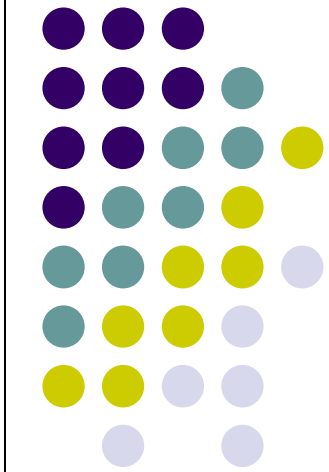
《嵌入式系统原理与开发》



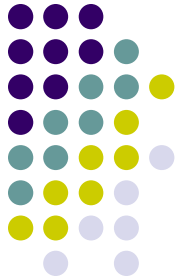
第5章 ARM指令集和汇编语言程序

南京大学计算机系　俞建新主讲

第5章 ARM指令集和汇编语言程序

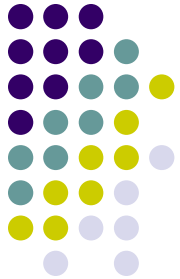
- 本章主要介绍以下内容:
  - ARM指令集的基本特点
    - 与Thumb指令集的区别
    - 与x86处理器的区别
    - ARM指令格式
  - ARM寻址方式
  - ARM指令集分类详解
  - ARM汇编语言的指示符
  - ARM汇编语言语句格式
  - ARM汇编语言程序格式
  - ARM汇编语句格式和程序格式进阶
  - ARM汇编语言程序举例

5.1 ARM指令集基本特点



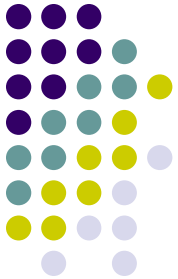
- 指令集的异同点
  - ARM、Thumb、x86
- ARM指令集的语法
  - ARM指令集的编码格式
- 指令条件码表
- 第2操作数

ARM指令集和Thumb指令集的共同点



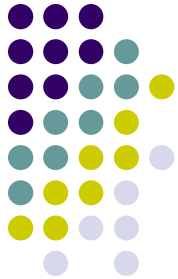
- ARM指令集和Thumb指令集具有以下共同点：
  1. 较多的寄存器，可以用于多种用途。
  2. 对存储器的访问只能通过Load/Store指令。
     - 两种指令集的差异特征在下页给出

ARM指令集和Thumb指令集的不同点



项目	ARM指令	Thumb指令
指令工作标志	CPSR的T位=0	CPSR的T位=1
操作数寻址方式	大多数指令为3地址	大多数指令为2地址
指令长度	32位	16位
内核指令	58条	30条
条件执行	大多数指令	只有分支指令
数据处理指令	访问桶形移位器和ALU	独立的桶形移位器和ALU指令
寄存器使用	15个通用寄存器+PC	8个通用低寄存器+7个高寄存器+PC
程序状态寄存器	特权模式下可读可写	不能直接访问
异常处理	能够全盘处理	不能处理

ARM指令集与x86指令集的主要不同点



- ARM指令集
  - 规整指令格式
    - 即：正交指令格式
  - 三地址指令
  - 由指令的附加位决定运算完毕后是否改变状态标志
  - 状态标志位只有4位
  - 有两种指令密度
  - 无整数除法指令
  - 大多数ARM指令都可以条件执行
  - 有适合DSP处理的乘加指令
  - Load/Store访存体系结构

- x86指令集
  - 非规整指令格式
    - 即：非正交指令格式
  - 二地址指令
  - 指令隐含决定运算完毕后是否改变状态标志
  - 状态标志位有6位
  - 单一指令密度
  - 有整数除法指令
  - 专用条件判断指令进行程序分支
  - 没有适合DSP处理的乘加指令
  - 运算指令能够访问存储器

ARM指令集的编码格式

- 参看ARM指令集编码格式PDF文件

ARM指令集的语法

- 一条典型的ARM指令语法如下所示:

<opcode>{<cond>}{S} <Rd>, <Rn> {,<Operand2>}

其中:

<opcode > 是指令助记符，决定了指令的操作。

例如: ADD表示算术加操作指令。

{<cond>} 是指令执行的条件，可选项。

{S} 决定指令的操作是否影响CPSR的值，可选项。

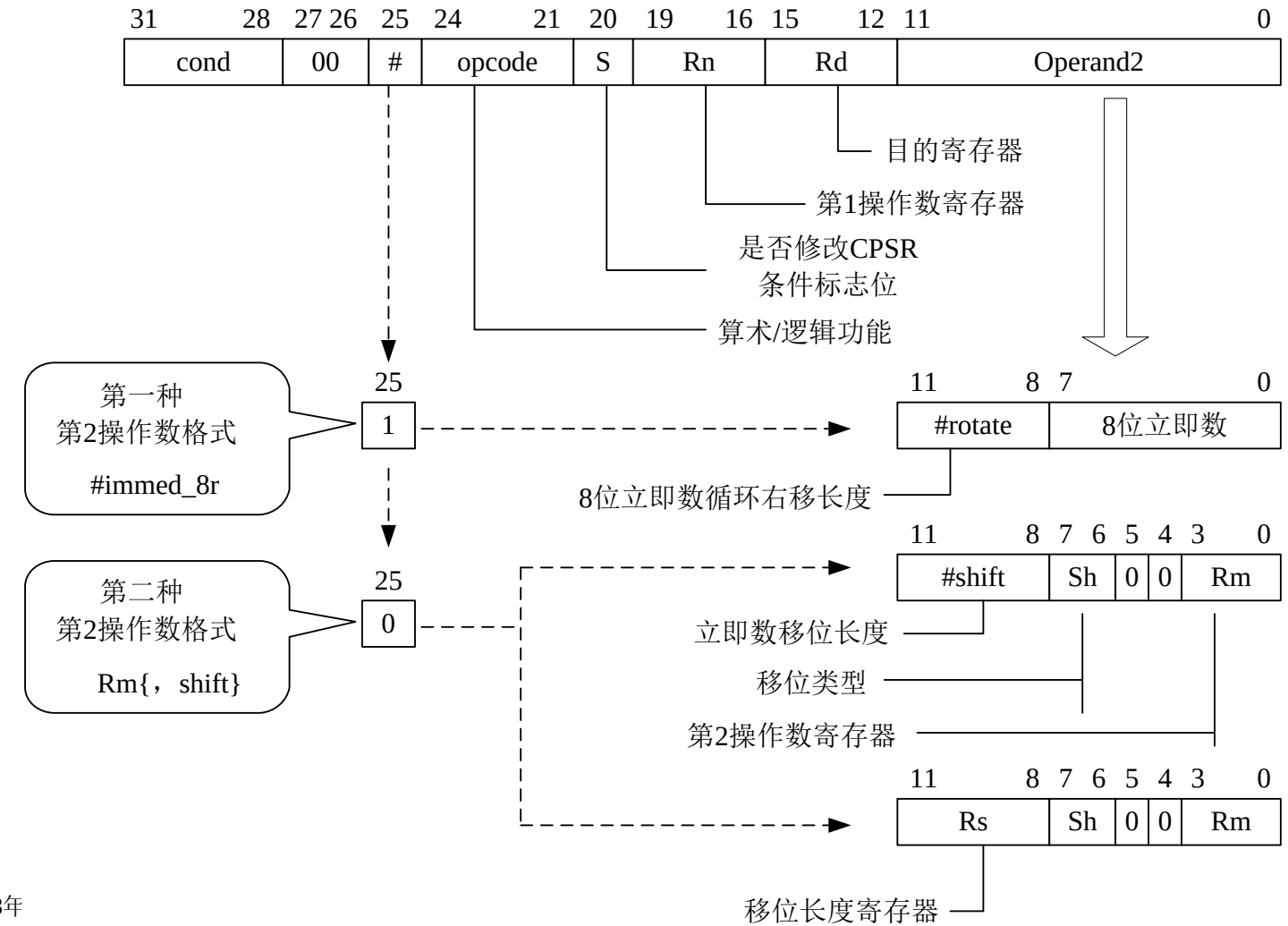
<Rd> 表示目标寄存器，必有项。

<Rn> 表示包含第1个操作数的寄存器，当仅需要一个源操作数时可省略。

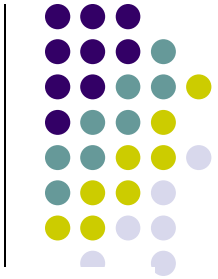
<Operand2> 表示第2个操作数，可选项。

第2操作数有两种格式: #immed_8r，Rm{, Shift}

ARM数据处理指令中第2操作数的编码格式图解

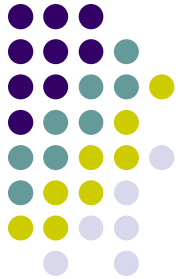


灵活的第2操作数



- 立即数型
  - **格式： #<32位立即数>**
  - 也写成#immed_8r
  - #<32位立即数>是取值为数字常量的表达式，并不是所有的32位立即数都是有效的。
  - 有效的立即数很少。它必须由一个8位的立即数循环右移偶数位得到。原因是32位ARM指令中条件码和操作码等占用了一些必要的指令码位，32位立即数无法编码在指令中。
  - 举例:
    - ADD r3, r7, #1020　;#immed_8r型第2操作数,
      ;1020是0xFF循环右移30位后生成的32位立即数
      ;推导：1020=0x3FC=0x000003FC

灵活的第2操作数（续1）

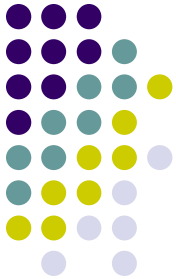


- 数据处理指令中留给Operand2操作数的编码空间只有12位，需要利用这12位产生32位的立即数。其方法是：把指令最低8位（bit[7:0]）立即数循环右移偶数次，循环右移次数由2*bit[11:8]（bit[11:8]是Operand2的高4位）指定。
- 例如： MOV R4, #0x8000000A

  ;其中的立即数#0x8000000A是由8位的0xA8循环右移0x4位得到。
- 又例如： MOV R4, #0xA0000002

  ;其中的立即数#0xA0000002是由8位的0xA8循环右移0x6位得到。

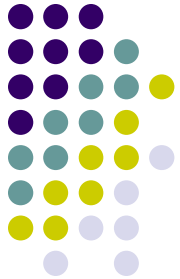
灵活的第2操作数（续2）



- 寄存器移位型
  - **格式：Rm{, <shift>}**
  - Rm是第2操作数寄存器，可对它进行移位或循环移位。<shift>用来指定移位类型（LSL，LSR，ASR，ROR或RRX）和移位位数。其中移位位数有两种表示方式，一种是5位立即数（#shift），另外一种是位移量寄存器Rs的值。参看下面的例子。例子中的R1是Rm寄存器。
  - ADD R5, R3, R1, LSL #2　　;R5←R3+R1*4
  - ADD R5, R3, R1, LSL R4　　;R5←R3+R1*2R4

    ;R4是Rs寄存器，Rs用于计算右移次数

详解第2操作数 # immed_8r



- 该常数必须对应8位位图，即常数是由一个8位的常数循环右移位偶数位得到。例如:
  - 合法常量：0x3FC、0、0xF0000000、200、0xF0000001。
  - 非法常量：0x1FE、511、0xFFFF、0x1010、0xF0000010。
  - 常数表达式应用举例:
    - MOV　R0，#1　　;R0=1
    - AND　R1，R2，#0x0F　　;R2与0x0F，结果保存在R1
    - LDR　R0，[R1]，#－4
    - SUB　R4，R2，#D4000002

      ;该立即数是0xBE循环右移6位

      ;课堂练习此第2操作数

详解第2操作数的Rm寄存器（1）

- RM寄存器通常是存放第2操作数的寄存器

<opcode>{<cond>}{S} <Rd>, <Rn> {,RM{, shift}}

- 举例:
  - SUB　　R1，R1，R2 ;R1－R2→R1
  - MOV　　PC，R0　;PC←R0，程序跳转到指定地址
  - LDR　　R0，[R1]，－R2
    ;读取R1地址上的存储器单元内容并存入R0，
    ;且R1 = R1－R2，后索引偏移
  - AND　　R0，R5，R2
    ;R2中存放的是第2操作数
    ;该数据属于寄存器方式的第2操作数

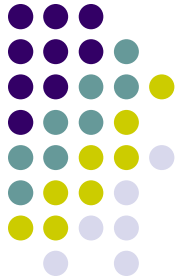
详解第2操作数的Rm寄存器（2）

```
ADD  R0, R0, R0, LSL #2        ;执行结果R0=5*R0
ADD  R5, R3, R1, LSL #2        ;R5←R3+R1*4
ADD  R5, R3, R1, LSL R4        ;R5←R3+R1*2R4
```

寄存器移位方式生成的第2操作数 Rm{, shift}

- 将寄存器的移位结果作为操作数，但Rm值保存不变，移位方法如下:
  - ASR　　# n　　;算术右移n位(1≤n≤32)。
  - LSL　　# n　　;逻辑左移n位(1≤n≤31)。
  - LSR　　# n　　;逻辑右移n位(1≤n≤32)。
  - ROR　　# n　　;循环右移n位(1≤n≤31)。
  - RRX　　;带扩展的循环右移1位。

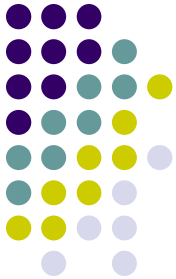
桶型移位器移位操作：Type Rs



- 其中，**Type**为ASR、LSL、LSR和ROR中的一种；Rs为偏移量寄存器，最低8位有效。若其值大于或等于32，则第2个操作数的结果为0(ASR、LSR例外)。
- 例如

  MOVS　R3, R1, LSL #7　;R3←R1*128

寄存器位移方式生成第2操作数应用举例



- ADD R1, R1, R1, LSL # 3

;R1=R1*9，因为R1← R1+R1*8。

- SUB R1, R1, R2, LSR # 2

;R1 = R1 − R2 ÷ 4,

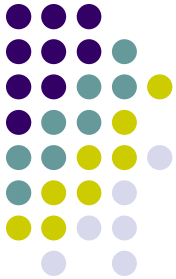
;因为R2右移2位相当于R2除以4。

- EOR R11, R12, R3, ASR #5

;R11= R12 ⊕ (R3 ÷ 32)

;第2操作数是R3的内容除以32

寄存器位移方式生成第2操作数应用举例（续）



- MOVS　R4, R4, LSR #32

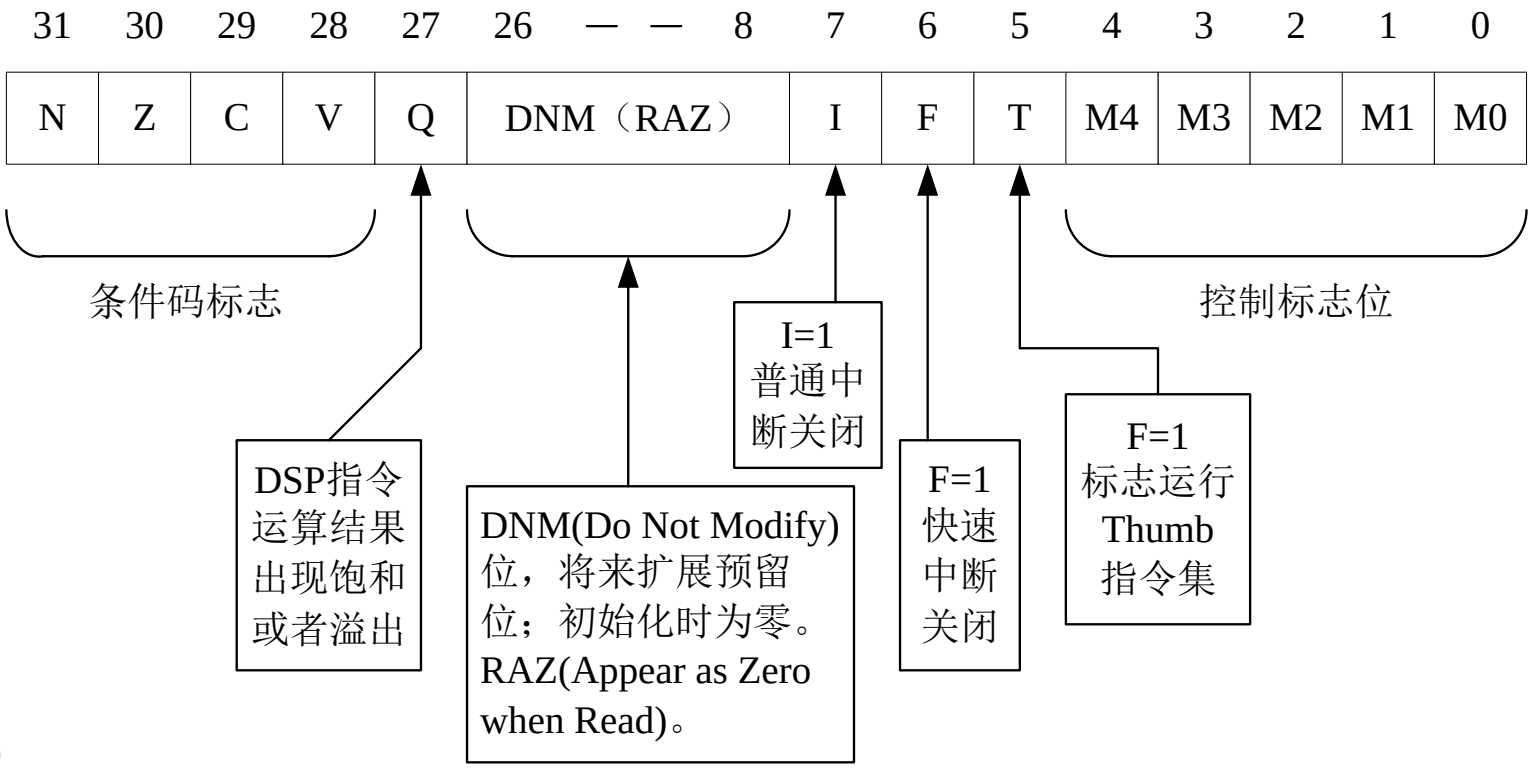
;C标志更新为R4的位[31]，R4清零。

;参看本课程教材第119页

- R15为处理器的程序计数器PC，一般不要对其进行操作，而且有些指令是不允许使用R15的，如UMULL指令。

ARM处理器的CPSR寄存器和SPSR寄存器的位定义格式图解

- 参看教材第4.2.3节



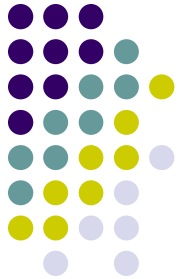
指令条件码表（1）

条件码助记符	标志	含义
EQ	Z = 1	相等
NE	Z = 0	不相等
CS/HS	C = 1	无符号数大于或等于
CC/LO	C = 0	无符号数小于
MI	N = 1	负数（ minus ）
PL	N = 0	正数或零
VS	V = 1	上溢出
VC	V = 0	没有上溢出

指令条件码表（2）

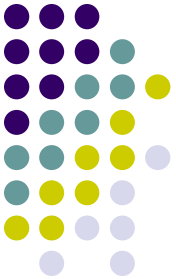
条件码助记符	标志	含义
HI	C = 1，Z = 0	无符号数大于
LS	C = 0，Z = l	无符号数小于或等于
GE	N = V	有符号数大于或等于
LT	N! = V	有符号数小于
GT	Z=0，N=V	有符号数大于
LE	Z = 1，N!=V	有符号数小于或等于
AL	任何	无条件执行(指令默认条件)
NV	ARMv3之前	该指令从不执行

5.2 ARM处理器寻址方式



- 寻址方式是根据指令中给出的地址码字段来实现寻找真实操作数地址的方式。
- ARM处理器具有8种基本寻址方式，以下列出：
  - 寄存器寻址
  - 立即寻址
  - 寄存器偏移寻址
  - 寄存器间接寻址
  - 基址寻址
  - 多寄存器寻址
  - 堆栈寻址
  - 相对寻址

寄存器寻址



- 操作数的值在寄存器中，指令中的地址码字段指出的是寄存器编号，指令执行时直接取出寄存器值来操作。寄存器寻址指令举例如下：
  - MOV R1，R2　　；读取R2的值送到R1
  - MOV R0，R0　　；R0=R0，相当于无操作
  - SUB R0，R1，R2　　；R0←R1-R2

  ；将R1的值减去R2的值，结果保存到R0
  - ADD R0, R1, R2　　；R0←R1+R2

  ；这条指令将两个寄存器（R1和R2）的内容相加，结果放入第3个寄存器R0中。必须注意写操作数的顺序：第1个是结果寄存器，然后是第一操作数寄存器，最后是第二操作数寄存器。

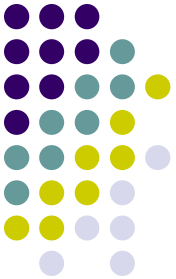
立即寻址

- 立即寻址指令中的操作码字段后面的地址码部分即是操作数本身。也就是说，数据就包含在指令当中，取出指令也就取出了可以立即使用的操作数(这样的数称为立即数)。立即寻址指令举例如下：
  - SUBS R0，R0，#1

