《ARM 嵌入式系统结构与编程》习题答案

第一章 绪论

1. 国内嵌入式系统行业对“嵌入式系统”的定义是什么？如何理解？

答：国内嵌入式行业一个普遍认同的定义是：以应用为中心，以计算机技术为基础，软硬件可裁剪，适应应用系统对功能，可靠性，成本，体积，功耗严格要求的专业计算机系统。

从这个定义可以看出嵌入式系统是与应用紧密结合的，它具有很强的专用性，必须结合实际系统需求进行合理的剪裁利用。因此有人把嵌入式系统比作是一个针对特定的应用而“量身定做”的专业计算机系统。

2．嵌入式系统是从何时产生的，简述其发展历程。

答：从 20 世纪 70 年代单片机的出现到目前各式各样的嵌入式微处理器，微控制器的大规模应用，嵌入式系统已经有了 30 多年的发展历史。

嵌入式系统的出现最初是基于单片机的。Intel 公司 1971 年开发出第一片具有 4 位总线结构的微处理器 4004，可以说是嵌入式系统的萌芽阶段。80 年代初的 8051 是单片机历史上值得纪念的一页。20 世纪 80 年代早期，出现了商业级的“实时操作系统内核”，在实时内核下编写应用软件可以使新产品的沿着更快，更节省资金。20 世纪 90 年代实时内核发展为实时多任务操作系统。步入 21 世纪以来，嵌入式系统得到了极大的发展。在硬件上，MCU 的性能得到了极大的提升，特别是 ARM 技术的出现与完善，为嵌入式操作系统提供了功能强大的硬件载体，将嵌入式系统推向了一个崭新的阶段。

3．当前最常用的源码开放的嵌入式操作系统有哪些，请举出两例，并分析其特点。

答：主要有嵌入式 Linux 和嵌入式实时操作内核 uC/OS-II

嵌入式 Linux 操作系统是针对嵌入式微控制器的特点而量身定做的一种 Linux 操作系统，包括常用的嵌入式通信协议和常用驱动，支持多种文件系统。主要有以下特点：源码开放，易于移植，内核小，功能强大，运行稳定，效率高等。

uC/OS 是源码工卡的实时嵌入式系统内核，主要有以下特点：源码公开，可移植性强，可固化，可剪裁，占先式，多任务，可确定性，提供系统服务等。

4．举例说明嵌入式设备在工控设备中的应用。

答：由于工业控制系统特别强调可靠性和实时性，“量身定做”的嵌入式控制系统恰能满足工业控制的需求。例如：工业过程控制，数字控制机床，电网设备监测，电力自动控制系统，石油化工监控等。

5．嵌入式技术的发展趋势有哪些？

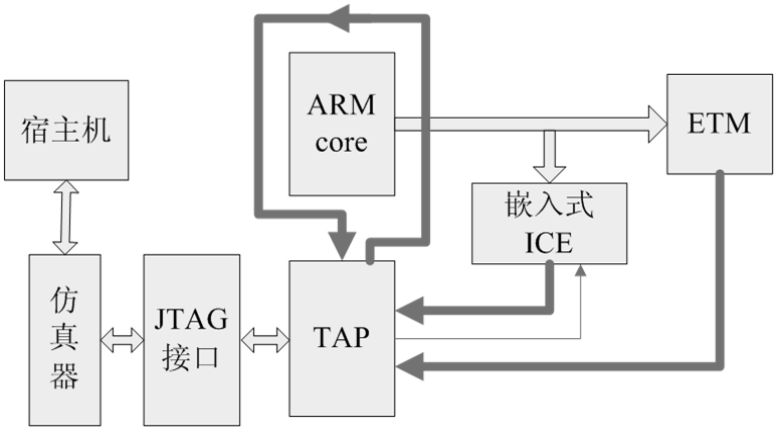
答：未来嵌入式系统的发展趋势有：1.随着信息化与数字化的发展，嵌入式设备进行网络互联是未来发展的趋势。2.优化嵌入式系统软硬件内核，提高系统运行速度，降低功耗和硬件成本。3.指令集的并行计算技术将引入嵌入式微处理器。4.嵌入式微处理器将会向多核技术发展。5.嵌入式技术将引领信息时代。

第 2 章 ARM 技术与 ARM 体系结构

1.简述 ARM 处理器内核调试结构原理

答：ARM 处理器一般都带有嵌入式追踪宏单元 ETM（Embedded Trace Macro），它是 ARM 公司自己推出的调试工具。ARM 处理器都支持基于 JTAG（Joint Test Action Group 联合测

试行动小组）的调试方法。它利用芯片内部的 Embedded ICE 来控制 ARM 内核操作，可完成单步调试和断点调试等操作。当 CPU 处理单步执行完毕或到达断点处时，就可以在宿主机端查看处理器现场数据，但是它不能在 CPU 运行过程中对实时数据进行仿真。



ETM 解决了上述问题，能够在 CPU 运行过程中实时扫描处理器的现场信息，并数据送往 TAP（Test Access Port）控制器。上图中分为三条扫描链（图中的粗实线），分别用来监视 ARM 核，ETM，嵌入式 ICE 的状态。

2. 分析 ARM7TDMI-S 各字母所代表的含义。

答：ARM7 T D M I – S 中

ARM 是 Advanced RISC Machines 的缩写

7 是系列号；

T：支持高密度 16 位的 Thumb 指令集；

D：支持 JTAG 片上调试；

M：支持用于长乘法操作（64 位结果）ARM 指令，包含快速乘法器；；

I：带有嵌入式追踪宏单元 ETM，用来设置断点和观察点的调试硬件；

S：可综合版本，意味着处理器内核是以源代码形式提供的。这种源代码形式又可以编译成一种易于 EDA 工具使用的形式。

3. ARM 处理器的工作模式有哪几种，其中哪些为特权模式，哪些为异常模式，并指出处理器在什么情况下进入相应的模式。

答：ARM 技术的设计者将 ARM 处理器在应用中可能产生的状态进行了分类，并针对同一类型的异常状态设定了一个固定的入口点，当异常产生时，程序会自动跳转到对应异常入口处进行异常服务。

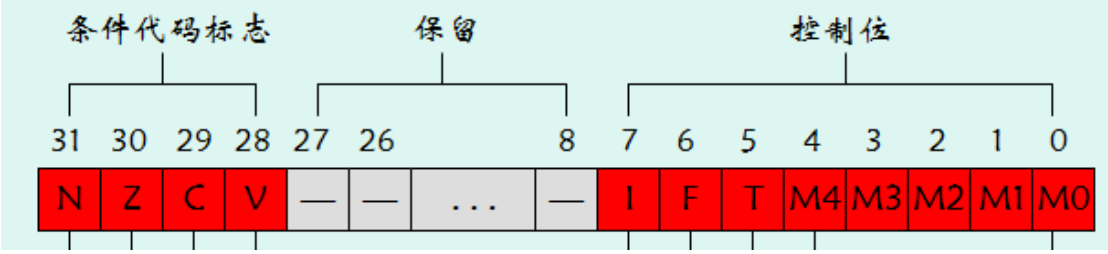
- 1．用户模式：非特权模式，也就是正常程序执行的模式，大部分任务在这种模式下执行。在用户模式下，如果没异常发生，不允许应用程序自行改变处理器的工作模式，如果有异常发生，处理器会自动切换工作模式
- 2．FIQ 模式：也称为快速中断模式，支持高速数据传输和通道处理，当一个高优先级(fast)中断产生时将会进入这种模式。
- 3．IRQ 模式：也称为普通中断模式，:当一个低优先级(normal)中断产生时将会进入

这种模式。在这模式下按中断的处理器方式又分为向量中断和非向量中断两种。通常的中断处理都在 IRQ 模式下进行。

- 4．SVC 模式：称之为管理模式，它是一种操作系统保护模式。当复位或软中断指令执行时处理器将进入这种模式。
- 5．中止模式：当存取异常时将会进入这种模式，用来处理存储器故障、实现虚拟存储或存储保护。
- 6．未定义指令异常模式：当执行未定义指令时会进入这种模式，主要是用来处理未定义的指令陷阱，支持硬件协处理器的软件仿真，因为未定义指令多发生在对协处理器的操作上。
- 7．系统模式：使用和 User 模式相同寄存器组的特权模式，用来运行特权级的操作系统任务。
- 在这 7 种工作模式中，除了用户模式以外，其他 6 种处理器模式可以称为特权模式，在这些模式下，程序可以访问所有的系统资源，也可以任意地进行处理器模式的切换。
- 在这 6 种特权模式中，除了系统模式外的其他 5 种特权模式又称为异常模式，每种异常都对应有自己的异常处理入口点。

