ARM指令集

李曦

llxx@ustc.edu.cn

内容提要

- **ARM指令集**
  - **ARM指令集分类与指令格式**
  - **ARM指令的寻址方式**
  - ***ARM指令集详解***
  - **Thumb指令及应用**
- **ARM程序设计基础**
  - 系统初始化
  - 调试环境

5.1 ARM微处理器的指令集概述

5.1.1 ARM微处理器的指令的分类与格式

- Load-store结构
- 指令分类
  - ◆数据处理指令 – 使用和改变寄存器的值
  - ◆数据传送指令 – 把存储器的值拷贝到寄存器中 (load)或把寄存器中的值拷贝到存储器中(store)
  - ◆控制流指令
    - ➢分支
    - ➢分支和链接，保存返回的地址,以恢复最先的次序
    - ➢陷入系统代码
  - ◆程序状态寄存器（PSR）处理指令
  - ◆协处理器指令
  - ◆异常产生指令

ARM指令集

- 跳转指令:
  - B　　跳转指令。　　BL　　带返回的跳转指令。
  - BLX　带返回和状态切换的跳转指令。　BX　　带状态切换的跳转指令。
- 数据处理指令
  - MOV　　数据传送指令　　MVN　　数据取反传送指令
  - CMP　　比较指令　　CMN　　反值比较指令
  - TST　　位测试指令　　TEQ　　相等测试指令
  - ADD　　加法指令28　　ADC　　带进位加法指令
  - SUB　　减法指令　　SBC　　带借位减法指令
  - RSB　　逆向减法指令　　RSC　　带借位的逆向减法指令
  - AND　　逻辑与指令　　ORR　　逻辑或指令
  - EOR　　逻辑异或指令　　BIC　　位清除指令
- 乘法指令与乘加指令
  - MUL　　32 位乘法指令　　MLA　　32 位乘加指令
  - SMULL　64 位有符号数乘法指令　　SMLAL　64 位有符号数乘加指令
  - UMULL　64 位无符号数乘法指令　　UMLAL　64 位无符号数乘加指令
- 程序状态寄存器存取指令
  - MRS 程序状态寄存器到通用寄存器的数据传送指令。
  - MSR 通用寄存器到程序状态寄存器的数据传送指令。

ARM指令集(续)

- 寄存器加载/存储指令：
  - LDR 字数据加载指令
  - LDRB 字节数据加载指令
  - LDRH 半字数据加载指令
  - STR 字数据存储指令
  - STRB 字节数据存储指令
  - STRH 半字数据存储指令
  - LDM 连续数据加载指令
  - STM 连续数据存储指令
- 数据交换指令 ：
  - SWP 字数据交换指令
  - SWPB 字节数据交换指令
- 移位元指令：
  - LSL 逻辑左移
  - ASL 算术左移
  - LSR 逻辑右移
  - ASR 算术右移
  - ROR 循环右移
  - RRX 带扩充的循环右移
- 协处理器指令
  - CDP 协处理器数据操作指令
  - LDC 协处理器数据加载指令
  - STC 协处理器数据存储指令
  - MCR ARM处理器寄存器到协处理器寄存器的数据传送指令
  - MRC 协处理器寄存器到ARM处理器寄存器的数据传送指令

5.1.2 指令的条件域

- 所有的ARM指令都可以条件执行
  - 指令的执行与否取决于CPSR寄存器的N, Z, C and V标志位
  - 所有的Thumb指令都可以解压成全部条件指令
- 每一条ARM指令包含4位的条件码，位于指令的最高4位[31:28]。条件码共有16种，每种条件码可用两个字符表示，这两个字符可以添加在指令助记符的后面和指令同时使用。
  - 例如，跳转指令B可以加上后缀EQ变为BEQ表示“相等则跳转”，即当CPSR中的Z标志置位时发生跳转。

- 在16种条件标志码中，只有15种可以使用。

指令的条件码(15种)

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

条件执行

- 条件执行可避免使用分支指令
- Example

```
        CMP r0, #5      ;
        BEQ BYPASS      ; if (r0!=5) {
        ADD r1, r1, r0  ;    r1:=r1+r0-r2
        SUB r1, r1, r2  ; }
BYPASS:         …
```

注意：条件不成立时，该指令被忽略，等于执行了一条NOP。

使用条件执行

```
        CMP r0, #5      ;
        ADDNE r1, r1, r0;
        SUBNE r1, r1, r2;
        …
```

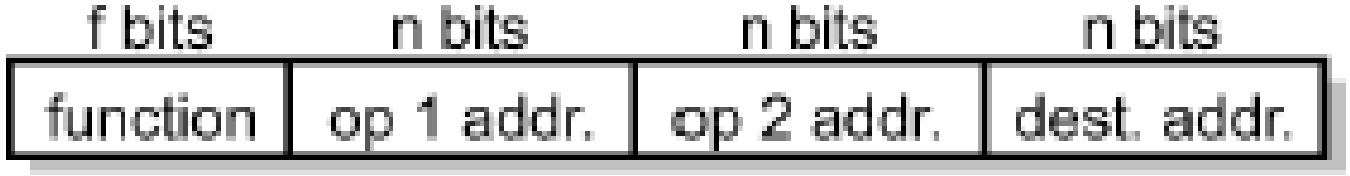
Note: add 2 –letter condition after the 3-letter opcode