  ；R0减1，结果放入R0，并且影响标志位
  - MOV R0，#0xFF000

  ；将十六进制立即数0xFF000装入R0寄存器
  - 立即数要以“#”号为前缀，16进制数值时以“0x”表示。

寄存器偏移寻址



- 寄存器偏移寻址是ARM指令集特有的寻址方式。当第2作数是寄存器偏移方式时，第2个寄存器操作数在与第1操作数结合之前，选择进行移位操作。
- 寄存器偏移寻址指令举例如下:
  - MOV R0, R2, LSL #3

;R2的值左移3位，结果放入R0，即R0 = R2 × 8

  - ANDS R1, R1, R2, LSL R3

;R2的值左移R3位，然后与R1相“与”，结果放入R1，并且影响标志位。

  - SUB R11, R12, R3, ASR #5

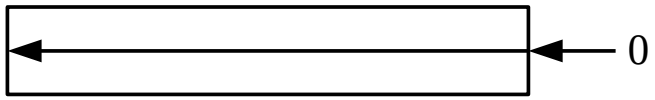
;R12—R3 ÷ 32，然后存入R11。

寄存器偏移寻址（续）

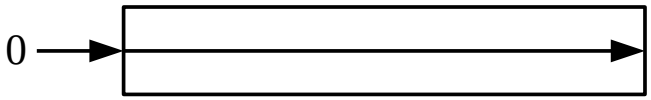
- 可采用的移位操作如下:
  - **LSL**: 逻辑左移(Logical Shift Left)，低端空出位补0。
  - **LSR**: 逻辑右移(Logical Shift Right)，高端空出位补0。
  - **ASR**: 算术右移(Arithmetic Shift Right)，移位过程中保持符号位不变，即若源操作数为正数，则字的高端空出的位补0；否则补1。
  - **ROR**: 循环右移(Rotate Right)，由字低端移出的位填入字高端空出的位。
  - **RRX**: 带扩展的循环右移(Rotate Right extended by l place)，操作数右移1位，高端空出的位用原C标志值填充。如果指定后缀“S”，则将Rm原值的位[0]移到进位标志。

移位操作示意图

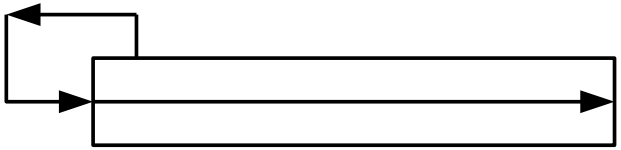
- 各种移位操作如下图所示：



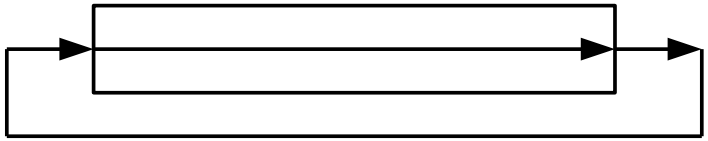
(a) LSL移位操作



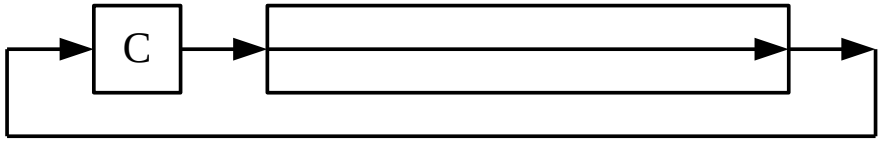
(b) LSR移位操作



(c) ASR移位操作

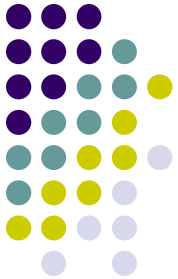


(d) ROR移位操作



(e) RRX移位操作

寄存器间接寻址

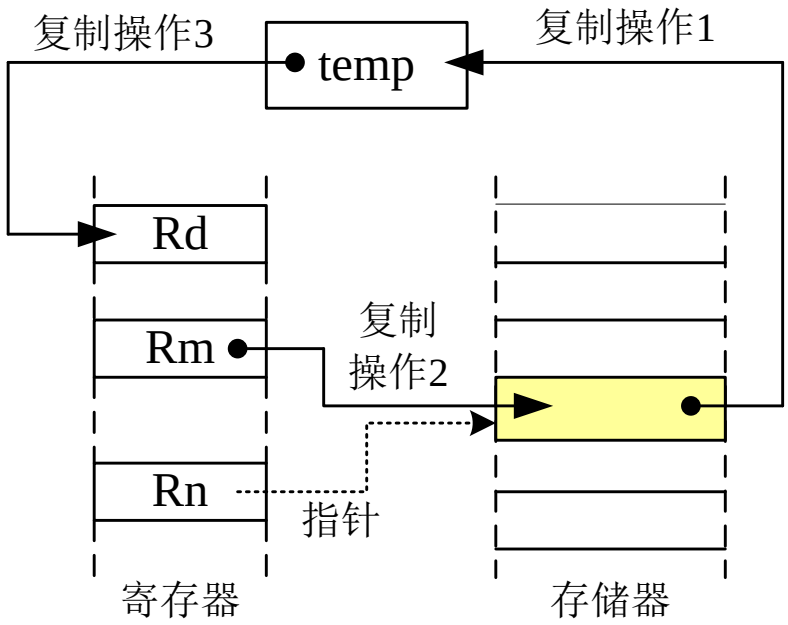


- 寄存器间接寻址指令中的地址码给出的是一个通用寄存器的编号，所需的操作数保存在寄存器指定地址的存储单元中，即寄存器为操作数的地址指针。寄存器间接寻址指令举例如下：
  - LDR R1，[R2]

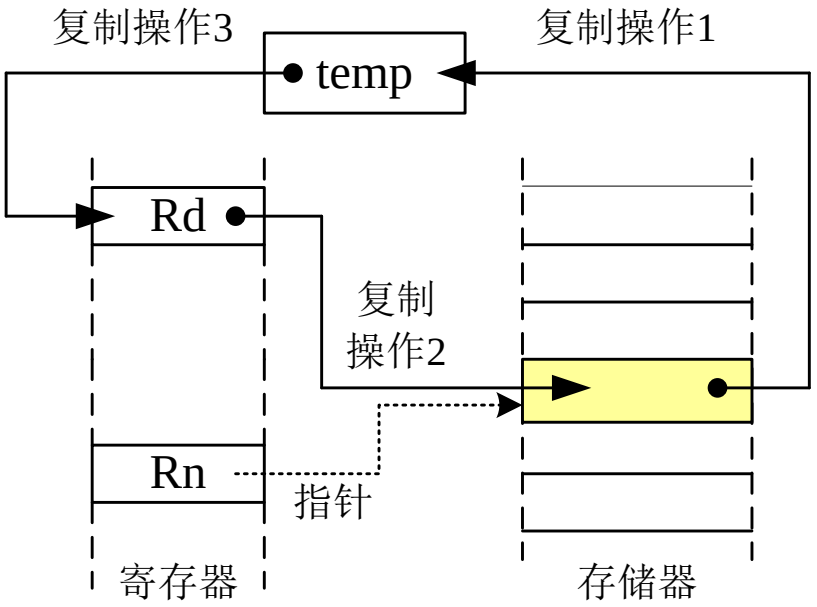
    将R2指向的存储单元的数据读出，保存在R1中。
  - SWP R1，R1，[R2]

    将寄存器R1的值与R2指定的存储单元的内容交换

SWP指令操作图解

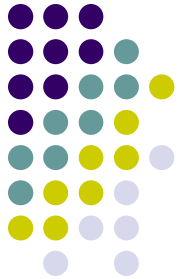


(a)SWP指令的一般性操作



(b)当Rm等同Rd时，SWP指令的操作

基址寻址

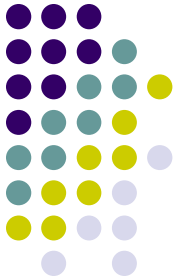


- 基址寻址就是将基址寄存器的内容与指令中给出的偏移量相加，形成操作数的有效地址。基址寻址用于访问基址附近的存储单元，常用于查表、数组操作、功能部件寄存器访问等。基址寻址指令举例如下：
  - LDR R2，[R3，#0x0C]

    读取R3 + 0x0C地址上的存储单元的内容，放入R2。
  - STR R1，[R0，#－4]!

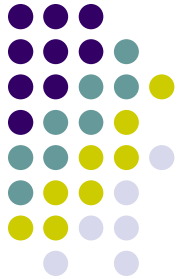
    ;[R0－4]←R1，R0=R0－4，符号“!”表明指令在完成数据传送后应该更新基址寄存器，否则不更新；属前索引。

基址寻址指令举例



- LDR R1，[R0，R3，LSL #1]
  ;将R0 + R3 × 2地址上的存储单元的内容读出，存入R1。
- LDR R0，[R1，R2，LSL #2]!
  ;将内存起始地址为R1+R2*4的字数据读取到R0中，
  ;同时修改R1，使得：R1=R1+R2*4。
- LDR R0, [R1, R2]!
  ;以R1+R2值为地址，访问内存。将该位置的字数据读
  ;取到R0中，同时修改R1，使得： R1=R1+R2。
  ;属于前索引指令

多寄存器寻址



- 多寄存器寻址即是一次可传送几个寄存器值，允许一条指令传送16个寄存器的任何子集或所有寄存器。多寄存器寻址指令举例如下:
  - LDMIA R1!，{R2－R7，R12}

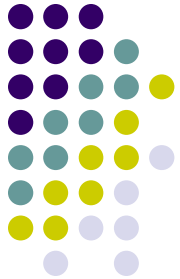
  ；将R1指向的单元中的数据读出到R2~R7、R12中

  ；(R1自动增加)
  - 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自动增加)
  - 使用多寄存器寻址指令时，寄存器子集的顺序是按由小到大的顺序排列，连续的寄存器可用“－”连接；否则用“，”分隔书写。

多寄存器寻址（续1）



- 多寄存器寻址指令举例
  - LDMIA R1!, {R0, R2, R5} ;

    ;R0←[R1]

    ;R2←[R1+4]

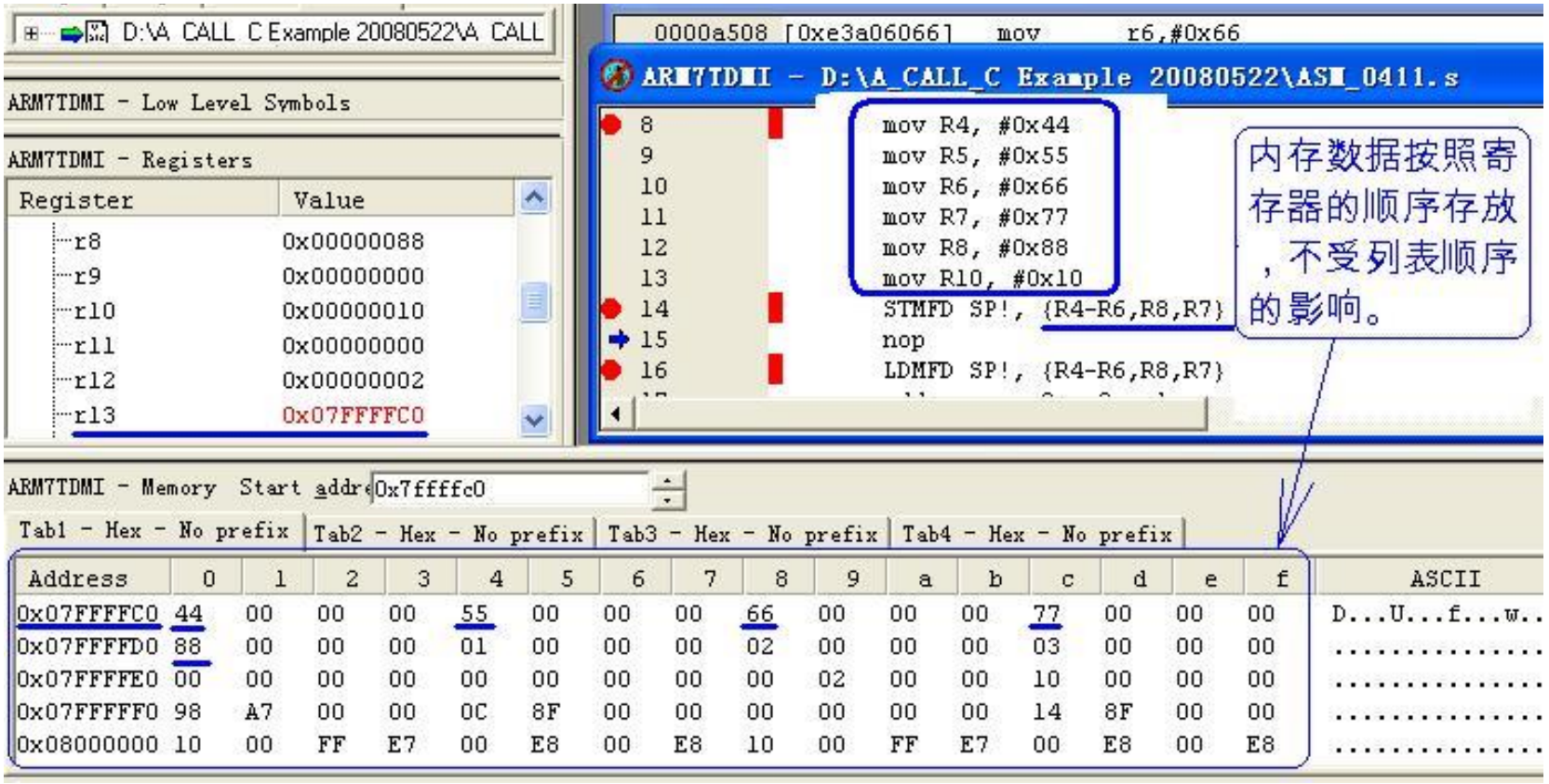
    ;R5←[R1+8]

    ;R1保持自动增值

    ;寄存器列表{R0, R2, R5}与{R2, R0, R5}等效
- 多寄存器指令的执行顺序与寄存器列表次序无关，而与寄存器的序号保持一致。

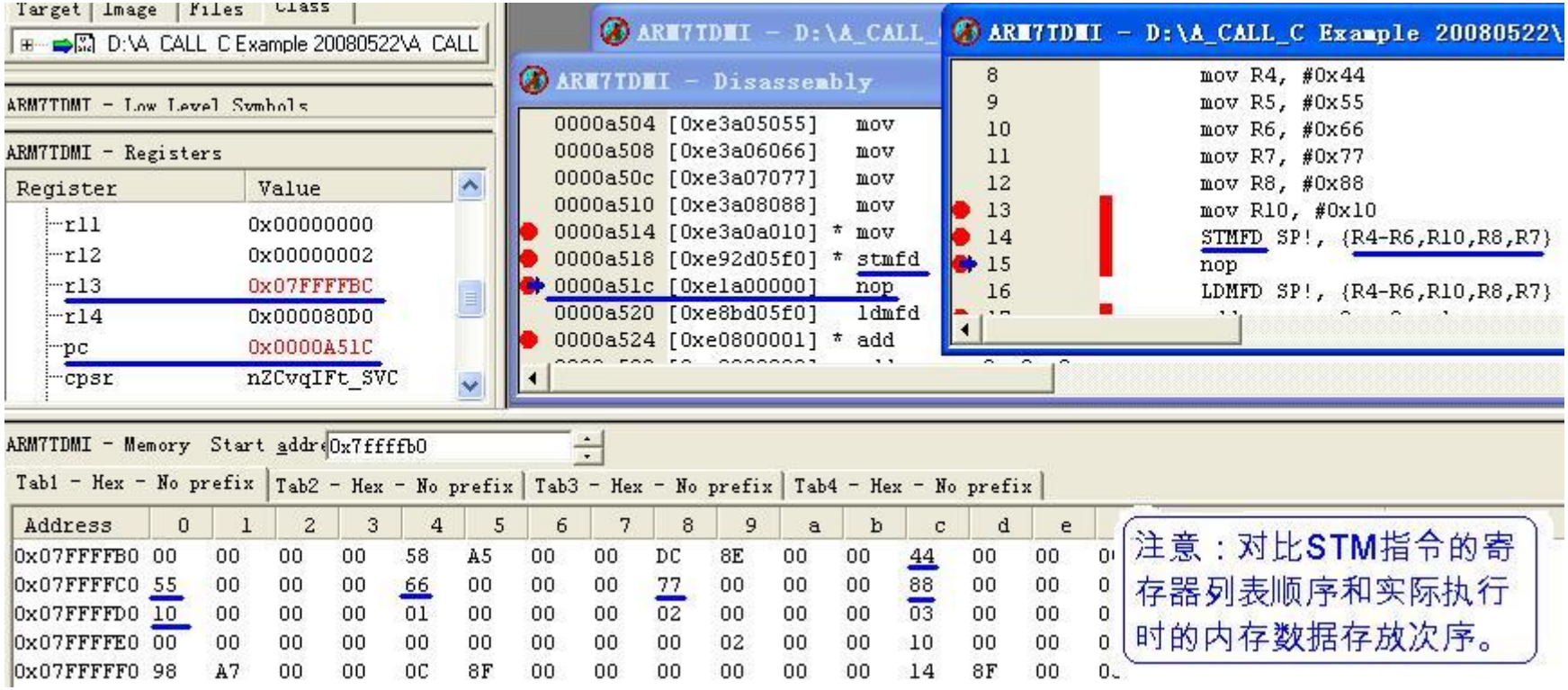
多寄存器指令的执行顺序举例1

- 通过ADS集成开发环境的AXD调试器窗口观察



多寄存器指令的执行顺序举例2

- 通过ADS集成开发环境的AXD调试器窗口观察



多寄存器寻址（续2）

- 下面是多寄存器传送指令STM举例如下:
  - STMIA R0!，{R1—R7}

;将R1~R7的数据保存到存储器中。存储指针在保存第一
;个值之后增加，增长方向为向上增长

  - STMIB R0!，{R1—R7}

;将R1~R7的数据保存到存储器中。存储指针在保存第一
;个值之前增加，增长方向为向上增长

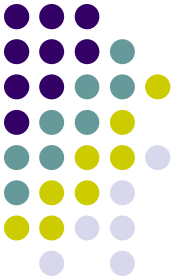
  - STMDA R0!, {R1—R7}

;将R1~R7的数据保存到存储器中。存储指针在保存第一
;个值之后增加，增长方向为向下增长

  - STMDB R0!，{R1—R7}

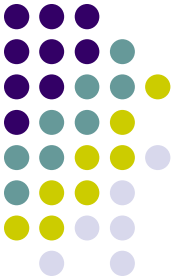
;将R1~R7的数据保存到存储器中。存储指针在保存第一
;个值之前增加，增长方向为向下增长

堆栈寻址



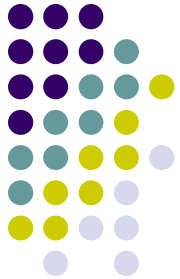
- 存储器堆栈可分为两种:
  - 向上生长：向高地址方向生长，称为递增堆栈。
  - 向下生长：向低地址方向生长，称为递减堆栈。
- 满堆栈
  - 堆栈指针指向最后压入的堆栈的有效数据项
- 空堆栈
  - 堆栈指针指向下一个待压入数据的空位置

堆栈寻址（续1）



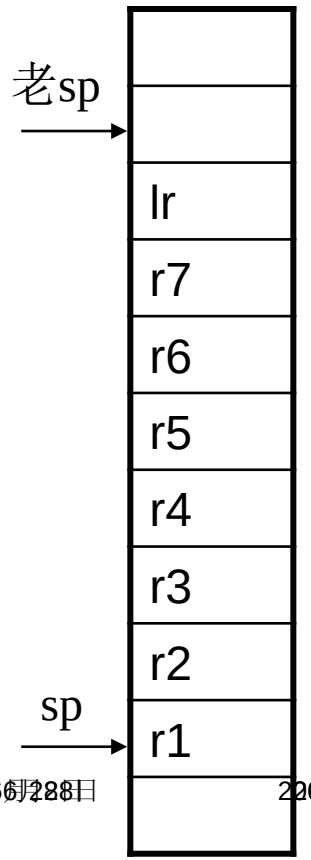
- 有4种类型的堆栈组合
  - 满递增：堆栈通过增大存储器的地址向上增长，堆栈指针指向内含有效数据项的最高地址。指令如LDMFA、STMFA等。
  - 空递增：堆栈通过增大存储器的地址向上增长，堆栈指针指向堆栈上的第一个空位置。指令如LDMEA、STMEA等。
  - 满递减：堆栈通过减小存储器的地址向下增长，堆栈指针指向内含有效数据项的最低地址。指令如LDMFD、STMFD等。
  - 空递减：堆栈通过减小存储器的地址向下增长，堆栈指针指向堆栈下的第一个空位置。指令如LDMED、STMED等。