4. 分析程序状态寄存器（PSR）各位的功能描述，并说明 C,Z,N,V 在什么情况下进行置位和清零。

答：



- 条件位：
  - N = 1-结果为负，　0-结果为正或 0
  - Z = 1-结果为 0，　0-结果不为 0
  - C =1-进位，　　0-借位
  - V =1-结果溢出，　0 结果没溢出
- Q 位：
  - 仅 ARM 5TE/J 架构支持
  - 指示增强型 DSP 指令是否溢出
- 中断禁止位：
  - I　= 1: 禁止　IRQ.
  - F = 1: 禁止　FIQ.
- T Bit
  - 仅 ARM　xT 架构支持
  - T = 0: 处理器处于 ARM 状态
  - T = 1: 处理器处于 Thumb 状态
- Mode 位(处理器模式位):

- 0b10000 User
- 0b10001 FIQ
- 0b10010 IRQ
- 0b10011 Supervisor
- 0b10111 Abort
- 0b11011 Undefined
- 0b11111 System

5. 简述 ARM 处理器异常处理和程序返回的过程。

答：只要正常的程序流被暂时中止，处理器就进入异常模式。例如响应一个来自外设的中断。在处理异常之前，ARM7TDMI 内核保存当前的处理器状态，这样当处理程序结束时可以恢复执行原来的程序。如果同时发生两个或更多异常，那么将按照固定的顺序来处理异常。

异常或入口	返回指令			向量表偏移
		处理器模式	优先级	
BL	*MOV PC,R14*			
SWI	MOVS PC,R14_svc	SVC	6	0x00000008
未定义的指令	MOVS PC,R14_und	UND	6	0x00000004
预取指中止	SUBS PC,R14_abt,#4	ABT	5	0x0000000C
快中断	SUBS PC,R14_fiq,#4	FIQ	3	0x0000001C
中断	SUBS PC,R14_irq,#4	IRQ	4	0x00000018
数据中止	SUBS PC,R14_abt,#8	ABT	2	0x00000010
复位	无	SVC	1	0x00000000

在异常发生后，ARM7TDMI 内核会作以下工作：

1.在适当的 LR 中保存下一条指令的地址

2.将 CPSR 复制到适当的 SPSR 中；

3. 将 CPSR 模式位强制设置为与异常类型相对应的值；

4.强制 PC 从相关的异常向量处取指。

ARM7TDMI 内核在中断异常时置位中断禁止标志，这样可以防止不受控制的异常嵌套。

注：异常总是在 ARM 状态中进行处理。当处理器处于 Thumb 状态时发生了异常，在异常向量地址装入 PC 时，会自动切换到 ARM 状态。

当异常结束时，异常处理程序必须：

1.将 LR 中的值减去偏移量后存入 PC，偏移量根据异常的类型而有所不同；

2.将 SPSR 的值复制回 CPSR；

3.清零在入口置位的中断禁止标志。

注：恢复 CPSR 的动作会将 T、F 和 I 位自动恢复为异常发生前的值。

6. ARM 处理器字数据的存储格式有哪两种？并指出这两种格式的区别。

答：ARM7TDMI 处理器可以将存储器中的字以下列格式存储

- 大端格式（Big-endian）
- 小端格式（Little-endian）
- 小端存储器系统：

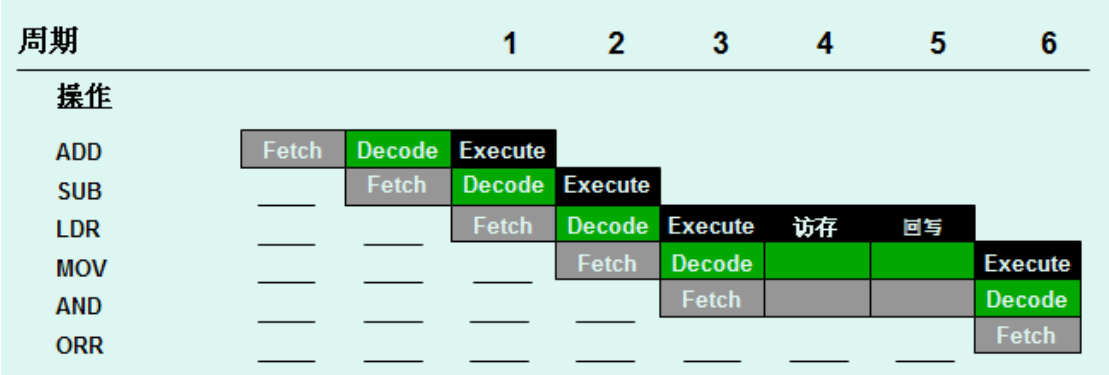
在小端格式中，高位数字存放在高位字节中。因此存储器系统字节 0 连接到数据线 7～0。

- 大端存储器系统：

在大端格式中，高位数字存放在低位字节中。因此存储器系统字节 0 连接到数据线 31～24。

7. 分析带有存储器访问指令（LDR）的流水线运行情况，并用图示说明流水线的运行机制。

答：存储器访问指令 LDR 流水线举例



取指的存储器访问和执行的数据路径占用都是不可同时共享的资源，对于多周期指令来说，如果指令复杂以至于不能在单个时钟周期内完成执行阶段，就会产生流水线阻塞。

对存储器的访问指令 LDR 是非单周期指令

LDR 指令的执行，访问存储器，回写寄存器（占用了 3 个周期）。造成了 MOV 指令的执行被阻断。

8. 简述 ARM9 的 5 级流水线每一级所完成的功能和实现的操作。

答：ARM920 在指令操作上采用 5 级流水线 .

取指：从指令 Cache 中读取指令。

译码：对指令进行译码，识别出是对哪个寄存器进行操作并从通用寄存器中读取操作数。

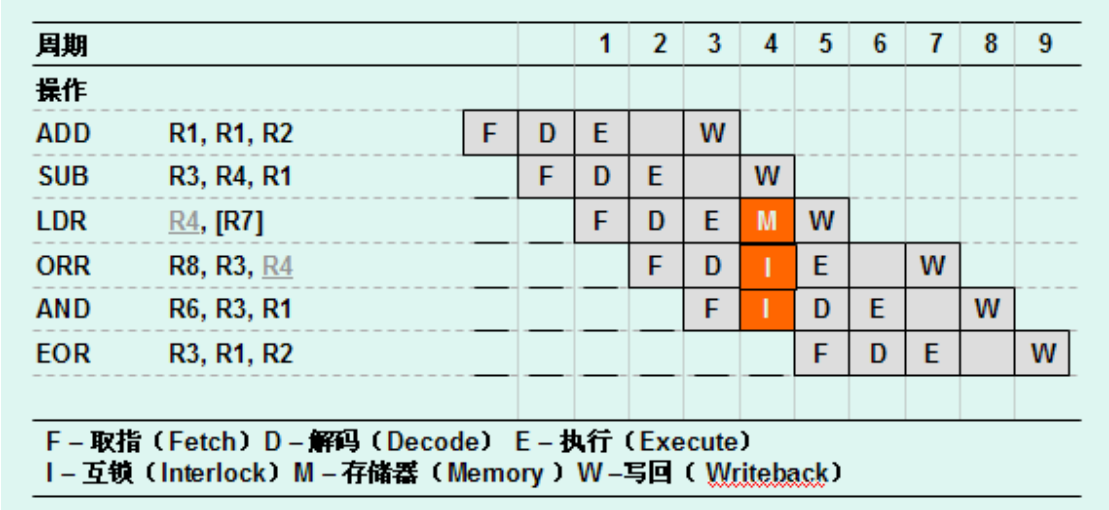
执行：进行 ALU 运算和移位操作，如果是对存储器操作的指令，则在 ALU 中计算出要访问的存储器地址。

存储器访问：如果是对存储器访问的指令，用来实现数据缓冲功能（通过数据 Cache）。

寄存器回写：将指令运算或操作结果写回到目标寄存器中。

9. 什么叫流水线互锁？应如何来解决，举例说明。

答：互锁：当前指令的执行需要前面指令的执行结果，但前面的指令没有执行完毕，引起流水线的等待。互锁发生时，硬件会停止指令的执行，直到数据准备好。



上边程序中 ORR 指令执行时需要使用 LDR 指令加载后的 R4 寄存器，因此造成了 ORR 指令的等待。

编译器以及汇编程序员可以通过重新设计代码的顺序或者其他办法来减少互锁的数量。

第 3 章 ARM 指令集寻址方式

1.在指令编码中，条件码占几位，最多有多少个条件，各个条件是如何形成的？

答：条件码占 4 位，最多有 15 个条件

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

2. 指令条件码中，V 标志位在什么情况下才能等于 1？

答：V—溢出标志位

对于加减法运算指令，当操作数和运算结果为二进制补码表示的带符号数时，V=1 表示符号

位溢出，其他指令通常不影响 V 位。

3. 在 ARM 指令中，什么是合法的立即数？判断下面各立即数是否合法，如果合法则写出在指令中的编码格式（也就是 8 位常数和 4 位移位数）

0x5430　0x108　0x304　0x501

0xfb10000　0x334000　0x3FC000　0x1FE0000

0x5580000　0x7F800 0x39C000　0x1FE80000

答：立即数必须由 1 个 8 位的常数通过进行 32 位循环右移偶数位得到，其中循环右移的位数由一个 4 位二进制的两倍表示。即一个 8 位的常数通过循环右移 2*rotate_4 位（即 0,2,4，。。。30）得到

