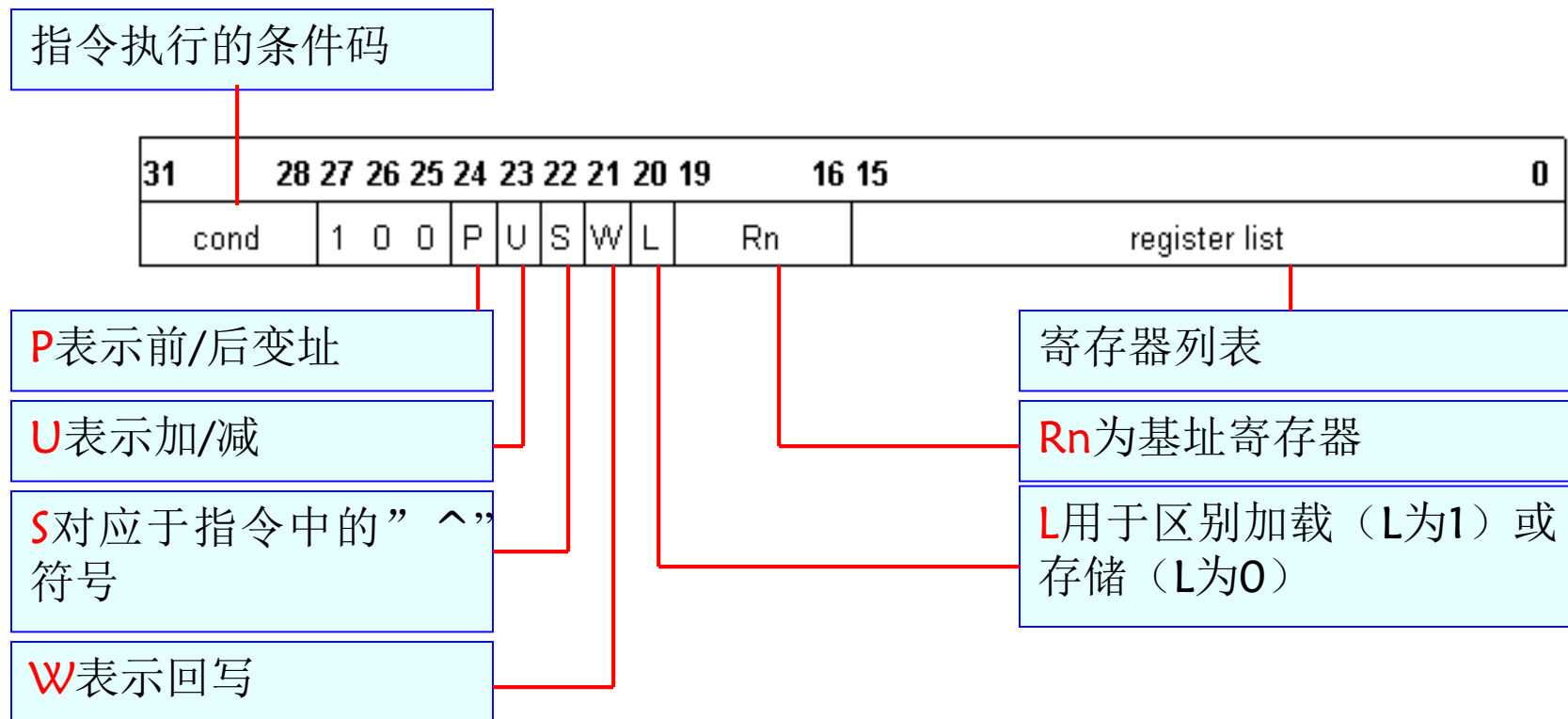
单寄存器存取指令编码

- LDR和STR——半字和有符号字节加载/存储指令编码



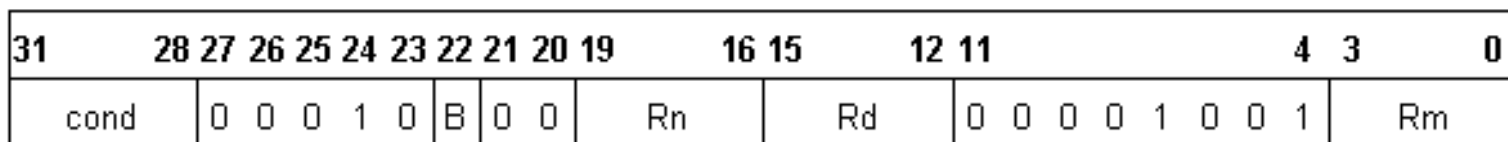
多寄存器存取指令编码

- LDM和STM——多寄存器加载/存储指令编码



单寄存器和存储器交换指令编码

- SWP和SWPB——寄存器和存储器交换指令编码



指令执行的条件码

B用于区别无符号字节
(**B**为1) 或字 (**B**为0)

Rm源寄存器

Rd目标寄存器

Rn为基址寄存器

- SWP指令应用示例:

SWP R1, R1, [R0]

SWPB R1, R2, [R0]

;将R1的内容与R0指向的存储单元的内容进行交换
;将R0指向的存储单元内的容读取一字节数据到R1中
;(高24位清零), 并将R2的内容写入到该内存单元中
;(最低字节有效)

4.2.1 存储器访问指令1-1

1. LDR 字数据加载指令

- 格式: $\text{LDR}\{\text{<cond>}\} \text{ <Rd>}, \text{<addr>};$
- 功能: 把addr所表示的内存地址中的字数据装载到目标寄存器Rd中, 同时还可以把合成的有效地址写回到基址寄存器。

地址addr可以是一个简单的值、一个偏移量, 或者是一个被移位的偏移量。

寻址方式: (Rn: 基址寄存器, Rm: 变址寄存器, Index: 偏移量, 12位的无符号数)

(1) $\text{LDR Rd}, [\text{Rn}]$

; 把内存中地址为Rn的字数据装入寄存器Rd中

(2) $\text{LDR Rd}, [\text{Rn}, \text{Rm}]$

; 将内存中地址为 $\text{Rn} + \text{Rm}$ 的字数据装入寄存器Rd中

4.2.1 存储器访问指令1-2

(3) LDR Rd, [Rn, # index]

;将内存中地址为 $Rn + index$ 的字数据装入Rd中

(4) LDR Rd, [Rn, Rm, LSL # 5]

;将内存中地址为 $Rn + Rm \times 32$ 的字数据装入Rd

(5) LDR Rd, [Rn, Rm] !

;将内存中地址为 $Rn + Rm$ 的字数据装入Rd, 并将新地址 $Rn + Rm$ 写入Rn

(6) LDR Rd, [Rn, # index] !

;将内存中地址为 $Rn + index$ 的字数据装入Rd, 并将新地址 $Rn + index$ 写入Rn

(7) LDR Rd, [Rn, Rm, LSL # 5] !

;将内存中地址为 $Rn + Rm \times 32$ 的字数据装入Rd, 并将新地址 $Rn + Rm \times 32$ 写入Rn

4.2.1 存储器访问指令1-3

LDR Rd, [Rn], Rm

; 将内存中地址为Rn的字数据装入寄存器Rd, 并将新地址 $Rn+Rm$ 写入Rn

LDR Rd, [Rn], # index

; 将内存中地址为Rn的字数据装入寄存器Rd, 并将新地址 $Rn+index$ 写入Rn

LDR Rd, [Rn], Rm, LSL # 5

; 将内存中地址为Rn的字数据装入寄存器Rd, 并将新地址 $Rn+Rm \times 32$ 写入Rn

例如:

LDR R0, [R1, R2, LSL # 5]!

; 将内存中地址为 $R1+R2 \times 32$ 的字数据装入寄存器R0, 并将新地址 $R1+R2 \times 32$ 写入R1

4.2.1 存储器访问指令1-4*

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

指令操作的伪代码:

```
if ConditionPassed(cond) then
if adderss[1:0] == 0b00 then
    Value = Memory[adderss,4]
else if adderss[1:0] == 0b01 then
    Value = Memory[adderss,4] Rotate_Right 8
else if adderss[1:0] == 0b10 then
    Value = Memory[adderss,4] Rotate_Right 16
else if adderss[1:0] == 0b11 then
    Value = Memory[adderss,4] Rotate_Right 24
if (Rd is R15) then
    if (architecture version 5 or above) then
        PC = value AND 0xFFFFFFFFE
        T Bit = value[0]
    else
        PC = value AND 0xFFFFFFF0
else
    Rd = value
```

4.2.2 存储器访问指令2

2. LDRB 字节数据加载指令

- 格式: `LDR{<cond>}B <Rd>, <addr>;`
- 功能: 同LDR指令, 但该指令只是从内存读取一个8位的字节数据而不是一个32位的字数据, 并将Rd的高24位清0。

- 例:

`LDRB R0, [R1]`

;将内存中起始地址为R1的一个字节数据装入R0中

4.2.3 存储器访问指令3

3. LDRBT 用户模式的字节数据加载指令

- 格式: $\text{LDR}\{\text{<cond>}\}\text{BT} \quad \text{<Rd>}, \text{<addr>;}$
- 功能: 同LDRB指令, 但无论处理器处于何种模式, 都将该指令当作一般用户模式下的内存操作。

4.2.4 存储器访问指令4

4. LDRH 半字数据加载指令

- 格式: $\text{LDR}\{\text{<cond>}\}\text{H} \quad \text{<Rd>}, \text{<addr>;}$
- 功能: 同LDR指令, 但该指令只是从内存读取一个16位的半字数据而不是一个32位的字数据, 并将Rd的高16位清0。

- 例:

LDRH R0, [R1]

;将内存中起始地址为R1的一个半字数据装入R0中

4.2.5 存储器访问指令5

5. LDRSB 有符号的字节数据加载指令

- **格式:** `LDR{<cond>}SB <Rd>, <addr>;`
- **功能:** 同LDRB指令, 但该指令将寄存器Rd的高24位设置成所装载的字节数据符号位的值。

- **例:**

`LDRSB R0, [R1]`

;将内存中起始地址为R1的一个字节数据装入R0中, R0的高24位设置成该字节数据的符号位

4.2.6 存储器访问指令6

6. LDRSH 有符号的半字数据加载指令

- 格式: $\text{LDR}\{\text{<cond>}\}\text{SH } \text{<Rd>}, \text{<addr>;}$
- 功能: 同LDRH指令, 但该指令将寄存器Rd的高16位设置成所装载的半字数据符号位的值。

- 例如:

`LDRSH R0, [R1]`

;将内存中起始地址为R1的一个16位半字数据装入R0中, R0的高16位设置成该半字数据的符号位

4.2.7 存储器访问指令7

7. LDRT 用户模式的字数据加载指令

- **格式：** $\text{LDR}\{\text{<cond>}\}\text{T} \quad \text{<Rd>}, \text{<addr>;}$
- **功能：** 同LDR指令，但无论处理器处于何种模式，都将该指令当作一般用户模式下的内存操作。
- **addr**所表示的有效地址必须是字对齐的，否则从内存中读出的数值需进行循环右移操作。

4.2.8 存储器访问指令8

8. STR 字数据存储指令

- **格式:** $\text{STR}\{<\text{cond}>\} \quad <\text{Rd}>, <\text{addr}>;$
- **功能:** 把寄存器Rd中的字数据（32位）保存到addr所表示的内存地址中，同时还可以把合成的有效地址写回到基址寄存器。

地址addr可以是一个简单的值、一个偏移量，或者是一个被移位的偏移量。

寻址方式同LDR指令。

- **例:**

$\text{STR} \quad \text{R0}, [\text{R1}, \#5]!$

;把R0中的字数据保存到以R1+5为地址的内存中，然后 $\text{R1}=\text{R1}+5$

4.2.9 存储器访问指令9

9. STRB 字节数据存储指令

- **格式：**STR {<cond>} B <Rd>, <addr>;
- **功能：**把寄存器Rd中的低8位字节数据保存到addr所表示的内存地址中。

其他用法同STR指令。

- **例：**

STRB R0, [R1]

