浙江大学
ZHEJIANG UNIVERSITY

嵌入式系统设计

第4课 嵌入式软件编程技术

王总辉
zhwang@zju.edu.cn

http://course.zju.edu.cn/

提　纲

- 嵌入式编程基本概念
- 嵌入式汇编编程技术
- 嵌入式高级编程技术
- 高级语言与低级语言混合编程

嵌入式软件编程技术

- 嵌入式编程基本概念
- 嵌入式汇编编程技术
- 嵌入式高级编程技术
- 高级语言与低级语言混合编程

嵌入式编程基本概念

- 汇编语言程序基本概念
- 高级语言程序设计概念
- 嵌入式开发工程概念

汇编语言程序基本结构

- ARM寄存器组织
- ARM指令格式
- ARM寻址方式
- 伪指令

ARM寄存器组织（1）

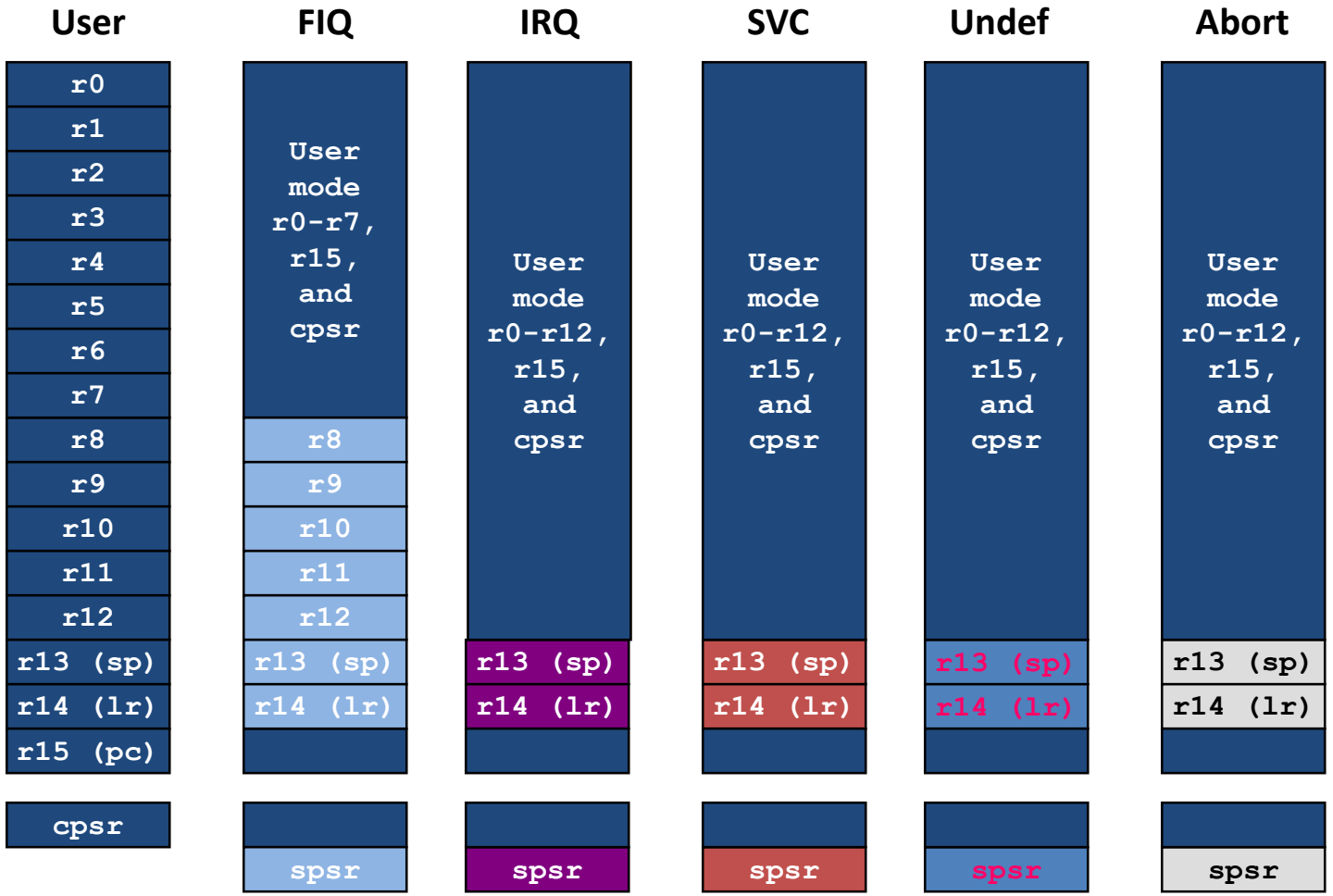
- ARM总共有7种不同的处理器模式
  - 用户模式(User)，这是程序正常执行的模式
  - 快中断模式(FIQ)，用于高速数据传输和通道处理
  - 外部中断模式(IRQ)，用于普通的外部中断请求处理
  - 管理模式(Supervisor)，供操作系统使用的保护模式
  - 数据访问中止模式(Abort)，用于虚拟存储和存储保护
  - 未定义模式(Undef)，用于支持硬件协处理器软件仿真
  - 系统模式(System)，用于运行特权级的操作系统任务