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

CMP r0, r1
CMPEQ r2, r3
ADDEQ r4, r4, #1
```

指令格式

- 3地址指令格式
  - 在ARM状态中使用



- 2地址指令格式
  - 在ARM和 THUMB 状态下使用

ARM7TDMI指令集编码

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 8	7 5	4	3 0	
Cond	0 0	I	Opcode	S	Rn	Rd	Operand 2				Data Processing PSR Transfer
Cond	0 0 0 0 0 0		A	S	Rd	Rn	Rs	1 0 0 1		Rm	Multiply
Cond	0 0 0 1 0		B	0 0	Rn	Rd	0 0 0 0	1 0 0 1		Rm	Single Data Swap
Cond	0 1	I	P U B W L		Rn	Rd	offset				Single Data Transfer
Cond	0 1 1		XXXXXXXXXXXXXXXXXXX						1	XXXX	Undefined
Cond	1 0 0		P U S W L		Rn	Register List					Block Data Transfer
Cond	1 0 1		L		offset						Branch
Cond	1 1 0		P U N W L		Rn	CRd	CP#	offset			Coproc Data Transfer
Cond	1 1 1 0		CP Opc		CRn	CRd	CP#	CP	0	CRm	Coproc Data Operation
Cond	1 1 1 0		CP Opc	L	CRn	Rd	CP#	CP	1	CRm	Coproc Register Transfer
Cond	1 1 1 1		ignored by processor								Software Interrupt

ARM存储器访问指令——单寄存器存储

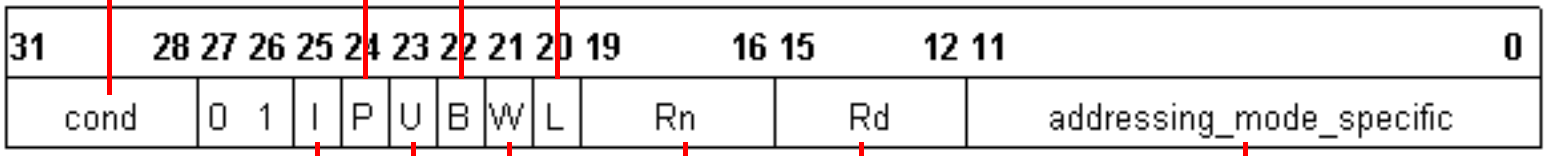
- LDR和STR——字和无符号字节加载/存储指令编码

B为1表示字节访问，为0表示字访问

P表示前/后变址

指令执行的条件码

L用于区别加载（L为1）或存储（L为0）



I为0时，偏移量为12位立即数，为1时，偏移量为寄存器移位

U表示加/减

为指令的寻址方式

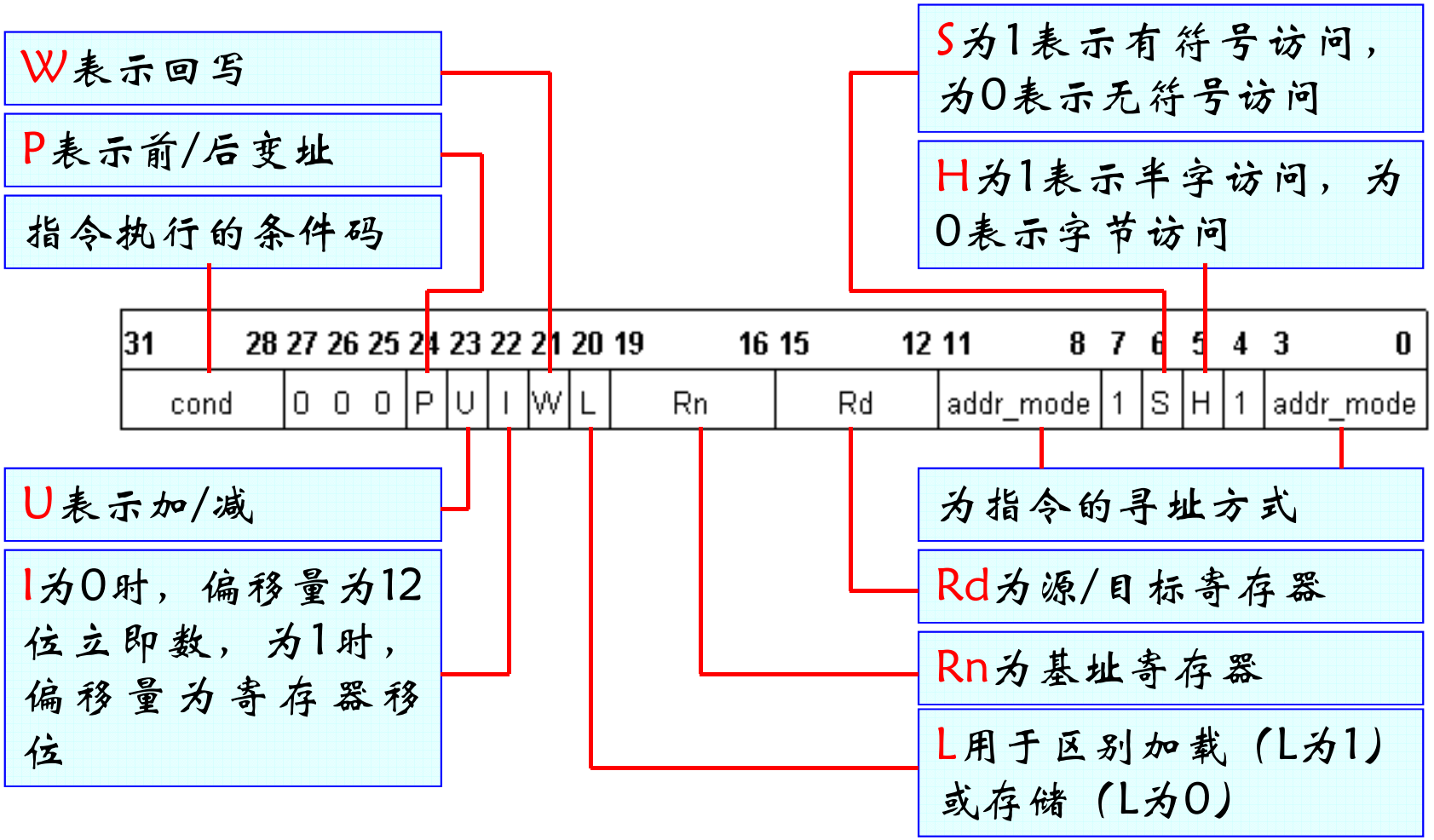
Rd为源/目标寄存器

Rn为基址寄存器

W表示回写

ARM存储器访问指令——单寄存器存储

- LDR和STR——半字和有符号字节加载/存储指令编码



5.2 ARM指令的寻址方式

- 立即寻址
- 寄存器寻址
- 寄存器间接寻址
- 基址变址寻址
- 多寄存器寻址
- 相对寻址
- 堆栈寻址

5.2.1 立即寻址

- 立即寻址也叫立即数寻址
- 操作数本身就在指令中给出，只要取出指令也就取到了操作数。这个操作数被称为立即数。
- 例如：

```
ADD R0, R0, #1          ; R0←R0+1
ADD R0, R0, #0x3f       ; R0←R0+0x3f
```

  - 第二个源操作数即为立即数，要求以“#”为前缀
    - 对于以十六进制表示的立即数，还要求在“#”后加上“0x”或“&”

有效立即数问题

- 并不是所有立即数都是有效的
- 在**32**位指令编码中存放**32**位立即数的方法是：
  - 在**ARM**数据处理指令中，第二操作数域**12**位
  - 每个立即数采用一个**8**位的常数（**bit[7:0]**）循环右移偶数位而间接得到，其中循环右移的位数由一个**4**位二进制（**bit[11:8]**）的两倍表示
    - 如果立即数记作**<immediate>**，**8**位常数记作**immed_8**，**4**位的循环右移值记作**rotate_imm**，有效的立即数是由一个**8**位的立即数循环右移偶数位得到，可以表示成：
- <immediate>=immed_8 循环右移（2×rotate_imm）

如：**mov R4，　＃0x8000 000A**

；　＃0x8000 000A由0xA8循环右移0x2位得到

5.2.2 寄存器寻址

- 寄存器中的数值作为操作数
  - 这种寻址方式是各类微处理器经常采用的一种方式，也是一种执行效率较高的寻址方式。
  - 以下指令：