;将寄存器R0中的低8位数据存入R1表示的内存地址中

4.2.10 存储器访问指令10

10. STRBT 用户模式的字节数据存储指令

- 格式：STR{<cond>}BT <Rd>, <addr>;
- 功能：同STRB指令，但无论处理器处于何种模式，该指令都将被当作一般用户模式下的内存操作。

4.2.11 存储器访问指令11

11. STRH 半字数据储存指令

- **格式:** STR{<cond>}H <Rd>, <addr>;
- **功能:** 把寄存器Rd中的低16位半字数据保存到addr所表示的内存地址中, 而且addr所表示的地址必须是半字对齐的。

其他用法同STR指令。

- **例:**

STRH R0, [R1]

;将寄存器R0中的低16位数据存入R1表示的内存地址中

4.2.12 存储器访问指令12

12. STRT 用户模式的字数据存储指令

- 格式: $\text{STR}\{\text{<cond>}\}\text{T} \quad \text{<Rd>}, \text{<addr>;}$
- 功能: 同STR指令, 但无论处理器处于何种模式, 该指令都将被当作一般用户模式下的内存操作。

4.2.13 存储器访问指令13-1

13. LDM 批量数据加载指令

- 格式:

LDM{<cond>} {<type>} <Rn>{!}, <regs>{^};

- 功能: 从一片连续的内存单元读取数据到各个寄存器中, 内存单元的起始地址为基址寄存器Rn的值, 各个寄存器由寄存器列表regs表示。

该指令一般用于多个寄存器数据的出栈。

type字段种类:

IA: 每次传送后地址加1。

IB: 每次传送前地址加1。

DA: 每次传送后地址减1。

DB: 每次传送前地址减1。

FD: 满递减堆栈。

ED: 空递减堆栈。

FA: 满递增堆栈。

EA: 空递增堆栈。

4.2.13 存储器访问指令13-2

- 例：

LDMIA/IB/DA/DB R13!, {R0-R1, R3};

各指令执行完后，结果如图所示。

0X123456a0	36338832
0X12345690	14543862
0X1234568c	15548545
0X12345688	54693645
0X12345684	66663333
0X12345680	00008888
0X1234567c	59595959
0X12345678	26262626

内存中的数据

R13
0X12345684

15548545	R3
54693645	R1
66663333	R0

IA

14543862	R3
15548545	R1
54693645	R0

IB

59595959	R3
00008888	R1
66663333	R0

DA

26262626	R3
59595959	R1
00008888	R0

DB

4.3.13 Load/Store指令13-3

- FD、ED、FA和EA指定是满栈还是空栈，是升序栈还是降序栈，用于堆栈寻址。
- 一个满栈的栈指针指向上次写的最后一个数据单元。
- 空栈的栈指针指向第一个空闲单元。
- 一个降序栈是在内存中反向增长而升序栈在内存中正向增长。

0X123456a0	36338832
0X12345690	14543862
0X1234568c	15548545
0X12345688	54693645
0X12345684	66663333
0X12345680	
0X1234567c	
0X12345678	

栈指针

FD

0X123456a0	
0X12345690	
0X1234568c	
0X12345688	54693645
0X12345684	66663333
0X12345680	00008888
0X1234567c	59595959
0X12345678	26262626

栈指针

FA

0X123456a0	36338832
0X12345690	14543862
0X1234568c	15548545
0X12345688	54693645
0X12345684	66663333
0X12345680	
0X1234567c	
0X12345678	

栈指针

ED

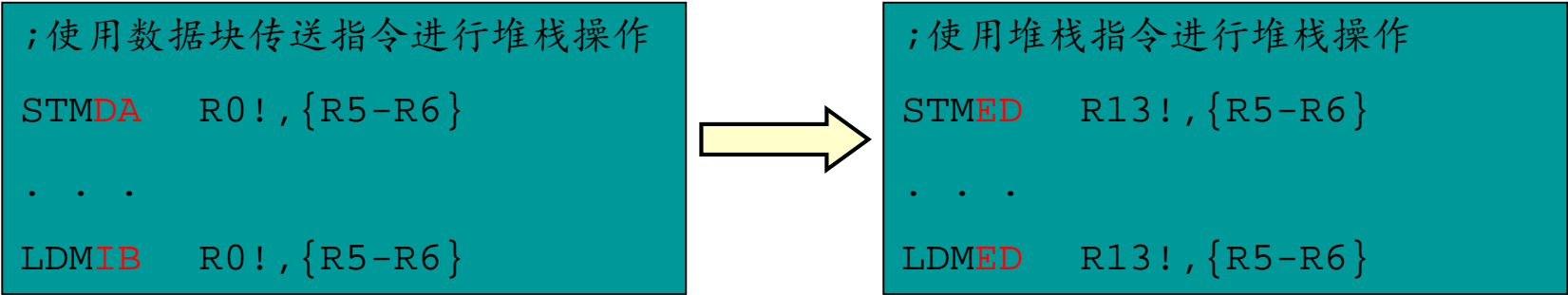
0X123456a0	
0X12345690	
0X1234568c	
0X12345688	54693645
0X12345684	66663333
0X12345680	00008888
0X1234567c	59595959
0X12345678	26262626

栈指针

EA

堆栈操作和数据块传送指令类似，也有4种模式，它们之间的关系如下表所示：

数据块传送 存储	堆栈操作 压栈	说明	数据块传送 加载	堆栈操作 出栈	说明
STMDA	STMED	空递减	LDMDA	LDMFA	满递增
STMIA	STMEA	空递增	LDMIA	LDMFD	满递减
STMDB	STMFD	满递减	LDMDB	LDMEA	空递增
STMIB	STMFA	满递增	LDMIB	LDMED	空递减



两段代码的执行结果是一样的，但是使用堆栈指令的压栈和出栈操作编程很简单（只要前后一致即可），而使用数据块指令进行压栈和出栈操作则需要考虑空与满、加与减对应的问题。

4.2.13 存储器访问指令13-4

- 格式:

LDM{<cond>} {<type>} <Rn>{!}, <regs>{^};

- {!}: 若选用了此后缀, 则当指令执行完毕后, 将最后的地址写入基址寄存器。
- {^}: 当regs中不包含PC时, 该后缀用于指示指令所用的寄存器为用户模式下的寄存器, 否则指示指令执行时, 将寄存器SPSR的值复制到CPSR中。

4.2.13 存储器访问指令13-5*

LDM {<cond>} <addressing_mode> <Rd>{!},
<registers>

指令操作的伪代码:

```
if ConditionPassed(cond) then
    adderss = start_address
    for i = 0 to 14
        if registers _list[i] == 1 then
            Ri = Memory[adderss, 4]
            adderss = address + 4
    if registers _list[15] == 1 then
        value = Memory[adderss, 4]
        if (architecture version 5 or above) then
            PC = value AND 0xFFFFFFFFE
            T Bit = value[0]
        else
            PC = value AND 0xFFFFFFF
        adderss = address + 4
    assert end_adderss = address - 4
```

```
LDM {<cond>}<addressing_mode> <Rd>{!},  
<registers_and_pc>^
```

指令操作的伪代码:

```
if ConditionPassed(cond) then  
    adderss = start_address  
    for i = 0 to 14  
        if registers _list[i] == 1 then  
            Ri = Memory[adderss,4]  
            adderss = address + 4  
    CPSR = SPSR  
    value = Memory[adderss,4]  
    if (architecture version 4T, 5 or above) and (T Bit  
= = 1) then  
        PC = value AND 0xFFFFFFFFFE  
    else  
        PC = value AND 0xFFFFFFFFFC  
    adderss = address + 4  
    assert end_adderss = address - 4
```

4.2.14 存储器访问指令14-1

14. STM 批量数据存储指令

- **格式:** STM{<cond>} {<type>}
<Rn>{!}, <regs>{^};
- **功能:** 将各个寄存器的值存入一片连续的内存单元中, 内存单元的起始地址为基址寄存器Rn的值, 各个寄存器由寄存器列表regs表示。

该指令一般用于多个寄存器数据的入栈。

{^}: 指示指令所用的寄存器为用户模式下的寄存器。

其他参数用法同LDM指令。

- **例:**

STMEA R13!, {R0-R12, PC}

;将寄存器R0~R12以及程序计数器PC的值保存到R13指示的堆栈中

4.2.14 存储器访问指令14-2*

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

指令操作的伪代码:

```
if ConditionPassed(cond) then
    adderss = start_address
    for i = 0 to 15
        if registers _list[i] == 1 then
            Memory[adderss,4] = Ri
            adderss = address + 4
    assert end_adderss = address - 4
```

4.3.15 数据存储指令15

15. SWP 字数据交换指令

- 格式: $\text{SWP}\{\langle\text{cond}\rangle\} \quad \langle\text{Rd}\rangle, \langle\text{op1}\rangle, [\langle\text{op2}\rangle];$
- 功能: $\text{Rd} = [\text{op2}], [\text{op2}] = \text{op1}$

从op2所表示的内存装载一个字并把这个字放置到目的寄存器Rd中，然后把寄存器op1的内容存储到同一内存地址中。

op1, op2均为寄存器。

- 例:

$\text{SWP} \quad \text{R0}, \text{R1}, [\text{R2}]$

;将R2所表示的内存单元中的字数据装载到R0，然后将R1中的字数据保存到R2所表示的内存单元中

4.3.16 Load/Store指令16

16. SWPB 字节数据交换指令

- **格式：** SWP{<cond>}B <Rd>, <op1>, [<op2>];
- **功能：** 从op2所表示的内存装载一个字节并把这个字节放置到目的寄存器Rd的低8位中，Rd的高24位设置为0；然后将寄存器op1的低8位数据存储在到同一内存地址中。
- **例：**

SWPB R0,R1,[R2]

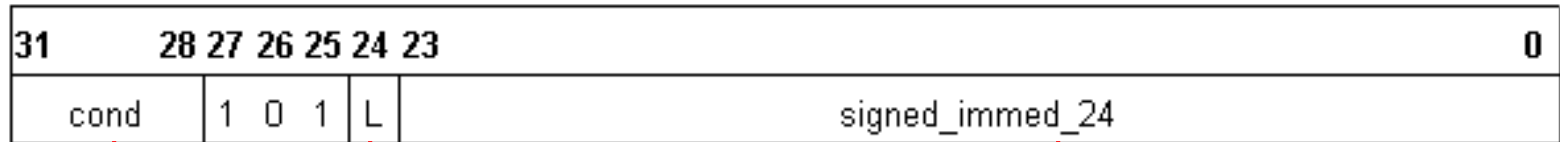
;将R2所表示的内存单元中的一个字节数据装载到R0的低8位，然后将R1中的低8位字节数据保存到R2所表示的内存单元中

4.3 跳转指令

- 跳转指令用于实现程序的跳转和程序状态的切换。
- ARM程序设计中，实现程序跳转有**两种方式**：
 - （1）跳转指令，
 - （2）直接向程序寄存器PC（R15）中写入目标地址值。
- 通过向程序计数器PC写入跳转地址值，可以实现在4GB的地址空间中的任意跳转。
- 使用跳转指令，其跳转空间受到限制。

• ARM分支指令——指令编码

• 分支指令B/BL指令编码格式

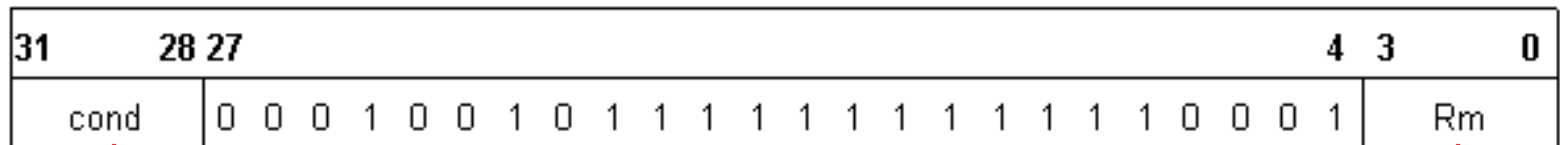


指令执行的条件码

L区别B指令（L为0）和BL指令（L为1）

24位有符号立即数（偏移量）

• 分支指令BX指令编码格式



指令执行的条件码

Rm目标地址寄存器，该寄存器装载跳转地址

4.3.1 跳转指令1

1. B 跳转指令

- 格式: $B\{\langle\text{cond}\rangle\} \quad \langle\text{addr}\rangle;$
- 功能: $PC = PC + \text{addr}$ 左移两位
- addr 的值是相对当前PC（即寄存器R15）的值的的一个偏移量，而不是一个绝对地址，它是24位有符号数。实际地址的值由汇编器来计算.
- addr 的值有符号扩展为32位后，左移两位，然后与PC值相加，即得到跳转的目的地址。
- 跳转的范围为 $-32\text{MB} \sim +32\text{MB}$ 。
- 例如:
- $B \quad \text{exit};$ 程序跳转到标号 exit 处
- ...
- $\text{exit} \cdots$