特权模式　异常模式

ARM寄存器组织（2）

- ARM 有37个32-Bits长的寄存器.
  - 1 个用作PC(program counter)
  - 1个用作CPSR(current program status register)
  - 5个用作SPSR(saved program status registers)
  - 30 个通用寄存器
- 当前处理器的模式决定着哪组寄存器可操作. 任何模式都可以存取：
  - 相应的r0-r12子集
  - 相应的 r13 (the stack pointer, sp) and r14 (the link register, lr)
  - 相应的 r15 (the program counter, pc)
  - 相应的CPSR(current program status register, cpsr)
- 特权模式 (除system模式) 还可以存取；
  - 相应的 spsr (saved program status register)

ARM寄存器组织（3）



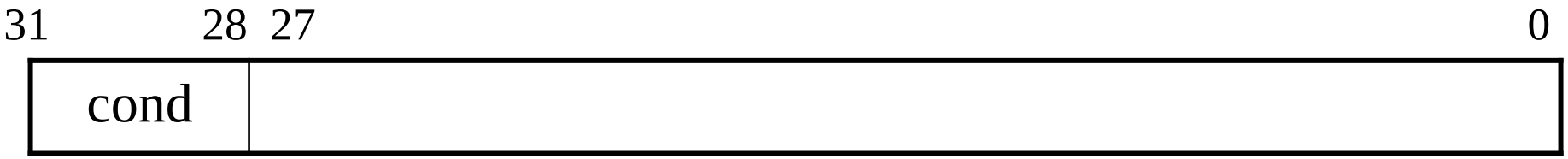
注意: System模式使用user模式寄存器集

指令格式——基本格式

<opcode>{<cond>}{S} <Rd>,<Rn> {, <shift_op2>}

opcode	操作码，即指令助记符
cond	条件码，描述指令执行的条件
S	自动更新CPSR条件码标志位
Rd	目标操作数
Rn	第1操作数寄存器
opcode2	第2操作数寄存器/立即数/ 位移运算的寄存器和立即数