```
ADD R0，R1，R2   ；R0←R1＋R2

ADD R3，R2，R1，LSL #3
    ;R3=R2+R1<<3
```

ARM指令——第2个操作数

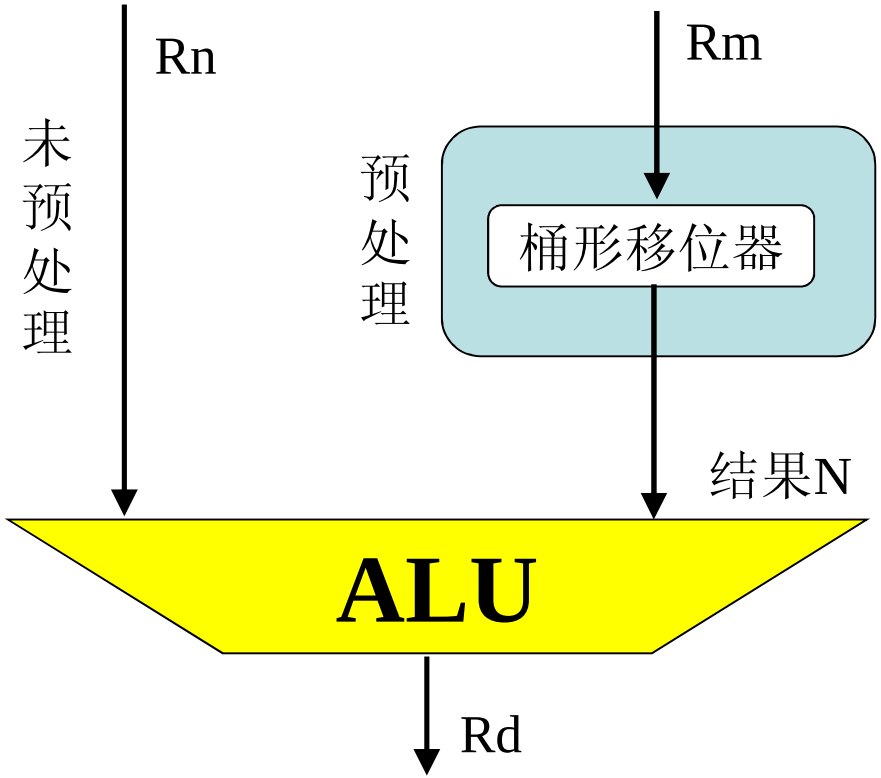
ARM指令的基本格式如下：

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

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

- #immed_8r——常数表达式，立即寻址；
- Rm——寄存器寻址方式；
- Rm,shift——寄存器移位方式；

ARM指令——第2个操作数



- Rm,shift——寄存器移位方式

将寄存器的移位结果作为操作数，但Rm值保持不变（移位操作不消耗额外的时间）。

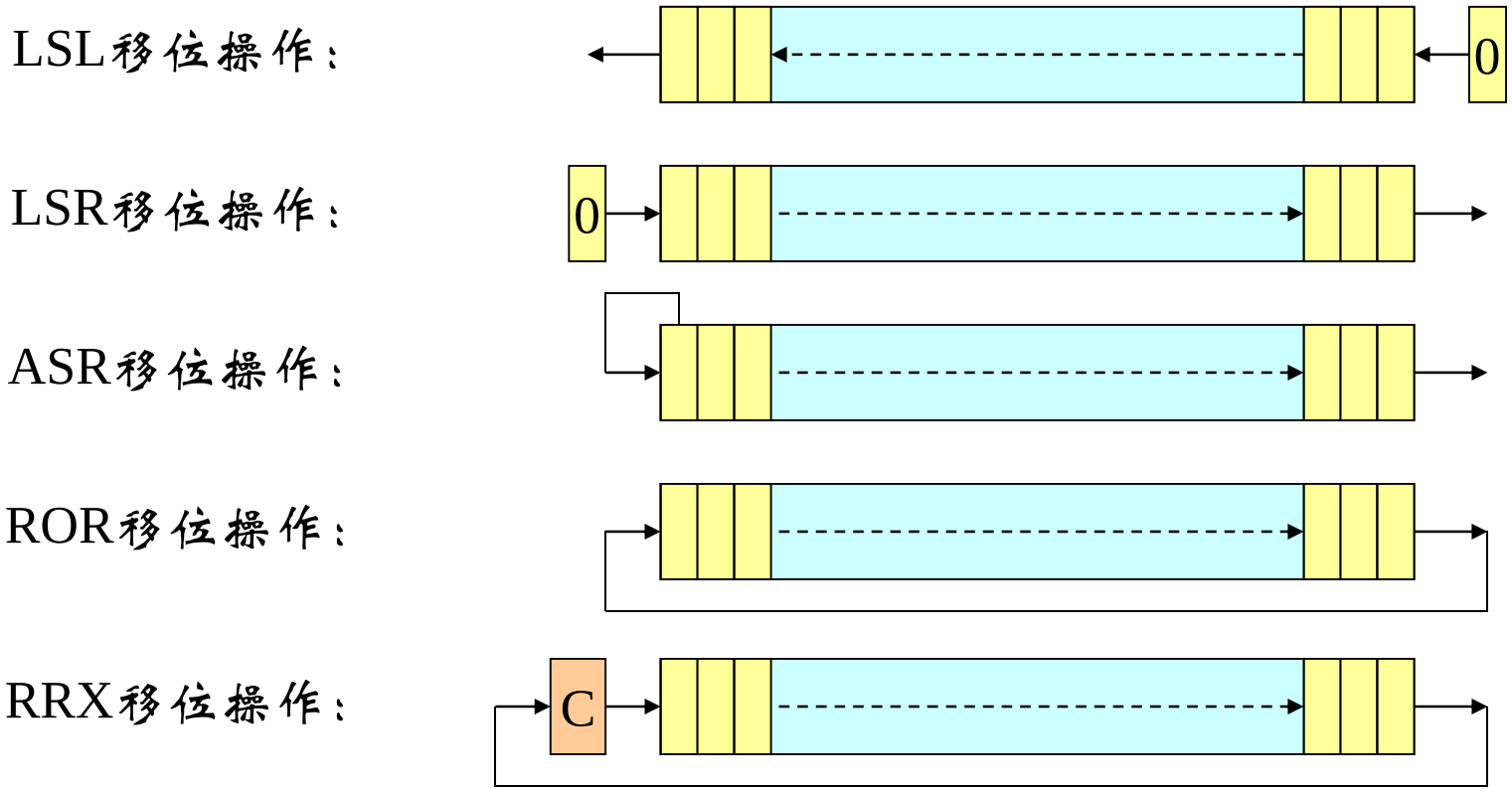
移位方法如下：

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。

桶形移位器操作

助记符	说明	移位操作	结果	Y值
LSL	逻辑左移	x LSL y	x<<y	#0-31 or Rs
LSR	逻辑右移	x LSR y	(unsigned)x>>y	#1-32 or Rs
ASR	算术右移	x ASR y	(signed)x>>Y	#1-32 or Rs
ROR	算术左移	x ROR y	((unsigned)x>>y|(x<<32-y))	#1-32 or Rs
RRX	扩展的循环右移	x RRX y	(c flag<<31)| ((unsigned)x>>1)	none

桶形移位器操作（续）



5.2.3 寄存器间接寻址

寄存器间接寻址：以寄存器中的值作为操作数的地址，而操作数本身存放在存储器中。

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

ADD R0，R1，[R2]     ；R0←R1＋[R2]
                     ；可用??? ，查
```