4.3.2 跳转指令2

2. BL 带返回的跳转指令

- **格式:** BL {<cond>} <addr>;
- **功能:** 同B指令，但BL指令执行跳转操作的同时，还将PC（寄存器R15）的值保存到LR寄存器（寄存器R14）中。
- 该指令用于实现**子程序调用**，程序的返回可通过把LR寄存器的值复制到PC寄存器中来实现。
- **例如:**
- BL func; 调用子程序func
- ...
- func
- ...
- MOV R15, R14; 子程序返回

4.3.3 跳转指令3

3. BLX 带返回和状态切换的跳转指令

- **格式:** BLX <addr>;或BLX <Rn>;
- **功能:** 处理器跳转到目标地址处, 并将PC (寄存器R15) 的值保存到LR寄存器 (R14) 中。
- 如果目标地址处为Thumb指令, 则程序状态从ARM状态切换为Thumb状态。
- 该指令用于子程序调用和**程序状态的切换**。
- **例如:**
- BLX T16; 跳转到标号T16处执行, T16后面的指令为Thumb指令
- ...
- CODE16
- T16 后面指令为Thumb指令
- ...

4.3.4 跳转指令4

4. BX 带状态切换的跳转指令

- 格式: `BX <Rn>;`
- 功能: 处理器跳转到目标地址处, 从那里继续执行;
- 目标地址为寄存器Rn的值和0xFFFFFFE作与操作的结果。
- 目标地址处的指令可以是ARM指令, 也可以是Thumb 指令。
- 例如:
 - `ADR R0, exit ;标号exit处的地址装入R0中`
 - `BX R0 ;跳转到exit处`

4.4 程序状态寄存器指令

- 用于状态寄存器和通用寄存器间传送数据。
- 总共有两条指令：MRS和MSR。
- 两者结合可用来修改程序状态寄存器的值。

4.4.1 程序状态寄存器指令1

1. MRS 程序状态寄存器到通用寄存器的数据传送指令

- **格式:** MRS {<cond>} <Rd>, CPSR/SPSR;
- **功能:** 用于将程序状态寄存器的内容传送到目标寄存器Rd中。
- 当进入中断服务程序或进程切换时，该指令可用来保存当前状态寄存器的值。
- **例如:**
- MRS R0, CPSR
- ;状态寄存器CPSR的值存入寄存器R0中

4.4.2 程序状态寄存器指令2

2. MSR 通用寄存器到程序状态寄存器的数据传送指令

- **格式:** MSR {<cond>} CPSR/SPSR_<field>, <op1>;
- **功能:** 用于将寄存器Rd的值传送到程序状态寄存器中。
- 当退出中断服务程序或进程切换时，该指令可用来恢复状态寄存器的值。
- 操作数op1可以是通用寄存器或立即数。
- <field>: 用来设置状态寄存器中需要操作的位。
- 32位的状态寄存器可以分为4个域:
- 位[31: 24]为条件标志位域，用f表示。
- 位[23: 16]为状态位域，用s表示。
- 位[15: 8]为扩展位域，用x表示。
- 位[7: 0]为控制位域，用c表示。
- **例如:** MSR CPSR_f, R0
- ;用R0的值修改CPSR的条件标志域
- MSR CPSR_fsxc, #5; CPSR的值修改为5

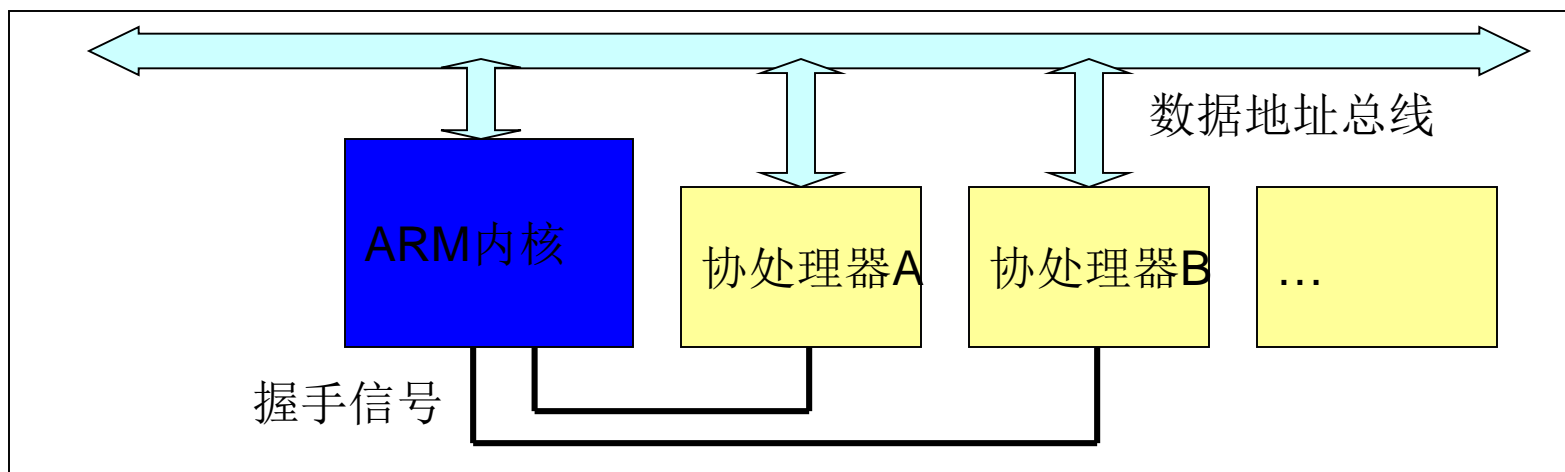
4.5 协处理器指令

- ARM处理器最多可支持16个协处理器，用于辅助ARM完成各种协处理操作。
- 在程序执行过程中，各协处理器只执行自身的协处理指令，而忽略属于ARM处理器和其他协处理器的指令。
- ARM协处理器指令可分为3类：
 - (1) ARM处理器用于初始化协处理器的数据操作指令（CDP）。
 - (2) 协处理器寄存器和内存单元之间的数据传送指令（LDC, STC）。
 - (3) ARM处理器寄存器和协处理器寄存器之间的数据传送指令（MCR, MRC）。

- ARM指令集——协处理器指令

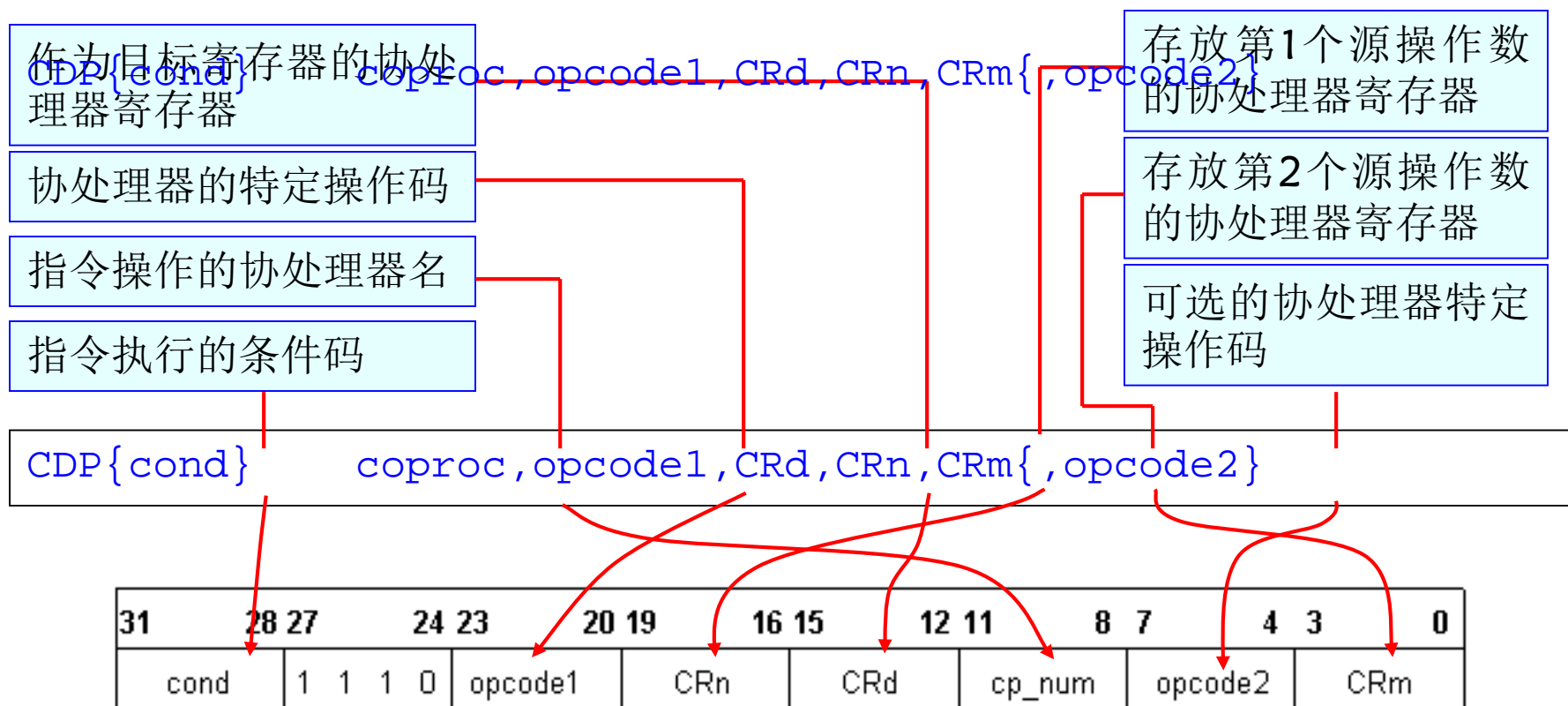
ARM内核支持协处理器操作，协处理器的控制要通过协处理器命令实现。

ARM内核与协处理器的关系



• ARM协处理器指令——数据操作指令

ARM处理器通过CDP指令通知ARM协处理器执行特定的操作。该操作由协处理器完成，即对命令的参数解释与协处理器有关，指令的使用取决于协处理器。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。指令格式如下：



数据操作指令编码

• ARM协处理器指令——数据操作指令

ARM处理器通过CDP指令通知ARM协处理器执行特定的操作。该操作由协处理器完成，即对命令的参数的解释与协处理器有关，指令的使用取决于协处理器。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。指令格式如下：

应用示例：

CDP p7,0,c0,c2,c3,0 ;对协处理器7操作，操作码为0，
;可选操作码为0

CDP p6,1,c3,c4,c5 ;对协处理器6操作，操作码为1

$$\text{CDP}\{\text{cond}\} \quad \text{coproc}, \text{opcode1}, \text{CRd}, \text{CRn}, \text{CRm}\{, \text{opcode2}\}$$

31	28 27	24 23	20 19	16 15	12 11	8 7	4 3	0
cond	1 1 1 0	opcode1	CRn	CRd	cp_num	opcode2	CRm	

• ARM协处理器指令——数据存取指令

协处理器数据存取指令LDC/STC指令可以将某一连续内存单元的数据读取到协处理器的寄存器中，或者将协处理器的寄存器数据写入到某一连续的内存单元中，传送的字数由协处理器来控制。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。