堆栈寻址（续2）

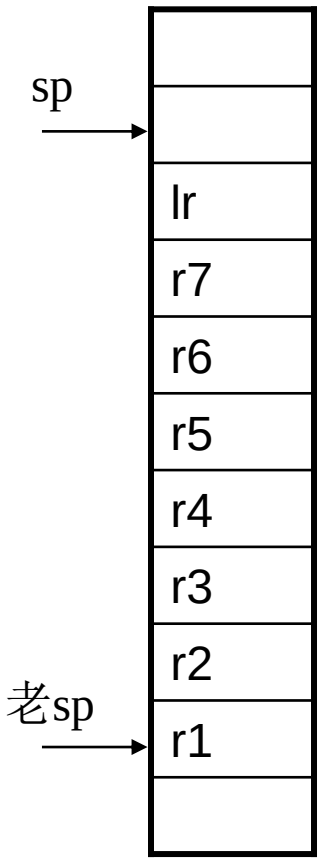


- 堆栈寻址指令举例如下:
  - STMFD SP!，{R1—R7，LR}
  ；将R1~R7、LR入栈（push），满递减堆栈。
  - LDMFD SP!，{R1—R7，LR}
  ；数据出栈（pop），放入R1~R7、LR寄存器。
  ；满递减堆栈

STMFD sp!, {r1-r7,lr}



LDMFD sp!, {r1-r7,lr}



高地址

低地址

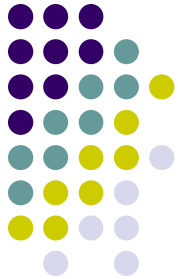
2008年6月28日

多寄存器传送指令映射表

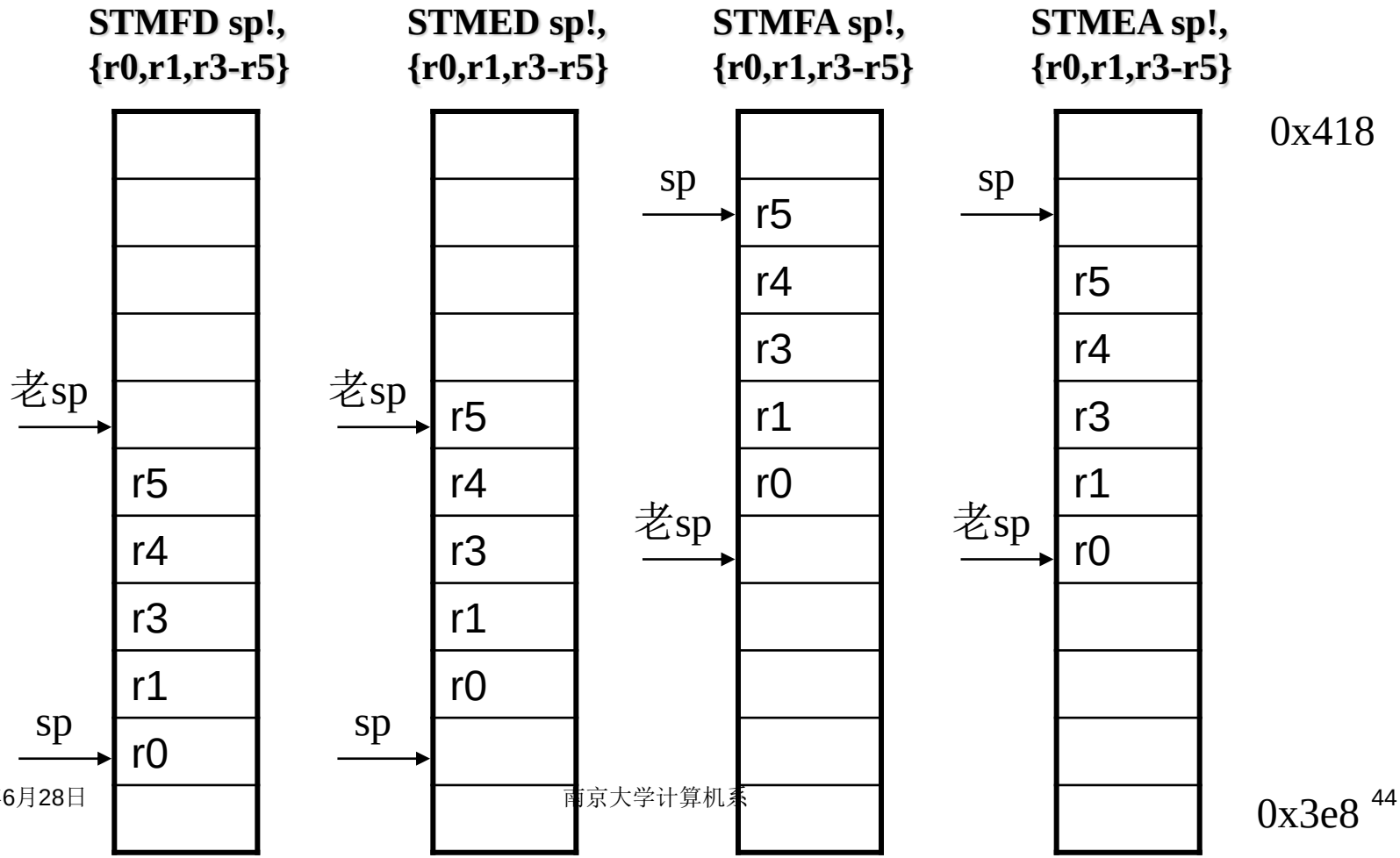
- STM =将寄存器内容存入内存单元（堆栈操作：入栈）
- LDM=将内存单元内容存入寄存器（堆栈操作：出栈）

传送与地址变化的关系 \ 地址变化的方向	向上生长		向下生长	
	指针满	指针空	指针满	指针空
地址增加在传送之前	**STMIB** **STMFA**			**LDMIB** **LDMED**
地址增加在传送之后		**STMIA** **STMEA**	**LDMIA** **LDMFD**	
地址减小在传送之前		**LDMDB** **LDMEA**	**STMDB** **STMFD**	
地址减小在传送之后	**LDMDA** **LDMFA**			**STMDA** **STMED**

多寄存器传送指令说明



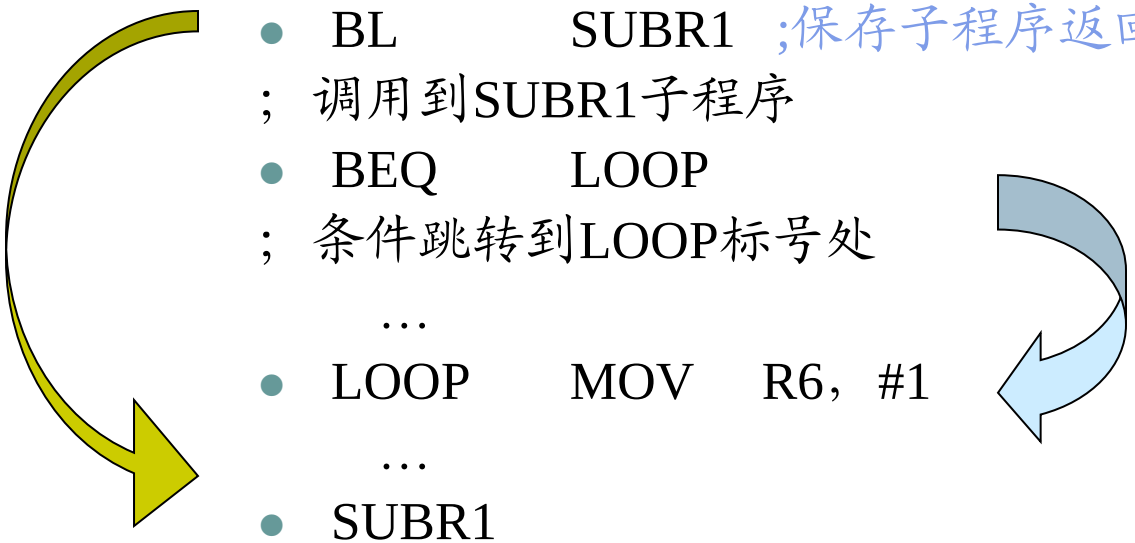
- 数据块传送:
  - I =向地址增大方向处理数据传送(Increment)
  - D =向地址减小方向处理数据传送(Decrement)
  - A =先传送数据后改变地址(after)
  - B =先改变地址后传送数据(before)
- 堆栈操作:
  - F =满栈顶指针(full)
  - E =空栈顶指针(empty)
  - A =堆栈向高地址方向增长(ascending stack)
  - D =堆栈向低地址方向增长(decending stack)


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

相对寻址

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

```
    BL      SUBR1   ;保存子程序返回地址
;   调用到SUBR1子程序
    BEQ     LOOP
;   条件跳转到LOOP标号处
        …
LOOP    MOV     R6，#1
        …
SUBR1
        …
```

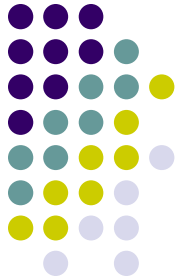


相对寻址举例

```
        BL  SUBR          ;转移到SUBR
        ……
        ……...
SUBR    …..               ;子程序入口
        …..
        MOV PC, R14       ;返回
```

; R14也就是LR

5.3 ARM指令集分类详解



- ARM指令集大致分为6类：分支指令、Load/Store指令、数据处理指令、程序状态寄存器指令、异常中断指令、协处理器指令。以下分别介绍其中的主要指令。

5.3.1 分支指令

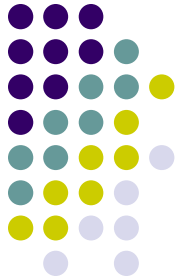
- ARM有两种方法可以实现程序分支转移。
  - 跳转指令
  - 所谓的长跳转
    - 直接向PC寄存器（R15）中写入目标地址。
  - ARM跳转指令有以下4种:
    - ① B 分支指令，语法
      - B{cond} label
    - ② BL 带链接分支指令
      - 语法: BL{cond} label
    - ③ BX 分支并可选地交换指令集
      - 语法: BX{cond} Rm
    - ④ BLX 带链接分支并可选择地交换指令集。
      - 语法: **BLX{cond} label | Rm**

BL指令举例

- BL指令的意义：Branch and Link
- 示例：

```
                ……. ……..
                bl MyPro          ;调用子程序MyPro
                ……….
MyPro                             ;子程序MyPro本体
                ……….
                ……….
                mov PC, LR        ;将R14的值送入R15，返回
```

BX指令使用举例



- 通过使用BX指令可以让ARM处理器内核工作状态在ARM状态和Thumb状态之间进行切换。
  - 参看下例:

```
;从ARM状态转变为Thumb状态
    LDR     R0, =Sub_Routine+1
    BX      R0
;从Thumb 状态转变为ARM状态
    LDR     R0, =Sub_Routine
    BX      R0
```

长跳转

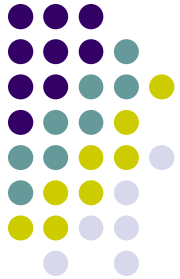
- 直接向PC寄存器写入目标地址值，可以实现4GB地址空间中的任意跳转。
  - 示例:
    - 以下的两条指令实现了4GB地址空间中的子程序调用。
  - MOV LR, PC

    ;保存返回地址
  - MOV R15, #0x00110000

    ;无条件转向绝对地址0x110000

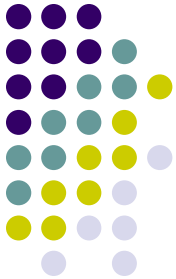
    ;此32位立即数地址应满足单字节循环右移偶数次

5.3.2 Load/Store指令



- Load/Store指令用于在存储器和处理器之间传输数据。Load用于把内存中的数据装载到寄存器，Store指令用于把寄存器中的数据存入内存。
- 共有3种类型的Load/Store指令:
  - 单寄存器传输指令
  - 多寄存器传输指令
  - 交换指令

单寄存器传送指令



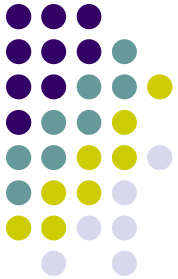
助记码	操作	指令描述
LDR	把一个字装入一个寄存器	Rd←mem32[address]
STR	从一个寄存器保存一个字	Rd→mem32[address]
LDRB	把一个字节装入一个寄存器	Rd←mem8[address]
STRB	从一个寄存器保存一个字节	Rd→mem8[address]
LDRH	把一个半字装入一个寄存器	Rd←mem16[address]
STRH	从一个寄存器保存一个半字	Rd→mem16[address]
LDRSB	把一个有符号字节装入寄存器	Rd←符号扩展(mem8[address])
LDRSH	把一个有符号半字装入寄存器	Rd←符号扩展 (mem16[address])

Load/Store指令变址模式

- 变址模式有四种：零偏移、前变址、后变址、回写前变址 。

变址模式	数据	基址寄存器	指令举例
零偏移	mem[base]	直接基址寄存器寻址	LDR r0, [r1]
回写前变址	mem[base+offset]	基址寄存器加偏移量	LDR r0, [r1, #4]!
前变址	mem[base+offset]	不变	LDR r0, [r1, #4]
后变址	mem[base]	基址寄存器加偏移量	LDR r0, [r1], #4

单寄存器传送指令举例



- LDR R2，[R3，#0x0C]

  读取R3 + 0x0C地址上的一个字数据内容，放入R2。属前变址。

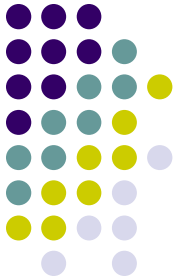
- STR R1，[R0，#－4]!

  [R0－4]←R1，R0=R0－4，符号“!”表明指令在完成数据传送后应该更新基址寄存器，否则不更新；属回写前变址。

- LDR R1，[R0，R3，LSL #1]

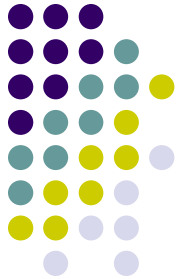
  将R0 + R3 × 2地址上的存储单元的内容读出，存入R1。

5.3.3 数据处理指令



- ARM数据处理指令大致分为以下6种类型。
  - 数据传送指令
  - 算术运算指令
  - 逻辑运算指令
  - 比较指令
  - 测试指令
  - 乘法指令

ARM数据处理指令

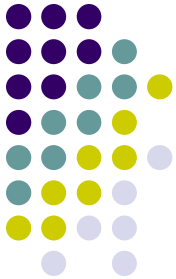


- ARM数据处理指令大致可分为3类:
  - 数据传送指令(如MOV、MVN);
  - 算术逻辑运算指令(如ADD、SUB、AND);
  - 比较指令(如CMP、TST)。
- 参见下面的表格
  - 数据处理指令只能对寄存器的内容进行操作。所有ARM数据处理指令均可选择使用S后缀，以影响状态标志。
  - 比较指令CMP、CMN、TST和TEQ不需要后缀S，它们会直接影响状态标志。

ARM数据处理指令集

助记符	说 明	操 作	条件码位置
MOV Rd，operand2	数据传送指令	Rd←operand2	MOV{cond}{S}
MVN Rd，operand2	数据非传送指令	Rd←(~operand2)	MVN{cond}{S}
ADD Rd，Rn，operand2	加法运算指令	Rd←Rn + operand2	ADD{cond}{S}
SUB Rd，Rn，operand2	减法运算指令	Rd←Rn－operand2	SUB{cond}{S}
RSB Rd，Rn，operand2	逆向减法指令	Rd←operand2－Rn	RSB{cond}{S}
ADC Rd，Rn，operand2	带进位加法指令	Rd←Rn + operand2 + Carry	ADC{cond}{S}
SBC Rd，Rn，operand2	带进位减法指令	Rd←Rn－operand2－(NOT)Carry	SBC{cond}{S}
RSC Rd，Rn，operand2	带进位逆向减法指令	Rd←operand2－Rn－(NOT)Carry	RSC{cond}{S}

ARM数据处理指令集（续）



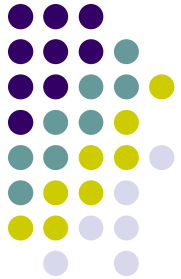
助记符	说 明	操 作	条件码位置
AND Rd，Rn，operand2	逻辑“与”操作指令	Rd←Rn & operand2	AND{cond}{S}
ORR Rd，Rn，operand2	逻辑“或”操作指令	Rd←Rn | operand2	ORR{cond}{S}
EOR Rd，Rn，operand2	逻辑“异或”操作指令	Rd←Rn^operand2	EOR{cond}{S}
BIC Rd，Rn，operand2	位清除指令	Rd←Rn&(~operand2)	BIC{cond}{S}
CMP Rn，operand2	比较指令	标志N,Z,C,V←Rn－operand2	CMP{cond}
CMN Rn，operand2	负数比较指令	标志N,Z,C,V ←Rn + operand2	CMN{cond}
TST Rn，operand2	位测试指令	标志N,Z,C,V ←Rn&operand2	TST{cond}
TEQ Rn，operand2	相等测试指令	标志N,Z,C,V ←Rn^operand2	TEQ{cond}

乘法指令

- ARM7TDMI(-S)具有32 × 32乘法指令、32 × 32乘加指令，32 × 32结果为64位的乘/乘加指令。ARM乘法指令如下表所列。

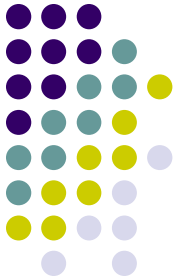
助记符	说 明	操 作	条件码
MUL Rd,Rm,Rs	32位乘法指令	Rd + Rm × Rs (Rd!=(Rm)	MUL{Cond}{S}
MLA Rd,Rm,Rs,Rn	32位乘加指令	Rd←Rm × Rs + Rn (Rd!=Rm)	MLA{cond}{S}
UMULL RdLo,RdHi,Rm,Rs	64位无符号乘法指令	(RdLo，RdHi)←Rm × Rs	UMULL{cond}{S}
UMLAL RdLo,RdHi,Rm,Rs	64位无符号乘加指令	(RdLo,RdHi←Rm × Rs + (RdLo,RdHi)	UMLAL{cond}{S}
SMULL RdLo,RdHi,Rm,Rs	64位有符号乘法指令	(RdLo,RdHi)←Rm × Rs	SMULL{cond}{S}
SMLAL RdLo,RdHi,Rm,Rs	64位有符号乘加指令	(RdLo,RdHi←Rm × Rs + (RdLo,RdHi)	SMLAL{cond}{S}

5.3.4 程序状态寄存器指令



- 读状态寄存器指令MRS
- 写状态寄存器指令MSR
- 指令举例
- 开中断与关中断

读状态寄存器指令MRS

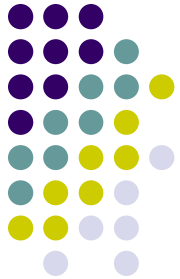


- 在ARM处理器中，只有MRS指令可以将状态寄存器CPSR或SPSR读出到通用寄存器中。
  - 指令格式如下:
    - MRS{cond} Rd，psr
  - 其中:
    - Rd 目标寄存器。Rd不允许为R15。
    - psr CPSR或SPSR。
  - 指令举例如下:
    - MRS R1，CPSR ； 将CPSR状态寄存器读取，保存到R1中。
    - MRS R2，SPSR ； 将SPSR状态寄存器读取，保存到R2中。

写状态寄存器指令MSR

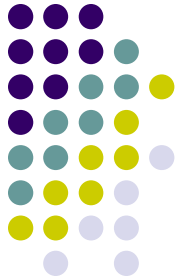
- 在ARM处理器中，只有MSR指令可以直接设置状态寄存器CPSR或SPSR。
  - 指令格式如下:
    - MSR{cond} psr_fields，#immed_8r
    - MSR{cond} psr_fields，Rm
  - 其中:
    - psr　　CPSR或SPSR。
    - fields　　指定传送的区域。

写状态寄存器指令MSR（续）



- fields可以是以下的一种或多种；(字母必须为小写)；c 控制域屏蔽字节 (psr[7…0])； x 扩展域屏蔽字节 (psr[15…8])； s 状态域屏蔽字节 (psr[23…16])； f 标志域屏蔽字节 (psr[31…24])。
- immed_8r　要传送到状态寄存器指定域的立即数，8位。
- Rm　　要传送到状态寄存器指定域的数据的源寄存器。

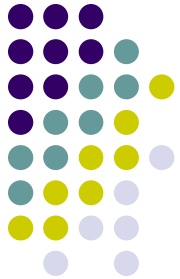
MSR指令举例



- MSR指令举例如下：

```
MSR  CPSR_c，#0xD3
；CPSR[7…0]=0xD3，即切换到管理模式，0b11010011
MSR  CPSR_cxsf，R3
；CPSR=R3
```

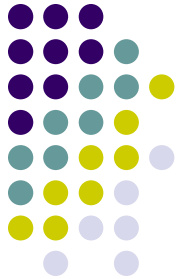
使能IRQ中断（开中断）



```
ENABLE_IRQ
    MRS     R0, CPSR
    BIC     R0, R0, #0x80
    MSR     CPSR_c, R0
    MOV     PC, LR
```