5.2.4 基址变址寻址

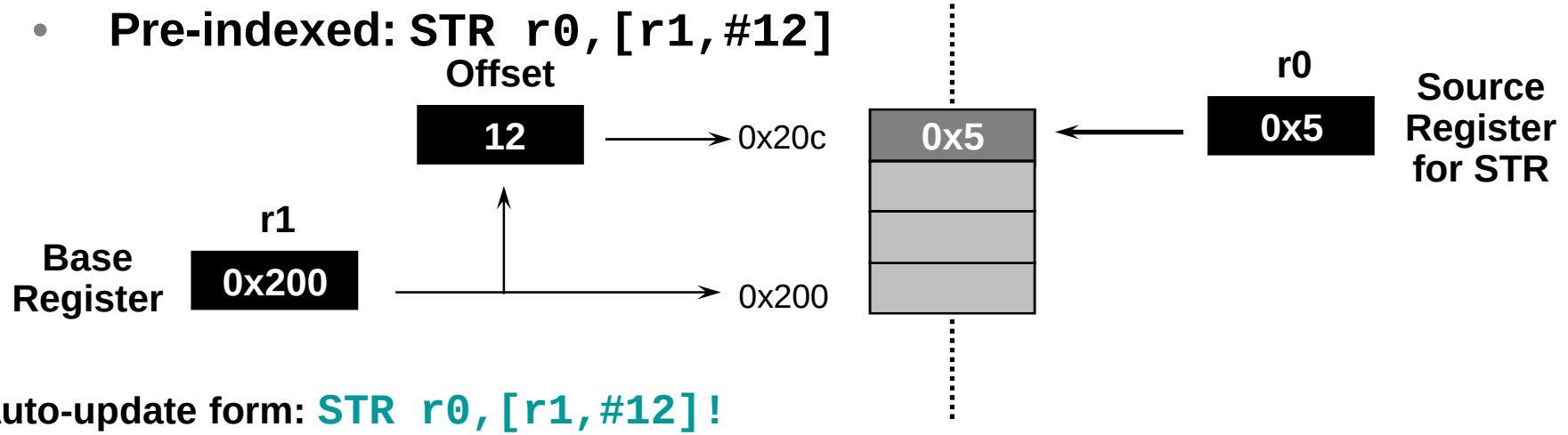
- 将基址寄存器的内容与指令中给出的地址偏移量相加，从而得到一个操作数的有效地址。
  - “变址”，改变基址
  - 变址寻址方式常用于访问某基地址附近的地址单元。
- 变址方式：

  前变址、自动变址、后变址、偏移变址

```
LDR R0，[R1，＃4]   ；R0←[R1＋4],前
LDR R0，[R1，＃4]！ ；R0←[R1＋4]、R1←R1＋4，自
LDR R0，[R1]，＃4   ；R0←[R1]、R1←R1＋4，后
LDR R0，[R1，R2]    ；R0←[R1＋R2]，偏
```

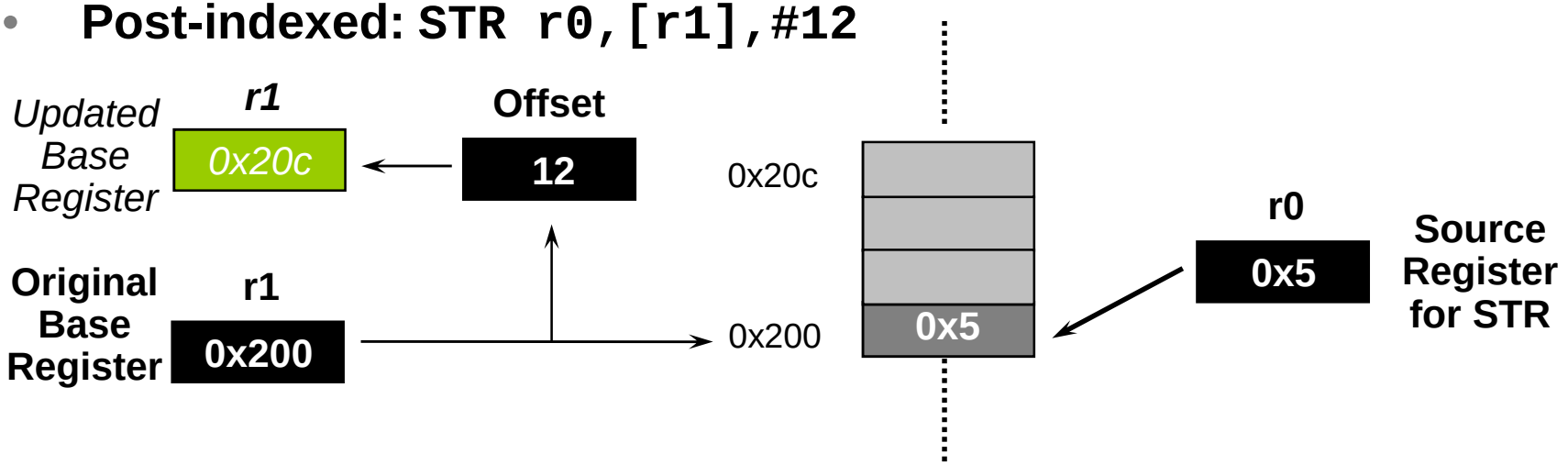
Pre or Post Indexed Addressing?