数据读取指令格式

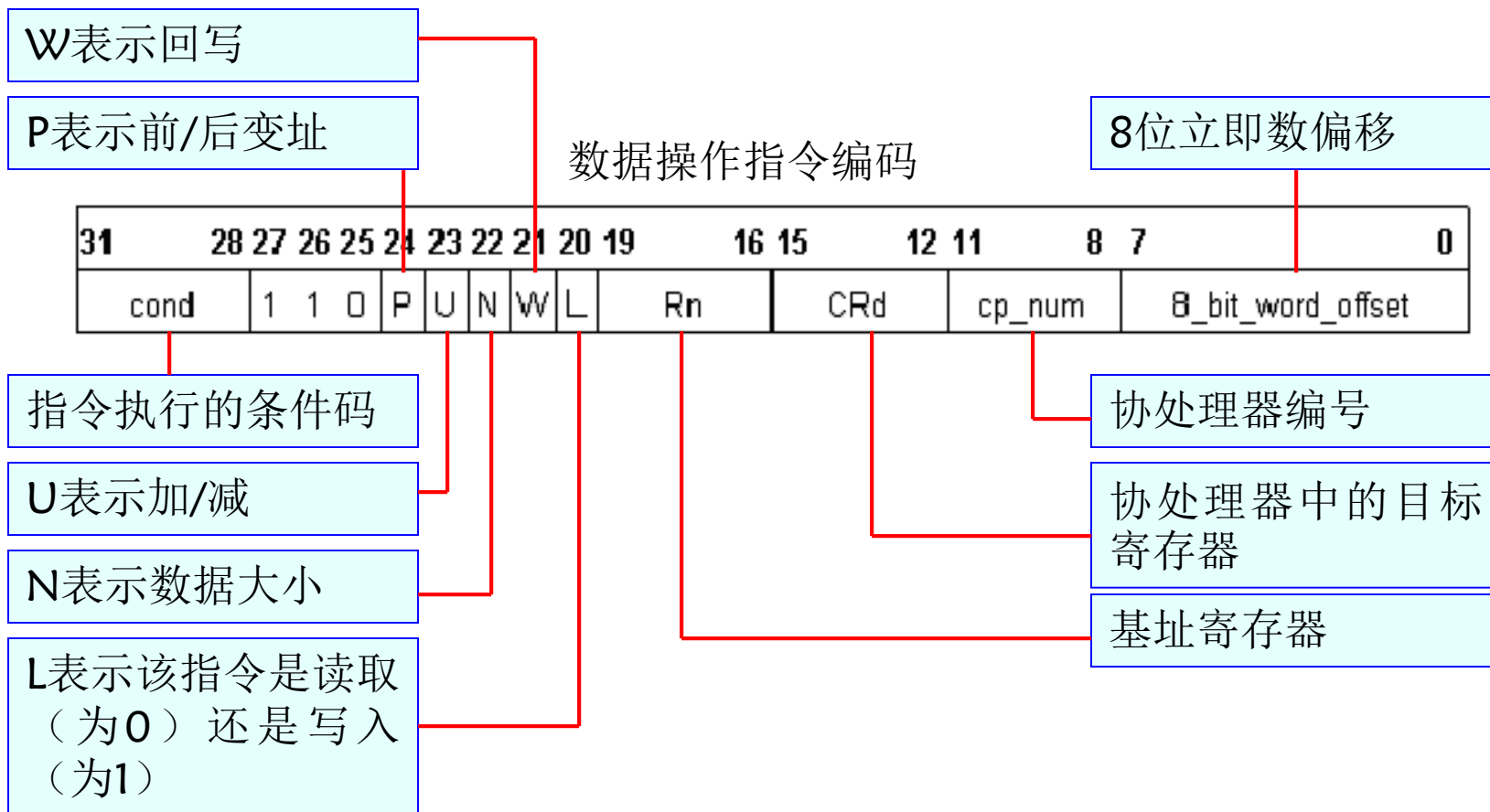
LDC{cond}{L} coproc, CRd, <地址>

数据存储指令格式

STC{cond}{L} coproc, CRd, <地址>

• ARM协处理器指令——数据存取指令

协处理器数据存取指令LDC/STC指令可以将某一连续内存单元的数据读取到协处理器的寄存器中，或者将协处理器的寄存器数据写入到某一连续的内存单元中，传送的字数由协处理器来控制。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。



• ARM协处理器指令——数据存取指令

协处理器数据存取指令LDC/STC指令可以将某一连续内存单元的数据读取到协处理器的寄存器中，或者将协处理器的寄存器数据写入到某一连续的内存单元中，传送的字数由协处理器来控制。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。

数据操作指令编码

31	28	27	26	25	24	23	22	21	20	19	16	15	12	11	8	7	0
cond	1	1	0	P	U	N	W	L	Rn	CRd	cp_num	8_bit_word_offset					

应用示例：

LDC p5, c2, [R2, #4] ;读取R2+4指向的内存单元的数据，
 ;传送到协处理器p5的c2寄存器中
STC p5, c1, [R0] ;将协处理器p5的C1寄存器内数据
 ;传送到R0指向的内存单元

• ARM协处理器指令——寄存器传送指令

如果需要在ARM处理器中的寄存器与协处理器中的寄存器之间进行数据传送，那么可以使用MCR/MRC指令。MCR指令用于将ARM处理器的寄存器中的数据传送到协处理器的寄存器。MRC指令用于将协处理器的寄存器中的数据传送到ARM处理器的寄存器中。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。

MCR指令格式（ARM→协处理器）

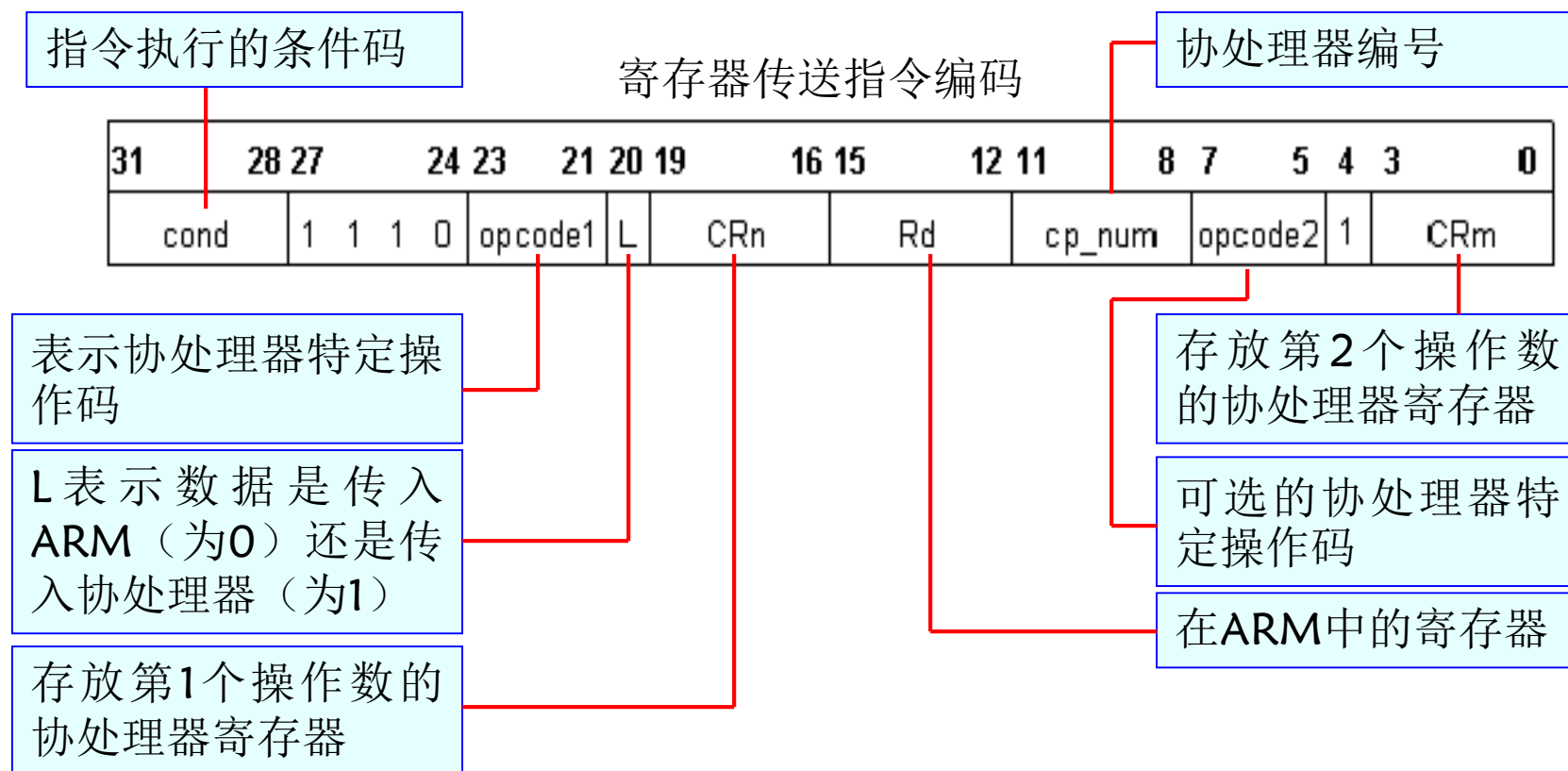
MCR{cond} coproc, opcode1, Rd, CRn, CRm{ , opcode2}
--

MRC指令格式（协处理器→ARM）

MRC{cond} coproc, opcode1, Rd, CRn, CRm{ , opcode2}
--

• ARM协处理器指令——寄存器传送指令

如果需要在ARM处理器中的寄存器与协处理器中的寄存器之间进行数据传送，那么可以使用MCR/MRC指令。MCR指令用于将ARM处理器的寄存器中的数据传送到协处理器的寄存器。MRC指令用于将协处理器的寄存器中的数据传送到ARM处理器的寄存器中。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。



• ARM协处理器指令——寄存器传送指令

如果需要在ARM处理器中的寄存器与协处理器中的寄存器之间进行数据传送，那么可以使用MCR/MRC指令。MCR指令用于将ARM处理器的寄存器中的数据传送到协处理器的寄存器。MRC指令用于将协处理器的寄存器中的数据传送到ARM处理器的寄存器中。若协处理器不能成功地执行该操作，将产生未定义指令异常中断。