0X5430

0000,0000,0000,0000,0101,0100,0011,0000 非法立即数

0X108

0000,0000,0000,0000,0000,0001,0000,1000　0x42 循环右移 30 位　（rotate_4=0xF）

0X304

0000,0000,0000,0000,0000,0011,0000,0100　0xC1 循环右移 30 位　（rotate_4=0xF）

0x501

0000,0000,0000,0000,0000,0101,0000,0001　非法立即数

0xfb10000

0000,1111,1011,0001,0000,0000,0000,0000　非法立即数

0x334000

0000,0000,0011,0011,0100,0000,0000,0000　0Xcd 循环右移 18 位（rotate_4=0x9）

0x3FC000

0000,0000,0011,1111,1100,0000,0000,0000　0XFF 循环右移 18 位（rotate_4=0x9）

0x1FE0000

0000,0001,1111,1110,0000,0000,0000,0000　非法立即数

0x5580000

0000,0101,0101,1000,0000,0000,0000,0000　非法立即数

0x7F800

0000,0000,0000,0111,1111,1000,0000,0000　非法立即数

0x39C000

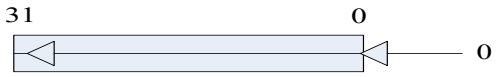
0000,0000,0011,1001,1100,0000,0000,0000　0XE7 循环右移 18 位（rotate_4=0x9）

0x1FE80000

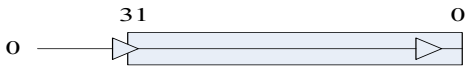
0001,1111,1110,1000,0000,0000,0000,0000　非法立即数

4.分析逻辑右移，算术右移，循环右移，带扩展的循环右移它们间的差别。

答：**LSL 逻辑左移 ：**

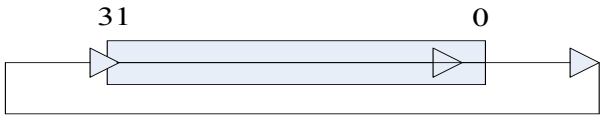


LSR 逻辑右移 ：

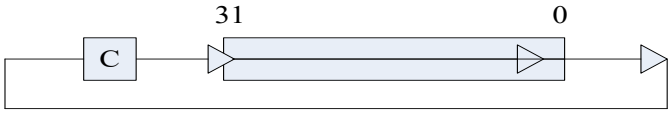


ASR 算术右移 ：

ROR 循环右移 ：



RRX 带扩展的循环右移：



5.ARM 数据处理指令具体的寻址方式有哪些，如果程序计数器 PC 作为目标寄存器，会产生什么结果？

答：数据处理指令寻址方式具体可分为 5 种类型：

1）第二操作数为立即数 2）第二操作数为寄存器 3）第二操作数为寄存器移位方式且移位的位数为一个 5 位立即数 4）第二操作数为寄存器移位方式且移位数值放在寄存器中 5）第二操作数位寄存器进行 RRX 移位得到。如果 PC （R15）用作目标寄存器，指令会产生不可预知的结果。

6.在 Load/Store 指令寻址中，字，无符号字节的 Load/Store 指令寻址和半字，有符号字节寻址，试分析它们之间的差别。

答：在 Load/Store 指令寻址中，

字，无符号字节的 Load/Store 指令寻址中共有以下 3 种内存地址构成格式：

1）Addressing_mode 中的偏移量为立即数 2）Addressing_mode 中的偏移量为寄存器的值 3）Addressing_mode 中的偏移量通过寄存器移位得到

半字，有符号字节的 Load/Store 指令寻址中共有以下 2 种内存地址构成格式：

1）Addressing_mode 中的偏移量为立即数 2）Addressing_mode 中的偏移量为寄存器的值

7.块拷贝 Load/Store 指令在实现寄存器组合连续的内存单元中数据传递时，地址的变化方式有哪几种类型，并分析它们的地址变化情况。

答：批量 Load/Store 指令在实现寄存器组合连续的内存单元中数据传递时，地址的变化方式有以下 4 种类型：

- **后增 IA (Increment After) ：每次数据传送后地址加 4；**
- **先增 IB (Increment Before) ：每次数据传送前地址加 4 ；**
- **后减 DA (Decrement After) ：每次数据传送后地址减 4 ；**
- **先减 DB (Decrement Before) ：每次数据传送前地址减 4 ；**

8.栈操作指令地址的变化方式有哪几种类型，并分析它们的地址变化情况，从而得出栈操作指令寻址和块拷贝 Load/Store 指令之间的对应关系。

答：根据堆栈指针的指向位置不同和堆栈的生长方向不同，共有 4 种类型的堆栈工作方式：

满递增堆栈 FA：堆栈指针指向最后压入的数据，且由低地址向高地址生成。

满递减堆栈 FD：堆栈指针指向最后压入的数据，且由高地址向低地址生成。

空递增堆栈 EA：堆栈指针指向下一个要放入数据的空位置，且由低地址向高地址生成。

空递减堆栈 ED：堆栈指针指向下一个要放入数据的空位置，且由高地址向低地址生成。

栈操作 / 块拷贝		地址变化方向			
		向上		向下	
		满	空	满	空
地址 增	先	STMIB STMFA			LDMIB LDMED
地址 增	后		STMIA STMEA	LDMIA LDMFD	
地址 减	先		LDMDB LDMEA	STMDB STMFD	
地址 减	后	LDMDA LDMFA			STMDA STMED

9.分析协处理器加载/存储指令的寻址方式种的内存地址索引格式中不同的汇编语法格式下内存地址的计算方法。

答：协处理器加载/存储指令的寻址方式种的内存地址索引格式中，索引格式类似于 LDR/STR 指令寻址中的立即数作为地址偏移量的形式。Addressing_mode 中的偏移量为 8 位立即数的汇编语法格式有以下 3 种：

- 前变址不回写形式：[<Rn>，#+/-<imm_offset8>*4]

  第一个内存地址编号为基地址寄存器 Rn 值加上/减去 imm_offset8 的 4 倍，后续的每一个地址是前一个内存地址加 4，直到协处理器发出信号，结束本次数据传输为止。

- 前变址回写形式：　[<Rn>，#+/-<imm_offset8>*4]！

  第一个内存地址编号为基地址寄存器 Rn 值加上/减去 imm_offset8 的 4 倍，后续的每一个地址是前一个内存地址加 4，直到协处理器发出信号，结束本次数据传输为止。当指令执行时，生成的地址值将写入基址寄存器。

- 后变址回写形式：　[<Rn>]，#+/-<imm_offset8>*4

  内存地址为基址寄存器 Rn 的值，当存储器操作完成后，将基地址寄存器 Rn 值加上/减去 imm_offset8 的 4 倍，后续的每一个地址是前一个内存地址加 4，直到协处理器发出信号，结束本次数据传输为止。最后将 Rn 值加上/减去 imm_offset8 的 4 倍写回到基址寄存器 Rn（更新基地址寄存器）。

10. 写出下列指令的机器码，并分析指令操作功能。

MOV　　R0，R1

MOV　　R1，,0X198

ADDEQS R1，R2，,0xAB

```
CMP     R2,#0Xab
LDR     R0,[R1,#4]
STR     R0,[R1,R1,LSL #2]!
LDRH    R0,[R1,#4]
LDRSB   R0,[R2,#-2]!
STRB    R1,[R2,#0Xa0]
LDMIA   R0,{R1,R2,R8}
STMDB   R0!,{R1-R5,R10,R11}
STMED   SP!{R0-R3,LR}
```

答：机器码部分略。

```
MOV     R0，R1                  ；R0《-----R1
MOV     R1，,0X198              ；R0《----0X198
ADDEQS R1，R2，,0xAB            ；当 Z=1 时，R1《---R2+0xAB 并影响标志位
CMP     R2,#0Xab                ；R2-0xAB，并影响标志位
LDR     R0,[R1,#4]              ；R0《---【R1+4】
STR     R0,[R1,R1,LSL #2]!      ；【R1+R1*4】《---R0，R1=R1+R1*4
LDRH    R0,[R1,#4]              ；R0《---【R1+4】半字，R0 的高 16 位清零
LDRSB   R0,[R2,#-2]!            ；R0《---【R2-2】字节，R0 有符号扩展为 32 位，R2=R2-2
STRB    R1,[R2,#0Xa0]           ；【R2+0Xa0】《----R1 低 8 位，
LDMIA   R0,{R1,R2,R8}
；将内存单元【R0】~【R+11】以字为单位读取到 R1，R2，R8 中
STMDB   R0!,{R1-R5,R10,R11}
```