- **Pre-indexed: `STR r0,[r1,#12]`**



Auto-update form: `STR r0,[r1,#12]!`

- **Post-indexed: `STR r0,[r1],#12`**



5.2.5 多寄存器寻址

采用多寄存器寻址方式，一条指令可以完成多个寄存器值的传送。这种寻址方式可以用一条指令完成传送最多16个通用寄存器的值。

以下指令：

```
LDMIA R0，{R1，R2，R3，R4}      ；R1←[R0]
                                ；R2←[R0＋4]
                                ；R3←[R0＋8]
                                ；R4←[R0＋12]
```

该指令的后缀IA表示在每次执行完加载/存储操作后，R0按字长度增加，因此，指令可将连续存储单元的值传送到R1～R4。

5.2.6 相对寻址

与基址变址寻址方式相类似，相对寻址以程序计数器PC的当前值为基地址，指令中的地址标号作为偏移量，将两者相加之后得到操作数的有效地址。

以下程序段完成子程序的调用和返回，跳转指令BL采用了相对寻址方式：

```
        BL      NEXT            ；跳转到子程序NEXT处执行
        ……
NEXT：
        ……
        MOV     PC，LR          ；从子程序返回
```

5.2.7 堆栈寻址

堆栈是一种数据结构，按先进后出（First In Last Out，FILO）的方式工作，使用一个称作“堆栈指针”的专用寄存器（SP）指示当前的操作位置，堆栈指针总是指向栈顶。

按SP指向的位置，分：

满堆栈(Full Stack)：当堆栈指针指向最后压入堆栈的数据时；

空堆栈(Empty Stack)：当堆栈指针指向下一个将要放入数据的空位置时；

根据堆栈的生成方式，分：

递增堆栈(Ascending Stack)：当堆栈由低地址向高地址生成时；

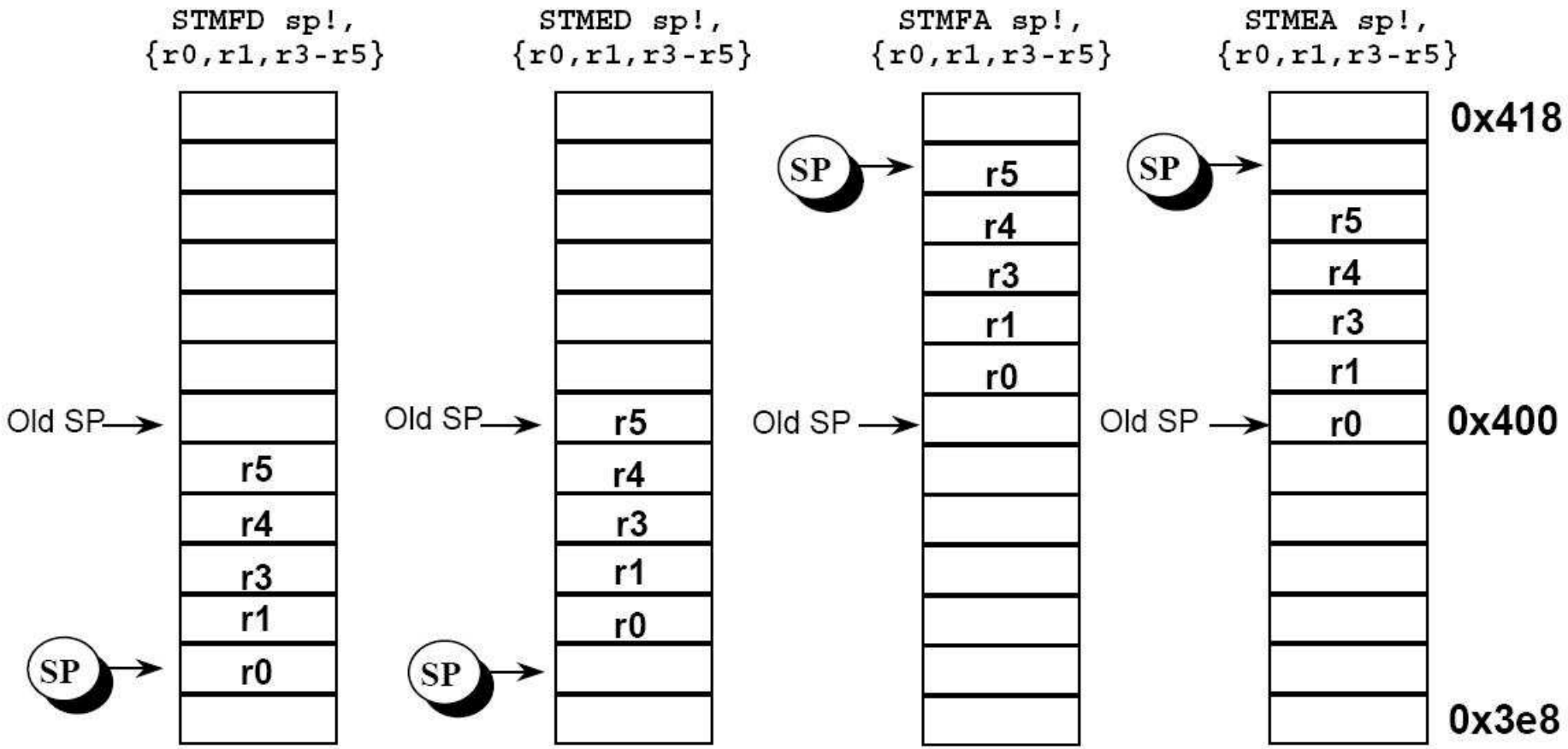
递减堆栈(Decending Stack)：当堆栈由高地址向低地址生成时；

ARM微处理器支持的堆栈工作方式

- 满递增堆栈FA
  - 堆栈指针指向最后压入的数据，且由低地址向高地址生成。
- 满递减堆栈FD
  - 堆栈指针指向最后压入的数据，且由高地址向低地址生成。
- 空递增堆栈EA
  - 堆栈指针指向下一个将要放入数据的空位置，且由低地址向高地址生成。
- 空递减堆栈ED
  - 堆栈指针指向下一个将要放入数据的空位置，且由高地址向低地址生成。
- 堆栈操作指令
  - **LDM**
  - **STM**

```
Ucosii移植：#define  OS_STK_GROWTH  1 /* 堆栈由高地址向低地址增长
```

栈操作图示


STMFD sp!, {r0,r1,r3-r5}
STMED sp!, {r0,r1,r3-r5}
STMFA sp!, {r0,r1,r3-r5}
STMEA sp!, {r0,r1,r3-r5}
Old SP
SP
r5
r4
r3
r1
r0
0x418
0x400
0x3e8