寄存器传送指令编码

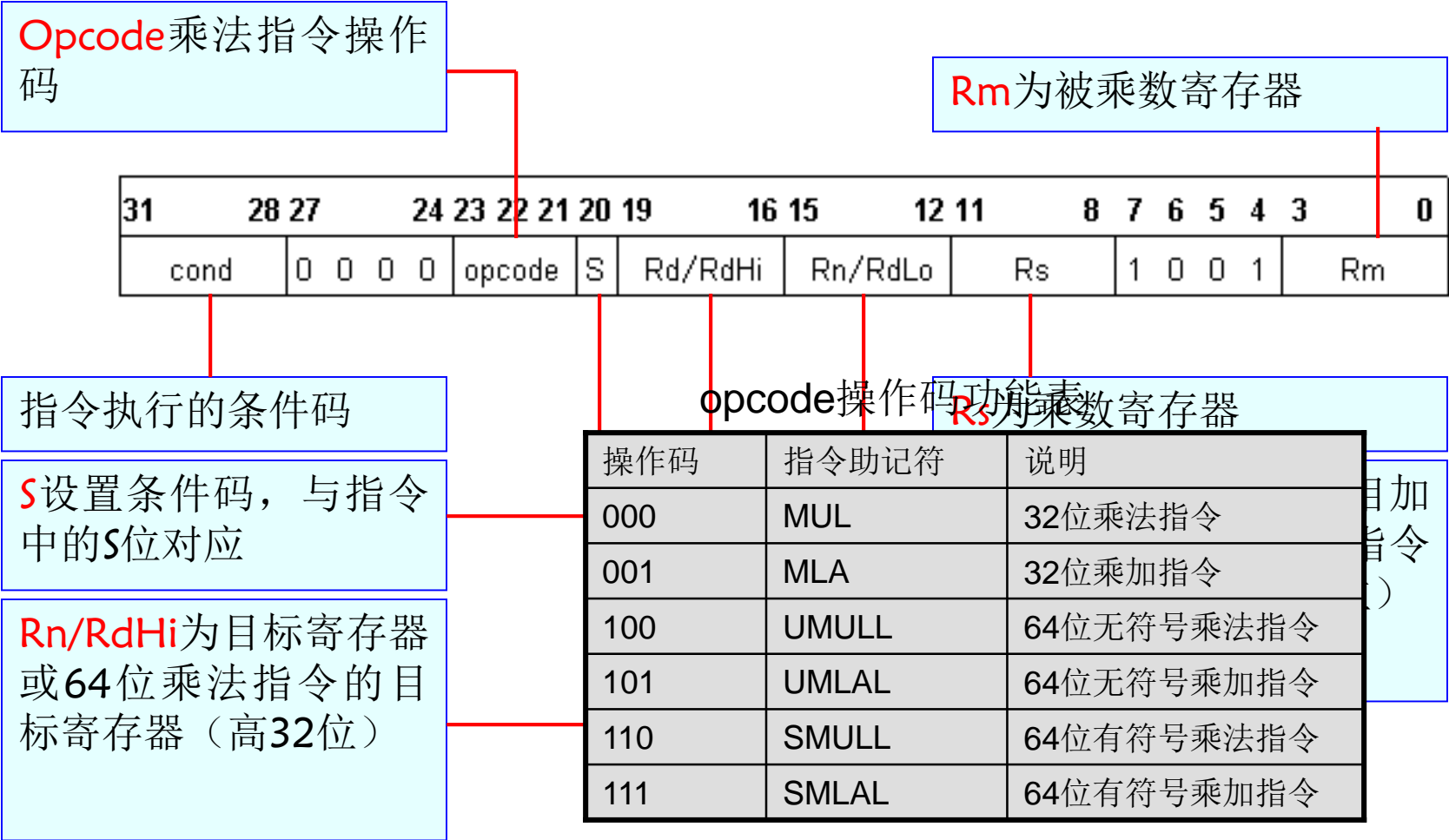
31	28	27	24	23	21	20	19	16	15	12	11	8	7	5	4	3	0		
cond		1	1	1	0	opcode1		L	CRn		Rd		cp_num		opcode2		1	CRm	

应用示例：

MCR	p6, 2, R7, c1, c2	;将ARM中的R7寄存器内容传递 ;到协处理器6的C1和C2寄存器
MRC	p5, 2, R2, c3, c2	;将协处理器5的C3和C2寄存器 ;内容传递到ARM中的R2寄存器

4.6 乘法指令*

乘法指令编码



4.6.1 乘法指令1

1. MUL 32位乘法指令

- 格式: $MUL\{<cond>\}\{S\} \quad <Rd>, <Rn>, <op2>;$
- 功能: $Rd = Rn \times op2$
- 该指令根据S标志, 决定操作是否影响CPSR的值; 其中op2必须为寄存器。Rn和op2的值为32位的有符号数或无符号数。

- 例:

MULS R0, R1, R2; $R0 = R1 \times R2$, 结果影响寄存器CPSR的值

4.6.2 乘法指令2

2. MLA 32位乘加指令

- 格式: $\text{MLA}\{<\text{cond}>\}\{S\} \quad <\text{Rd}>, <\text{Rn}>, <\text{op2}>, <\text{op3}>;$

- 功能: $\text{Rd} = \text{Rn} \times \text{op2} + \text{op3}$

op2和op3必须为寄存器。Rn、op2和op3的值为32位的有符号数或无符号数。

- 例:

$\text{MLA} \quad \text{R0}, \text{R1}, \text{R2}, \text{R3}; \quad \text{R0} = \text{R1} \times \text{R2} + \text{R3}$

4.6.3 数据处理指令3

3. SMULL 64位有符号数乘法指令

- 格式:

SMULL{<cond>} {S} <Rdl>, <Rdh>, <Rn>, <op2>;

- 功能:

$Rdh\ Rdl = Rn \times op2$

Rdh、Rdl和op2均为寄存器。Rn和op2的值为32位的有符号数。

- 例:

SMULL R0,R1,R2,R3; $R0 = R2 \times R3$ 的低32位
 ; $R1 = R2 \times R3$ 的高32位

4.6.4 数据处理指令4

4. SMLAL 64位有符号数乘加指令

- 格式:

$\text{SMLAL}\{<\text{cond}>\}\{S\} \quad <\text{Rd1}>, <\text{Rdh}>, <\text{Rn}>, <\text{op2}>;$

- 功能:

$\text{Rdh} \text{ Rd1} = \text{Rn} \times \text{op2} + \text{Rdh} \text{ Rd1}$

Rdh、Rd1和op2均为寄存器。Rn和op2的值为32位的有符号数，Rdh Rd1的值为64位的加数。

- 例:

$\text{SMLAL} \quad \text{R0}, \text{R1}, \text{R2}, \text{R3}; \quad \text{R0} = \text{R2} \times \text{R3} \text{的低32位} + \text{R0}$
 $\quad \quad \quad ; \quad \text{R1} = \text{R2} \times \text{R3} \text{的高32位} + \text{R1}$

4.6.5 乘法指令5

13. UMULL 64位无符号数乘法指令

- 格式:

UMULL {<cond>} {S} <Rd1>, <Rdh>, <Rn>, <op2>;

- 功能: 同SMULL指令, 但指令中Rn和op2的值为32位的无符号数。

- 例如:

- UMULL R0, R1, R2, R3
- ;R0=R2×R3的低32位
- ;R1=R2×R3的高32位
- 其中R2, R3的值为无符号数

4.6.6 数据处理指令6

6. UMLAL 64位无符号数乘加指令

- 格式:

UMLAL {<cond>} {S} <Rd1>, <Rdh>, <Rn>, <op2>;

- 功能: 同SMLAL指令, 但指令中Rn, op2的值为32位的无符号数, Rdh Rd1的值为64位无符号数。

- 例如:

- UMLAL R0, R1, R2, R3
- ;R0=R2×R3的低32位+R0
- ;R1=R2×R3的高32位+R1
- 其中R2, R3的值为32位无符号数
- R1, R0的值为64位无符号数

4.7.1 异常中断指令1

1. SWI 软件中断指令

- **格式:** SWI {<cond>} 24位的立即数;
- **功能:** 用于产生软件中断, 以使用户程序调用操作系统的系统例程。
- 指令中24位的立即数指定用户程序调用系统例程的类型, 其参数通过通用寄存器传递。
- 当24位的立即数被忽略时, 系统例程类型由寄存器R0指定, 其参数通过其他通用寄存器传递。
- **例如:**
- SWI 0X05; 调用编号为05的系统例程。

4.7.2 异常中断指令2

- SWI {<cond>} <immed_24>
- 指令操作的伪代码:
- if ConditionPassed(cond) then
- R14_svc = address of next instruction after the SWI instruction
- SPSR_svc = CPSR
- CPSR[4:0] = 0b10011 /* Enter Supervisor mode */
- CPSR[5] = 0 /* Execute in ARM state, T Bit = 0*/
- /* CPSR[6] is unchanged, fiq不变*/
- CPSR[7] = 1 /* Disable normal interrupt, I Bit = 1, 禁止irq */
- if high vectors configured then
- PC = 0xFFFF0008
- else
- PC = 0x00000008

4.7.3 异常中断指令3

2. BKPT 断点中断指令

- **格式:** BKPT 16位的立即数;
- **功能:** 用于产生软件断点中断, 以便软件调试时使用。16位的立即数用于保存软件调试中额外的断点信息。
- **指令操作的伪代码:**
- if (not overriden by debug hardware) then
- R14_abt = address of BKPT instruction + 4
- SPSR_abt = CPSR
- CPSR[4:0] = 0b10111 /* 中止模式 */
- CPSR[5] = 0 /* 使程序处于 ARM 状态, T Bit = 0*/
- /* CPSR[6] is unchanged, fiq不变*/
- CPSR[7] = 1 /* 禁止正常中断, I Bit = 1, 禁止irq */
- if high vectors configured then
- PC = 0xFFFF000C
- else
- PC = 0x0000000C

4.8 Thumb指令集

- Thumb指令集可以看做ARM指令集的一个子集，用于支持存储系统数据总线为16位的应用系统。
- Thumb指令长度为16位，这样，与32位的ARM指令集相比，有效地节省了系统的存储空间。
- 但Thumb指令集中的数据处理指令的操作数仍然是32位的，指令寻址地址也是32位的。
- **Thumb指令集中没有：** 乘加指令、64位乘法指令、协处理器指令、数据交换指令、程序状态寄存器指令，而且指令的第二操作数受到限制，除了跳转指令B有条件执行功能外，其他指令均为无条件执行。
- **Thumb指令集有4大类：** 数据处理指令、跳转指令、Load/Store指令、软件中断指令。

数据处理指令汇总1

格 式	功 能
MOV Rd, imm_8;	Rd= imm_8; Rd 为 R0~R7, imm_8 为 8 位立即数
MOV Rd, Rn;	Rd= Rn; Rd、Rn 为 R0~R15
MVN Rd, Rn;	Rd= ~Rn; Rd、Rn 为 R0~R7
NEG Rd, Rn;	Rd= - Rn; Rd、Rn 为 R0~R7
ADD Rd, Rn, imm;	Rd= Rn+ imm; Rd 为 R0~R7, Rn 为 R0~R7 或 PC 或 SP; Rn 为 PC 或 SP 时, imm 为 10 位立即数; 否则, imm 为 3 位立即数
ADD Rd, Rn, Rm;	Rd= Rn+ Rm; Rd、Rn、Rm 为 R0~R7
ADD Rd, imm;	Rd= Rd+ imm; Rd 为 R0~R7 或 SP Rd 为 SP 时, imm 为- 508~+ 508 间的 4 整数倍的数 否则, imm 为 8 位立即数
ADD Rd, Rn;	Rd= Rd+ Rn; Rd、Rn 为 R0~R15
ADC Rd, Rn;	Rd= Rd+ Rn+ carry; Rd、Rn 为 R0~R7, carry 为进位标志值
SUB Rd, Rn, imm_3;	Rd= Rn- imm_3; Rd、Rn 为 R0~R7, imm_3 为 3 位立即数
SUB Rd, Rn, Rm;	Rd= Rn- Rm; Rd、Rn、Rm 为 R0~R7,
SUB Rd, imm;	Rd= Rd- imm; Rd 为 R0~R7 或 SP Rd 为 SP 时, imm 为- 508~+ 508 间的 4 整数倍的数 否则, imm 为 8 位立即数
SBC Rd, Rn;	Rd= Rd- Rn- !carry; Rd、Rn 为 R0~R7, carry 为进位标志值
MUL Rd, Rn;	Rd= Rd×Rn; Rd、Rn 为 R0~R7

数据处理指令汇总2

格 式	功 能
AND Rd, Rn;	$Rd = Rd \& Rn$; Rd、Rn 为 R0~R7
ORR Rd, Rn;	$Rd = Rd Rn$; Rd、Rn 为 R0~R7
EOR Rd, Rn;	$Rd = Rd \wedge Rn$; Rd、Rn 为 R0~R7
BIC Rd, Rn;	$Rd = Rd \& (\sim Rn)$; Rd、Rn 为 R0~R7
ASR Rd, Rn;	$Rd = Rd$ 算术右移 Rn 位; Rd、Rn 为 R0~R7
ASR Rd, Rn, imm_5;	$Rd = Rn$ 算术右移 imm_5 位; Rd、Rn 为 R0~R7, imm_5 为 1~32 间的数值
LSL Rd, Rn;	$Rd = Rd$ 逻辑左移 Rn 位; Rd、Rn 为 R0~R7
LSL Rd, Rn, imm_5;	$Rd = Rn$ 逻辑左移 imm_5 位; Rd、Rn 为 R0~R7
LSR Rd, Rn;	$Rd = Rd$ 逻辑右移 Rn 位; Rd、Rn 为 R0~R7
LSR Rd, Rn, imm_5;	$Rd = Rn$ 逻辑右移 imm_5 位; Rd、Rn 为 R0~R7,
ROR Rd, Rn;	$Rd = Rd$ 循环右移 Rn 位; Rd、Rn 为 R0~R7
CMP Rn, Rm;	根据 $Rn - Rm$ 的值, 修改 CPSR 的状态标志位; Rn、Rm 为 R0~R7
CMP Rn, imm_8;	根据 $Rn - imm_8$ 的值, 修改 CPSR 的状态标志位; Rn 为 R0~R7
CMN Rn, Rm;	根据 $Rn + Rm$ 的值, 修改 CPSR 的状态标志位; Rn、Rm 为 R0~R7
TST Rn, Rm;	根据 $Rn \& Rm$ 的值, 修改 CPSR 的状态标志位; Rn、Rm 为 R0~R7