指令格式——基本格式

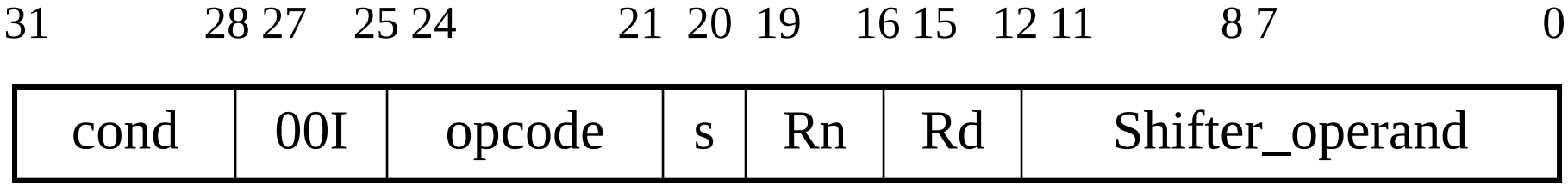


cond：指令执行的条件编码

指令格式——条件码

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

指令格式——数据处理指令

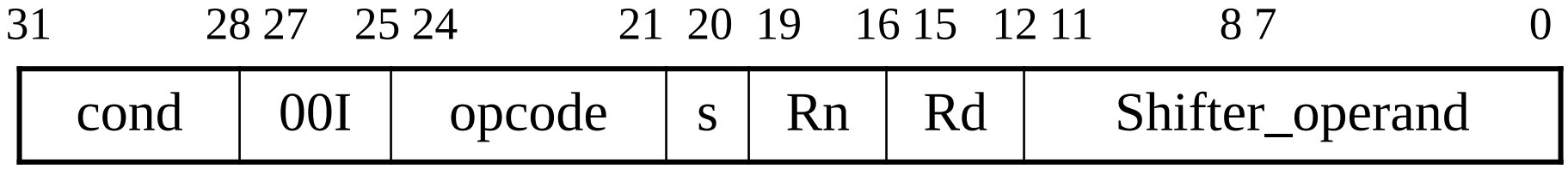


- Cond：指令执行的条件编码
- Opcode：指令操作符编码
- I: 指明Shifter_operand是立即数还是寄存器
- S：决定指令的操作是否影响CPSR的值
- Rd：操作目标寄存器编码
- Rn：包含第一操作数的寄存器编码
- Shifter_operand：表示第二操作数

指令格式——例子（1）

- **MOV R1, R0　　E1A01000**

1110 000 1101 0 0000 0001 000000000000

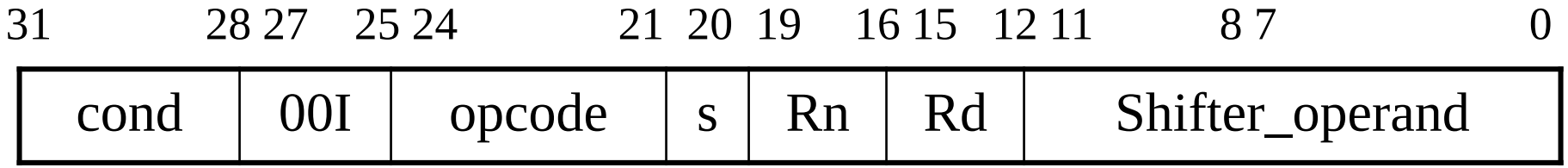


- **Cond： 1110**
- **Opcode： 1101**
- **I: 0**
- **S： 0**
- **Rd： 0001**
- **Rn： 总是0**
- **Shifter_operand： 0000 0000 0000**

指令格式——例子（2）

- **MOV R1, #0x34　　E3A01034**

1110 001 1101 0 0000 0001 000000110100

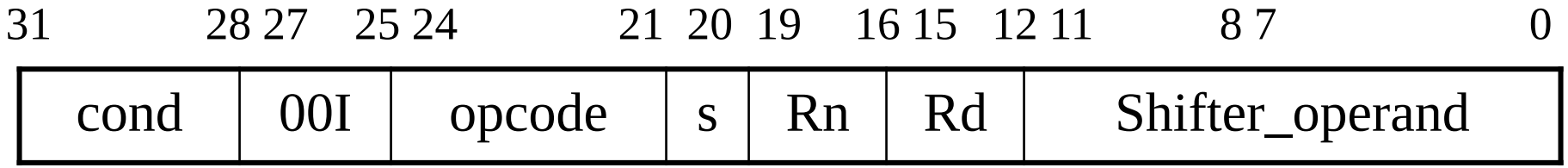


- **Cond： 1110**
- **Opcode： 1101**
- **I: 1**
- **S： 0**
- **Rd： 0001**
- **Rn： 总是0**
- **Shifter_operand： 0000 0011 0100**

指令格式——例子（3）

- **ADDGTS R1, R0, R2　　C0901002**

1100 000 0100 1 0000 0001 000000000010



- **Cond： 1100**
- **Opcode：0100**
- **I: 0**
- **S：1**
- **Rd：0001**
- **Rn：0000**
- **Shifter_operand：0000 0000 0010**

ARM寻址方式

- 寄存器寻址
- 立即数寻址
- 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
- 块拷贝寻址
- 相对寻址