子程序调用

```
BL SUBR                    ; branch to SUBR
…                          ; return to here
…
SUBR
…                          ; subroutine entry point
…
MOV pc, r14                ; return
```

- 转跳时返回地址存放在r14中，子程序返回时只要把原先保存在r14的地址装入pc即可（没有类似RET的语句）
- 如果子程序要使用r14的话，需要保存存在里面的返回地址。

程序嵌套调用

子程序里再调用子程序时会改变r14，因此遇到多层程序调用时得保存r14

```
BL SUB1                          ; branch to SUB!
…
…

SUB1
STMFA r13!, {r0-r2, r14}         ; save regs
BL SUB2
…
…
LDMFA r13!, {r0-r2, pc}          ; return

SUB2
…
MOV pc, r14                      ; return
```

软中断

```
SWI SWI_WriteC        ; output character in r0
SWI SWI_Exit          ; return to monitor
```

- **SWI**将处理器置于监控（**SVC**）模式，从地址**0x08**开始执行指令。
- 系统调用类型：这些中断调用的作用与实现的系统有关。
  - 立即数“SWI_WriteC”和“SWI_Exit”出现在SWI指令的0~23比特，它被操作系统用来判断用户程序调用系统例程的类型，由中断服务程序决定是否使用。
  - 当指令中**24**位的立即数被忽略时，用户程序调用系统例程的类型由通用寄存器**R0**的内容决定，同时，参数通过其他通用寄存器传递。

BKPT

- 断点中断指令，用于产生软件断点，供调试程序用。

软中断服务程序

SWI异常中断处理程序中，取出SWI指令中立即数（服务类型）的步骤为：

- 首先确定引起软中断的SWI指令是ARM指令还是Thumb指令，这可通过对SPSR访问得到；
- 然后取得该SWI指令的地址，这可通过访问LR寄存器得到；
- 接着读出该SWI指令，分解出立即数。

```
SWI_Handler
    STMFD   SP!, {R0-R3, R12, LR}   ; 现场保护
    MRS     R0, SPSR                ; 读取SPSR
    STMFD   SP!, {R0}               ; 保存SPSR
    TST     R0, #0x20               ; 测试T标志位
    LDRNEH  R0, [LR,#-2]            ; 若是Thumb指令，读取指令码(16位)
    BICNE   R0, R0, #0xFF00         ; 取得Thumb指令的8位立即数（低8位）
    LDREQ   R0, [LR,#-4]            ; 若是ARM指令，读取指令码(32位)
    BICEQ   R0, R0, #0xFF000000     ; 取得ARM指令的24位立即数（低23位）
        ...
    LDMFD   SP!, {R0-R3, R12, PC}^ ; SWI异常中断返回
```

NE：Z=0
EQ：Z=1

异常处理

- 处理器允许多个异常同时发生，按固定优先级进行处理。
- 进入异常处理的基本步骤：
  - 将下一条指令的地址存入相应连接寄存器LR
  - 将CPSR复制到相应的SPSR中
  - 根据异常类型，强制设置CPSR的运行模式位
  - 强制PC从相关的异常向量地址取下一条指令执行，从而跳转到相应的异常处理程序处
    - 如果异常发生时，处理器处于Thumb状态，则当异常向量地址加载入PC时，处理器自动切换到ARM状态。
- 从异常返回：
  - 将连接寄存器LR的值减去相应的偏移量后送到PC中
  - 将SPSR复制回CPSR中
  - 若在进入异常处理时设置了中断禁止位，要在此清除。

状态寄存器操作指令

下面是从特殊寄存器到普通寄存器的传输指令

```
MRS r0, CPSR   ; r0 := CPSR
MRS r3, SPSR   ; r3 := SPSR
```

下面是把数据传输到特殊寄存器的指令

```
MSR CPSR, R0           ; 复制 R0 到 CPSR 中
MSR SPSR, R0           ; 复制 R0 到 SPSR 中
MSR CPSR_flg, R0       ; 复制 R0 的标志位到 CPSR 中
MSR CPSR_flg, #0xF0000000      ; NVCZ=“1111”
 MSR     SPSR_all, Rm          ;
 MSRNE CPSR_ctl, Rm            ;
 MRS     Rd, CPSR
```

- 在 user模式中，不能改变CPSR的控制位，只能改变条件标志。在其他模式中，可获得整个 CPSR。
- 在 user模式中，不能尝试访问 SPSR，因为它不存在!

使用特殊寄存器指令的例子

设置 V 标志:

```
MSR CPSR_flg, #&10000000      ; 设置V标志但
                              ; 不影响控制位
```

要改变模式:

```
MRS   R0, CPSR_all            ; 复制PSR
BIC   R0, R0, #&1F            ; 清除模式位
ORR   R0, R0, #new_mode       ; 把模式位设置为新模式
MSR   CPSR_all, R0            ; 写回PSR，变更模式
```

协处理器指令

- **ARM**微处理器可支持多达**16**个协处理器
  - 每个协处理器只执行针对自身的协处理指令
- **ARM**处理器初始化协处理器的数据处理操作
  - 在**ARM**处理器的寄存器和协处理器的寄存器之间传送数据
    - **MCR** **ARM**处理器寄存器→协处理器寄存器的数据传送指令
    - **MRC** 协处理器寄存器→**ARM**处理器寄存器的数据传送指令
  - 在协处理器的寄存器和存储器之间传送数据。
    - **CDP** 协处理器数据操作指令
    - **LDC** 协处理器数据加载指令
    - **STC** 协处理器数据存储指令
  - 若协处理器不能成功完成传送操作，则产生未定义指令异常

- CDP 指令用于 ARM 处理器通知 ARM 协处理器执行特定的操作
- 格式：

CDP {条件} 协处理器编码，协处理器操作码 1，目的寄存器，源寄存器 1，源寄存器 2，协处理器操作码 2；

其中协处理器操作码 1 和协处理器操作码 2 为协处理器将要执行的操作，目的寄存器和源寄存器均为协处理器的寄存器，指令不涉及 ARM 处理器的寄存器和存储器。

指令示例：
CDP P3，2，C12，C10，C3，4 ；该指令完成协处理器P3的初始化

ARM920T协处理器

- CP14 调试通信通道协处理器
  - 运行于 ARM 内核上的应用程序与运行于主机上的调试器之间的通信
- CP15 系统控制协处理器
  - 配置和控制caches、MMU、保护系统、配置时钟模式（在bootloader时钟初始化用到）
    - Cache清空
    - MMU初始化、enable
    - TLB清空
    - 。 。 。

协处理器CP15

Register	Read	Write
0	ID code [a]	Unpredictable
0	Cache type [a]	Unpredictable
1	Control	Control
2	Translation table base	Translation table base
3	Domain access control	Domain access control
4	Unpredictable	Unpredictable
5	Fault status [b]	Fault status [b]
6	Fault address	Fault address
7	Unpredictable	Cache operations
8	Unpredictable	TLB operations
9	Cache lockdown [b]	Cache lockdown [b]
10	TLB lockdown [b]	TLB lockdown [b]
11	Unpredictable	Unpredictable
12	Unpredictable	Unpredictable
13	FCSE PID	FCSE PID
14	Unpredictable	Unpredictable
15	Test configuration	Test configuration

a. Register location 0 provides access to more than one register. The register accessed depends on the value of the opcode_2 field. See the register description for details.
b. Separate registers for instruction and data. See the register description for details.