跳转指令汇总

格 式	功 能
B{cond} label	PC=label; 若有 cond ，则 label 必须在当前指令的 -256~+256 字节范围内； 否则， label 必须在当前指令的 -2K~+2K 字节范围内
BL label	R14=PC+4, PC=label; label 必须在当前指令的 -4M~+4M 字节范围内
BX Rn	PC=Rn ，且切换处理器状态

Load/Store指令汇总1

格 式	功 能
LDR Rd, [Rn,imm];	Rd= 地址 (Rn+ imm) 中的字数据；Rd 为 R0~R7， Rn 为 R0~R7 或 SP 或 PC；若 Rn 为 PC 或 SP， imm 为 5 位立即数，否则 imm 为 8 位立即数
LDR Rd, [Rn,Rm];	Rd= 地址 (Rn+ Rm) 中的字数据；Rd、 Rn、 Rm 为 R0~R7
LDRH Rd, [Rn,imm_5];	Rd= 地址 (Rn+ imm_5) 中的无符号半字数据； Rd、 Rn 为 R0~R7，imm_5 为 5 位立即数
LDRH Rd, [Rn,Rm];	Rd= 地址 (Rn+ Rm) 中的无符号半字数据； Rd, Rn, Rm 为 R0~R7
LDRB Rd, [Rn,imm_5];	Rd= 地址 (Rn+ imm_5) 中的无符号字节数据； Rd、 Rn 为 R0~R7
LDRB Rd, [Rn,Rm];	Rd= 地址 (Rn+ Rm) 中的无符号字节数据； Rd, Rn, Rm 为 R0~R7
LDRSH Rd, [Rn,Rm];	Rd= 地址 (Rn+ Rm) 中的有符号半字数据； Rd, Rn, Rm 为 R0~R7
LDRSB Rd, [Rn,Rm];	Rd= 地址 (Rn+ Rm) 中的有符号字节数据； Rd, Rn, Rm 为 R0~R7
LDR Rd, label;	Rd= 地址 (label) 中的字数据；Rd 为 R0~R7

Load/Store指令汇总2

格 式	功 能
STR Rd, [Rn,imm];	地址 (Rn + imm) 处的字数据 = Rd ; Rd 为 R0~R7, Rn 为 R0~R7 或 SP 或 PC ; 若 Rn 为 PC 或 SP , imm 为 5 位立即数 , 否则 imm 为 8 位立即数
STR Rd, [Rn,Rm];	地址 (Rn + Rm) 处的字数据 = Rd ; Rd、 Rn、 Rm 为 R0~R7
STRH Rd, [Rn,imm_5];	地址 (Rn + imm_5) 处的无符号半字数据 = Rd ; Rd、 Rn 为 R0~R7
STRH Rd, [Rn,Rm];	地址 (Rn + Rm) 处的无符号半字数据 = Rd ; Rd、 Rn、 Rm 为 R0~R7
STRB Rd, [Rn,imm_5];	地址 (Rn + imm_5) 处的无符号字节数据 = Rd ; Rd、 Rn 为 R0~R7
STRB Rd, [Rn,Rm];	地址 (Rn + Rm) 处的无符号字节数据 = Rd ; Rd、 Rn、 Rm 为 R0~R7
LDMIA Rd(!),regs;	Regs = 以 Rd 为起始地址的连续字数据 ; regs 为寄存器列表
STMIA Rd(!),regs;	以 Rd 为起始地址的连续字数据 = regs ; regs 为寄存器列表
PUSH regs{,LR};	[SP...] = regs{,LR} ; LR 即 R14 , SP 即 R13
POP regs{,PC};	regs{,PC} = [SP...] ; PC 即 R15 , SP 即 R13

软中断指令汇总

格 式	功 能
SWI 8位立即数	8 位立即数为中断号

5. ARM汇编伪指令和伪操作

概念

5.1 ARM汇编伪指令

5.2 ARM汇编伪操作

5.3 ARM汇编宏指令

5.4 ADS编译环境下的伪操作

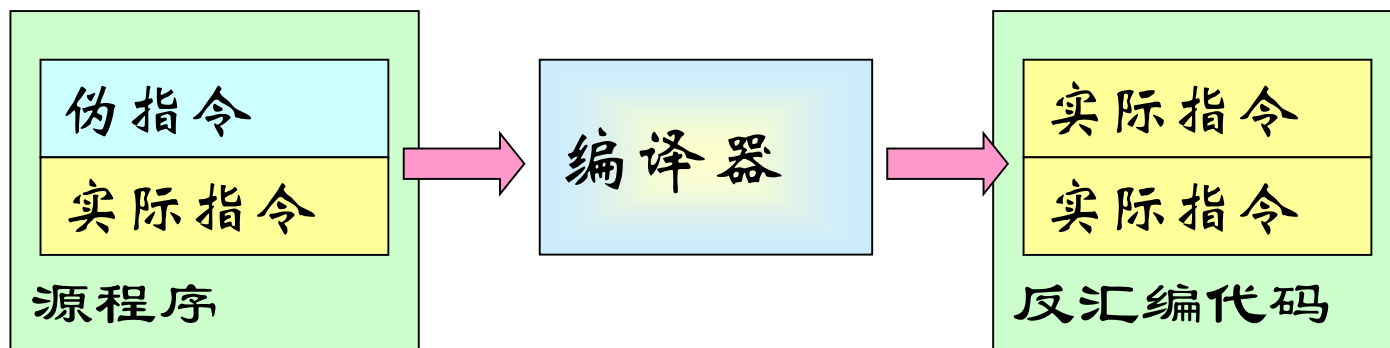
伪指令、伪操作和宏指令概念

- **伪指令(Pseudo Instructions)**——是汇编语言程序里的特殊指令助记符，在汇编时被合适的机器指令替代。
- **伪操作(Directives)**——为汇编程序所用，在源程序进行汇编时由汇编程序处理，只在汇编过程起作用，不参与程序运行。
- **宏指令**——通过伪操作定义的一段独立的代码。在调用它时将宏体插入到源程序中。也就是常说的宏。

说明：所有的伪指令、伪操作和宏指令，均与具体的开发工具中的编译器有关(这里主要针对“**ADS/SDT IDE**”开发工具，最新的工具**MDK5、DS-5**等也基本相似)。

5.1 伪指令

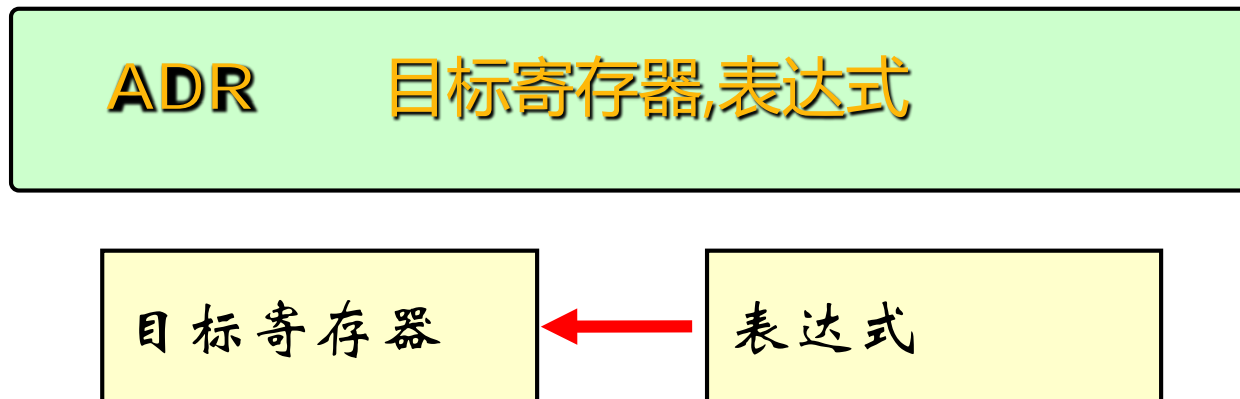
ARM伪指令不属于ARM指令集中的指令，是为了编程方便而定义的。伪指令可以像其它ARM指令一样使用，但在编译时这些指令将被等效的ARM指令代替。



1. 小范围地址读取指令：ADR
2. 中等范围地址读取指令：ADRL
3. 大等范围地址读取指令：LDR
4. 空操作指令：NOP

5.1.1 ARM伪指令——小范围的地址读取

ADR伪指令将基于PC相对偏移的地址值或基于寄存器相对偏移的地址值读取到寄存器中。在汇编编译器编译源程序时，ADR伪指令被编译器替换成一条合适的指令，若不能用一条指令实现，则产生错误，编译失败。

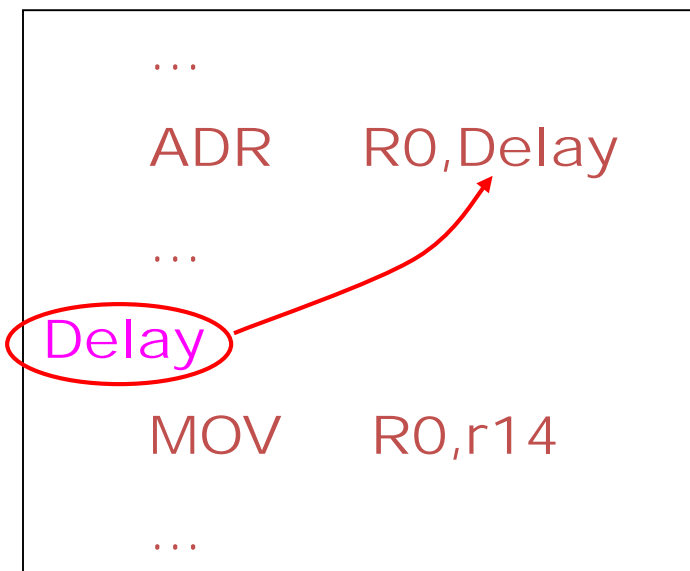


地址表达式expr的取指范围：

- 当地址值不是字对齐时，其取指范围为 ± 255 ；
- 当地址值是字对齐时，其取指范围为 ± 1020 ；
- 当地址值是16字节对齐时，其取指范围将更大。

应用示例（源程序）：

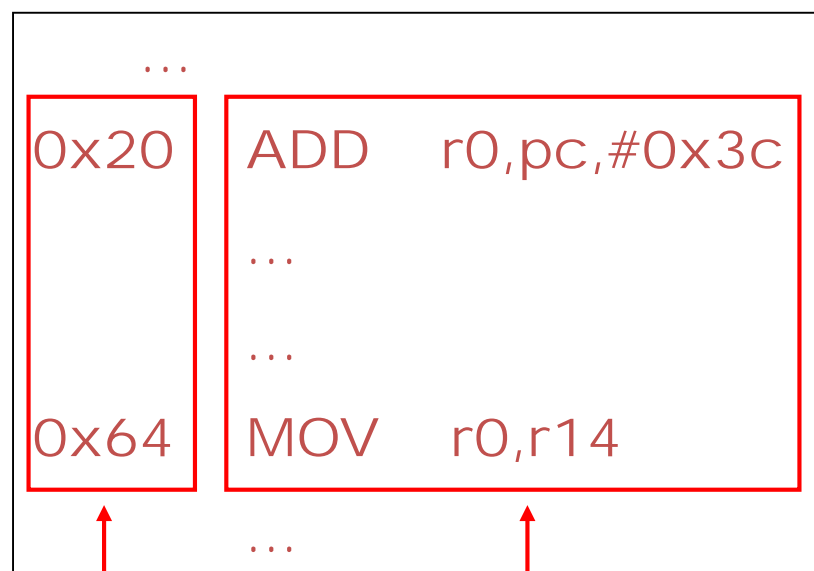
```
...  
ADR    R0,Delay  
...  
Delay  
MOV    R0,r14  
...
```



使用伪指令将程序标号Delay的地址存入R0

编译后的反汇编代码：

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



地址

程序代码

应用示例（源程序）：

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

使用伪指令将程序标号Delay的地址存入R0

编译后的反汇编代码：

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

ADR伪指令被汇编成一条指令

;查表应用示例:

ADR R0,DISP_TAB ;加载转换表地址

LDRB R1,[R0,R2] ;使用R2作为参数,进行查表

...