ARM寻址方式——寄存器寻址

- **MOV R1, R2 ;R2->R1**
- **SUB R0, R1,R2 ;R1-R2 -> R0**

ARM寻址方式——立即数寻址

- **SUBS R0,R0,#1　　;R0-1 -> R0**

- **MOV　R1,#0xFF00　　;0xFF00 -> R0**

编译结果　E3A01CFF，二进制表示如下：

1110 001 1101 0 0000 0001 110011111111

第二操作数分成两个部分，高4位为移位次数/2，低8位为基数

#0x1FF00？

ARM寻址方式——寄存器偏移寻址

- ADD R1, R1, R2 , ROR #0x2
  ; R2循环右移两位后与R1相加，结果放入R1

- MOV R1, R0, LSL R2
  ; R0逻辑左移R2位后放入R1中

ARM寻址方式——寄存器间接寻址

- STR R1, [R2]

  ; 将R1的值存入以R2内容为地址的存储器中

- SWP R1, R1, [R2]

  ; 交换以R2为地址的存储器内容和R1内容

ARM寻址方式——基址变址寻址

- op Rd, [Rn, R1]
  ；零偏移，Rn+R1是操作数地址
- op Rd, [Rn, FlexOffset]
  ；前索引偏移，Rn+FlexOffset是地址
- op Rd, [Rn, FlexOffset]!
  ；带写回的前索引偏移，地址存入Rn寄存器
- op Rd, [Rn], FlexOffset
  ；后索引偏移，Rn是地址，Rn+FlexOffset写入Rn

ARM寻址方式——多寄存器寻址

- LDMIA R0, {R1, R2, R3, R4, R5}

  ; R1<- R0, R2<- R0+4, …, R5<-R0+16

- STMIA R0, {R2-R5, R7}

  ; R0<-R2, R0+4<-R3, … , R0+12<-R5, R0+16<-R7

ARM寻址方式——堆栈寻址

- STMFD SP!, {R1-R7, LR}

  ；将R1-R7，LR存放到堆栈中，这条指令一般用来保护现场

ARM寻址方式——相对寻址

- BL Label

  ；转跳到Label标签处

ARM指令分类（1）

- 跳转指令
- 通用数据处理指令
- 乘法指令
- Load/Store访存指令

ARM指令分类（2）

- ARM协处理器指令
- 杂项指令
- 饱和算术指令
- ARM伪指令

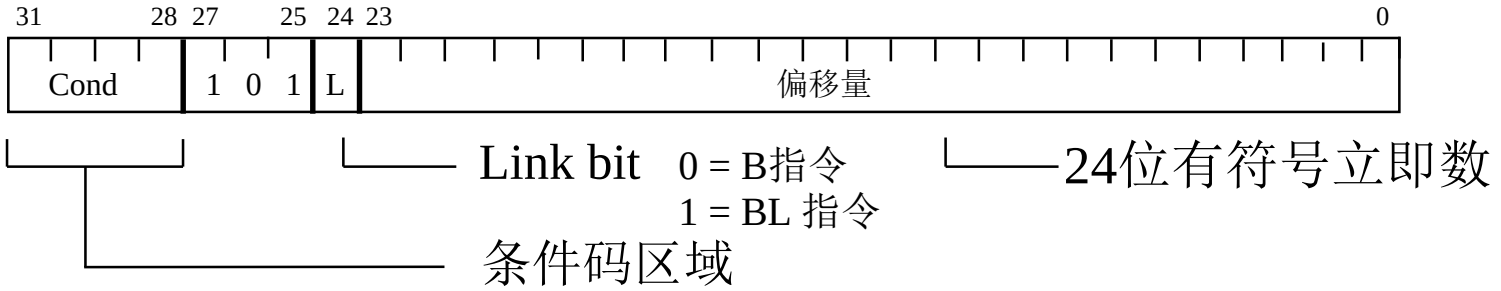
Thumb指令分类

- Thumb跳转指令
- Thumb通用数据处理指令
- Thumb算术指令
- Thumb内存访问指令
- Thumb软中断和断电指令
- Thumb伪指令

分支跳转指令

- Branch : B{<cond>} label
- Branch with Link :

  BL{<cond>} subroutine_label



通常与以下指令一起使用

```
CMP  R0, R1
BNE  LOOP
```

数据处理指令

- 数学运算: ADD ADC SUB SBC RSB RSC
- 逻辑运算: AND ORR EOR BIC
- 比较: CMP CMN TST TEQ
- 数据搬移: MOV MVN