ARM伪汇编指令ADR

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。通常，编译器用一条ADD指令或SUB指令来实现该ADR伪指令的功能，若不能用一条指令实现，则产生错误，编译失败。

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     r1,pc,#0x3c
        ...
        ...
0x64    MOV     r0,r14
        ...
```

ADR伪指令被汇编成一条指令

ARM伪汇编指令LDR

向寄存器加载立即数：当立即数的大小超出MOV或MVN指令能操作的范围时，生成额外的代码（利用pc得到立即数）。

注意：这样使用的话，代码的运行地址必须在Link时固定，不能移动。

```
LDR r1,=0xfff   ; loads 0xfff into r1
                ; =>      LDR r1,[pc,offset_to_litpool]
                ;           ...
                ; litpool DCD 0xfff
LDR r2,=place   ; loads the address of “place” into r2
                ;
                ; =>      LDR r2,[pc,offset_to_litpool]
                ;           ...
                ; litpool DCD place
```

下面的代码应该用MOV，写成LDR时被汇编器自动转回MOV语句

```
LDR r3,=0xff0       ; loads 0xff0 into r3
                    ; => MOV r3,#0xff0
```

ARM伪汇编指令NOP

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。

```
    mov     R1,#0x1234
Delay
    NOP                     ;空操作
    NOP
    NOP
    SUBS    R1,R1,#1        ;循环次数减一
    BNE     Delay           ;如果循环没有结束，跳转Delay继续
    MOV     PC,LR           ;子程序返回
```

简单的ARM程序

```
;文件名：TEST1.S
;功
;说                                    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
```

使用“；”进行注释

标号顶格写

实际代码段

声明文件结束

Example: Hello ARM World!