I位=0　开中断

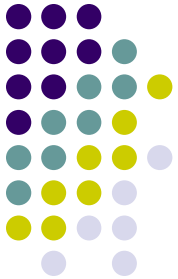
禁能IRQ中断（关中断）



```
DISABLE_IRQ
    MRS  R0  CPSR
    ORR  R0,  R0, #0x80
    MSR  CPSR_c,  R0
    MOV  PC,  LR
```

I位=1 关中断

MSR指令说明

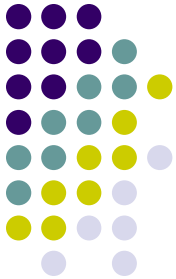


- 程序中不能通过MSR指令直接修改CPSR中的T控制位来实现ARM状态/Thumb状态的切换，必须使用BX指令完成处理器状态的切换(因为BX指令属分支指令，它会打断流水线状态，实现处理器状态切换)。
- MRS与MSR配合使用，实现CPSR或SPSR寄存器的读—修改—写操作，可用来进行处理器模式切换、允许/禁止IRQ/FIQ中断等设置，如下面的程序清单所示。

堆栈指令初始化

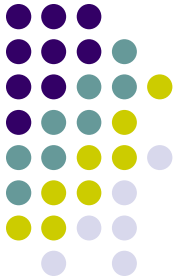
```
INITSTACK
    MOV  R0，LR
    ；保存返回地址
    MSR  CPSR_c，#0xD3
    LDR  SP，StackSvc
    ；设置管理模式堆栈，M[4:0]=0b10011
    MSR  CPSR_c，#0xD2
    LDR  SP，StackIrq
    ；设置中断模式堆栈，M[4:0]=0b10010
    MOV  PC，R0
```

5.3.5 软中断指令SWI



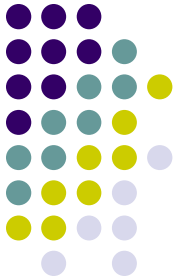
- SWI指令用于产生软中断，从而实现从用户模式变换到管理模式，CPSR保存到管理模式的SPSR中，执行转移到SWI向量。在其它模式下也可使用SWI指令，处理器同样地切换到管理模式。
- 指令格式如下:
  - SWI{cond} immed_24 // Thumb指令是 immed_8
  - 其中： immed_24是24位立即数，值为0~16,777,215之间的整数。立即数用于指定指令请求的具体SWI服务。
- 指令举例如下:
  - SWI 0 ；软中断，中断立即数为0
  - SWI 0x123456 ；软中断，中断立即数为0x123456

获得SWI指令的立即数



- 在SWI异常中断处理程序中，取出SWI立即数的步骤为：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;
- 接着读出指令，分解出立即数。
- 程序清单如下所示。

获得SWI指令的立即数（续）



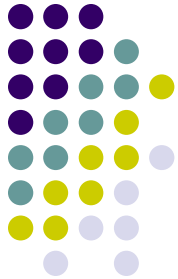
```
T_bit       EQU       0x20
SWI_Handler
    STMFD SP!，{R0－R3，R12，LR}；现场保护
    MRS  R0，SPSR                          ；读取SPSR
    STMFD  SP!，{R0}                       ；保存SPSR
    TST   R0，#T_bit                  ；测试T标志位，CPSR第M5位
    ；T=1表明执行Thumb指令，参看讲义上集91页
    LDREQH R0，[LR,#－2]              ；若是Thumb指令，则读取指令码(16位)
    BICEQ  R0，R0,#0xFF00             ；取得Thumb指令的8位立即数
    LDRNE R0，[LR,#－4]               ；若是ARM指令，则读取指令码(32位)
    BICNE   R0, R0, #0xFF000000       ；取得ARM指令的24位立即数
    …
    LDMFD  SP!, {R0－R3，R12，PC}          ；SWI异常中断返回
```

5.3.6 ARM协处理器指令

- ARM支持协处理器操作。协处理器控制通过协处理器命令实现。

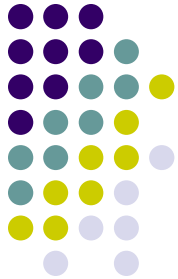
助记符	说 明	操 作	条件码
CDP coproc,opcode1,CRd,CRn, CRm{,opcode2}	协处理器数据操作指令	取决于协处理器	CDP{cond}
LDC{1} coproc,CRd,<地址>	协处理器数据读取指令	取决于协处理器	LDC{cond}{L}
STC{1} coproc,CRd,<地址>	协处理器数据写入指令	取决于协处理器	STC{cond}{L}

ARM协处理器指令（续）



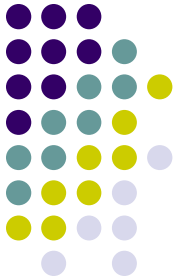
助记符	说 明	操 作	条件码
MCR coproc,opcode1, Rd,CRn,CRm{,opcode2}	ARM寄存器到协处理器寄存器的数据传送指令	取决于协处理器	MCR{cond}
MRC coproc,opcode1, Rd,CRn,CRm{,opcode2)	协处理器寄存器到ARM寄存器的数据传送指令	取决于协处理器	MRC{cond}

5.3.7 ARM伪指令



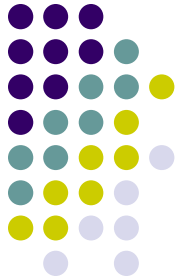
- ARM伪指令不是ARM指令集中的指令，只是为了编程方便编译器定义了伪指令。可以像其它ARM指令一样使用伪指令，但在编译时这些指令将被等效的ARM指令代替。
- ARM伪指令有4条，分别为ADR伪指令、ADRL伪指令、LDR伪指令和NOP伪指令。

ADR伪指令



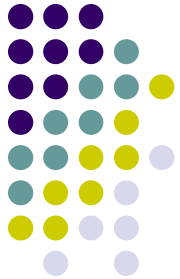
- 小范围的地址读取伪指令
- 该指令将基于PC的地址值或者基于寄存器的地址值读取到寄存器中
- 语法:
  - ADR {<cond>} register, expr
    - 其中，register为目标寄存器。expr为基于PC或者基于寄存器的地址表达式，其取值范围如下:
    - 当地址值不是字对齐时，其取值范围为-255~255。
    - 当地址值是字对齐时，其取值范围为-1020~1020。
    - 当地址值是16字节对齐时，其取值范围将更大。

ADR伪指令使用举例



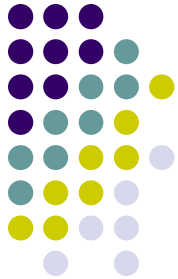
- 下面是一个使用ADR伪指令的例子:
- start　　MOV　R0,　#1000
- 　　ADR　　R4,　start
- ; 案例ARM处理器是三级流水线，PC值为当前指令地址值加8字节
- ; 因此本ADR伪指令将被编译器替换成机器指令
- ; SUB R4, PC, #0xC

ADRL伪指令



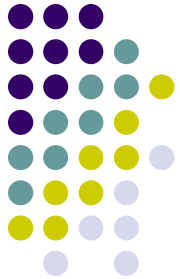
- 中等范围的地址读取伪指令。该指令将基于PC或基于寄存器的地址值读取到寄存器中。ADRL伪指令比ADR伪指令可以读取更大范围的地址。
- ADRL伪指令在汇编时被编译器替换成两条指令。

ADRL伪指令语法



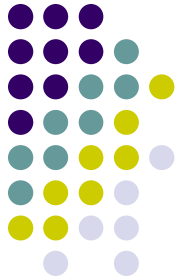
- 语法:
  - ADRL {<cond>} register, expr
  - 其中，register为目标寄存器。expr为基于PC或者基于寄存器的地址表达式，其取值范围如下:
  - 当地址值不是字对齐时
    - 其取值范围为-64KB~64KB。
  - 当地址值是字对齐时
    - 其取值范围为-256KB~256KB。
  - 当地址值是16字节对齐时
    - 其取值范围将更大。

ADRL指示符的代码范例



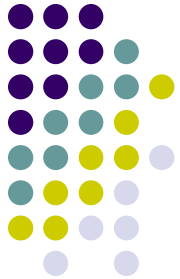
```
            AREA    adrlabel, CODE,READONLY
            ENTRY                           ; Mark first instruction to execute
Start
            BL      func                    ; Branch to subroutine
stop        MOV     r0, #0x18               ; angel_SWIreason_ReportException
            LDR     r1, =0x20026            ; ADP_Stopped_ApplicationExit
            SWI     0x123456                ; ARM semihosting SWI
            LTORG                           ; Create a literal pool
func        ADR     r0, Start               ; => SUB r0, PC, #offset to Start
            ADR     r1, DataArea            ; => ADD r1, PC, #offset to DataArea
            ; ADR   r2, DataArea+4300       ; This would fail because the offset
                                            ; cannot be expressed by operand2
                                            ; of an ADD
            ADRL    r2, DataArea+4300       ; => ADD r2, PC, #offset1
                                            ;    ADD r2, r2, #offset2
            MOV     pc, lr                  ; Return
DataArea    SPACE   8000                    ; Starting at the current location,
                                            ; clears a 8000 byte area of memory
                                            ; to zero
            END
```

空操作伪指令NOP



- NOP伪指令在汇编时将会被替代成ARM中的空操作，比如可能为“MOV R0，R0"指令等。
  - 伪指令格式如下:
  - NOP

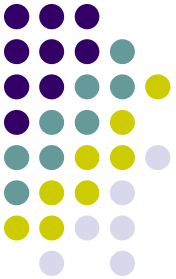
NOP指令的用法



- NOP可用于延时操作，如下面的程序清单所示。
- 软件延时程序清单

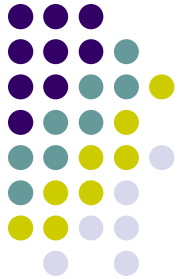
```
    …
DELAY1
    NOP
    NOP
    NOP
    SUBS    R1, R1，#1
    BNE     DELAY1
    …
```

大范围地址读取伪指令LDR



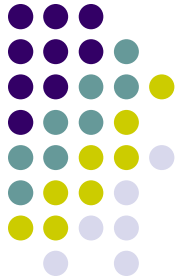
- LDR伪指令用于加载32位的立即数或一个地址值到指定寄存器。
- 在汇编编译源程序时，LDR伪指令被编译器替换成一条合适的指令。
- 若加载的常数未超出MOV或MVN的范围，则使用MOV或MVN指令代替该LDR伪指令;
- 否则汇编器将常量放入文字池，并使用一条程序相对偏移的LDR指令从文字池读出常量。
- 与ARM存储器访问指令的LDR相比，伪指令的LDR的参数有“=”符号。

伪指令LDR格式



- 伪指令格式如下:
  - LDR{cond} register， = expr/label－expr
- 其中:
  - register　　加载的目标寄存器。
  - expr　　32位立即数。
  - Label－expr　基于PC的地址表达式或外部表达式。

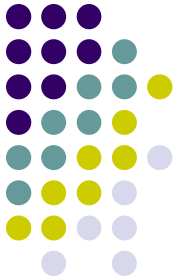
伪指令LDR举例



- LDR伪指令举例如下:

```
LDR   R0，  = 0x12345678
；加载32位立即数0x12345678
LDR   R0，=DATA_BUF + 60
；加载DATA_BUF地址 + 60
…
LTORG
；声明文字池
…
```

加载32位立即数程序举例



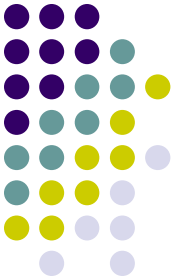
- 伪指令LDR常用于加载芯片外围功能部件的寄存器地址(32位立即数)，以实现各种控制操作，如下面的程序清单所示。

```
…
LDR  R0，=IOPIN  ;加载寄存器IOPIN的地址
LDR  R1，[R0]    ;读取IOPIN寄存器的值
…
LDR  R0，  = IOSET
LDR  R1，=0x00500500
STR  R1，[R0]    ;IOSET=0x00500500
```

第9讲重点

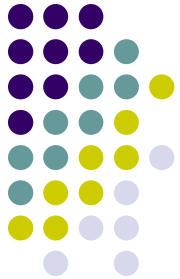
- ARM指令集与x86指令集的主要不同之处
- ARM指令的基本格式
- 第2操作数
- ARM处理器的8种寻址方式
- ARM存储器访问指令
  - 加载/存储指令的4种地址计算方式
  - 多寄存器加载/存储指令
- ARM数据处理指令
- 软中断指令SWI
- ARM伪指令

第9讲复习题与思考题



- 除了本课件列出的ARM指令集与x86指令集不同点外，请你试列出更多的不同点。
- 就ARM处理器而言，相对寻址时的基准地址是什么？
- 数据块传送指令与堆栈指令有何不同？
- 为什么讲ARM的SWP指令并非简单地执行SWAP（数据交换）操作。
- 请说明ARM指令第2个操作数是如何定义的？
- 如何区别前索引偏移和后索引偏移？
- 使用LDR指令时需要特别注意的是什么？
- 如何编写开中断和关中断的汇编指令段？
- 如何让中断服务子程序获得SWI指令带有的立即数？

第4次习题布置

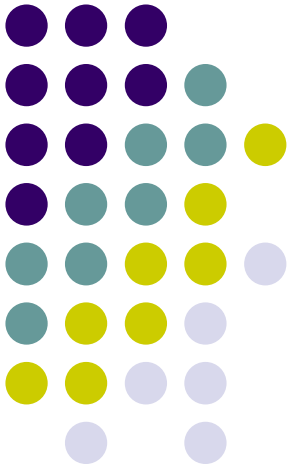


- 题1，使用两种类型的第2操作数，分别编写3条ARM指令，并且说明这些指令中的第2操作数的形成方法。
- 题2，如何辨别LDR指令是ARM机器指令，还是伪指令。请你各举出3条数据传送LDR指令的例子和3条LDR伪指令的例子。
- 题3，LDR和STR指令有前变址、后变址和惠写前变址三种变址模式，请你举例说明之。

《嵌入式系统原理与开发》

2008年春季

第10讲



南京大学计算机系　俞建新主讲

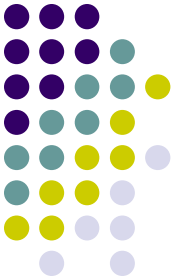
第5章 ARM指令集和汇编语言程序

- 本章主要介绍以下内容:
  - ARM指令集的基本特点
    - 与Thumb指令集的区别
    - 与x86处理器的区别
    - ARM指令格式
  - ARM寻址方式
  - ARM指令集分类详解
  - ARM汇编语言的指示符
  - ARM汇编语言语句格式
  - ARM汇编语言程序格式
  - ARM汇编语句格式和程序格式进阶
  - ARM汇编语言程序举例

本讲主要参考文献

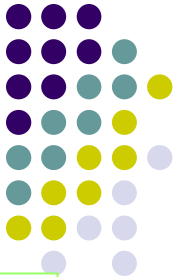
- **ARM公司英文资料:**
  - ADS_AssemblerGuide_B.pdf
  - DDI0100E_ARM_ARM.pdf
- **中文图书**
  - 《ARM体系结构与编程》，清华大学出版社
  - 《嵌入式系统基础教程》，机械工业出版社

5.4 ARM汇编语言程序的指示符



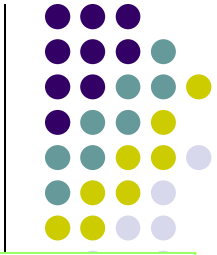
- ARM汇编语言源程序中语句由指令、指示符和宏指令组成。
- 在ARM中将directive称做指示符
  - ARM的指示符指令相当于x86的伪指令
- 在ARM中pseudo-instruction被称为伪指令
  - ARM指令集中只有4条伪指令
- 而宏指令则是通过指示符定义的。
  - 使用MACRO和 MEND指示符

5.4.1 符号定义指示符



- 符号定义(Symbol definition)指示符用于定义ARM汇编程序中的变量，对变量进行赋值以及定义寄存器名称。包括以下指示符:
  - GBLA，GBLL及GBLS 声明全局变量;
  - LCLA，LCLL及LCLS 声明局部变量;
  - SETA，SETL及SETS 给变量赋值;
  - RLIST 为通用寄存器列表定义名称;
  - CN 为协处理器的寄存器定义名称;
  - CP 为协处理器定义名称;
  - DN及SN 为VFP的寄存器定义名称;
  - FN 为FPA的浮点寄存器定义名称。

5.4.2 数据定义指示符

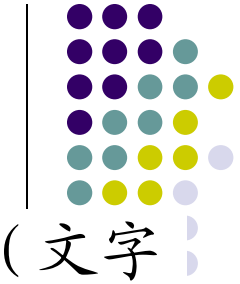


- 数据定义(Data definition)指示符包括以下的指示符：
  - LTORG 声明一个数据缓冲池(literal pool)的开始;
  - MAP 定义一个结构化的内存表(storage map)的首地址;
  - FIELD 定义结构化的内存表中的一个数据域(field);
  - SPACE 分配一块内存单元，并用0初始化;
  - DCB 分配一段字节的内存单元，并用指定的数据初始化;
  - DCD及DCDU 分配一段字的内存单元，并用指定的数据初始化;
  - DCDO 分配一段字的内存单元，并将单元的内容初始化成该单元相对于静态基值寄存器的偏移量。

数据定义指示符（续）

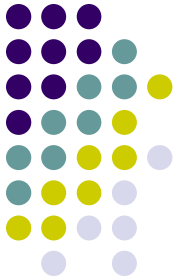
- DCFD及DCFDU 分配一段双字的内存单元，并用双精度的浮点数据初始化。
- DCFS及DCFSU 分配一段字的内存单元，并用单精度的浮点数据初始化。
- DCI 分配一段字节的内存单元，用指定的数据初始化，指定内存单元中存放的是代码，而不是数据。
- DCQ及DCQU 分配一段双字的内存单元，并用64位的整数数据初始化。
- DCW及DCWU 分配一段半字的内存单元，并用指定的数据初始化。
- DATA 在代码段中使用数据。现已不再使用，仅用于保持向前兼容。

LTORG指示符



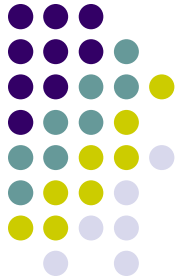
- LTORG指示符要求汇编器立即安排一个数据缓冲池（文字池）。
- 语法:
  - LTORG
  - 使用说明：通常ARM汇编器把数据缓冲池放在代码段的最后面，即下一个代码段开始之前，或者END指示符之前。
  - 当程序中使用LDMFD之类的指令时，数据缓冲池可能越界。这时可以使用LTORG指示符定义数据缓冲池。以防止越界发生。通常，大的代码段可以使用多个数据缓冲池。
  - LTORG指示符通常放在无条件跳转指令之后。或者子程序返回之后，这样处理器就不会错误地把数据缓冲池中的数据当作指令来执行。

LTORG指示符使用举例



```
                AREA  EXP_1,  CODE,  READONLY
START           BL      FUNC1
FUNC1
;CODE
                LDR  R1,=55555555
;产生形如LDR R1, [PC, #offset to Literal Pool]的指令
;CODE
                MOV  PC, LR ; 子程序结束
                LTORG       ; 定义数据缓冲池 &55555555
DATA            SPACE   46  ; 从当前位置开始分配46字节
                END         ; 默认的数据缓冲池为空
```

使用MAP和FIELD描述数据结构

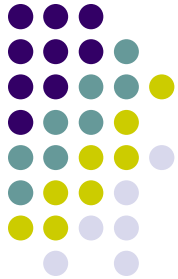


- ADS编译器使用MAP和FIELD两个指示符描述数据结构。
- MAP定义了数据结构的起始地址。
- FIELD用来定义数据结构中的字段

MAP指示符

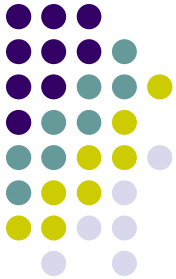
- ^是MAP的同义词
- MAP语法
  - MAP expr {, base-register}
  - 其中：expr为数字表达式或者是程序中的标号。
  - 当指令中没有base-register时，expr即为结构化内存表的首地址。此时内存表的位置计数器{VAR}设置成该地址值。
  - 当expr 为程序中的标号，该标号必须是已经定义过的。
  - base-register为一个寄存器。当指令中包含这一项时，结构化内存表（数据结构）的首地址为expr和base-register寄存器的和。

MAP指示符使用举例



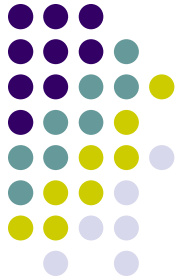
- 下面是MAP指示符汇编语句举例
  - MAP 0xA0FF, R8　　;数据结构首地址为R8+0xA0FF
  - ; 也就是VAR的值成为R8+0xA0FF
  - MAP 0xC12305　　;数据结构起始位置是0xC12305
  - MAP readdata+86　　;数据结构起始位置是readdata+86
  - ; readdata标号已经定义过了
  - MAP 0, R7　　;数据结构起始位置是R7中的数据
  - MAP 0x1FF, R6　　;数据结构起始位置是R7中的数据
  - ; 加上0x1FF