将寄存器 R1~R5，R10，R11 的值以字为单位依次写入【R0】中，每写一个字之前 R0=R0-4

```
STMED   SP!{R0-R3,LR}
```

将寄存器 R0~R3，LR 的的值以字为单位依次写入【SP】中，每写一个字之后 SP=SP-4

第 4 章 ARM 指令集系统

1.ARM 指令可分为哪几类？说出哪几条指令是无条件执行的。

答：ARM 微处理器的指令集可以分为：数据处理指令，分支指令，加载/存储指令，批量加载/存储指令，交换指令，程序状态寄存器（PSR）处理指令，协处理器操作指令和异常产生指令八大类。

几乎所有的 ARM 指令都是可以有条件执行的。带链接和状态切换的跳转指令 BLX，当目标地址由程序标号给出时，即：BLX <target_address>

由于指令码中是没有条件编码位的，所以指令是无条件执行的。

2.如何实现两个 64 位数的加法操作，如何实现两个 64 位数的减法操作，如何求一个 64 位数的负数？

答：1）使用 ADC 实现 64 位加法，结果存于 R1、R0 中：

```
ADDS    R0,R0,R2    ;R0 等于低 32 位相加，并影响标志位
ADC     R1,R1,R3    ;R1 等于高 32 位相加，并加上低位进位
```

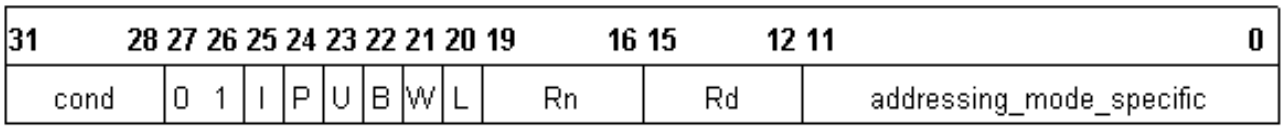
2）使用 SBC 实现 64 位减法，结果存于 R1、R0 中：

```
SUBS    R0,R0,R2    ; 低 32 位相减，并影响标志位
SBC     R1,R1,R3    ;高 32 位相减，并减去低位借位
```

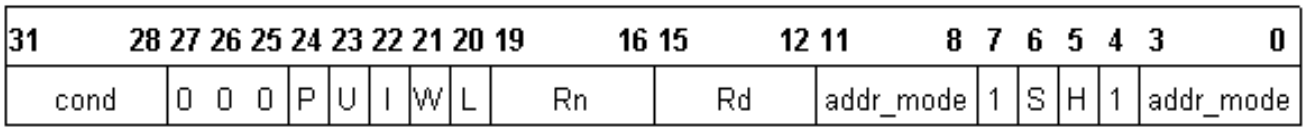
3）使用 RSC 指令实现求 64 位数值的负数 ：

RSBS R2,R0,#0
RSC R3,R1,#0

3.写出 LDRB 指令与 LDRSB 指令的二进制编码格式，并指出它们之间的区别。
答：LDRB 指令的二进制编码格式：



LDRSB 指令的二进制编码格式：



LDRB 指令用于将内存中的一个 8 位字节数据读取到指令中的目标寄存器的低 8 位中，寄存器的高 24 位用零扩展。
LDRSB 指令用于将内存中的一个 8 位字节数据读取到指令中的目标寄存器的低 8 位中，寄存器的高 24 位用符号位扩展。

4.分析下列每条语句的功能，并确定程序段所实现的操作。
CMP R0，,0
MOVEQ R1，,0
MOVGT R1，,1
答：CMP R0，,0 ；比较 R0 与 0 的大小
MOVEQ R1，,0 ；若 R0==0，则 R1=0
MOVGT R1，,1 ；若 R0>0，则 R1=1

5.请使用多种方法实现将字数据 0xFFFFFFFF 送入寄存器 R0
答：1）MVN R0，#0
2）MOV R0，#1
RSB R0，R0，#0

6.写一条 ARM 指令，分别完成下列操作：
(1)R0=16
(2)R0=R1/16
(3)R1=R2*3
(4)R0=-R0
答：
(1)R0=16 MOV R0，#16
(2)R0=R1/16 MOV R0，R1，LSR #4
(3)R1=R2*3 MOV R3，#3 MUL R1，R2，R3
(4)R0=-R0 RSB R0，R0，#0

7.编写一个 ARM 汇编程序，累加一个队列中的所有元素，碰到 0 时停止。结果放入 R4。

答：假设队列为地址从 R0 开始递增的字队列：

```
LOOP
        LDR R1, [R0,#4]!
        MOVS R2, R1
        BEQ END
        ADD R4,R4,R2
        B LOOP
END
```

8.写出实现下列操作的 ARM 指令：

当 Z=1 时，将存储器地址为 R1 的字数据读入寄存器 R0.

当 Z=1 时，将存储器地址为 R1+R2 的字数据读入寄存器 R0

将存储器地址为 R1-4 的字数据读入寄存器 R0。

将存储器地址为 R1+R6 的字数据读入寄存器 R0，并将新地址 R1+R6 写入 R1。

答：1)LDREQ R0, [R1]

2)LDREQ R0, [R1,R2]

3)LDR R0, [R1,#-4]

4)LDR R0, [R1,R6]!

9.写出下列 ARM 指令所实现的操作：

```
LDR     R2，[R3，#-4]!
LDR     R0，[R0],R2
LDR     R1，[R3，R2，LSL #2]!
LDRSB   R0，[R2，#-2]!
STRB    R1，[R2，,0xA0]
LDMIA   R0,{R1,R2,R8}
STMDB   R0!,{R1-R5,R10,R11}
```

答：LDR R2，[R3，#-4]! ;R2<-[R3-4], R3=R3-4

LDR R0，[R0],R2 ;R0<-[R0], R0=R0+R2

LDR R1，[R3，R2，LSL #2]! ;R1<-[R3+R2*4], R3=R3+R2*4

LDRSB R0，[R2，#-2]!

;R0 低 8 位<-[R2-2]字节数据,，R0 高 24 位符号扩展，R2=R2-2

STRB R1，[R2，#0xA0]

R1 低 8 位->【R2+0xA0】

LDMIA R0,{R1,R2,R8}

从地址 R0 开始的内存中依次读取字数据，送入寄存器 R1，R2，R8

STMDB R0!,{R1-R5,R10,R11}

将寄存器 R11，R10，R5-R1 的字数据，依次写入地址 R0 中，每次写入前 R0=R0-4

10.SWP 指令的优势是什么？

答：SWP 指令支持原子操作，它能在一条指令中完成存储器和寄存器之间的数据交换。

11. 如何用带 PSR 操作的批量字数据加载指令实现 IRQ 中断的返回？

答：在进入 IRQ 中断处理程序时，首先计算返回地址，并保存相关的寄存器

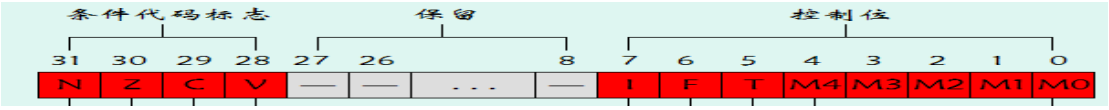
- SUB R14,R14,#4 ;
- STMFD R13!, {R0-R3, R12, LR}

如果 IRQ 中断处理程序返回到被中断的进程则执行下面的指令。该指令从数据栈中恢复寄存器 R0~R3 及 R12 的值，将返回地址传送到 PC 中，并将 SPSR_irq 值复制到 CPSR 中

- LDMFD R13!, {R0-R3, R12, PC}^

12. 用 ARM 汇编语言编写代码，实现将 ARM 处理器切换到用户模式，并关闭中断。

答：




```
；禁能 IRQ 中断
MRS     R0 CPSR
ORR     R0, R0,#0x80
MSR     CPSR, R0
；切换到用户模式
MRS     R0 CPSR
BIC     R0, #0x0F
MSR     CPSR, R0
```

第 5 章 Thumb 指令

1.与 32 位的 ARM 指令集相比较，16 位的 Thumb 指令集具有哪些优势？

答：在 ARM 体系结构中，ARM 指令集是 32 位的，具有很高的执行效率。但是对于嵌入式而言，其存储空间极其有限，由于每条 ARM 指令都要占用 4 个字节，对存储空间的要求较高。为了压缩代码的存储，增加代码存储密度，ARM 公司设计了 16 位的 Thumb 指令。Thumb 代码所需的存储空间约为 ARM 代码的 60%~70%。