DISP_TAB

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

5.1.2 ARM伪指令——中等范围的地址读取

ADRL伪指令将基于PC相对偏移的地址值或基于寄存器相对偏移的地址值读取到寄存器中，比ADR伪指令可以读取更大范围的地址。在汇编编译器编译源程序时，ADRL伪指令被编译器替换成两条合适的指令。若不能用两条指令实现，则产生错误，编译失败。

ADRL 标寄存器,表达式

目标寄存器



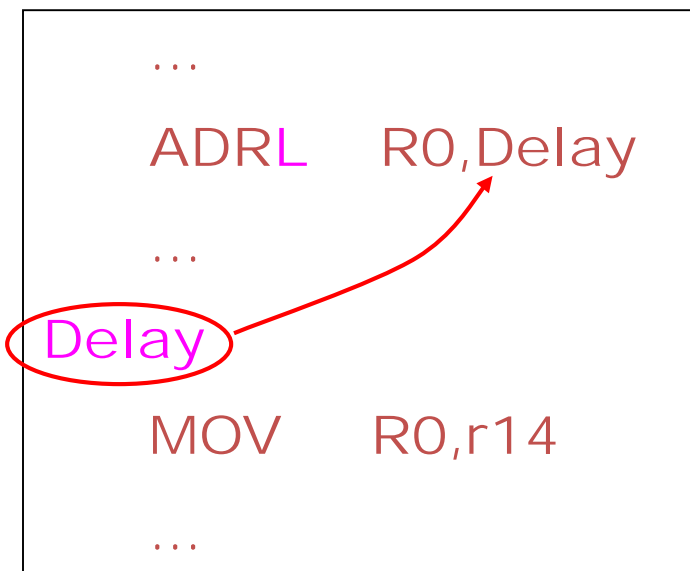
表达式

地址表达式expr的取指范围：

- 当地址值不是字对齐时，其取指范围为 $\pm 64K$
- 当地址值是字对齐时，其取指范围为 $\pm 256K$
- 当地址值是16字节对齐时，其取指范围将更大

应用示例（源程序）：

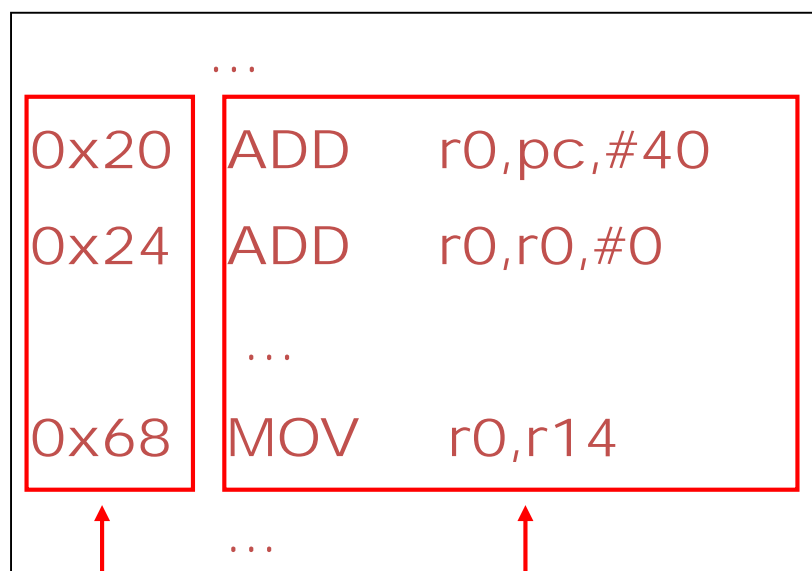
```
...  
ADRL  R0,Delay  
...  
Delay  
MOV   R0,r14  
...
```



使用伪指令将程序标号Delay的地址存入R0

编译后的反汇编代码：

```
...  
0x20  ADD    r0,pc,#40  
0x24  ADD    r0,r0,#0  
...  
0x68  MOV    r0,r14  
...
```



地址

程序代码

应用示例（源程序）：

```
...  
ADRL  R0,Delay  
...  
Delay  
MOV   R0,r14  
...
```

使用伪指令将程序标号Delay的地址存入R0

编译后的反汇编代码：

```
...  
0x20  ADD    r0,pc,#40  
0x24  ADD    r0,r0,#0  
...  
0x68  MOV    r0,r14  
...
```

ADRL伪指令被汇编成两条指令，
尽管第2条指令并没有意义

5.1.3 ARM伪指令——大范围的地址读取

LDR伪指令用于加载32位的立即数或一个地址值到指定寄存器。在汇编编译源程序时，LDR伪指令被编译器替换成一条合适的指令。若加载的常数未超出MOV或MVN的范围，则使用MOV或MVN指令代替该LDR伪指令，否则汇编器将常量放入文字池，并使用一条程序相对偏移的LDR指令从文字池读出常量。

LDR 标寄存器, = 表达式

目标寄存器

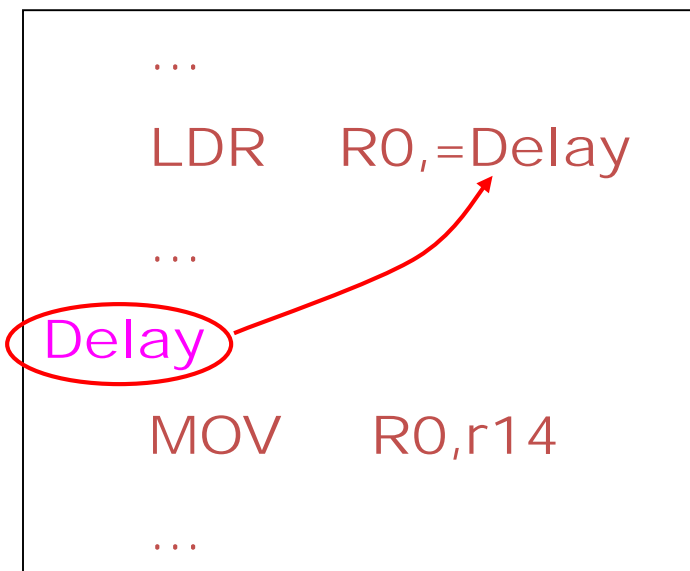


表达式

地址表达式expr的取指范围为任意值

应用示例（源程序）：

```
...  
LDR    R0,=Delay  
...  
Delay  
MOV    R0,r14  
...
```



使用伪指令将程序标号Delay的地址存入R0

编译后的反汇编代码：

...	...
0x60	LDR R0,[pc 0x4c]
...	...
0x64	MOV R0, LR
...	...
0xb4	DCD 0x64

↑ ↑

地址 程序代码

应用示例（源程序）：

```
...  
LDR R0,=Delay  
...  
Delay  
MOV R0,r14  
...
```

必须加入“=”

使用伪指令将程序标号Delay的地址存入R0

编译后的反汇编代码：

```
...  
0x60 LDR R0,[pc 0x4c]  
...  
0x64 MOV R0, LR  
...  
0xb4 DCD 0x64
```

必须小于4KB

LDR伪指令被汇编成一条LDR指令，并在文字池中定义了一个常量，该常量为Delay标号的地址

5.1.4 ARM伪指令——空操作伪指令

NOP伪指令在汇编时将会被代替成ARM中的空操作，比如可能是“MOV R0, R0”指令等。**NOP**可用于延时操作。

指令格式：



NOP

5.2 ARM汇编伪操作

ADS编译环境下的伪操作可分为以下几类：

- 符号定义（Symbol Definition）伪操作
- 数据定义（Data Definition）伪操作
- 汇编控制（Assembly Control）伪操作
- 其它（Miscellaneous）伪操作

5.2.1 符号定义伪操作

- GBLA, GBLL, GBLS: 声明全局变量。
- LCLA, LCLL, LCLS: 声明局部变量。
- SETA, SETL, SETS: 给变量赋值。
- RLIST: 为通用寄存器列表定义名称。

Test3 SETA 0xaa ; 将该变量赋值为0xaa

5.2.2 数据定义伪操作

- LTORG: 声明一个数据缓冲池的开始
- SPACE: 分配一块字节内存单元, 并用0初始化
- DCB: 分配一段字节内存单元, 并初始化
- DCD、DCDU: 分配一段字内存单元, 并初始化
- MAP: 定义一个结构化的内存表的首地址
- FIELD: 定义结构化内存表中的一个数据域

MAP 0x100, R0 ; 定义结构化内存表首地址的值为 $0x100 + R0$

A FIELD 16 ; 定义A的长度为16字节, 位置为 $0x110 + R0$

LTORG ; 定义数据缓冲池&0x8000

Data SPACE 4200 ; 从当前位置开始分配4200字节的内存单元,
; 并初始化为0。

5.2.2.1 LTORG

用于声明一个数据缓冲池（文字池）的开始。

语法格式:

LTORG

- 例: start BL func ;

• • • • •

```
func    LDR    R1,=0x8000    ;子程序
```

● ● ● ● ● ●

MOV PC, LR ;子程序返回

LTORG ;定义数据缓冲池&0x8000

Data SPACE 4200 ;从当前位置开始分配4200字节的内存单元，并初始化为0。

END

默认数据缓冲池为空

注意：

- LTORG伪操作通常放在无条件跳转指令之后，或者子程序返回指令之后，这样处理器不会错误地将数据缓冲池中的数据当作指令来执行。
- 通常ARM汇编编译器把数据缓冲池放在代码段的最后面，即下一个代码段开始之前，或者END伪操作之前。

5.2.2.3 DCB——也可以用符号“=”表示

用于定义并且初始化一个或者多个字节的内存区域。

语法格式：

{label} DCB expr {, expr}

或 {label} = expr {, expr}

其中expr表示：

- -128到255之间的一个数值常量或者表达式。
- 一个字符串。

注意：当DCB后面紧跟一个指令时，可能需要使用ALIGN确保指令是字对齐的。

例：

short DCB 1 ;为short分配了一个字节, 并初始化为1。

string DCB “string”, 0 ;构造一个以0结尾的字符串

5.2.2.4 DCD、DCDU分配一段字内存单元

(1) DCD ——分配一段字对齐的内存单元

用于分配一段字对齐的内存单元，并初始化。 DCD也可以用符号”&”表示

语法格式：

{label} DCD expr {, expr}

或 {label} & expr {, expr} expression

其中：

expr : 数字表达式或程序中的标号。

注意：DCD伪操作可能在分配的第一个内存单元前插入填补字节以保证分配的内存是字对齐的。

例：

data1 DCD 2, 4, 6

;为data1分配三个字, 内容初始化为2, 4, 6

data2 DCD label+4

;初始化data2为label+4对应的地址

(2) DCDU ——分配一段字非严格对齐的内存单元

DCDU与DCD的不同之处在于DCDU分配的内存单元并不严格字对齐。

5.2.2 汇编控制伪操作

- IF, ELSE及ENDIF: 有条件选择汇编
- WHILE及WEND: 有条件循环（重复）汇编
- MACRO, MEND及MEXIT: 宏定义汇编

5.2.4 其它伪操作

- AREA: 定义一个代码段或数据段
- CODE16、CODE32: 告诉编译器后面的指令序列位数
- ENTRY: 指定程序的入口点
- ALIGN: 将当前的位置以某种形式对齐 ALIGN或ALIGN n: 以字或n字节对齐
- END: 源程序结尾
- EQU: 为数字常量、基于寄存器的值和程序中的标号定义一个字符名称。
- EXPORT、GLOBAL: 声明源文件中的符号可以被其他源文件引用
- IMPORT、EXTERN: 声明某符号是在其他源文件中定义的
- GET、INCLUDE: 将一个源文件包含到当前源文件中，并将被包含的文件在其当前位置进行汇编处理。
- INCBIN: 将一个文件包含到当前源文件中，而被包含的文件不进行汇编处理