FIELD指示符



- #是FIELD的同义词
- FIELD语法
  - {label} FIELD expr
  - 其中： {label} 为可选项。当指令中包含{label} 时，标号label的值为当前内存表位置计数器{VAR}的值。汇编编译器执行完此条FIELD指示指令后，内存表位置计数器的值将加上expr。
  - expr表示本字段（数据项）在内存表中所占的字节数。

MAP和FIELD指示符使用说明



- FIELD指示符说明一个数据项所需要的内存空间，并且为这个数据项提供一个标号。
- MAP指示符中的base-register寄存器对于其后所有的FIELD指示符定义的数据项是默认使用的，直至遇到新的包含base-register寄存器的MAP指示符。
- MAP指示符和FIELD指示符仅仅定义数据结构，它们并不实际分配内存空间（这一条需要核实）。

MAP和FIELD联合使用举例

```
startofdata     EQU     0x0100   ;
                MAP     startofdata
int_a           field   4
int_b           field   4
str_one         field   64
k_array         field   128
bit_mask        field   4
```

本例中数据结构的起始位置映射到固定（绝对）地址0x0100，然后为各个字段分配空间。也可以用LDR和STR之类的指令对结构体中的字段进行操作。例如使用下面的语句：

```
LDR   R4, int_a
```

MAP和FIELD联合使用举例（续1）

```
; 本代码清单是上一个幻灯片的等效代码
startofdata     EQU     0x0100
                ^       startofdata
int_a           #   4
int_b           #   4
str_one         #   64
k_array         #   128
bit_mask        #   4
```

MAP和FIELD联合使用举例（续2）

对上述MAP和FIELD联合使用举例代码中的int_a赋值的代码清单如下:

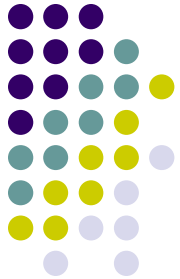
```
MOV   R0, #20
LDR   R1, =int_a
;也可以使用语句LDR  R1, int_a
STR   R0, [R1]
```

用EQU指示符完成MAP和FIELD联合使用举例代码的等效操作

实际上，使用MAP和FIELD描述的数据结构也可以使用EQU描述。参见下面的代码清单。

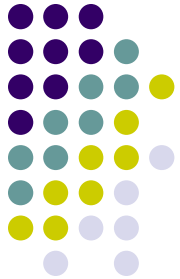
```
startofdata     EQU     0x0100
int_a           EQU     startofdata
int_b           EQU     int_a+4
str_one         EQU     int_b+4
k_array         EQU     str_one+64
bit_mask        EQU     k_array+128
; 这种EQU指示符的代码结构修改繁琐，不推荐使用
```

SPACE指示符



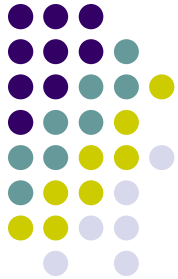
- SPACE用于分配一块内存单元，并且用0初始化。
- %是SPACE的同义词
- 语法:
  - {label} MAP expr
  - 其中： label是可选项； expr是本指示符分配的内存字节数。
  - 示例
  - Pro_data SPACE 1290 ;分配1290个字节内存单元
    ;并且将该存储区初始化为0

相对地址



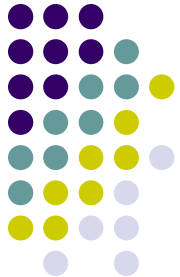
- 相对地址有两种:
  - 一种是以寄存器作为基地址的相对偏移
    - 可以分为两类
      - 基于相对地址的内存表
      - 基于寄存器的内存表
  - 另外一种是基于程序的相对偏移

基于寄存器的相对偏移



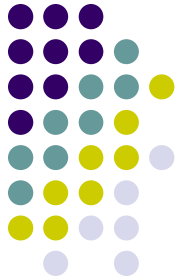
```
; 基于相对地址的内存表示代码如下
                MAP     0
int_a           FIELD   4       ;为int_a分配4个字节
int_b           FIELD   4       ;为int_b分配4个字节
str_one         FIELD   64      ;为str_one分配64个字节
k_array         FIELD   128     ;为k_array分配128个字节
bit_mask        FIELD   4       ;为bit_mask分配4个字节
;这种情况下的基址寄存器是不固定的
```

程序相对偏移



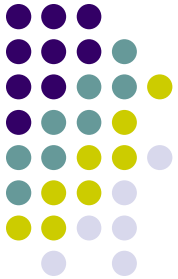
- 以程序偏移地址作为基地址相对以寄存器为基地址要简单得多。首先要为数据结构申请存储空间，然后将申请的空间首部地址作为基地址。
- 代码如下面的幻灯片所示：

程序相对偏移（续）



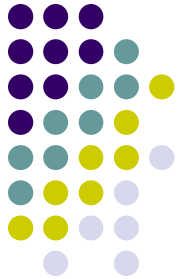
```
; 使用程序相对偏移为基地址
s_label         SPACE   280
                ; 申请空间用来存放数据结构
                MAP     s_label
                ; 将分配空间的起始地址作为基地址
int_a           FIELD   4
int_b           FIELD   4
strone          FIELD   64
k_array         FIELD   128
bit_mask        FIELD   4   ;这种情况下基地址由编译器安排
```

5.4.3 汇编控制指示符



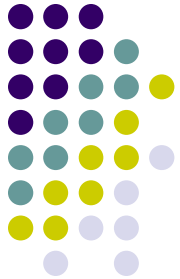
- 汇编控制(Assembly control)指示符包括下面的指示符:
  - IF，ELSE及ENDIF
    - 汇编或者不汇编一段源代码
  - WHILE及WEND
    - 条件重复汇编相同的一段源代码
  - MACRO及MEND
    - 标识宏定义开始与结束
  - MEXIT
    - 用于从宏跳转出去

5.4.4 信息报告指示符



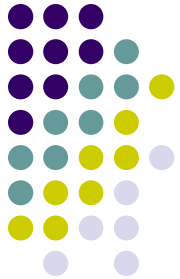
- 信息报告(Reporting)指示符包括下列指示符:
  - **ASSERT**
    - 在汇编编译器对汇编程序的第二趟扫描中，如果其中的ASSERTION中条件不成立，ASSERT伪操作将报告该错误信息。
  - **INFO**
    - 支持第一二趟汇编扫描时报告诊断信息。
  - **OPT**
  - **TTL及SUBT**

5.4.5 其他指示符



- 这些杂类的指示符包括:
  - ALIGN
  - AREA
  - CODE16及CODE32
  - END
  - ENTRY
  - EQU
  - EXPORT或GLOBAL

其他的指示符（续）



- EXTERN
- GET或INCLUDE
- IMPORT
- INCBIN
- KEEP
- NOFP
- REQUIRE
- REQUIRE8及PRESERVE8
- RN
- ROUT

5.4.5.1 AREA

- AREA指示符用于定义一个代码段或者数据段。
  - 语法格式
  - AREA sectionname{，attr}{，attr}....
  - 其中:
    - sectionname为所定义的代码段或者数据段的名称。如果该名称是以数字开头的，则该名称必须用“|”括起来，如|1_datasec|。还有一些代码段具有约定的名称，如|.text|表示C语言编译器产生的代码段或者是与C语言库相关的代码段。
    - Attr是该代码段(或者程序段)的属性。
    - 在AREA指示符中，各属性间用逗号隔开。

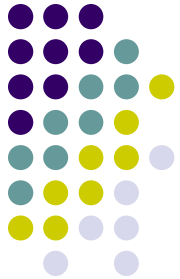
AREA的属性

- 下面列举主要的属性:
  - ALIGN=expression。默认的情况下，ELF的代码段和数据段是4字节对齐的。
    - Expression可以取0～31的数值，相应的对齐方式为($2^{expression}$)字节对齐。如expression=3时为8字节对齐。
  - ASSOC=section。指定与本段相关的ELF段。任何时候连接section段也必须包括sectionname段。
  - CODE 定义代码段。默认属性为READONLY。
  - COMDEF 定义一个通用的段。该段可以包含代码或者数据。在个源文件中，同名的COMDEF段必须相同。

AREA的属性（续）

- COMMON 定义一个通用的段。该段不包含任何用户代码和数据，连接器将其初始化为0。各源文件中同名的COMMON 段公用同样的内存单元，连接器为起分配合适的尺寸。
- DATA 定义数据段。默认属性为READWRITE。
- NOINIT 指定本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，或者将个内存单元值初始化为0。
- READONLY 指定本段为只读，代码段的默认属性为READONLY。
- READWRITE 指定本段为可读可写，数据段的默认属性为READWRITE。

AREA指示符举例



- 举例
  - 下面的指示符定义了一个代码段，代码段的名称为Mainpro，属性为READONLY。

```
AREA Mainpro, CODE, READONLY
;code segment
```

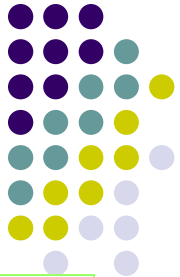
5.4.5.2 EQU

- EQU指示符为数字常量、基于寄存器的值和程序中的标号(基于PC的值)定义一个字符名称。
- *是EQU的同义词。
  - 语法格式
  - name EQU expr{, type}
  - 其中:
    - expr为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。
    - name为EQU指示符为expr定义的字符名称。
    - type 当expr为32位常量时，可以使用type指示expr表示的数据的类型。

EQU（续）

- type有下面3种取值:
  - CODE16
  - CODE32
  - DATA
- 使用说明
  - EQU指示符的作用类似于C语言中的#define，用于为一个常量定义字符名称。
- 示例
  - abcd EQU 2　　;定义abcd符号的值为2
  - abcd EQU label+16　;定义abcd符号的值(label+16)
  - addr1 EQU 0xlC, CODE32　;定义addr1符号值为;绝对地址值0xlC，而且该处为ARM指令。

5.4.5.3 ENTRY



- ENTRY指示符指定程序的入口点
  - 语法格式
  - ENTRY
  - 使用说明
    - 一个程序(可以包含多个源文件)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)。
  - 示例

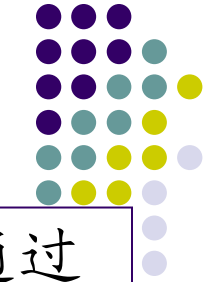
    AREA example, CODE, READONLY

    ENTRY　;应用程序的入口点

5.4.5.4 CODE16和CODE32

- CODE16指示符告诉汇编编译器后面的指令序列为16位的Thumb指令。
- CODE32指示符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
  - 使用说明
    - 当汇编源程序中同时包含ARM指令和Thumb指令时，使用CODE16指示符告诉汇编编译器后面的指令序列为16位的Thumb指令；使用CODE32指示符告诉汇编编译器后面的指令序列为32位的ARM指令。但是，CODE16指示符和CODE32指示符只是告诉编译器后面指令的类型，该指示符本身并不进行程序状态的切换。

CODE16/CODE32举例



在下面的例子中，程序先在ARM状态下执行，然后通过BX指令切换到Thumb状态，并跳转到相应的Thumb指令处执行。在Thumb程序入口处用CODE16指示符标识下面的指令为Thumb指令。参看下面的指令段:

```
                ……
        AREA  ChangeState, CODE, READONLY
        CODE32          ;指示下面的指令为ARM指令
        LDR  r0, =start+1
        BX  r0          ;切换到Thumb，并跳转到start处执行
        CODE16          ;指示下面的指令为Thumb指令
start   MOV r1, #10
```

5.4.5.5 END

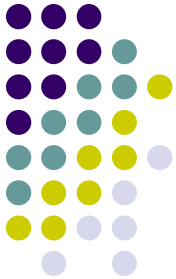
- END指示符告诉编译器已经到了源程序结尾。
  - 语法格式:
  - END
  - 使用说明:
    - 每一个汇编源程序都包含END指示符，以告诉本源程序的结束。
  - 示例:

```
AREA example CODE，READONLY
……
……
END
……
```

5.4.5.6 ALIGN

- ALIGN指示符通过添加补丁字节使当前位置满足一定的对齐方式。
  - 语法格式
  - ALIGN {expr{，offset}}
  - 其中，expr为数字表达式，用于指定对齐方式。可能的取值为2的次幂，如1、2、4、8等。如果指示符中没有指定expr，则当前位置对齐到下一个字边界处。offset为数字表达式。当前位置对齐到下面形式的地址处：offset+n*expr。

ALIGN（续1）

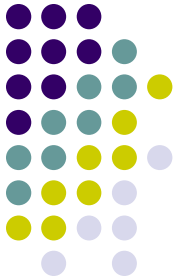


- 使用说明
  - 下面的情况中，需要特定的地址对齐方式:
  - Thumb的宏指令ADR要求地址是字对齐的，而Thumb代码中地址标号可能不是字对齐的。这时就要使用指示符ALIGN 4使Thumb代码中的地址标号字对齐。
  - 由于有些ARM处理器的CACHE采用了其他对齐方式，如16字节的对齐方式，这时使用ALIGN指示符指定合适的对齐方式可以充分发挥该CACHE的性能优势。
  - LDRD及STRD指令要求内存单元是8字节对齐的。这样在为这两个指令分配的内存单元前要使用ALIGN 8实现8字节对齐方式。
  - 地址标号通常自身没有对齐要求。而在ARM代码中要求地址标号是字对齐的，在Thumb代码中要求字节对齐。这样需要使用合适的ALIGN指示符来调整对齐方式。

5.4.5.7 EXPORT及GLOBAL

- EXPRORT声明一个符号可以被其他文件引用。相当于声明了一个全局变量。GLOBAL是EXPORT的同义词。
  - 语法格式
  - EXPORT symbol {[WEAK]}
    - 其中，symbol为声明的符号名称，大小写敏感。
    - [WEAK]选项声明其他的同名符号优先于本符号被引用。
  - 使用说明
    - 使用EXPORT指示符声明一个源文件中的符号，使得该符号可以被其他源文件引用。
  - 示例
    - AREA Example，CODE，READONLY
    - EXPORT Do_Add ；函数名称DoAdd可以被引用

5.4.5.8 IMPORT



- IMPORT指示符告诉编译器当前的符号不是在本源文件中定义的，而是在其他源文件中定义的，在本源文件中可能引用该符号，而且不论本源文件是否实际引用该符号，该符号都将被加入到本源文件的符号表中。
  - 语法格式
  - IMPORT symbol {[WEAK]}
    - 其中:
    - symbol为声明的符号的名称。它是区分大小写的。
    - [WEAK] 指定这个选项后，如果symbol在所有的源文件中都没有被定义，编译器也不会产生任何错误信息，同时编译器也不会到当前没有被INCLUDE进来的库中去查找该符号。

IMPORT（续）

- 使用说明
  - 使用IMPORT指示符声明一个符号是在其他源文件中定义的。如果连接器在连接处理时不能解析该符号，而IMPORT指示符中没有指定[WEAK]选项，则连接器将会报告错误。如果连接器在连接处理时不能解析该符号，而IMPORT指示符中指定了[WEAK]选项，则连接器将不会报告错误，而是进行下面的操作:
  - <1>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或者BL指令相当于一条NOP指令。
  - <2>其他情况下该符号被设置为0。

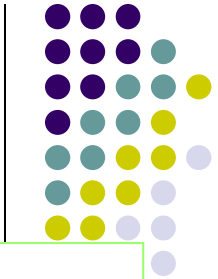
5.4.5.9 EXTERN

- EXTERN指示符告诉编译器当前的符号不是在本源文件中定义的，而是在其他源文件中定义的，在本源文件中可能引用该符号。如果本源文件没有实际引用该符号，该符号都将不会被加入到本源文件的符号表中。
  - 语法格式
  - EXTERN symbol {[WEAK]}
    - 其中，symbol为声明的符号的名称。它是区分大小写的。
    - [WEAK] 指定该选项后，如果symbol在所有的源文件中都没有被定义，编译器也不会产生任何错误信息，同时编译器也不会到当前没有被INCLUDE进来的库中去查找该符号。

EXTERN（续1）

- 使用说明
  - 使用EXTERN指示符声明一个符号是在其他源文件中定义的。如果连接器在连接处理时不能解析该符号，而EXTERN指示符中没有指定[WEAK]选项，则连接器将会报告错误。如果连接器在连接处理时不能解析该符号，而EXTERN指示符中指定了[WEAK]选项，则连接器将不会报告错误，而是进行下面的操作:
  - <1>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或者BL指令相当于一条NOP指令。
  - <2>其他情况下该符号被设置为0。

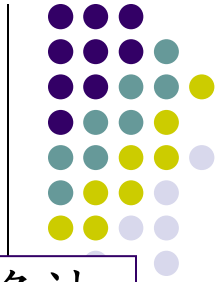
EXTERN（续2）



- 示例
  - 下面的代码测试是否连接了C++库，并根据结果执行不同的代码

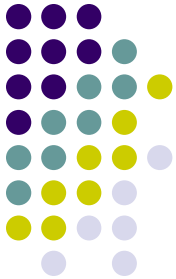
```
AREA  Example，  CODE，READONLY
EXTERN  _CPP_INITIALIZE[WEAK]
;如果连接了c++库则读取
;函数_CPP_INITIALIZE地址
LDR  r0，_CPP_INITIALIZE
CMP  r0，#0              ;Test if zero.
BEQ  nocplusplus
;如果没有连接C++库，则跳转到nocplusplus
```

5.4.5.10 GET及INCLUDE



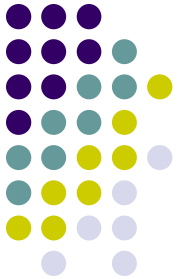
- GET指示符将一个源文件包含到当前源文件中，并将被包含的文件在其当前位置进行汇编处理。
- INCLUDE是GET的同义词。
  - 语法格式
  - GET filename
    - 其中，filename为被包含的源文件的名称。这里可以使用路径信息。
  - 使用说明（1）
    - 通常可以在一个源文件中定义宏，用EQU定义常量的符号名称，用MAP和FIELD定义结构化的数据类型，这样的源文件类似于C语言中的.H文件。然后用GET指示符将这个源文件包含到它们的源文件中，类似于在C源程序的“include *.h”。

GET及INCLUDE（续1）



- 使用说明（2）
  - 编译器通常在当前目录中查找被包含的源文件。可以使用编译选项-I添加其他的查找目录。同时，被包含的源文件中也可以使用GET指示符，即GET指示符可以嵌套使用。如在源文件A中包含了源文件B，而在源文件B中包含了源文件C。编译器在查找C源文件时将把源文件B所在的目录作为当前目录。
  - GET指示符不能用来包含目标文件(二进制执行文件)。
  - 包含目标文件需要使用INCBIN指示符。

GET及INCLUDE（续2）



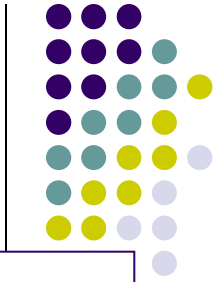
- 示例

```
AREA  Example，CODE，READONLY
GET file1.s                    ;包含源文件file1.s
GET c:\project\file2.s
;包含源文件file2.s，可以包含路径信息
GET c:\program files\file3.s
;包含源文件file3.s，路径信息中可以包含空格
```

5.4.5.11 INCBIN

- INCBIN指示符将一个文件包含到(INCLUDE)当前源文件中，被包含的文件不进行汇编处理。
  - 语法格式
  - INCBIN filename
    - 其中，filename为被包含的文件的名称。这里可以使用路径信息。
  - 使用说明
    - 通常可以使用INCBIN将一个执行文件或者任意的数据包含到当前文件中。被包含的执行文件或数据将被原封不动地放到当前文件中。编译器从INCBIN指示符后面开始继续处理。

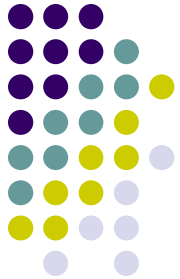
INCBIN（续）



- 使用说明
  - 编译器通常在当前目录中查找被包含的源文件。可以使用编译选项-I添加其他的查找目录。同时，被包含的源文件中也可以使用GET指示符，即GET指示符可以嵌套使用。如在源文件A中包含了源文件B，而在源文件B中包含了源文件C。编译器在查找C源文件时将把源文件B所在的目录作为当前目录。
  - 这里所包含的文件名及路径信息中都不能有空格。
  - 示例

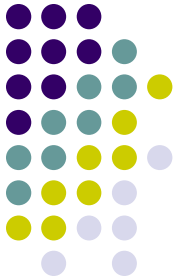
```
AREA Example，CODE，READONLY
INCBIN file1.dat                    ;包含文件file1.dat
INCBIN c:\project\file2.txt          ;包含文件file2.txt
```