2.Thumb 指令可分为哪几类？Thumb 指令有条件执行指令吗，如果有请说明哪些指令是条件执行的。

答：Thumb 指令可分为数据处理指令，存储器操作指令，分支指令，软中断指令。

Thumb 指令集只有一条分支指令是有条件的，其余所有指令都是无条件的；

B{cond} label

3.分析下面的 Thumb 指令程序代码，指出程序所完成的功能。

```
.global _start
.text
.equ    num 20
_start:
    MOV     SP,#0x400
    ADR     R0,Thumb_start+1
    BX      R0
.thumb
```

```
Thumb_start:
    ASR     R2,R0,#31
    EOR     R0,R2
    SUB     R3,R0,R2
stop:
    B       stop
.end
```

答：上述代码首先将处理器状态切换到 Thumb 状态，

```
    ASR     R2,R0,#31    ;  用 R0 的符号位填充 R2
    EOR     R0,R2        ；如果 R0 为正数，则 R0 不变；如果 R0 为负数，则 R0 取反
    SUB     R3,R0,R2     ；R0-R2->R3    (R2 为全零或全 1)
```

4.在 Thumb 状态中，用多种方法实现将寄存器 R0 中的数据乘以 10

答：1）MOV R1，#10

```
    MUL R0,R1
```

2) LSL R1，R0,#3

```
    LSL R2，R0,#1
    ADD R0,R1,R2
```

5.带链接的分支指令 BL 提供了一种在 Thumb 状态下程序间相互调用的方法，当从子程序返回时，可以采用哪种返回方式？

答：通常使用下面的方式之一：

```
MOV   PC, LR
BX    LR
POP {PC}   ;需要在子程序中使用 PUSH {LR}
```

6.指出下列的 Thumb 程序代码所完成的功能：

```
ASR     R0,R1,#31
EOR     R1,R0
SUB     R1,R0
```

答：

```
ASR     R0,R1,#31    ；用 R1 的符号位填充 R0
EOR     R1,R0        ；如果 R1 为正数，则 R1 不变；如果 R1 为负数，则 R1 取反
SUB     R1,R0        ；R1-R0->R1    (R0 为全零或全 1)
```

第 6 章 ARM 汇编伪指令与伪操作

1.在 ARM 汇编语言程序设计中，伪操作与伪指令的区别是什么？

答：伪指令是 ARM 处理器支持的汇编语言程序里的特殊助记符，它不再处理器运行期间由机器执行，只是在汇编时被合适的机器指令代替成 ARM 或 Thumb 指令，从而实现真正的指令操作。

伪操作是 ARM 汇编语言程序里的一些特殊的指令助记符，其作用主要是为了完成汇编程序做各种准备工作，对源程序运行汇编程序处理，而不是在计算机运行期间由处理器执行。也就是说，这些伪操作只是在汇编过程中起作用，一旦汇编结束，伪操作也就随之消失。

2.分析 ARM 汇编语言伪指令 LDR，ADRL，ADR 的汇编结果，说明它们之间的区别。

答：　　LDR 伪指令将一个 32 位的常数或者一个地址值读取到寄存器中，可以看作是加载寄存器的内容。如果加载的常数符合 MOV 或 MVN 指令立即数的要求，则用 MOV 或 MVN 指令替代 LDR 伪指令。如果加载的常数不符合 MOV 或 MVN 指令立即数的要求，汇编器将常量放入内存文字池，并使用一条程序相对偏移的 LDR 指令从内存文字池读出常量。

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

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

3.在 ADS 编译环境下，写出下列操作的伪操作：

（1）声明一个局部的算术变量 La_var 并将其初始化 0；

（2）声明一个局部的逻辑变量 Ll_var 并将其初始化 FALSE；

（3）声明一个局部的字符串变量 Ls_var 并将其初始化 空串；

（4）声明一个全局的逻辑变量 Gl_var 并将其初始化 FALSE；

（5）声明一个全局的字符串变量 Gs_var 并将其初始化 空串；

（6）声明一个全局的算术变量 Ga_var 并将其初始化 0xAA；

（7）声明一个全局的逻辑变量 Gl_var 并将其初始化 TRUE；

（8）声明一个全局的字符串变量 Gs_var 并将其初始化 “CHINA”；

答：

（1）声明一个局部的算术变量 La_var 并将其初始化 0；

```
LCLA  La_var
```

（2）声明一个局部的逻辑变量 Ll_var 并将其初始化 FALSE；

```
LCLL  Ll_var
```

（3）声明一个局部的字符串变量 Ls_var 并将其初始化 空串；

```
LCLS  Ls_var
```

（4）声明一个全局的逻辑变量 Gl_var 并将其初始化 FALSE；

```
GCLL  Gl_var
```

（5）声明一个全局的字符串变量 Gs_var 并将其初始化 空串；

```
GCLS  Gs_var
```

（6）声明一个全局的算术变量 Ga_var 并将其初始化 0xAA；

```
GCLA  Ga_var
Ga_ var  SETA  0xAA
```

（7）声明一个全局的逻辑变量 Gl_var 并将其初始化 TRUE；

```
GCLL  Gl_var
Gl_ var  SETL  TRUE
```

（8）声明一个全局的字符串变量 Gs_var 并将其初始化 “CHINA”；

```
GCLS  Gs_var
Gs_ var  SETS  "CHINA"
```

4.用 ARM 开发工具伪操作将寄存器列表 R0-R5，R7，R8 的名称定义为 Reglist。

答：Reglist　RLST　{R0-R5，R7，R8}

5.完成下列数据定义伪操作：
（1）申请以 data_buffer1 为起始地址的连续的内存单元，并依次用半字数据 0x11,0x22,0x33,0x44,0x55 进行初始化；
（2）申请以 Str_buffer 为起始地址的连续的内存单元，并用字符串“ARM7 and ARM9”进行初始化；
答：(1)　data_buffer1　　DCW　　0x11,0x22,0x33,0x44,0x55
　　(2)　Str_buffer　　DCB　　“ARM7 and ARM9”

6.定义一个结构化的内存表，其首地址固定为 0x900，该结构化内存表包含 2 个域，Fdatal 长度为 8 个字节，Fdata2 长度为 160 个字节。
答：MAP　0x900
　　Fdata1　FIELD　8
　　Fdata2　FIELD　160

7.在 GNU-ARM 编译环境下，写出实现下列操作的伪操作：
（1）分配一段字节内存单元，并用 57,0x11,031，'Z'，0x76 进行初始化；
（2）分配一段半字内存单元，并用 0xFFE0,0xAABB，0x12 进行初始化；
（3）分配一段字内存单元，　并用 0x12345678,0xAABBCCDD 进行初始化；
（4）分配一段内存单元，并用长为 8 字节的数值 0x11 填充 100 次；
答：
（1）分配一段字节内存单元，并用 57,0x11,031，'Z'，0x76 进行初始化；
　.byte　57,0x11,031，'Z'，0x76
（2）分配一段半字内存单元，并用 0xFFE0,0xAABB，0x12 进行初始化；
　.hword　0xFFE0,0xAABB，0x12
（3）分配一段字内存单元，　并用 0x12345678,0xAABBCCDD 进行初始化；
　.word　0x12345678,0xAABBCCDD
（4）分配一段内存单元，并用长为 8 字节的数值 0x11 填充 100 次；
　.fill　100 , 8, 0x11

8.写出与 GNU-ARM 编译环境下伪操作.arm　，　.thumb 功能相同的 ARM 标准开发工具编译环境下的伪操作。
答：.arm　对应 ARM 或 CODE32
　.thumb 对应 THUMB 或 CODE16

第 7 章 汇编语言程序设计

1.分别写出 ARM 集成开发环境下 ARM 汇编语句格式与 GNU ARM 环境下 ARM 汇编语句通用格式，并分析它们的区别。
答：ADS 环境下 ARM 汇编语句格式如下：

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

- Symbol ：标号（地址）
- Instruction ：指令（ARM/Thumb）
- Directive ：伪操作
- pseudo-instruction：伪指令

GNU 环境下 ARM 汇编语言语句格式如下：

- {label :} {instruction} {@comment}
- {label :} {directive} {@comment}
- {label :} { pseudo-instruction} {@comment}