伪指令

- 数据定义伪指令
- 符号定义伪指令
- 过程定义伪指令
- 汇编控制伪指令
- 其他伪指令

常用伪指令

- AREA：用于定义一个代码段或数据段
- CODE16 ： 通知编译器其后的指令为16位的Thumb指令
- CODE32： 通知编译器其后的指令为32位的ARM指令
- ENTRY：用于指定汇编程序的入口点。
- END：用于通知编译器已经到了源程序的结尾。
- EXPORT：用于在程序中声明一个全局的标号，该标号可在其他的文件中引用。EXPORT可用GLOBAL代替。
- IMPORT：用于通知编译器要使用的标号在其他的源文件中定义，但要在当前源文件中引用

常用汇编指令介绍—指令后缀

- ARM中的指令可以带后缀，从而丰富该指令的功能，这种形式叫做指令族，常用的后缀有：
  - B(byte)：功能不变，操作长度变为8位(依赖CPU位数，以下相同)
  - H(Halfword)：功能不变，操作长度变为16位
  - S(signed)：功能不变，操作数变为有符号数
  - S(S标识)：影响CPSR里的NZCV标识位
- 举例
  - ldr指令族：ldrb,ldrh,ldrsb ldrsh，从内存中加载指定长度的数据
  - mov指令族：movs r0,#0，结果是0，赋值会影响CPSR的NZCV标识，将Z位置为1

常用汇编指令介绍—CPU状态

- 有四个条件代码
  - （1）N：正负，N=1,结果为负，N=0，结果为正或0
  - （2）Z：零，Z=1结果为0，Z=0，结果非零
  - （3）C：进位，加法产生进位则C=1，不然为C=0；借位，减法运算产生了借位则C=0，否则C=1
  - （4）V：溢出，V=1，有溢出，V=0，无溢出

常用汇编指令介绍—数据传输指令

- mov：move，在两个寄存器之间或者立即数和寄存器之间传递数据，将后一个寄存器上的值或者立即数赋值给前一个寄存器，例如：
  - mov r1,r0
  - mov r1,#0xFF：将立即数0xFF赋值给寄存器r1
- mvn：和mov用法一致，区别是mvn会把后一个寄存器的值或者立即数按位取反后赋值给前一个寄存器，例如：
  - mvn r0,#0xFF，则r0的值为0xffffff00(32位数据)

常用汇编指令介绍—算术指令

- add：加法运算
- sub：减法运算
- rsb：逆向减运算
- adc：带进位的加法运算（ADC指令用于把两个操作数相加，再加上CPSR中的C条件标志位的值）
- sbc：带进位的减法运算
- rsc：带进位的反减指令

常用汇编指令介绍—逻辑指令

- and：与操作
- orr：或操作
- eor：异或操作
- bic：位清除操作
  - 例如：BIC R0，R0，#%1011；该指令清除 R0 中的位 0、1、和3，其余的位保持不变。

常用汇编指令介绍—比较指令

- cmp：比较大小
- cmn：取反比较
- tst：按位与运算
- teq：按位异或运算

常用汇编指令介绍—乘法指令

- mul：32位乘法
- mla：32位乘加
- umull：64位无符号乘法
- umlal：64位无符号乘加
- smull：64位有符号乘法
- smlal：64位有符号乘加

常用汇编指令介绍—前导0计数和CPSR访问

- 前导0计数指令
  - clz：统计一个数的二进制位前面有几个0
- CPSR访问指令
  - mrs：用于读取CPSR和SPSR
  - msr：用于写CPSR和SPSR

常用汇编指令介绍—跳转分支指令

- b指令：跳转
- bl指令：跳转，保存返回地址
- bx指令：跳转，切换到ARM/Thumb模式
- blx指令：跳转，保存返回地址，切换到ARM/Thumb模式

常用汇编指令介绍—内存访问指令

- ldr：加载指定内存地址的数据到寄存器，按照字节访问
- str：加载指定寄存器数据到内存地址中，按照字节访问
- ldm：和ldr功能一样，一次多字节多寄存器访问
- stm：和str功能一样，一次多字节多寄存器访问
- swp：内存和寄存器互换指令，一边读一边写，
  - 例如：swp r1,r2,[r0]：读取指针r0的数据到r1中，同时把r2的数据赋值给r0指针指向的变量