5.4.5.12 NOFP



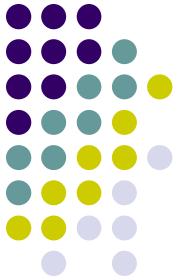
- 使用NOFP指示符禁止源程序中包含浮点运算指令。
  - 语法格式
  - NOFP
  - 使用说明
    - 当系统中没有硬件或软件仿真代码支持浮点运算指令时，使用NOFP指示符禁止在源程序中使用浮点运算指令。这时如果源程序中包含浮点运算指令，编译器将会报告错误。同样如果在浮点运算指令的后面使用NOFP指示符，编译器同样将会报告错误。

5.4.5.13 REQUIRE



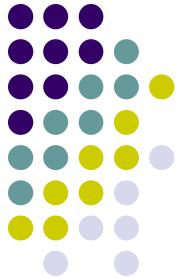
- REQUIRE指示符指定段之间的相互依赖关系。
  - 语法格式
  - REQUIRE label
    - 其中，label为所需要的标号的名称。
  - 使用说明
    - 当进行连接处理时包含了有REQUIRE label指示符的源文件，则定义label的源文件也将被包含。

5.4.5.14 RN



- RN指示符为一个特定的寄存器定义名称。
  - 语法格式
  - name RN expr
    - 其中：expr为某个寄存器的编码。name为本指示符给寄存器expr定义的名称。
  - 使用说明
    - RN指示符用于给一个寄存器定义名称。方便程序员记忆该寄存器的功能。
  - 举例：database RN R11
    - ；将寄存器R11重命名为database
    - ；增加可读性

5.5 ARM汇编语言语句格式



- ARM汇编语言语句格式如下所示:

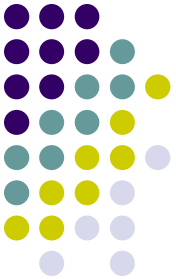
  {symbol}{instruction | directive | pseudo-instruction}
  {;comment}

- 其中:
  - instruction为指令。在ARM汇编语言中，指令不能从一行的行头开始。在一行语句中，指令的前面必须有空格或者符号。
  - Directive是指示符。
  - pseudo-instruction是伪指令。

ARM汇编语言语句格式（续）

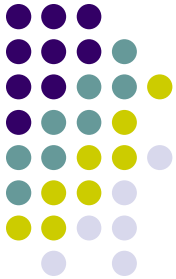
- symbol为符号。在ARM汇编语言中，符号必须从一行的行头开始，并且符号中不能包含空格。在指令和伪指令中符号用作地址标号(label)；在有些指示符中，符号用作变量或者常量。
- comment为语句的注释。在ARM汇编语言中注释以分号“;”开头。注释的结尾即为一行的结尾。注释也可以单独占用一行。

ARM汇编程序编写规范



- 汇编语句格式
  - ARM汇编中，所有标号必须在一行的顶格书写，其后面不要添加符号“:”。
  - 而所有指令均不能顶格书写。
  - ARM汇编器对标识符大小写敏感(即区分大小写字母)，书写标号及指令时字母大小写要一致。
  - 在ARM汇编程序中，ARM指令、伪指令、寄存器名可以全部为大写字母，也可以全部为小写字母，但不要大小写混合使用。
  - 源程序中，语句之间可以插入空行，以使得源代码的可读性更好。

ARM汇编程序编写规范（续）

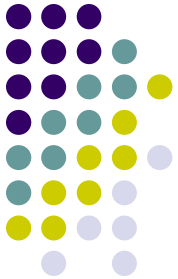


- 格式如下:
  - **[标号] <指令|条件|S> <操作数> [;注释]**
  - 源程序中允许有空行。适当地插入空行，可以提高源程序的可读性。
  - 如果单行代码太长，可以使用字符“\”将其分行。“\”后不能有任何字符，包括空格和制表符等。
  - 对于变量的设置、常量的定义，其标识符必须在一行的顶格书写。
- 下面给出了汇编指令正确和错误的例子

汇编指令正确的例子

```
        …
Str1            SETS  “My  String1.”     ;设置字符串变量Str1
Count           RN  R0                    ;定义寄存器名Count
USR_STACK       EQU  64                   ;定义常量

START           LDR  R0，=0x12345678  ;1235678H
                MOV  R1，#0
LOOP
                MOV  R2，#1
     …
```

汇编指令错误的例子

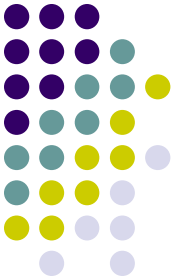


```
  DOB  MOV  R0，#1        ;标号DOB没有顶格书写
ABC:   MOV  R1，#2        ;标号不允许用符号 “: ”修饰
MOV    R2，#3             ;命令不允许顶格书写
Loop   Mov   R2,#3        ;指令中大小写混合
       B     Loop         ;无法跳转到loop标号，大小写
                          ;不一致
```

5.6 ARM汇编语言程序格式

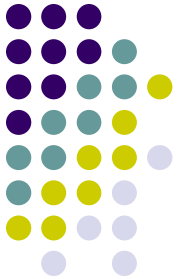
- ARM汇编语言以段(section)为单位组织源文件。
- 段是相对独立的、具有特定名称的、不可分割的指令或者数据序列。
- 段又可以分为代码段和数据段，代码段存放执行代码，数据段存放代码运行时需要用到的数据。
- 一个ARM源程序至少需要一个代码段，大的程序可以包含多个代码段和数据段。

ARM汇编源程序和映像文件



- ARM汇编语言源程序经过汇编处理后生成一个可执行的映像文件(类似于Windows系统下的EXE文件)。该可执行的映像文件通常包括下面3部分:
  - 一个或多个代码段。代码段通常是只读的。
  - 零个或多个包含初始值的数据段。这些数据段通常是可读写的。
  - 零个或多个不包含初始值的数据段。这些数据段被初始化为0，通常是可读写的。
- 连接器根据一定的规则将各个段安排到内存中的相应位置。源程序中段之间的相邻关系与执行的映像文件中段之间的相邻关系并不一定相同。

ARM汇编源程序基本结构举例



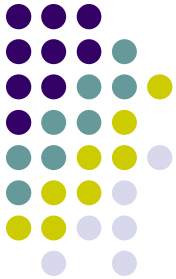
源程序基本结构如下示出:

```
        AREA EXAMPLE1, CODE,  READONLY
        ENTRY
start
        MOV  r0,  #10
        MOV  r1,  #3
        ADD  r0,  r0, r1
        END
```

案例ARM汇编源程序说明

- 在ARM汇编语言源程序中，指示符AREA定义一个段。
- AREA指示符表示了一个段的开始，同时定义了这个段的名称及相关属性。在本例中定义了一个只读的代码段，其名称为EXAMPLE1。
- ENTRY指示符标识了程序执行的第一条指令。一个ARM程序中可以有多个ENTRY，至少要有一个ENTRY。初始化部分的代码以及异常中断处理程序中都包含了ENTRY。如果程序包含了C代码，C语言库文件的初始化部分也包含了ENTRY。
- 本程序的程序体部分实现了一个简单的加法运算。
- END指示符告诉汇编编译器源文件结束。每一个汇编模块必须包含一个END指示符，指示本模块结束。

汇编语言子程序调用



- 在ARM汇编语言中，子程序调用是通过BL指令完成的。BL指令的语法格式如下:
  - BL subname
  - 其中，subname是调用的子程序的名称。
- BL指令完成两个操作：将子程序的返回地址放在LR寄存器中，同时将PC寄存器值设置成目标子程序的第一条指令地址。
- 在子程序返回时可以通过将LR寄存器的值传送到PC寄存器中来实现。
- 子程序调用时通常使用寄存器R0~R3来传递参数和返回结果。

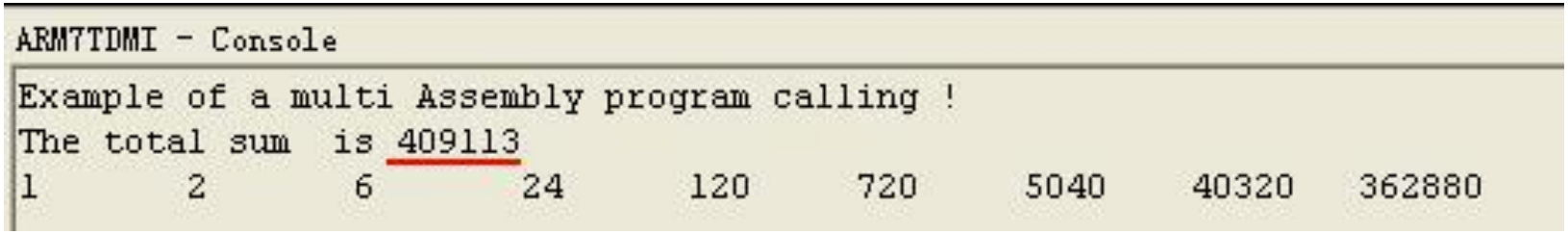
汇编子程序调用举例

- 子程序DOADD完成加法运算，操作数放在R0和R1寄存器中，结果放在R0中。

```
                AREA EXAMPLE2, CODE, READONLY
                ENTRY
start           MOV  r0, #10          ;  R0设置输入参数
                MOV  r1, #3           ;  R1设置输入参数
                BL     doadd          ;  调用子程序doadd
doadd           ADD  r0,  r0,  r1     ;  子程序
                MOV   pc，lr          ;  从子程序中返回
                END
```

ARM汇编子程序的嵌套调用举例-1

- 这里给出的ARM汇编程序嵌套调用范例程序做如下计算:
  - **求自然数1到n的阶乘的总和，半主机方式输出**
  - 运算结果如下图所示。第3行显示的是1~9的阶乘，第2行显示的是1!+2!+3!+4!+5!+6!+7!+8!+9!之总和。



ARM汇编子程序的嵌套调用举例-2

```
/* This program is semihosting output mode */
/* First the main function call assembly summing subprogram */
/* Then the summing subprogram call assembly factorial subprogram */
#include <stdio.h>
extern int asmFac(int n);

struct factorial_sum{
int cal_fn;
int sum_fn;
int fn[9];
};

extern struct factorial_sum * summing(struct factorial_sum * arg1);
```

ARM汇编子程序的嵌套调用举例-3

```
int main(void)
{    int j;
     struct factorial_sum fac={ 9, 0, {1,1,1,1,1,1,1,1,1}};   //设置参数
     struct factorial_sum * result;      //申请变量作为返回值

     printf("Example of a multi Assembly program calling !\n");

     result=summing(&fac); //调用求和函数 R0存放的是FAC变量的首地址
     printf("The total sum  is %d\n", result->sum_fn);  //输出结果
     for(j=0; j<9; j++){
                         printf("%d\t",(result->fn)[j]);
                         }
}
```

ARM汇编子程序的嵌套调用举例-4

```
;the details of parameters transfer comes from ATPCS
;if there are more than 4 args, stack will be used
        EXPORT summing
        IMPORT asmFac       ;说明用到了其他文件中的子汇编程序
        AREA SUMMING, CODE, READONLY
summing
            STMFD SP!, {R4-R5}
            ldr     r1,  [r0]           ;r1=cal_fn
            mov     r2, #1              ;将r2设置为当前需要计算的阶乘数,
                                        ;它从1变化到cal_fn
            add     r3, r0, #8          ;将r3指向fn数组
            mov     r5, #0              ;r5为总和,初始值置为0
loop
            cmp     r1, r2              ;将cal_fn与当前所需计算的阶乘值比较
            blt     back                ;如果小于,则返回
```

ARM汇编子程序的嵌套调用举例-5

```
        STMFD SP!, {R0-R3,lr}   ;保存ro~r3,lr
                                ;因为调用了外部文件的汇编子程序
        mov     r0, r2          ;将r0设置为当前所需计算的阶乘值
        bl      asmFac          ;调用阶乘函数
        mov     r4, r0          ;将返回值(阶乘)存在r4中
        ldmfd   SP!, {R0-R3,lr} ;恢复先前保存的寄存器值
        str     r4, [r3]        ;将计算所得的阶乘值存入数组中
        add     r3, r3, #4      ;将r3指向数组的下一个
        add     r5, r5, r4      ;将阶乘值加入总和
        add     r2, r2, #1      ;计算下一个阶乘
        b loop                  ;循环
back
        str     r5, [r0, #4]    ;将计算所得的总和存入结构体中
        ldnfd   SP!, {R4-R5}    ;恢复寄存器值
        mov     pc, lr          ; summing子程序返回
        END                     ;汇编代码结束
```

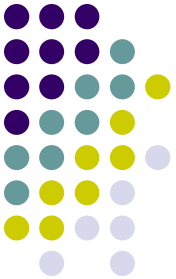
ARM汇编子程序的嵌套调用举例-6

```
;the details of parameters transfer comes from ATPCS
;if there are more than 4 args, stack will be used
;这个ARM汇编函数也可以被C函数调用，符合ATPCS规范
;int asmFac(int n)
    EXPORT asmFac
    AREA ASMFILE, CODE, READONLY

asmFac
        mov         r1, r0      ;r1=n
loop
        subs    r1,     r1, #1  ;将r1减1
        mulgt   r0,     r1, r0  ;如果大于0,则乘上r1(相当与n*(n-1))
        bgt             loop    ;如果大于0,继续
        mov  pc, lr             ;asmFac子程序返回
        END                     ;汇编代码结束
```

第10讲重点

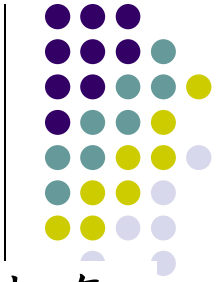
- ARM汇编语言代码中使用的指示符
  - 符号定义指示符
  - 数据定义指示符
  - 汇编控制指示符
  - 信息报告指示符
- ARM汇编语言语句格式
- ARM汇编语言程序格式

第10讲复习题与思考题



1 何谓ARM汇编语言的指示符?
2 ARM汇编语言的pseudo-instruction是什么指令?
3 ARM汇编语言中的变量是如何定义的?
4 全局变量和局部变量的作用范围各是什么?
5 ARM汇编程序的变量是如何赋值的?
6 数据缓冲池(literal pool，也译成文字池)是怎么一回事?
7 MAP指示符的功能是什么?
8 SPACE指示符的功能是什么?
9 DCD指示符的功能是什么?

第10讲复习题与思考题（续）



10 请说明下列3条指令执行完毕后，分配了多少字节的内存空间。

```
DISPTAB     DCB   0x30, 0x43, 0x76, 0x20
            DCB   -120, 255, 36, 42
Nullstring  DCB   “Welcome to Nanjing!”, 0
            ;构造一个以NULL结尾的字符串
```

11 ARM汇编程序中的宏是如何定义的？

12 ARM汇编语言有哪些杂类的伪操作？

13 如何告诉汇编编译器下面的指令序列是ARM指令还是Thumb指令？

14 如何使用本源文件以外的符号？

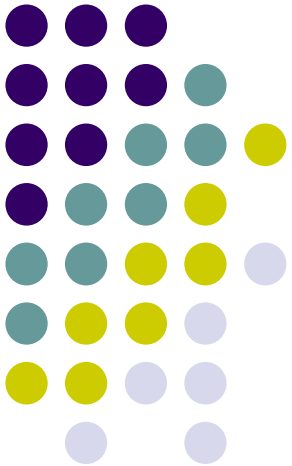
15 如何将本源文件的符号开放给外部源文件使用？

16 EQU伪指令相当于C语言的什么语句？

《嵌入式系统原理与开发》

2008年春季

第11讲



南京大学计算机系　俞建新主讲

第5章 ARM指令集和汇编语言程序

- 本章主要介绍以下内容:
  - ARM指令集的基本特点
    - 与Thumb指令集的区别
    - 与x86处理器的区别
    - ARM指令格式
  - ARM寻址方式
  - ARM指令集分类详解
  - ARM汇编语句格式和程序格式
  - ARM汇编语言的指示符
  - ARM汇编程序标准与规范
  - 典型ARM汇编语言程序举例

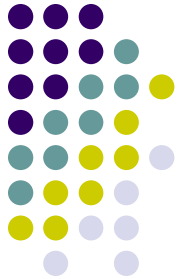
本讲主要参考文献

- **ARM公司英文资料:**
  - ADS_AssemblerGuide_B.pdf
  - DDI0100E_ARM_ARM.pdf
- **中文图书**
  - 《ARM体系结构与编程》，清华大学出版社
  - 《嵌入式系统基础教程》，机械工业出版社

讲授内容

- ATPCS 和AAPCS规范要点
- ARM编译器保有的特定关键字
- 典型ARM汇编语言程序举例
  - ARM汇编程序基本结构
  - C程序和CPP程序调用ARM汇编子程序
  - ARM汇编程序调用ARM汇编子程序
  - 整数除法子程序
  - S3C44B0X处理器的启动代码
    - 44BInit.s、option.s、memcfg.s

5.7 ARM汇编程序规范

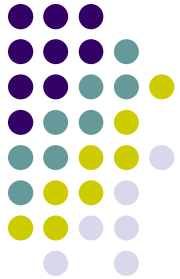


- 寄存器的使用规则
- 堆栈使用规则
- 参数传递规则

ATPCS

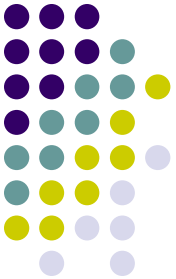
- ATPCS（ARM-Thumb Procedure Call Standard）规定了一些子程序间调用的基本规则，这些规则包括子程序调用过程中寄存器的使用规则，数据栈的使用规则，参数的传递规则。有了这些规则之后，单独编译的C语言程序就可以和汇编程序相互调用。
- 使用ADS的C语言编译器编译的C语言子程序满足用户指定的ATPCS类型。而对于汇编语言来说，则需要用户来保证各个子程序满足ATPCS的要求。

AAPCS



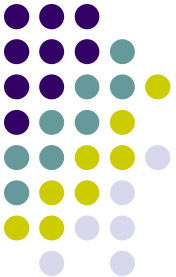
- 2007年ARM公司正式推出了AAPCS标准
  - ARM Archtecture Procedure Call Standard
- AAPCS是ATPCS的改进版
- 目前，AAPCS和ATPCS都是可用的标准

寄存器的使用规则



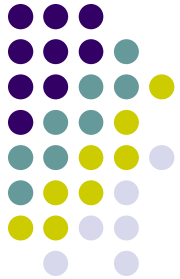
- 子程序间通过寄存器R0～R3来传递参数。这时，寄存器R0～R3可记作a0～a3。被调用的子程序在返回前无需恢复寄存器R0～R3的内容。
- 在子程序中，使用寄存器R4～R11来保存局部变量。这时，寄存器R4～R11可以记作v1～v8。如果在子程序中使用了寄存器v1～v8中的某些寄存器，则子程序进入时必须保存这些寄存器的值，在返回前必须恢复这些寄存器的值。在Thumb程序中，通常只能使用寄存器R4～R7来保存局部变量。
- 寄存器R12用作过程调用中间临时寄存器，记作IP。在子程序之间的连接代码段中常常有这种使用规则。

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

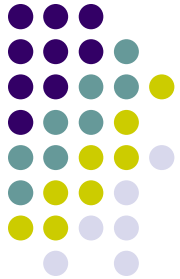


- ATPCS规定堆栈为FD类型，即满递减堆栈，并且对堆栈的操作是8字节对齐。
- 对于汇编程序来说，如果目标文件中包含了外部调用，则必须满足下列条件:

  （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。

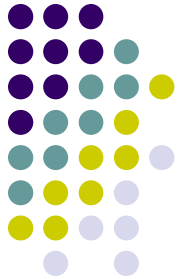
参数个数可变子程序参数传递规则

- 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

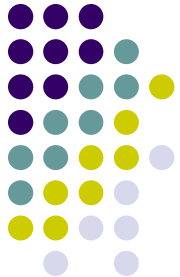


- 子程序中结果返回的规则如下:
  - 结果为一个32位整数时，可以通过寄存器R0返回;
  - 结果为一个64位整数时，可以通过寄存器R0和Rl返回;
  - 结果为一个浮点数时，可以通过浮点运算部件的寄存器f0、d0或s0来返回;
  - 结果为复合型浮点数（如复数）时，可以通过寄存器f0 ~ fn或d0 ~ dn来返回;
  - 对于位数更多的结果，需要通过内存来传递。

5.7.2 ARM编译器保有的特定关键字

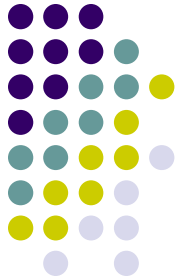
- ARM编译器支持一些对ANSI C进行扩展的关键词。这些关键词用于声明变量、声明函数、对特定的数据类型进行一定的限制。

用于声明函数的关键词（双下划线起头）



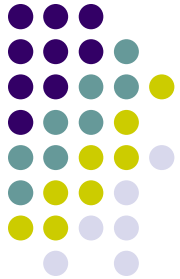
- __asm，内嵌汇编
- __inline，内联展开
- __irq，声明IRQ或FIQ的ISR
- __pure，函数不修改该函数之外的数据
- __softfp，使用软件的浮点连接件
- __swi，软中断函数
- __swi_indirect，软中断函数