2.局部标号提供分支指令在汇编程序的局部范围内跳转，它的主要用途是什么，并举一实例加以说明。

答：局部标号

- 局部标号的语法格式如下：
- n {routname} （0~99）
- 被引用的局部标号语法规则是：
- % {F｜B} {A｜T} n {routname}
- 其中：
- n 是局部标号的数字号。（0~99）
- routname 是当前局部范围的名称。
- %表示引用操作。
- F 指示汇编器只向前搜索。
- B 指示汇编器只向后搜索。
- A 指示汇编器搜索宏的所有嵌套层次。
- T 指示汇编器搜索宏的当前层次。
- 局部标号提供分支指令在汇编程序在局部范围内的跳转

3.先对内存地址 0xB000 开始的 100 个字内存单元填入 0x10000001~0x10000064 字数据，然后将每个字单元进行 64 位累加，结果保存于【R9：R8】（R9 中存放高 32 位）。

答：解：先对内存地址 0xB000 开始的 100 个字内存单元填入 0x10000001~0x10000064 字数据，然后将每个字单元进行 64 位累加，结果保存于【R9：R8】（R9 中存放高 32 位）。

在 ARM 集成开发环境下编程：

```
/*-------------------------------------------------------------------------------
********寄存器使用说明************************
***R0:存放地址值
***R2:递减计数器
***R9:64 位递加结果的高 32 位
***R8:64 位递加结果的低 32 位
*-------------------------------------------------------------------------------*/
        AREA    Fctrl,CODE,READONLY         ;声明代码段 Fctrl
        ENTRY                               ;标识程序入口
        CODE32                              ;声明 32 位 ARM 指令
START
        MOV     R0,#0xB000                  ；初始化寄存器
```

```
        MOV     R1,#0x10000001
        MOV     R2,#100
loop_1                                  ；第一次循环赋值
        STR     R1,[R0],#4
        ADD     R1,R1,#1
        SUBS    R2,R2,#1
        BNE     loop_1

        MOV     R0,#0xB000
        MOV     R2,#100
        MOV     R9,#0
        MOV     R8,#0
loop_2                                  ；第二次循环累加
        LDR     R1,[R0],#4
        ADDS    R8,R1,R8                ；R8=R8+R1，进位影响标志位
        ADDC    R9,R9,#0                ；R9=R9+C，C 为进位位
        SUBS    R2,R2,#1
        BNE     loop_2
Stop
        B       Stop                    ；文件结束
        END
```

4.在 GNU 环境下用 ARM 汇编语言编写程序，初始化 ARM 处理器各模式下的堆栈指针 SP_mode (R13)，各模式的堆栈指针地址如下：

```
.equ   _ISR_STARTADDRESS, 0xCFFF000             @设置栈的内存基地址
.equ    UserStack,      _ISR_STARTADDRESS       @用户模式堆栈地址
.equ    SVCStack,       _ISR_STARTADDRESS+64    @管理模式堆栈地址
.equ    UndefStack,     _ISR_STARTADDRESS+64*2  @未定义模式堆栈地址
.equ    AbortStack,     _ISR_STARTADDRESS+64*3  @中止模式堆栈地址
.equ    IRQStack,       _ISR_STARTADDRESS+64*4  @IRQ 模式堆栈地址
.equ    FIQStack,       _ISR_STARTADDRESS+64*5  @FIQ 模式堆栈地址
```

答：

在 GNU ARM 开发环境下编程：

```
.equ   _ISR_STARTADDRESS, 0xCFFF000             @设置栈的内存基地址
.equ    UserStack,      _ISR_STARTADDRESS       @用户模式堆栈地址
.equ    SVCStack,       _ISR_STARTADDRESS+64    @管理模式堆栈地址
.equ    UndefStack,     _ISR_STARTADDRESS+64*2  @未定义模式堆栈地址
.equ    AbortStack,     _ISR_STARTADDRESS+64*3  @中止模式堆栈地址
.equ    IRQStack,       _ISR_STARTADDRESS+64*4  @IRQ 模式堆栈地址
.equ    FIQStack,       _ISR_STARTADDRESS+64*5  @FIQ 模式堆栈地址

.equ    USERMODE    0x10    @用户模式
.equ    FIQMODE     0x11    @FIQ 模式
.equ    IRQMODE     0x12    @IRQ 模式
```

```
.equ    SVCMODE     0x13    @管理模式
.equ    ABORTMODE   0x17    @中止模式
.equ    UNDEFMODE   0x1B    @未定义模式
.equ    SYSMODE     0x1F    @系统模式
.equ    MODEMASK    0x1F    @模式位掩码控制字
.global _start
.text
.arm
_start:
        MRS     R0,CPSR                 @读取当前 CPSR
        BIC     R0,R0,#MODEMASK         @清除模式位

        @设置系统模式下的 SP
        ORR     R1,R0,#SYSMODE
        MSR     CPSR_c,R1
        LDR     SP,=UserStack

        @设置中止模式下的 SP
        ORR     R1,R0,#ABORTMODE
        MSR     CPSR_c,R1
        LDR     SP,=AbortStack

        @设置管理模式下的 SP
        ORR     R1,R0,#SVCMODE
        MSR     CPSR_c,R1
        LDR     SP,=SVCStack

        @设置 IRQ 模式下的 SP
        ORR     R1,R0,#IRQMODE
        MSR     CPSR_c,R1
        LDR     SP,=IRQStack

        @设置 FIQ 模式下的 SP
        ORR     R1,R0,#FIQMODE
        MSR     CPSR_c,R1
        LDR     SP,=FIQStack

Stop :
        B       Stop
.end                                    @文件结束
```

5.内存数据区定义如下：

Src:

```
        .long 1,2,3,4,5,6,7,8,9,0xA,0xB,0xC,0xD,0xE,0xF,0x10
        .long 1,2,3,4,5,6,7,8,9,0xA,0xB,0xC,0xD,0xE,0xF,0x10
Src_Num: .long 32
Dst:
        .long 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        .long 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

请用 ARM 指令编写程序，实现将数据从源数据区 Src 拷贝到目标数据区 Dst，要求以 6 个字为单位进行块拷贝，如果不足 6 个字时，则以字为单位进行拷贝（其中数据区 Src_Num 处存放源数据的个数）。

答：解：程序设计思路：每进行 6 个字的批量拷贝前，先判断 SRC_NUM 是否大于 6，是则进行 6 字的批量拷贝并将 SRC_NUM 减去 6，否则则进行单字的拷贝，在使用寄存器组时还要注意保存现场。

在 ARM 集成开发环境下编程：

```
/*-------------------------------------------------------------------------------------
********寄存器使用说明************************
***R0:源数据区指针
***R1:目标数据区指针
***R2:单字拷贝字数
***R3:块拷贝字数
***R5~~R10:批量拷贝使用的寄存器组
***SP:栈指针
*-------------------------------------------------------------------------------------*/
SRC_NUM         EQU     32                          ；设置要拷贝的字数
        AREA    Copy_Data,CODE,READONLY  ;声明代码段 Copy_Data
        ENTRY                                       ;标识程序入口
        CODE32                                      ;声明 32 位 ARM 指令
START
    LDR     R0,=Src                 ;R0=源数据区指针
    LDR     R1，=Dst                ；R1=目标数据区指针
    MOV     R2,#SRC_NUM             ；R2=单字拷贝字数
    MOV     SP,#0x9000

    CMP     R2,#6
    BLS     Copy_Words          ；R2<=6，则拷贝单字
    STMFD   SP!,{R5-R10}        ；保存将要使用的寄存器组 R5-R10
    ;进行块拷贝，每次拷贝 6 个字
Copy_6Word
    LDMIA   R0!,{R5-R10}
    STMIA   R1!,{R5-R10}
    SUBS    R2,R2,#6
    BHI     Copy_6Word          ;R2>6
    LDMFD   SP!,{R5-R10}        ；恢复寄存器组 R5-R10
    ;将剩余的数据区以字为单位拷贝
```

```
Copy_Words                          ;拷贝剩余字节
    LDR     R3,[R0],#4
    STR     R3,[R1],#4
    SUBS    R2,R2,#1
    BNE     Copy_Word
Stop
    B       Stop
    LTORG
Src
    DCD     1,2,3,4,5,6,7,8,9,0xA,0xB,0xC,0xD,0xE,0xF,0x10
    DCD     1,2,3,4,5,6,7,8,9,0xA,0xB,0xC,0xD,0xE,0xF,0x10
Dst
    DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    DCD     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    END
```

6.将一个存放在【R1:R0】中的 64 位数据（其中 R1 中存放高 32 位）的高位和低位对称换位，如第 0 位与第 63 位调换，第 1 位与第 62 位调换，第 2 位与第 61 位调换，。。。。第 31 位与第 32 位调换。

答：解：程序设计思路：对于单个 32 位寄存器的对称换位操作，我们可以采用移位操作的方法，通过依次从低位取出目标寄存器的各个位，再将其放置到目标寄存器的最低位，然后通过移位操作，送入相应位。对于【R1:R0】到【R3:R2】的 64 位对称换位操作，我们可以采用 R1-》R2 和 R0-》R3 的两个 32 位的换位操作来完成。

在 ARM 集成开发环境下编程：