5.2.4.1 AREA

用于定义一个代码段或是数据段。

语法格式：

```
AREA    sectionname {, attr} {, attr} ... attribute
```

其中：

- sectionname: 为所定义的段的名称。
- attr: 该段的属性。具有的属性为:
 - CODE: 定义代码段。
 - DATA: 定义数据段。
 - READONLY: 指定本段为只读, 代码段的默认属性。
 - READWRITE: 指定本段为可读可写, 数据段的默认属性。
 - ALIGN: 指定段的对齐方式为2expression。expression的取值为0~31。
 - COMMON: 指定一个通用段。该段不包含任何用户代码和数据。
 - NOINIT: 指定此数据段仅仅保留了内存单元, 而没有将各初始值写入内存单元, 或者将各个内存单元值初始化为0。

注意：一个大的程序可包含多个代码段和数据段。一个汇编程序至少包含一个代码段。

5.2.4.2 CODE16和CODE32

CODE16告诉汇编编译器后面的指令序列为16位的Thumb指令。

CODE32告诉汇编编译器后面的指令序列为32位的ARM指令。

语法格式：

CODE16

CODE32

注意：CODE16和CODE32只是告诉编译器后面指令的类型，该伪操作本身不进行程序状态的切换。

例:

```
AREA      ChangeState, CODE, READONLY
```

```
ENTRY
```

```
CODE32
```

;下面为32位ARM指令

```
LDR      R0, =start+1
```

```
BX       R0
```

```
.....
```

```
CODE16
```

;下面为16位Thumb指令

```
start    MOV      R1, #10
```

```
.....
```

```
END
```

5.2.4.3 ENTRY

指定程序的入口点。

语法格式：

ENTRY

注意：

一个程序（可包含多个源文件）中至少要有一个ENTRY（可以有多个ENTRY），但一个源文件中最多只能有一个ENTRY（可以没有ENTRY）

5.2.4.4 ALIGN

ALIGN伪操作通过填充0将当前的位置以某种形式对齐。

语法格式：

ALIGN {expr[, offset]}

其中：

- expr：一个数字，表示对齐的单位。这个数字是2的整数次幂，范围在 $2^0 \sim 2^{31}$ 之间。

如果没有指定expr，则当前位置对齐到下一个字边界处。

- Offset：偏移量，可以为常数或数值表达式。

不指定offset表示将当前位置对齐到以expr为单位的起始位置。

例1：

```
short    DCB    1           ;本操作使字对齐被破坏
ALIGN                                ;重新使其为字对齐
MOV      R0, 1
```

例2：

```
ALIGN      8                ;当前位置以2个字的方式对齐
```

5.2.4.5 END

END伪操作告诉编译器已经到了源程序结尾。

语法格式：

END

注意：

每一个汇编源程序都必须包含END伪操作，以表明本源程序的结束。

5.2.4.6 EQU ——也可以用符号“*”表示

EQU伪操作为数字常量、基于寄存器的值和程序中的标号定义一个字符名称。

语法格式：

name EQU expr{, type}

其中：

- name: 为expr定义的字符名称。
- expr: 基于寄存器的地址值、程序中的标号、32位的地址常量或者32位的常量。表达式，为常量。
- type: 当expr为32位常量时，可以使用type指示expr的数据的类型。取值为：
 - CODE32
 - CODE16
 - DATA

例:

abcd	EQU	2	;定义abcd符号的值为2
abcd	EQU	label+16	
			;定义abcd符号的值为(label+16)
abcd	EQU	0x1c, CODE32	
			;定义abcd符号的值为绝对地址
			;值0x1c, 而且此处为ARM指令

5.2.4.7 EXPORT及GLOBAL

声明一个源文件中的符号，使此符号可以被其他源文件引用。

语法格式：

```
EXPORT/GLOBAL symbol {[weak]}
```

其中：

- symbol：声明的符号的名称。（区分大小写）
- [weak]：声明其他同名符号优先于本符号被引用。

例：

```
AREA    example, CODE, READONLY
EXPORT  DoAdd
DoAdd   ADD    R0, R0, R1
```

5.2.4.8 IMPORT及EXTERN

声明一个符号是在其他源文件中定义的。

语法格式：

```
IMPORT  symbol {[weak]}
```

```
EXTERN  symbol {[weak]}
```

其中：

- symbol：声明的符号的名称。

- [weak]:
 - 当没有指定此项时，如果symbol在所有的源文件中都没有被定义，则连接器会报告错误。
 - 当指定此项时，如果symbol在所有的源文件中都没有被定义，则连接器不会报告错误，而是进行下面的操作。
 - 如果该符号被B或者BL指令引用，则该符号被设置成下一条指令的地址，该B或BL指令相当于一条NOP指令。
 - 其他情况下此符号被设置成0。

5.2.4.9 GET及INCLUDE

将一个源文件包含到当前源文件中，并将被包含的文件在其当前位置进行汇编处理。

指令格式：

GET filename

INCLUDE filename

其中：

- filename: 包含的源文件名，可以使用路径信息（可包含空格）。

例：GET d:\arm\file.s

5.2.4.10 INCBIN

将一个文件包含到当前源文件中，而被包含的文件不进行汇编处理
指令格式：

```
INCBIN    filename
```

其中：

- filename：被包含的文件名称，可使用路径信息（不能有空格）。

适用情况：通常使用此伪操作将一个可执行文件或者任意数据包含到当前文件中。

例：INCBIN d:\arm\file.txt

例：编写一具有完整汇编格式的程序，实现冒泡法排序功能。设无符号字数据存放在从0x400004开始的区域，字数据的数目字存放在0x400000中。

```
AREA    SORT, CODE, READONLY
        ENTRY
START
        MOV        R1, #0x400000
LP
        SUBS       R1, R1, #1
        BEQ        EXIT
        MOV        R7, R1
        LDR        R0, =0x400004
LP1
        LDR        R2, [R0], #4
        LDR        R3, [R0]
        CMP        R2, R3
        STRLO      R3, [R0, # -4]
        STRLO      R2, [R0]
        SUBS       R7, R7, #1
        BNE        LP1
        B          LP
EXIT
        END
```

6. ARM汇编中的文件格式

源程序文件	文件名	说 明
汇编程序文件	*. S	用ARM汇编语言编写的ARM程序或Thumb程序。
C程序文件	*. C	用C语言编写的程序代码。
头文件	*. H	为了简化源程序，把程序中常用到的常量命名、宏定义、数据结构定义等等单独放在一个文件中，一般称为头文件。

7. ARM汇编与C的混合编程

内嵌汇编

语法:

_asm

{

指令[;指令] /*注释*/

...

[;指令]

}

_inline void

enable_IRQ(viod)

{

int tmp;

_asm

{

MRS tmp,CPSR

BIC tmp,tmp,#0x80

MSR CPSR_c,tmp

}

}

8.AAPCS

- 2007年ARM公司正式推出了AAPCS标准
 - ARM Archtecture Procedure Call Standard
- AAPCS是ATPCS的改进版
- 目前， AAPCS和ATPCS都是可用的标准

寄存器的使用规则

- 子程序间通过寄存器R0~R3来传递参数。这时，寄存器R0~R3可记作a0~a3。被调用的子程序在返回前无需恢复寄存器R0~R3的内容。
- 在子程序中，使用寄存器R4~R11来保存局部变量。这时，寄存器R4~R11可以记作v1~v8。如果在子程序中使用了寄存器v1~v8中的某些寄存器，则子程序进入时必须保存这些寄存器的值，在返回前必须恢复这些寄存器的值。在Thumb程序中，通常只能使用寄存器R4~R7来保存局部变量。
- 寄存器R12用作过程调用中间临时寄存器，记作IP。在子程序之间的连接代码段中常常有这种使用规则。

寄存器的使用规则（续）

- 寄存器R13用作堆栈指针，记作SP。在子程序中寄存器R13不能用作其他用途。寄存器SP在进入子程序时的值和退出子程序时的值必须相等。
- 寄存器R14称为连接寄存器，记作LR。它用于保存子程序的返回地址。如果在子程序中保存了返回地址，寄存器R14则可以用作其他用途。
- 寄存器R15是程序计数器，记作PC。它不能用作其它用途。

堆栈使用规则

- **ATPCS**规定堆栈为**FD**类型，即满递减堆栈，并且对堆栈的操作是**8**字节对齐。
- 对于汇编程序来说，如果目标文件中包含了外部调用，则必须满足下列条件：
 - （1）外部接口的堆栈必须是**8**字节对齐的。
 - （2）在汇编程序中使用**PRESERVE8**伪指令告诉连接器，本汇编程序数据是**8**字节对齐的。

参数传递规则

- 根据参数个数是否固定，可以将子程序分为参数个数固定的子程序和参数个数可变化的子程序。
- 这两种子程序的参数传递规则是不一样的。

参数个数可变子程序参数传递规则

- 对于参数个数可变的子程序，当参数个数不超过4个时，可以使用寄存器R0~R3来传递参数；当参数超过4个时，还可以使用堆栈来传递参数。
- 在传递参数时，将所有参数看作是存放在连续的内存字单元的字数据。然后，依次将各字数据传递到寄存器R0，R1，R2和R3中。**如果参数多于4个，则将剩余的字数据传递到堆栈中。**入栈的顺序与参数传递顺序相反，即最后一个字数据先入栈。

参数个数固定子程序参数传递规则

- 如果系统不包含浮点运算的硬件部件，浮点参数会通过相应的规则转换成整数参数（若没有浮点参数，此步省略），然后依次将各字数据传送到寄存器R0~R3中。如果参数多于4个，将剩余的字数据传送堆栈中，入栈的顺序与参数顺序相反，即最后一个字数据先入栈。在参数传递时，将所有参数看作是存放在连续的内存字单元的字数据。

子程序结果返回规则

- 子程序中结果返回的规则如下：
 - 结果为一个32位整数时，可以通过寄存器R0返回；
 - 结果为一个64位整数时，可以通过寄存器R0和R1返回；
 - 结果为一个浮点数时，可以通过浮点运算部件的寄存器f0、d0或s0来返回；
 - 结果为复合型浮点数（如复数）时，可以通过寄存器f0~fn或d0~dn来返回；
 - 对于位数更多的结果，需要通过内存来传递。

ARM编译器保有的特定关键字

- ARM编译器支持一些对ANSI C进行扩展的关键词。这些关键词用于声明变量、声明函数、对特定的数据类型进行一定的限制。

用于声明函数的关键词 (双下划线起头)

- `__asm`, 内嵌汇编
- `__inline`, 内联展开
- `__irq`, 声明IRQ或FIQ的ISR
- `__pure`, 函数不修改该函数之外的数据
- `__softfp`, 使用软件的浮点连接件
- `__swi`, 软中断函数
- `__swi_indirect`, 软中断函数

用于声明变量的关键词

- **register**
 - 声明一个变量，告诉编译器尽量保存到寄存器中。
- **_int64**
 - 该关键词是long long的同义词。
- **_global_reg**
 - 将一个已经声明的变量分配到一个全局的整数寄存器中。

ARM汇编程序设计

- 文件格式

 - 汇编文件 *.s

 - 引入文件 *.INC

 - C程序 *.C

 - 头文件 *.H

- 汇编语句格式

 - [标号] <指令|条件|S> <操作数>[: 注释]

标号顶格写，指令不可顶格书写

标号后没有：

程序中可以有空行

分行用\ (用于比较长的语句)