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。

5.8 典型ARM汇编语言程序举例



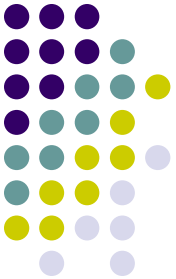
- 请参看本课程教材《嵌入式系统基础教程》中第151页开始的第5.2节。

5.8.1 条件执行举例

- 求a和b两整数最大公约数的C程序

```
While a!=b)
{
    If(a>b) a-=b; else b-=a;
}
```

条件执行举例（续）



- 如果用ARM汇编子程序来实现，就是求r1和r2两个寄存器中的两个整数的最大公约数。使用条件执行指令表示只有以下4句代码:

```
gcd
cmp     r1, r2  ;cmp与subs功能类似但不存结果
subgt   r1, r1, r2      ;如果r1>r2执行此指令
sublt   r2, r2, r1      ;如果r1<r2执行此指令
bne     gcd             ;如果r1<>r2则转gcd标号
```

注： 函数结束时r1=r2，都可以用作返回值。

5.8.2 32位地址送入一个寄存器中

- 下面指令段中的load指令根据输入参数决定调用那个函数。具体做法是将函数的绝对地址通过LDR指令存入在r4寄存器中，由于是32的绝对地址，LDR会被解释成以下操作：将函数的绝对地址放入一个文字池（Literal pool，嵌入在代码中的用以存放常数的区域）。产生一条形如：LDR rn [pc, #offset to literal pool]的指令来将这个绝对地址读入到指定的寄存器中。
- 类似地LDR指令也通过上述方法读入一个32位的绝对数。在下例中，LDR指令将32位绝对数0x11552634读入到r0寄存器中，用作调用routine1或者routine2的参数。

32位地址送入一个寄存器中（续）

```
;void load(int i)
;void routine1(int 1);
;void routine2(int 2);
        AREA LOAD, CODE, READONLY
        IMPORT routine1
        IMPORT routine2
        EXPORT load
load
        stmfd   r13!, {r4, r14}
        ldr     r4, = routine1      ;首先将32位地址存放在附近的区域
        cmps    r0, #1
        ldrne   r4, = routine2
        ldr     r0 = 0x11552634 ;函数的第1个int参数
        bx      r4
        ldmfd   r13!, {r4,  r14}
        bx      r14
```

5.8.3 从IRQ和FIQ异常处理程序返回

- 从IRQ和FIQ异常处理程序返回时，返回地址应该是LR-4。
- 有三种不同的编程方法，分别列出如下:
- 返回方式1

```
INT_HANDLER
        <异常处理代码>
        ……
        SUBS PC, LR, #4       ; PC=R14-4
```

从IRQ和FIQ异常处理程序返回（2）

- 返回方式2

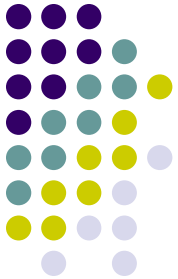
```
INT_HANDLER
        SUB  R14, R14, #4     ; R14 -=4
        ……
        <异常处理代码>
        ……
        MOVS PC, LR
```

从IRQ和FIQ异常处理程序返回（3）

- 返回方式3

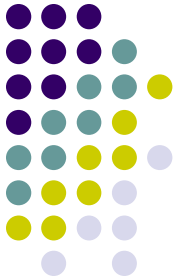
```
INT_HANDLER
        SUB  R14, R14, #4     ; R14 = R14 – 4
        STMFD R13!, {R0-R3, R14}
              ……
        <异常处理代码>
              ……
        LDMFD R13!, {R0-R3, R15}
```

5.8.4 调用ARM汇编语言子程序



- 在ARM汇编语言中，子程序调用是通过BL指令完成的。BL指令的语法格式如下:
  - BL subname
  - 其中，subname是调用的子程序的名称。
- BL指令完成两个操作：将子程序的返回地址放在LR寄存器中，同时将PC寄存器值设置成目标子程序的第一条指令地址。
- 在子程序返回时可以通过将LR寄存器的值传送到PC寄存器中来实现。
- 子程序调用时通常使用寄存器R0~R3来传递参数和返回结果。

调用汇编子程序举例



- 子程序DOADD完成加法运算，操作数放在R0和R1寄存器中，结果放在R0中。

```
                AREA EXAMPLE2, CODE, READONLY
                ENTRY
start           MOV  r0, #10          ;R0设置输入参数
                MOV  r1, #3           ;R1设置输入参数
                BL     doadd          ;调用子程序doadd
doadd           ADD  r0,  r0,  r1     ;子程序实体
                MOV   pc，lr          ;从子程序中返回
                END
```

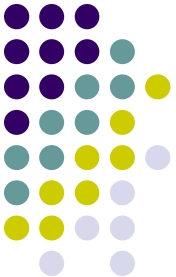
5.8.5 循环结构

- 在ARM汇编中，没有专门的指令用来实现循环，一般通过跳转指令加条件码的形式来实现。可以采用比较指令CMP或者减法指令SUB等实现。参看下面的指令段:

```
LOOP
        ADD    R4, R4, R0
        ADD    R0, R0, #1
        CMP    R0, R1
        BLE    LOOP   ; R0小于等于R1场合跳转
```

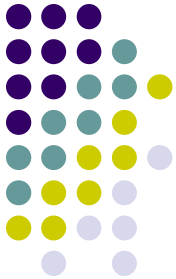
- 在做完了两次加法操作后，比较R0，R1的值，影响条件标志。最后的条件跳转语句根据CMP指令执行的结果来决定是否进行循环。

5.8.6 数据块复制示范程序



- 本程序将数据从源数据区复制到目标数据区
- 复制时，以8个字为单位进行。
- 对于最后所剩不足8个字的数据，以字为单位进行复制，这时程序跳转到copywords处执行。
- 在进行以8个字为单位的数据复制时，保存了所用的8个工作寄存器。程序清单如下面所示。

数据块复制示范程序（1）



```
; 设置本段程序的名称(Block)及属性
AREA Block, CODE, READONLY
; 设置将要复制的字数
num  EQU  20
; 标识程序入口点
    ENTRY

Start
; r0寄存器指向源数据区src
     LDR    r0，    =src
; r1寄存器指向目标数据区dst
     LDR    r1，    =dst
```

数据块复制示范程序（2）

```
; r2指定将要复制的字数
    MOV   r2，  #num
; 设置数据栈指针(r13)，用于保存工作寄存器数值
    MOV   sp，  #0x400
; 进行以8个字为单位的数据复制
blockcopy
; 需要进行的以8个字为单位的复制次数
    MOVS   r3，r2，  LSR  #3
; 对于剩下不足8个字的数据，跳转到copywords，以字为单位复制
    BEQ   copywords
; 保存工作寄存器，压栈
    STMFD  sp!，  {r4~r11}
```

数据块复制示范程序（3）

```
octcopy
; 从源数据区读取8个字的数据，放到8个寄存器中，并更新目标数据区指针r0
    LDMIA  r0!，  {r4-r11}
; 将这8个字数据写入到目标数据区中，并更新目标数据区指针r1
    STMIA  r1!，  {r4-r11}
; 将块复制次数减1
    SUBS   r3，  r3，  #1
; 循环，直到完成以8个字为单位的块复制
    BNE    octcopy
; 恢复工作寄存器值，出栈
    LDMFD  sp!，  {r4-r11}
```

数据块复制示范程序（4）

```
copywords
; 剩下不足8个字的数据的字数
    ANDS   r2，  r2，  #7
; 数据复制完成
    BEQ    stop
wordcopy
; 从源数据区读取1个字的数据，放到r3寄存器中，并
    更新目标数据区指针r0，后索引偏移
    LDR    r3，  [r0]，  #4
; 将这r3中数据写入到目标数据区中，并更新目标数据
    区指针r1，后索引偏移
    STR    r3，  [r1]，  #4
```

数据块复制示范程序（5）

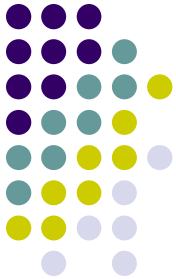
```
; 将字数减1
    SUBS   r2，  r2，  #l
; 循环，直到完成以字为单位的数据复制
    BNE    wordcopy
; 定义数据区BlockData
AREA BlockData，  DATA，  READWRITE
; 定义源数据区src及目标数据区dst
src   DCD   1，2，3，4，5，6，7，8，1，2，3，4，5
    ，6，7，8，1，2，3，4
dst   DCD   0，0，0，0，0，0，0，0，0，0，0，0，0
    ，0，0，0，0，0，0，0
; 结束汇编
    END
```

5.8.7 内嵌汇编

- 内嵌汇编（inline assembly）的语法如下：

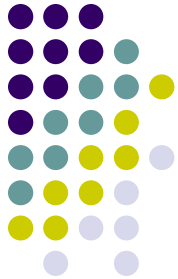
```
__asm
{
   指令[;指令]  /* 注释 */
   …
   [指令]
}
```

内嵌汇编的指令用法



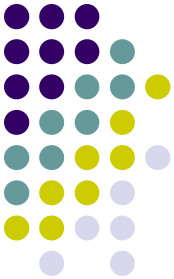
- 内嵌在C或者C++程序中的ARM汇编指令与普通（ADS）格式的ARM汇编指令有所不同。其主要原因在于C/C++编译器在编译C/C++源代码的同时要兼顾处理内嵌汇编程序，因此CPU的内部寄存器资源使用有额外约束。以下讲解内嵌ARM汇编指令的用法。

ARM内嵌汇编程序的操作数



- 内嵌汇编指令中作为操作数的寄存器和常量可以是表达式。这些表达式可以是char，short或int类型，而且这些表达式都是作为无符号数进行操作。若需要带符号，用户需要自己处理与符号有关的操作。编译器将会计算这些表达式的值，并为其分配寄存器。

ARM内嵌汇编程序的物理寄存器



- 内嵌汇编程序中使用物理寄存器有以下限制。

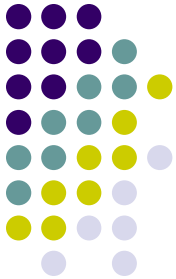
  不能直接向PC寄存器赋值，程序跳转只能使用B或BL指令实现

  不要使用过于复杂的C表达式，因为将会需要较多的物理寄存器，这将导致与其他指令中用到的物理寄存器产生使用冲突。

  编译器可能会使用R12或R13存放编译的中间结果，在计算表达式的值时可能会将寄存器R0～R3，R12和R14用于子程序调用。因此在内嵌的汇编指令中不要将这些寄存器同时指定为指令中的物理寄存器。

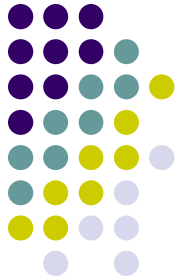
  通常内嵌的汇编指令中不要指定物理寄存器，因为这可能会影响编译器分配寄存器，进而影响代码的效率。

其他内嵌汇编程序的编写注意点



- **常量：**在内嵌汇编指令中，常量前面的“#”可以省略。
- **指令展开：**内嵌汇编指令中，如果包含常量操作数，该指令可能被内嵌汇编器展开成几条指令。
- **标号：**C程序中的标号可以被内嵌的汇编指令使用，但是只有指令B可以使用C程序中的标号，而指令BL则不能使用。
- **内存单元的分配：**所有的内存分配均由C编译器完成，分配的内存单元通过变量供内嵌汇编器使用。内嵌汇编器不支持内嵌程序中用于内存分配的伪指令。

内嵌汇编程序中的SWI和BL指令



- SWI和BL指令：在两个指令使用到内嵌汇编中，除了正常的操作数域外，还必须增加以下3个可选的寄存器列表:
  - ◇用于输入参数的寄存器列表。
  - ◇用于存储返回结果的寄存器列表。
  - ◇用于表示那些寄存器将有可能会被修改的寄存器列表。

内嵌汇编代码举例字符串复制（1）

```
#include<stdio.h>
void str_cpy(const char *src,char *dst)
{
    int ch;
    __asm
    {
    loop: //普通ARM汇编代码中的标号后面不能跟冒号。C程序中
          //的标号可以被内嵌的汇编指令使用。 ARM内嵌汇编代码中
          //只有B指令可以使用C的标号，而BL指令不能够使用C代码
          //的标号。 C程序的标号后面跟冒号，由Goto语句转向标号处。
          LDRB    ch, [src], #1
          STRB    ch, [dst], #1
          CMP              ch,        #0
          BNE      loop
    }
}
```

内嵌汇编代码举例字符串复制（2）

```
int main(void)
{
    const char * a="Hello world!\n";
    char b[20];
    //  do inline assembly routine str_cpy(a,b)
    __asm
    {
    MOV R0,a  // 将串a的串首地址送到R0寄存器
    MOV R1,b  // 将串b的串首地址送到R1寄存器
    BL  str_cpy, {R0, R1}   // 调用C函数str_cpy()
    }
    printf("Original string: %s\n",a);
    printf("Copied   string: %s\n",b);  //  半主机方式显示复制前后的两个串
    return(0);
}
```

5.8.8 ARM汇编、C和C++混合编程

- 在C/C++程序中如果必须使用汇编指令来完成某些操作，可以采用两种方法：1.采用内嵌汇编，即在C/C++源程序中嵌入一块汇编代码，让这块汇编代码来完成特定的操作；2.将必须使用汇编代码的部分独立编写成在一个文件中，形成一个子程序，C/C++程序可以调用这些汇编程序来完成特定的操作。

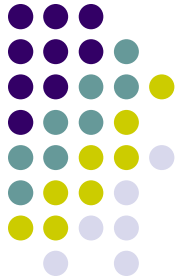
C/C++程序与ARM汇编语言程序的相互调用

- C/C++程序与汇编程序相互调用时，应遵守相应的ATPCS，主要有五种调用。
  - ARM汇编子程序调用C语言子程序
  - ARM汇编子程序调用C++语言子程序
  - C语言程序调用ARM汇编语言子程序
  - C++语言程序调用ARM汇编语言子程序
  - C语言程序调用C++语言子程序
- 下面就每种具体情况逐一举例说明。

C/C++程序调用ARM汇编子程序要点

- 设计汇编程序必须遵守ATPCS，保证程序调用时参数的正确传递。
- 在汇编程序中使用EXPORT指示符声明本程序可以被别的程序调用。
- 在C语言程序中使用extern关键词声明该汇编程序可以被调用，C++语言程序使用extern “C”来声明该汇编程序可以被调用。

例1 C程序调用ARM汇编子程序



```
/* main_0522.c semihosting output mode */
#include <stdio.h>
extern int asmfile(int arg1, int arg2, int arg3);

int main(void)
{
int a1=1,a2=2,a3=4;
printf("Example of C Program calling Assembly program!\n");
printf("(%d + %d + %d) * 600 = %d\n",a1,a2,a3,asmfile( a1, a2,
    a3));
}
```

C程序调用ARM汇编子程序（续）

```
; ASM_0522.s
        EXPORT asmfile
        AREA My_pro, CODE, READONLY
asmfile
            STMFD SP!, {R4-R6,R8,R7}
            add       r0, r0, r1
            add       r0, r0, r2
            mov       r4, #600
            mul       r3, r0, r4
            mov       r0, r3
            LDMFD SP!, {R4-R6,R8,R7}
            mov  pc, lr
            END
```

例2 ARM汇编程序调用C语言子程序

- 本案例程序比较两个IP地址的大小，a1~a4存放IP地址1的值（按照ATPCS传递参数），b1~b4存放IP地址2的值（通过栈传递参数），如果IP地址1的值大于IP地址2的值则返回1，如果IP地址1的值小于IP地址2的值则返回-1，如果两者相等则返回零。
- IP地址1取值： 192, 168, 1, 152,
- IP地址2取值： 172, 0, 0, 151

例2 ARM汇编程序调用C子程序（续）

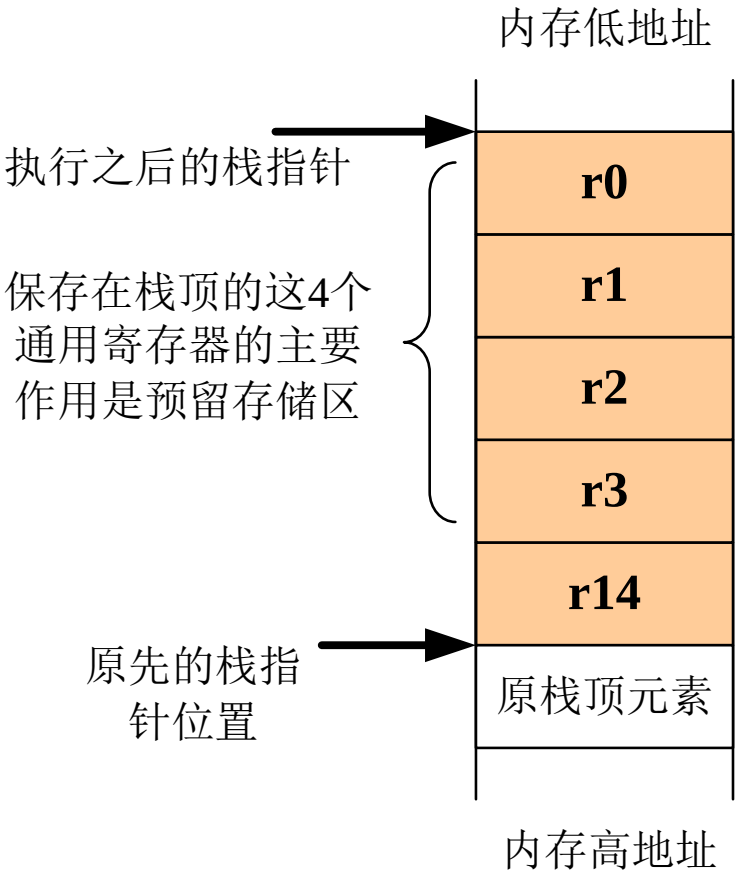
```
/* C代码部分 */
#include <stdio.h>
extern int function(void);   /* 声明function是外部函数 */
int compare_ip(int a1, int a2, int a3, int a4, int b1, int b2, int b3, int b4){
    if(a1!=b1)
        return a1>b1?1:-1;
    if(a2!=b2)
        return a2>b2?1:-1;
    if(a3!=b3)
        return a3>b3?1:-1;
    if(a4!=b4)
        return a4>b4?1:-1;
    return 0;  }
int main(){
    printf("This is a example of semihosting\n");
    printf("result is %d\n",function());  }
```

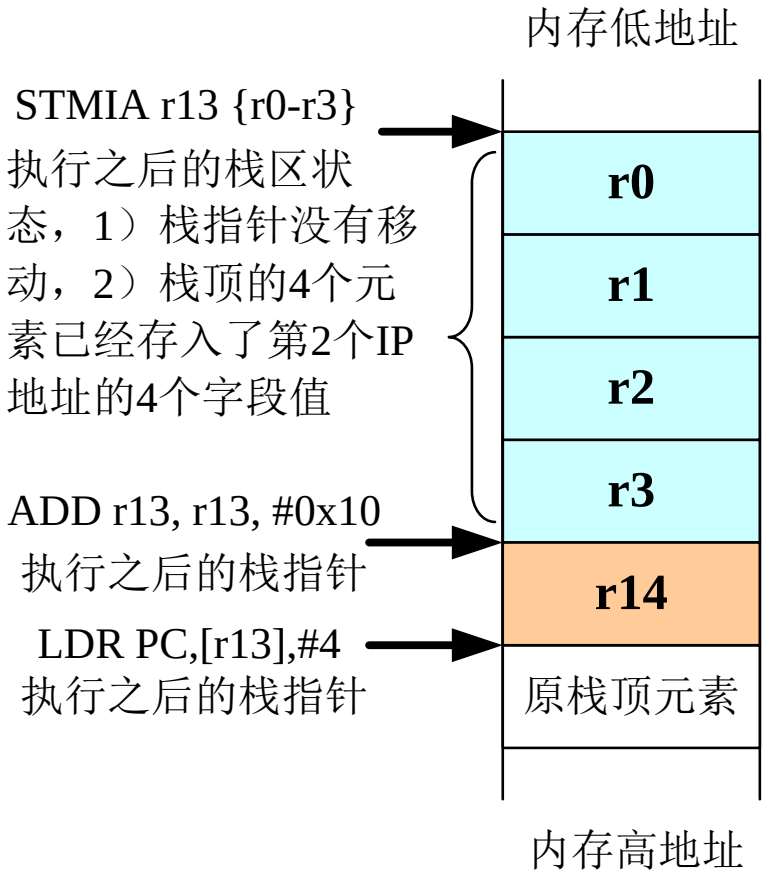
例2 ARM汇编调用C子程序（续2）

```
    AREA FUNCTION, CODE, READONLY   ;ARM汇编子程序
    IMPORT compare_ip
    EXPORT function
function
    STMFD       r13!,{r0-r3,r14}   ;保存寄存器到栈区
    MOV         r3,#0x97           ;存入IP地址2的4个数, 0x97=151
    MOV         r2,#0              ; 存入0
    MOV         r1,#0              ; 存入0
    MOV         r0,#0xac           ; 存入0xac=172
    STMIA       r13, {r0-r3}       ; R0-R4覆盖存入栈区的R0-R4位置
    MOV         r3,#0x98           ; 存入IP地址1的4个数, 0x98=152
    MOV         r2,#1              ; 存入1
    MOV         r1,#0xa8           ; 存入0xa8=168
    MOV         r0,#0xc0           ; 存入0xc0=192
    BL          compare_ip  ; 0x0          ; 调用C语言函数进行IP值比较
    ADD         r13,r13,#0x10      ; 栈指针上移4个字（元素）
    LDR         pc,[r13],#4        ; 将保存的r14值加载到PC，而后r13加4
    END                            ; 本段汇编代码的操作图解参看下一页
```