```
/*-------------------------------------------------------------------------------
********寄存器使用说明**********************
***R1，R0:源数据
***R3，R2:目标数据
***R4:计数器，初值为 32，递减至 0
*-------------------------------------------------------------------------------*/
        AREA    Bit_Exch,CODE,READONLY      ;声明代码段 Bit_Exch
        ENTRY                               ;标识程序入口
        CODE32                              ;声明 32 位 ARM 指令
START
        LDR     R0,=0x55555555      ;输入源数据
        MOV     R3,#0               ；目标数据
        MOV     R5,#0               ；数据临时缓冲区
        MOV     R4,#32              ；计数器
Bitex_H32
        AND     R5,R0,#1            ；取出源数据的最低位送 R5
        ORR     R3,R5,R3,LSL #1     ；将目标数据左移一位，并将取出的数据
                                    ;送入其最低位
```

```
        MOV     R0,R0,LSR #1        ;源数据右移一位
        SUBS    R4,R4,#1            ; 递减计数
        BNE     Bitex_L

        LDR     R1,=0x55555555      ;输入源数据
        MOV     R2,#0               ; 目标数据
        MOV     R5,#0               ; 数据临时缓冲区
        MOV     R4,#32              ; 计数器
Bitex_L32
        AND     R5,R1,#1            ; 取出源数据的最低位送 R5
        ORR     R2,R5,R2,LSL #1     ; 将目标数据左移一位，并将取出的数据
                                    ;送入其最低位
        MOV     R1,R1,LSR #1        ;源数据右移一位
        SUBS    R4,R4,#1            ; 递减计数
        BNE     Bitex_L

Stop
        B       Stop
        END
```

7.内存数据区定义如下：

```
DataZone
    DCD   0x12345678,   0x87654321,   0xABCDEF12,   0xCDEFAB45
    DCD   0x12345678,   0x87654321,   0xABCDEF12,   0xCDEFAB45
    DCD   0x12345678,   0x87654321,   0xABCDEF12,   0xCDEFAB45
    DCD   0x12345678,   0x87654321,   0xABCDEF12,   0xCDEFAB45
    DCD   0x12345678,   0x87654321,   0xABCDEF12,   0xCDEFAB45
    DCD   0x12345678,   0x87654321,   0xABCDEF12,   0xCDEFAB45
    DCD   0x12345678,   0x87654321,   0xABCDEF12,   0xCDEFAB45
    DCD   0x12345678,   0x87654321,   0xABCDEF12,   0xCDEFAB45
```

以上可以看做一个 8*4 矩阵，请用 ARM 汇编语言在 ARM 集成开发环境下设计程序，实现对矩阵的转置操作。

如果改为在 GNU ARM 环境下编程，程序应如何修改。

答：解：使用 R0 指向源数据区，R1 指向目标数据区。从源数据区中按列取数据（每次取 8 个），然后顺序存入目标数据区。

在 ARM 集成开发环境下编程：

```
/*------------------------------------------------------------------------------------------
********寄存器使用说明************************
***R0:源数据
***R1:目标数据
***R2:行计数器，初值为 8，递减至 0
***R3:列计数器，初值为 4，递减至 0
*-------------------------------------------------------------------------------------------*/
```

```
        AREA    Bit_Exch,CODE,READONLY      ;声明代码段 Bit_Exch
        ENTRY                               ;标识程序入口
        CODE32                              ;声明 32 位 ARM 指令
START
        LDR     R0,=Src
        LDR     R1,=Dst
        MOV     R3,#4
COL
        MOV     R2,#8
ROW
        LDR     R4,[R0],#16         ；按列取数据
        STR     R4,[R1],#4          ；按行存数据
        SUBS    R2,R2,#1            ；行计数递减
        BNE     ROW

        SUBS    R3,R3,#1            ；列计数递减
        BNE     COL

Stop
        B       Stop
        END
```

第 8 章 ARM 汇编语言与嵌入式 C 混合编程

1.严格按照嵌入式 C 语言的编程规范，写一个 C 语言程序，实现将一个二维数组内的数据行和列进行排序。

答：略

2.嵌入式 C 程序设计中常用的移位操作有哪几种，请说明每种运算所对应的 ARM 指令实现。

答：移位操作分为左移操作与右移操作

左移运算符“<<”实现将“<<”左边的操作数的各个二进制位向左移动“<<”右边操作数所指定的位数，高位丢弃，低位补 0。其值相当于乘以：2“左移位数”次方。

右移运算符“>>”实现将“>>”左边的操作数的各个二进制位向右移动“<<”右边操作数所指定的位数。

- 对于空位的补齐方式，无符号数与有符号数是有区别的。
- 对无符号数进行右移时，低位丢弃，高位用 0 补齐，其值相当于除以：2“右移位数”次方
- 对有符号数进行右移时，根据处理器的不同选择逻辑右移或算术右移

3.volatile 限制符在程序中起到什么作用。请举例说明。

答：volatile 的本意为 “暂态的”或.“易变的”，该说明符起到抑制编译器优化的作用。

如果在声明时用“volatile”关键进行修饰，遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供特殊地址的稳定访问。

- 例:硬件端口寄存器读取

- Char x=0,y=0,z=0;
- x=ReadChar(0x54000000);//读端口
- y=x;
- x=ReadChar(0x54000000);//再读端口
- z=x;
- 以上代码可能被编译器优化为
- Char x=0,y=0,z=0;
- x=ReadChar(0x54000000);//读端口
- y=x;
- z=x;
- 为了确保 x 的值从真实端口获取，声明时应该为
- Volatile char x;
- Char y,z;

4.请分析下列程序代码的执行结果。