```
                AREA    HelloW,CODE,READONLY  ;声明代码区
SWI_WriteC      EQU  &0                       ；输出r0中的字符
SWI_Exit        EQU  &11                      ；程序结束
                ENTRY                         ；代码入口
START       ADR      r1,TEXT                  ；r1---“Hello World”
LOOP        LDRB     r0,[r1],#1               ;读取下一字节
            CMP      r0,#0                    ；检查文本终点
            SWINE    SWI_WriteC               ；若非终点，则打印
            BNE      LOOP                     ；并返回LOOP
            SWI      SWI_Exit                 ；执行结束
TEXT        =        “Hello World”,&0a,&0d,0
            END                               ；程序结束
```

ARM ADS汇编伪指令

- ALIGN
  地址对齐（2^n边界）
- AREA
  指定Section
- CODE16
  指示下面是Thumb（16-bit）代码
- CODE32
  指示下面是ARM（32-bit）代码
- DATA
  指示当前代码段里的标号是存放数据的
- DCB，DCD
  定义初始化的数据
- ENTRY
  程序入口
- EQU
  为常量起名
- EXPORT
  定义一个可以在其他模块访问的名称
- EXTERN，IMPORT
  申明一个存在于其他模块的名称
- GET
  相当于C中的#include
- OPT
  打印控制
- SETS，SETA
  设置字符串变量和数据变量
- SPACE
  申请空白数据空间

APCS(ARM Procedure Call Standard)

- 对寄存器使用的限制
- 使用栈的惯例
  - **APCS**规定数据栈为**FD**（满递减）类型
- 在函数调用之间传递/返回参数
  - 当参数不超过**4**个时，可以使用寄存器**R0~R3**来传递参数；
- 可以被‘回溯’的基于栈的结构的格式，用来提供从失败点到程序入口的函数(和给予的参数)的列表

- 开发一个系统，不要求一定要符合APCS。

寄存器命名

寄存器号	APCS名字	意义
R0	a1	可以改变的输入变量
R1	a2	可以改变的输入变量
R2	a3	可以改变的输入变量
R3	a4	可以改变的输入变量
R4	v1	不可改变的输入变量
R5	v2	不可改变的输入变量
R6	v3	不可改变的输入变量
R7	v4	不可改变的输入变量
R8	v5	不可改变的输入变量
R9	v6	不可改变的输入变量
R10	sl	栈限制
R11	fp	桢指针
R12	ip	内部过程调用寄存器
R13	sp	栈指针
R14	lr	连接寄存器
R15	pc	程序计数器

BootLoader

- 在系统加电后，CPU 将首先执行 Boot Loader 程序
  - 在操作系统内核运行之前，初始化硬件设备、建立内存空间的映射图，将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核准备好正确的环境。
- Boot Loader 的两种操作模式
  - 启动加载模式
    - 将操作系统加载到 RAM 中运行
  - 下载模式
    - 通过串口连接或网络连接等通信手段从主机（Host）下载文件，如：内核映像和根文件系统映像等

BOOT的一般步骤

- 设置中断向量表
- 初始化存储设备
- 初始化堆栈
- 初始化用户执行环境
- 调用主应用程序

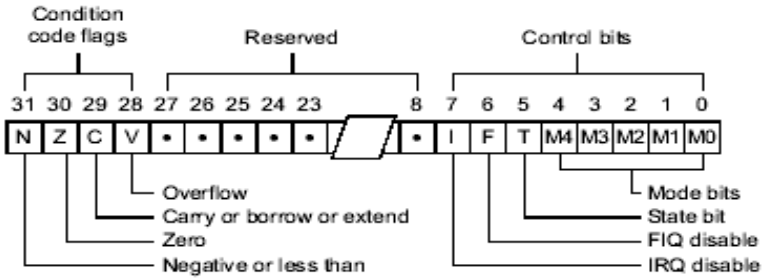
设置中断向量表

- 每个中断只占据向量表中1个字的存储空间，放置一条ARM跳转指令；
- 中断向量表的程序实现通常如下表示：
- AREA Boot ,CODE, READONLY
  - ENTRY
  - B??? ResetHandler
  - B??? UndefHandler
  - B??? SWIHandler
  - B??? PreAbortHandler
  - B??? DataAbortHandler
  - B
  - B??? IRQHandler
  - B??? FIQHandler
  - 其中关键字ENTRY是指定编译器保留这段代码，因为编译器可能会认为这是一段亢余代码而加以优化。链接的时候要确保这段代码被链接在0地址处，并且作为整个程序的入口。

初始化存储设备

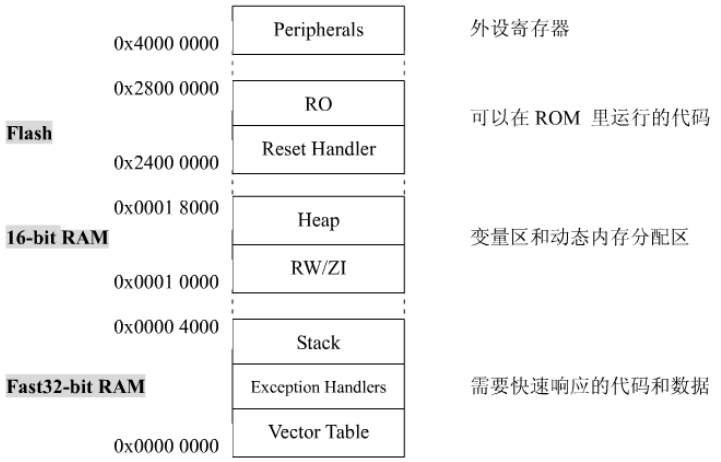
初始化堆栈

- 对程序中需要用到的每一种模式都要给SP定义一个堆栈地址
  - 方法：改变状态寄存器的状态位，使处理器切换到不同的状态，然后给SP赋值。
    - 注意：不要切换到User模式进行User模式的堆栈设置
- 堆栈初始化的代码示例
  - mrs r0,cpsr ； 读取cpsr寄存器的值
  - bic r0,r0,#MODEMASK ； 把模式位清零
  - orr r1,r0,#UNDEFMODE|NOINT
  - msr cpsr_cxsf,r1 ;UndefMode
  - ldr sp,=UndefStack
- 堆栈地址的定义一般如下：
  - ^ (_ISR_STARTADDRESS-0x1400)
  - UserStack # 1024 ;#=field,定义一个数据域，长度为1024
  - SVCStack # 1024
  - UndefStack # 1024
  - AbortStack # 1024
  - IRQStack # 1024
  - FIQStack # 0



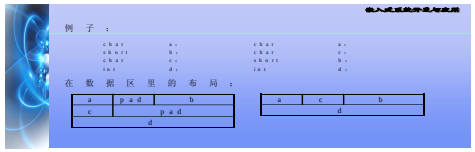
初始化用户执行环境

- 完成必要的从ROM到RAM的数据传输和内容清零
  - 主要是把RO、RW、ZI三段拷贝到指定的位置
- 一个ARM映像文件构成
  - RO，代码段，可在ROM、RAM中
  - RW，已初始化的全局变量，需在RAM中
  - ZI，未初始化的全局变量，需在RAM中
- 调用主应用程序
  - IMPORT main
  - B??? Main

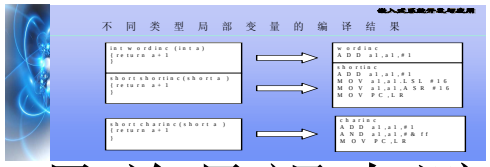


一、变量定义

- 在变量声明的时候，最好把所有相同类型的变量放在一起定义——对齐，这样可以优化存储器布局。由下例可以看出：



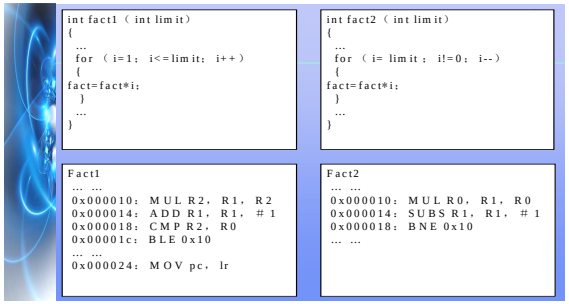
- 对于局部变量类型的定义，使用short或char来定义变量并不是总能节省存储空间。有时使用32位int或unsinged int局部变量更有效率一些，如下图所示：



- 变量定义中，为了精简程序，程序员总是竭力避免使用冗余变量。但有时使用冗余变量可以减少存储器访问的次数，这可以提高系统性能。

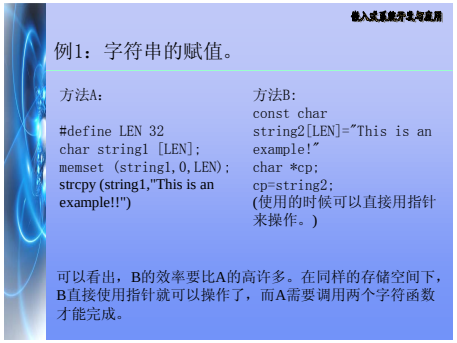
二、循环条件

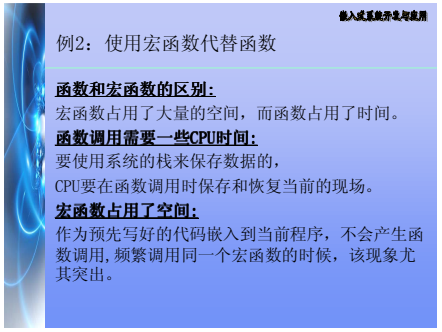
- 计数循环是程序中十分常用的流程控制结构，一般有以下两种形式：
- for （loop=1；loop<=limit；loop++）
- for （loop=limit；loop!=0；loop--）
- 这两种循环形式在逻辑上并没有效率差异，但是映射到具体的体系结构中时，就产生了很大的不同，如下图所示。



三、以空间换时间

➢计算机程序中最大的矛盾是空间和时间的矛盾，从这个角度出发逆向思维来考虑程序的效率问题，比如若系统的实时性要求很高，内存还有剩余，则我们就有可以用以空间换时间的方法来提高程序执行的效率。





四、数学方法解决问题

- 所以在编写程序的时候，适当地采用一些数学方法会对程序的执行效率有数量级的提高，如下例所示：

方法1：	方法2：

int i,,j;
For (i=1;i<=100;i=++){
j+=i;
}
	int i;
	i=(100*(1+100))/2

五、使用位操作

- 一般的位操作是用来控制硬件的，或者做数据变换使用，但是，灵活的位操作可以减少除法和取模运算，有效地提高程序运行的效率, 如下例所示：

方法1：	方法2：
int I,J;	int I,J;
I=257/8;	I=257>>3;
J=456%32;	J=456-(456>>5<<5);


Thank You

寄存器命名

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_abt	SPSR_und	SPSR_irq	SPSR_fiq