例2 ARM汇编调用C子程序（续3）

- ARM汇编语言子程序Function的栈区操作图解





例3 ARM汇编程序调用 C++子程序

```
// 这是C++调用ARM汇编的范例程序
// 文件名： try_C++_call_S
// main函数位于这个.CPP文件
extern "C" void asmfunc(void);

int main(void)
{
   asmfunc();
}
```

例3 ARM汇编调用 C++子程序（续）

```
; 这个ARM汇编语言程序被main函数调用
; 该函数又调用ARM汇编函数cppfunc
;
        AREA mypro, CODE, READONLY
        IMPORT cppfunc
        EXPORT asmfunc
asmfunc
        STMFD sp!,{lr}
        BL cppfunc
        LDMFD sp!,{pc}
        END
```

例3 ARM汇编调用 C++子程序（续2）

```
#include <iostream>
#include <cmath>
using namespace std;
const double PI = 3.1415926;
class point
{
public:
    point(){x=0; y=0;};
    point(double a, double b) {x=a; y=b;};
    point(point &a) {x=a.x; y=a.y;};
    ~point(){};
```

例3 ARM汇编调用 C++子程序（续3）

```
double GetX() {return x;};
double GetY() {return y;};
point &Move(const point &a) {this->x+=a.x;
this->y+=a.y; return *this;};
double GetArea(point &a) {return abs((this->x-a.x))*abs((a.y-
    this->y));};
point &Rotate(const int degree)
{
    double tempX=this->x, tempY=this->y, length, a;
    a = atan(tempY/tempX);
    length = sqrt(tempX*tempX+tempY*tempY);
```

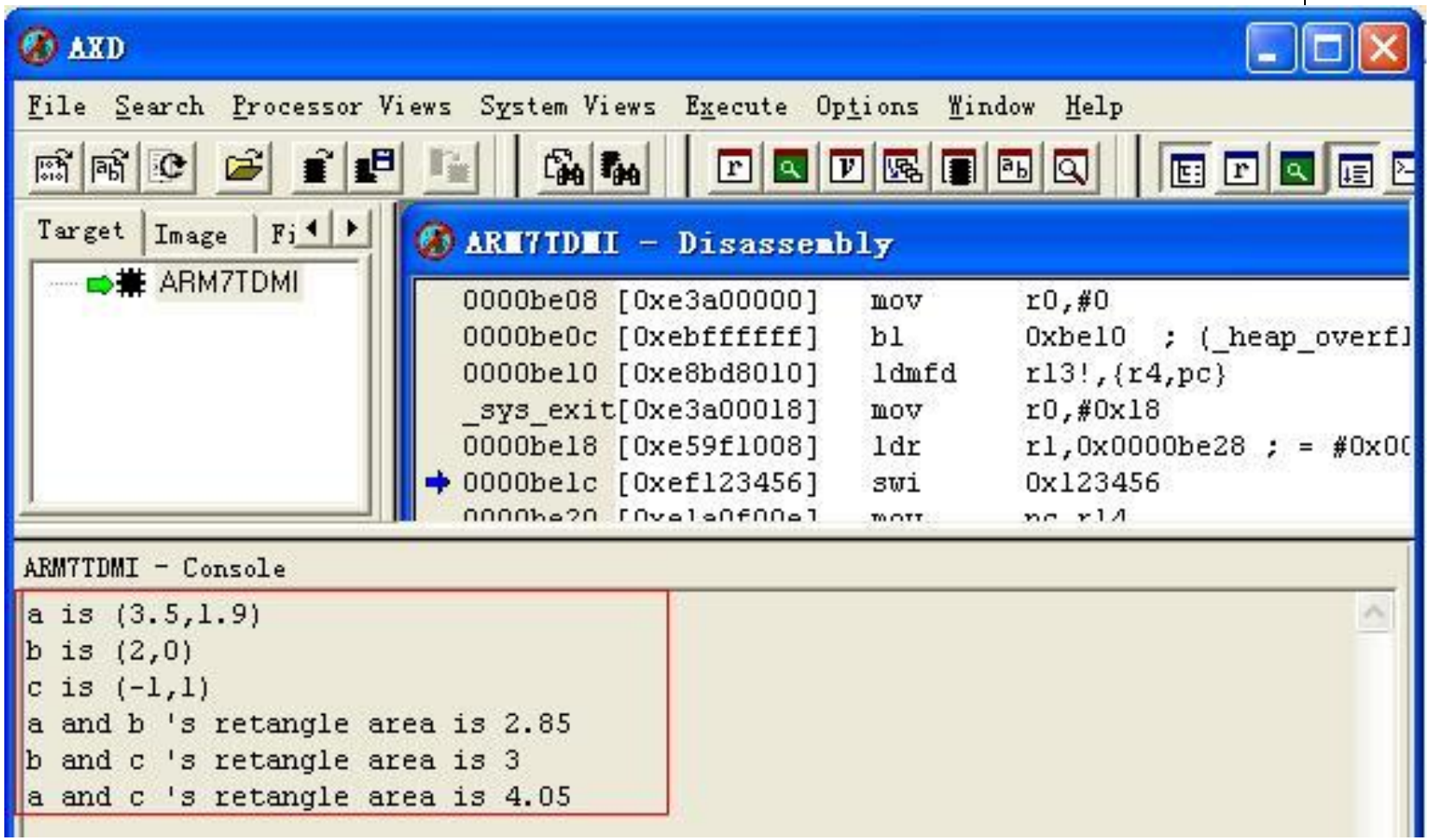
例3 ARM汇编调用 C++子程序（续4）

```
    this->x = length * cos(a+degree*PI/180);
    this->y = length * sin(a+degree*PI/180);
    return *this;
    };
private:
    double x, y;
};
```

例3 ARM汇编调用 C++子程序（续5）

```
extern "C" int cppfunc()
{
    point a(1.5, 1.9);
    point b(2.0, 0.0);
    point c(1.0, 1.0);
    a.Move(b);
    c.Rotate(90);
    cout<<"a is ("<<a.GetX()<<","<<a.GetY()<<")"<<endl;
    cout<<"b is ("<<b.GetX()<<","<<b.GetY()<<")"<<endl;
    cout<<"c is ("<<c.GetX()<<","<<c.GetY()<<")"<<endl;
    cout<<"a and b 's retangle area is "<<a.GetArea(b)<<endl;
    cout<<"b and c 's retangle area is "<<b.GetArea(c)<<endl;
    cout<<"a and c 's retangle area is "<<a.GetArea(c)<<endl;
}
```

例3 ARM汇编程序调用C++子程序的半主机方式运行结果输出截图



例4 C语言程序调用 C++子程序

```
/* main.c 源代码清单 */
#include <stdio.h>
struct S {
    char ca[12];
};
extern void cppfunc(struct S *p);
/* Declaration of the C++ function to be called from C */
void f(struct S * s) {    /*  f是C语言函数  */
        char *tmp = "Hello";
        char * source = s->ca;
        while(*source++ = *tmp++); //copy "hello" to s->ca
        cppfunc(s);  /* 调用C++函数cppfunc()，初始化 's'  */
}
```

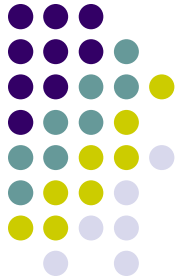
例4 C语言程序调用 C++子程序（续1）

```
int main(){
        struct S s;
        f(&s);   /* 调用C语言的函数f ( )  */
        printf("%s\n",s.ca);
        return 0;
}
```

例4 C语言程序调用 C++子程序（续2）

```
// CPP.CPP函数清单
class S {       // has no base classes or virtual functions
public:         char ca[12];
                S()  {ca[0]='\0'; }
};
extern "C" void cppfunc(S *p) {
// Definition of the C++ function to be called from C.
// The function is written in C++, only the linkage is C
        char * tmp="world!";
        char * source=p->ca;
        while(*source !='\0')   //扫描过原先在c程序中的赋值
                source++;
        *source++ =' ';                         //将结束符改为空格
        while(*source++ = *tmp++); //copy "world" to p->ca   }
```

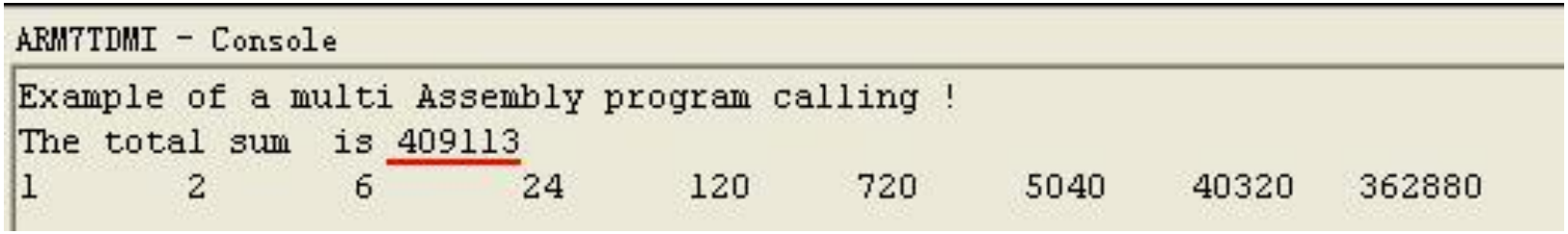
ARM汇编（子）程序的相互调用



- 基本要点:
  - 如果一个ARM汇编语言程序文件含有调用外部汇编语言程序文件中子程序（函数）的指令，则需要用IMPORT指示符来指明将要调用的子程序名称。
  - 如果本汇编语言程序文件中的某个子程序（函数），需要被外部的ARM汇编语言程序文件中的语句调用，则需要用EXPORT指示符来指明将要被调用的子程序（函数）名称。
  - 被执行的汇编子程序在运行前，要注意将寄存器组压入栈区，返回时要注意将栈区保存的工作现场恢复到处理器的寄存器组。

例5 ARM汇编子程序嵌套调用举例-1

- 这里给出的ARM汇编程序嵌套调用范例程序做如下计算:
  - **求自然数1到n的阶乘的总和，半主机方式输出**
  - 运算结果如下图所示。第3行显示的是1~9的阶乘，第2行显示的是1!+2!+3!+4!+5!+6!+7!+8!+9!之总和。



例5 ARM汇编子程序嵌套调用举例-2

```
/* This program is semihosting output mode */
/* First the main function call assembly summing subprogram */
/* Then the summing subprogram call assembly factorial subprogram */
#include <stdio.h>
extern int asmFac(int n);

struct factorial_sum{
int cal_fn;
int sum_fn;
int fn[9];
};

extern struct factorial_sum * summing(struct factorial_sum * arg1);
```

例5 ARM汇编子程序嵌套调用举例-3

```
int main(void)
{    int j;
     struct factorial_sum fac={ 9, 0, {1,1,1,1,1,1,1,1,1}};   //设置参数
     struct factorial_sum * result;      //申请变量作为返回值

     printf("Example of a multi Assembly program calling !\n");

     result=summing(&fac); //调用求和函数 R0存放的是FAC变量的首地址
     printf("The total sum  is %d\n", result->sum_fn);  //输出结果
     for(j=0; j<9; j++){
                         printf("%d\t",(result->fn)[j]);
                         }
}
```

例5 ARM汇编子程序嵌套调用举例-4

```
;the details of parameters transfer comes from ATPCS
;if there are more than 4 args, stack will be used
        EXPORT summing
        IMPORT asmFac       ;说明用到了其他文件中的子汇编程序
        AREA SUMMING, CODE, READONLY
summing
            STMFD SP!, {R4-R5}
            ldr     r1,  [r0]           ;r1=cal_fn
            mov     r2, #1              ;将r2设置为当前需要计算的阶乘数,
                                        ;它从1变化到cal_fn
            add     r3, r0, #8          ;将r3指向fn数组
            mov     r5, #0              ;r5为总和,初始值置为0
loop
            cmp     r1, r2              ;将cal_fn与当前所需计算的阶乘值比较
            blt     back                ;如果小于,则返回
```

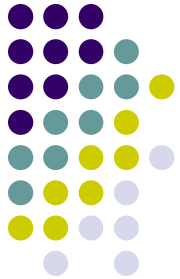
例5 ARM汇编子程序嵌套调用举例-5

```
        STMFD SP!, {R0-R3,lr}   ;保存ro~r3,lr
                                ;因为调用了外部文件的汇编子程序
        mov     r0, r2          ;将r0设置为当前所需计算的阶乘值
        bl      asmFac          ;调用阶乘函数
        mov     r4, r0          ;将返回值(阶乘)存在r4中
        ldmfd   SP!, {R0-R3,lr} ;恢复先前保存的寄存器值
        str     r4, [r3]        ;将计算所得的阶乘值存入数组中
        add     r3, r3, #4      ;将r3指向数组的下一个
        add     r5, r5, r4      ;将阶乘值加入总和
        add     r2, r2, #1      ;计算下一个阶乘
        b loop                  ;循环
back
        str     r5, [r0, #4]    ;将计算所得的总和存入结构体中
        ldnfd   SP!, {R4-R5}    ;恢复寄存器值
        mov     pc, lr          ; summing子程序返回
        END                     ;汇编代码结束
```

例5 ARM汇编子程序嵌套调用举例-6

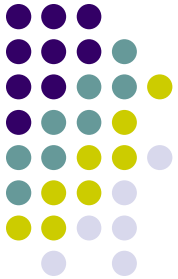
```
;the details of parameters transfer comes from ATPCS
;if there are more than 4 args, stack will be used
;这个ARM汇编函数也可以被C函数调用，符合ATPCS规范
;int asmFac(int n)
        EXPORT asmFac
        AREA ASMFILE, CODE, READONLY

asmFac
            mov             r1, r0      ;r1=n
loop
            subs    r1,     r1, #1      ;将r1减1
            mulgt   r0,     r1, r0      ;如果大于0,则乘上r1(相当与n*(n-1))
            bgt             loop        ;如果大于0,继续
            mov  pc, lr                 ;asmFac子程序返回
            END                         ;汇编代码结束
```

ARM9五级流水线互锁现象举例



- 互锁使得指令的执行周期增加，下面举例说明:
  - LDR r1, [r2,#8]
  - ADD r0, r0, r1
  - MOV r2, #0

5.8.9 5级流水线的互锁问题



- ARM9指令流水线有5级。 五个流水线阶段分别是：取指、译码、ALU、LS1、LS2。
- 流水线的增加增加了系统的吞吐量，但也引起了流水线互锁（pipeline interlock），即一条指令需要前一条指令的执行结果，而这时结果还没有出来，那么处理器就会等待。

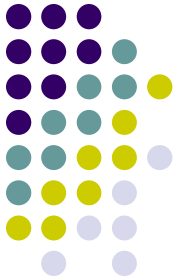
ARM9五级流水线互锁现象的图解

流水线	取值	译码	ALU	LS1	LS2
周期1	LDR				
周期2	ADD	LDR			
周期3	MOV	ADD	LDR	…	
周期4	…	MOV	ADD	LDR	…
周期5		MOV	**ADD**	–	–
周期6			MOV	–	

ARM9五级流水线互锁现象的解决

- 如果将指令顺序该为:
  - LDR　r1, [r2,#8]
  - MOV　r2, #0
  - ADD　r0, r0, r1
- 这样的指令顺序执行由于没有产生流水线互锁，执行上述指令只需要3个周期，即一个时钟周期完成一条指令。在保持结果正确无误的前提下，比修改之前减少了一个时钟周期。

5.8.10 ARM汇编语言程序中的宏定义和宏指令



- 指示符MACRO和MEND
- MACRO指示符标识宏定义的开始，MEND标识宏定义的结束。
- 用MACRO及MEND定义一段代码，称为宏定义体，这样在程序中就可以通过宏指令多次调用该代码段。

MACRO和MEND指示符

- 语法:

  MACRO

  {$label} macroname {$paprameter{,
  $paprameter}……}

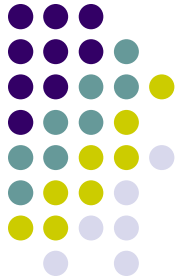
  ;code

  ……..

  ;code

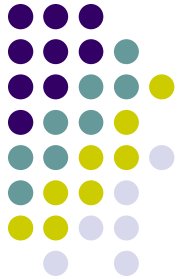
  MEND

MACRO和MEND指示符说明



- 其中:
  - $label在宏指令被展开时，label可以被替换成相应的符号，通常是一个标号。在一个符号前面使用$表示程序被汇编时将使用相应的值来替代$后面的符号。
  - Macroname为所定义的宏的名称
  - $parameter为宏指令的参数。当宏指令被展开时将被替换成相应的值，类似于函数中的形式参数。可以在宏定义时为参数指定相应的默认值。

ARM宏定义和宏指令举例



```
                MACRO
$label  xmac $p1, $p2
; code
$label.loop1    ; code
; code
                BGE $label.loop1
$label.loop2    ; code
                BL      $p1
                BGT $label.loop2
                ; code
                ADR     $p2
                ; code
                MEND
```

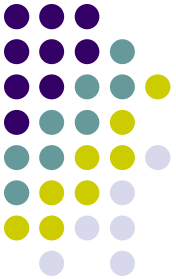
ARM宏定义和宏指令举例（续）

```
;在程序中调用xmac宏
lpp     xmac   subro1, xde
;程序被宏展开后，宏展开的结果如下:
;code
lpploop1        ;code
                ;code
                BGE lpploop1
lpploop2        ;code
                BL subro1
                BGT   lpploop2
                ;code
                ADR   xde
```

5.8.11 GNU格式的ARM汇编语言程序设计

- GNU格式ARM汇编语言程序主要用于基于ARM硬件平台和Linux操作系统的嵌入式开发。
- GNU提供的相关汇编器是arm-elf-as（简称为as），连接器是arm-elf-ld（简称为ld）。GNU格式ARM汇编程序与ADS格式ARM汇编程序有较大的区别。

基于ARM处理器的GNU汇编语句格式如下：

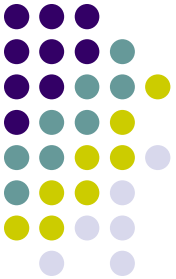


- **[<label>:] [<instruction or dircetive>] @注释**
- 其中的第1个字段**label**是标号，第2个字段**instruction or dircetive**是指令或者GNU的汇编指示符。
- 与ADS格式的汇编语言程序不同，在GNU格式汇编语言程序的源代码行里不需要缩进指令和指示符。
- 限于篇幅，更多的教学内容请大家阅读教材第5.4节。

第11讲重点

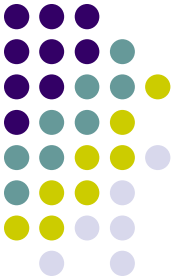
- ATCPS和AAPCS
- ARM编译器保有的特定关键字
- 典型ARM汇编语言程序设计举例
- ARM汇编、C和C++混合编程
- 5级流水线的互锁问题
- ARM汇编语言程序中的宏定义和宏指令
- ARM处理器的GNU汇编语句格式

第11讲复习题与思考题



- 什么是ATPCS和AAPCS?
- 子程序或者过程调用时，如果参数超过4个，如何编程实现正确地参数传递?
- 从IRQ和FIQ处理程序返回时，如何编写返回指令?
- 如何把32位地址送入一个寄存器中?
- ARM内嵌汇编的指令用法有哪些注意点?
- 在ADS环境中如何实现以下的子程序调用?
  - C语言程序调用 C++子程序
  - C/C++程序调用ARM汇编子程序要点
  - ARM汇编子程序的嵌套调用
- 试给出ARM的GNU汇编程序设计要点？它与ARM的ADS汇编程序设计有什么区别?

第5次习题布置（上机实习题）



- 题1，以 ARM汇编子程序为主，实现下列功能。
  - 从开发板的键盘输入5~10个的字符串，串长不超过20个字符。连续两次输入Enter键结束输入操作。
  - 先对输入进来的串进行首尾次序颠倒操作，显示输出。而后再对串进行升序排序操作并输出结果。
- 题2，编写汇编子程序，尽可能地求出5个连续的最大费波那契数列的整数，或者5个连续的最大整数，而后用半主机方式或者在开发板的LCD上逐行显示。说明你的算法和可能的最大值。