```
#include<stdio.h>
main(){
    int value=0xFF1;
    int *p1,**p2,***p3,****p4;
    p1=&value;
    p2=&p1;
    p3=&p2;
    p4=&p3;
    printf("****p4=%d\n",****p4);
}
```

答：程序输出结果为：****p4=4081

5.分析宏定义#define POWER(x) x*x 是否合理，举例说明。如果不合理，应如何更改？
答：#define POWER(x) x*x 不合理；对于带参数的宏，其参数应该用括号括起来。
例：如果按照下边方式使用该宏
POWER(2+3) 则宏展开后为 2+3*2+3
该宏应修改为：#define POWER(x) （x）*（x）

6.条件编译在程序设计中有哪些用途？
答：条件编译包括了 6 条预处理指令#ifdef, #ifndef. ##if, #elif, #else, #endif.。条件编译的功能在于对源程序中的一部分内容只有满足某种条件的情况下才进行编译。

7.何为可重入函数？如果使程序具有可重入性，在程序设计中应该注意哪些问题？
答：如果某个函数可以被多个任务并发使用，而不会造成数据错误，我们就说这个函数具有可重入性（reentrant） 。
可重入函数可以使用局部变量，也可以使用全局变量。
如果使用全局变量，则应通过关中断、信号量（即 P、V 操作）等手段对其加以保护，若不加以保护，则此函数就不具有可重入性，即当多个进程调用此函数时，很有可能使得此全局变量变为不可知状态。

8.现有模块 module_1， module_2， module_3, 要求在模块 module_1 中提供可供模块 module_2， module_3 使用的 int 型变量 xx，请写出模块化程序设计框架。
答：首先在 module_1 的.c 文件中定义 int xx；
/*module_1.c*/
int xx=0;
然后在 module_1 的.h 文件中声明 xx 为外部变量
/*module_1.h*/
extern int xx;
接下来在 module_2 源文件中包括 module_1 的 .h 文件
/*module_2.c*/
#include "module_1.h"
在 module_3 源文件中包括 module_1 的 .h 文件
/*module_3.c*/
#include "module_1.h"
这样在 module_2， module_3 中就可以使用 module_1 中提供的 int 型变量 xx 了。

9.ATPCS 与 AAPCS 的全称是是什么，它们有什么差别？掌握子程序调用过程中寄存器的使用规则，数据栈的使用规则及参数的传递规则，在具体的函数中能够熟练应用。
答：过程调用标准 ATPCS（ARM-Thumb Produce Call Standard）规定了子程序间相互调用的基本规则， ATPCS 规定子程序调用过程中寄存器的使用规则、数据栈的使用规则及参数的传递规则。
2007 年，ARM 公司推出了新的过程调用标准 AAPCS（ARM Architecture Produce Call Standard）,它只是改进了原有的 ATPCS 的二进制代码的兼容性。

10.内嵌式汇编有哪些局限性？编写一段代码采用 C 语言嵌入式汇编程序，在汇编程序中实现字符串的拷贝操作。
答：内嵌汇编的局限性
（1）操作数
- ARM 开发工具编译环境下内嵌汇编语言，指令操作数可以是寄存器、常量或 C 语言表达式。可以是 char、short 或 int 类型，而且是作为无符号数进行操作。
- 当表达式过于复杂时需要使用较多的物理寄存器，有可能产生冲突。
- GNU ARM 编译环境下内嵌汇编语言 ARM 开发工具稍有差别，不能直接引用 C 语言中的变量。

（2）物理寄存器
不要直接向程序计数器 PC 赋值，程序的跳转只能通过 B 或 BL 指令实现。
一般将寄存器 R0～R3、R12 及 R14 用于子程序调用存放中间结果，因此在内嵌汇编指令中，一般不要将这些寄存器同时指定为指令中的物理寄存器。
在内嵌的汇编指令中使用物理寄存器时，如果有 C 语言变量使用了该物理寄存器，则编译器将在合适的时候保存并恢复该变量的值。需要注意的是，当寄存器 SP、SL、FP 以及 SB 用作特定的用途时，编译器不能恢复这些寄存器的值。
通常在内嵌汇编指令中不要指定物理寄存器，因为有可能会影响编译器分配寄存器，进而可能影响代码的效率。

（3）标号、常量及指令展开

- C 语言程序中的标号可以被内嵌的汇编指令所使用。但是只有 B 指令可以使用 C 语言程序中的标号，BL 指令不能使用 C 语言程序中的标号。

（4）内存单元的分配

- 内嵌汇编器不支持汇编语言中用于内存分配的伪操作。所用的内存单元的分配都是通过 C 语言程序完成的，分配的内存单元通过变量以供内嵌的汇编器使用。
- （5）SWI 和 BL 指令
- SWI 和 BL 指令用于内嵌汇编时，除了正常的操作数域外，还必须增加如下 3 个可选的寄存器列表：
- 用于存放输入的参数的寄存器列表。
- 用于存放返回结果的寄存器列表。
- 用于保存被调用的子程序工作寄存器的寄存器列表。

第 8 章 ARM 汇编语言与嵌入式 C 混合编程

1.严格按照嵌入式 C 语言的编程规范，写一个 C 语言程序，实现将一个二维数组内的数据行和列进行排序。

答：略

2.嵌入式 C 程序设计中常用的移位操作有哪几种，请说明每种运算所对应的 ARM 指令实现。

答：移位操作分为左移操作与右移操作

左移运算符“<<”实现将“<<”左边的操作数的各个二进制位向左移动“<<”右边操作数所指定的位数，高位丢弃，低位补 0。其值相当于乘以：2“左移位数”次方。

右移运算符“>>”实现将“>>”左边的操作数的各个二进制位向右移动“<<”右边操作数所指定的位数。

- 对于空位的补齐方式，无符号数与有符号数是有区别的。
- 对无符号数进行右移时，低位丢弃，高位用 0 补齐，其值相当于除以：2“右移位数”次方
- 对有符号数进行右移时，根据处理器的不同选择逻辑右移或算术右移

3.volatile 限制符在程序中起到什么作用。请举例说明。

答：volatile 的本意为 “暂态的”或.“易变的”，该说明符起到抑制编译器优化的作用。

如果在声明时用“volatile”关键进行修饰，遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供特殊地址的稳定访问。

- 例:硬件端口寄存器读取
- Char x=0,y=0,z=0;
- x=ReadChar(0x54000000);//读端口
- y=x;
- x=ReadChar(0x54000000);//再读端口
- z=x;
- 以上代码可能被编译器优化为
- Char x=0,y=0,z=0;
- x=ReadChar(0x54000000);//读端口
- y=x;

- z=x;
- 为了确保 x 的值从真实端口获取，声明时应该为
- Volatile char x;
- Char y,z;

4.请分析下列程序代码的执行结果。

```
#include<stdio.h>
main(){
    int value=0xFF1;
    int *p1,**p2,***p3,****p4;
    p1=&value;
    p2=&p1;
    p3=&p2;
    p4=&p3;
    printf("****p4=%d\n",****p4);
}
```

答：程序输出结果为：****p4=4081

5.分析宏定义#define POWER(x) x*x 是否合理，举例说明。如果不合理，应如何更改？
答：#define POWER(x) x*x 不合理；对于带参数的宏，其参数应该用括号括起来。
例：如果按照下边方式使用该宏
POWER(2+3) 则宏展开后为 2+3*2+3
该宏应修改为：#define POWER(x) （x）*（x）

6.条件编译在程序设计中有哪些用途？
答：条件编译包括了 6 条预处理指令#ifdef, #ifndef. ##if, #elif, #else, #endif.。条件编译的功能在于对源程序中的一部分内容只有满足某种条件的情况下才进行编译。

7.何为可重入函数？如果使程序具有可重入性，在程序设计中应该注意哪些问题？
答：如果某个函数可以被多个任务并发使用，而不会造成数据错误，我们就说这个函数具有可重入性（reentrant） 。
可重入函数可以使用局部变量，也可以使用全局变量。
如果使用全局变量，则应通过关中断、信号量（即 P、V 操作）等手段对其加以保护，若不加以保护，则此函数就不具有可重入性，即当多个进程调用此函数时，很有可能使得此全局变量变为不可知状态。

8.现有模块 module_1, module_2, module_3, 要求在模块 module_1 中提供可供模块 module_2, module_3 使用的 int 型变量 xx，请写出模块化程序设计框架。
答：首先在 module_1 的.c 文件中定义 int xx；

```
/*module_1.c*/
int xx=0;
```

然后在 module_1 的.h 文件中声明 xx 为外部变量

```
/*module_1.h*/
extern int xx;
```

接下来在 module_2 源文件中包括 module_1 的 .h 文件
/*module_2.c*/
#include "module_1.h"
在 module_3 源文件中包括 module_1 的 .h 文件
/*module_3.c*/
#include "module_1.h"
这样在 module_2, module_3 中就可以使用 module_1 中提供的 int 型变量 xx 了。

9.ATPCS 与 AAPCS 的全称是是什么，它们有什么差别？掌握子程序调用过程中寄存器的使用规则，数据栈的使用规则及参数的传递规则，在具体的函数中能够熟练应用。
答：过程调用标准 ATPCS（ARM-Thumb Produce Call Standard）规定了子程序间相互调用的基本规则， ATPCS 规定子程序调用过程中寄存器的使用规则、数据栈的使用规则及参数的传递规则。
2007 年，ARM 公司推出了新的过程调用标准 AAPCS（ARM Architecture Produce Call Standard）,它只是改进了原有的 ATPCS 的二进制代码的兼容性。

10.内嵌式汇编有哪些局限性？编写一段代码采用 C 语言嵌入式汇编程序，在汇编程序中实现字符串的拷贝操作。
答：内嵌汇编的局限性
（1）操作数
- ARM 开发工具编译环境下内嵌汇编语言，指令操作数可以是寄存器、常量或 C 语言表达式。可以是 char、short 或 int 类型，而且是作为无符号数进行操作。
- 当表达式过于复杂时需要使用较多的物理寄存器，有可能产生冲突。
- GNU ARM 编译环境下内嵌汇编语言 ARM 开发工具稍有差别，不能直接引用 C 语言中的变量。

（2）物理寄存器

不要直接向程序计数器 PC 赋值，程序的跳转只能通过 B 或 BL 指令实现。

一般将寄存器 R0～R3、R12 及 R14 用于子程序调用存放中间结果，因此在内嵌汇编指令中，一般不要将这些寄存器同时指定为指令中的物理寄存器。

在内嵌的汇编指令中使用物理寄存器时，如果有 C 语言变量使用了该物理寄存器，则编译器将在合适的时候保存并恢复该变量的值。需要注意的是，当寄存器 SP、SL、FP 以及 SB 用作特定的用途时，编译器不能恢复这些寄存器的值。

通常在内嵌汇编指令中不要指定物理寄存器，因为有可能会影响编译器分配寄存器，进而可能影响代码的效率。

（3）标号、常量及指令展开
- C 语言程序中的标号可以被内嵌的汇编指令所使用。但是只有 B 指令可以使用 C 语言程序中的标号，BL 指令不能使用 C 语言程序中的标号。

（4）内存单元的分配
- 内嵌汇编器不支持汇编语言中用于内存分配的伪操作。所用的内存单元的分配都是通过 C 语言程序完成的，分配的内存单元通过变量以供内嵌的汇编器使用。
- （5）SWI 和 BL 指令
- SWI 和 BL 指令用于内嵌汇编时，除了正常的操作数域外，还必须增加如下 3 个可选的寄存器列表：
- 用于存放输入的参数的寄存器列表。

- 用于存放返回结果的寄存器列表。
- 用于保存被调用的子程序工作寄存器的寄存器列表。