常用汇编指令介绍—ldm和stm

- stmia sp {r0 - r12}：stm表示进行批量数据操作，ia的意思是将r0存入SP的内存地址处，然后SP内存地址+4(32位)，将r1存入该地址，内存地址再+4，存入r2，依次存到r12
- **不同功能的后缀：**
  - **ia：increase after，后增加，表示每个操作时，先传输数据，后增加内存地址，**
  - **ib：increase before，先增加，表示在每个操作时，先增加内存地址，再进行数据传输**
  - **da：decrease after：和ia一样，差别在于减少地址**
  - **db：decrease before：和ib一样，差别在于减少地址**
  - **fd：full decrease：满递减堆栈，查看栈的描述**
  - **ed：empty decrease：空递减堆栈**
  - **fa：满递增堆栈**
  - **ea：空递增堆栈**

常用汇编指令介绍—软中断和协处理器指令

- 软中断指令
  - swi(software interrupt)，在软件层模拟产生一个中断，这个中断会传送给CPU，常用于实现系统调用

- 协处理器指令（支持16个协处理器）
  - mrc：读取协处理器中的寄存器
  - mcr：向协处理器中的寄存器写数据

高级语言程序设计概念-可重入函数

- 可重入性（reentrant）

  如果某个函数可被多个任务并发调用而不会造成数据错误，则称该函数具有可重入性

  可重入函数可在任意时刻被中断，稍后继续运行时不会造成错误

- 不可重入性（non-reentrant）

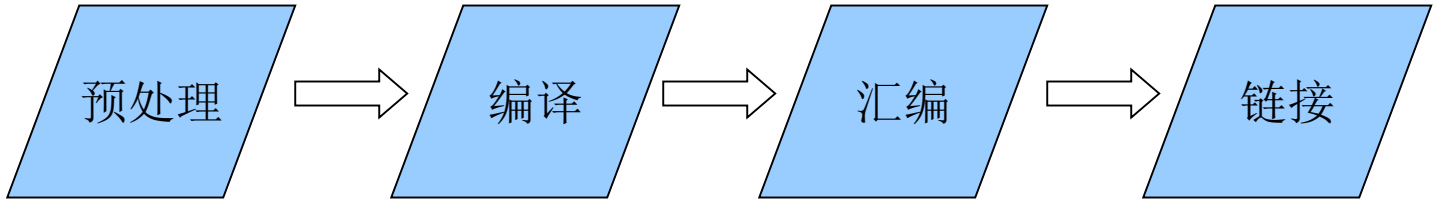
  不可重入函数不能被多个任务共享，除非采用信号量等机制确保函数的互斥调用，或者在代码的关键部分禁止中断

高级语言程序设计概念-中断及处理

- 嵌入式系统开发中，经常会用到中断：硬中断、软中断、异常等。
- 标准C中不包含中断服务的自动处理。
- 许多编译器厂商增加关键字对中断服务程序的支持。

嵌入式开发工程概念-编译过程

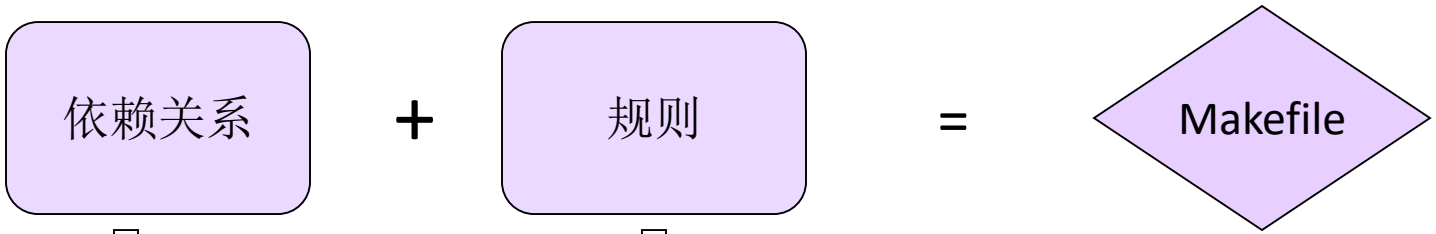
- 在使用 GCC 编译程序时，编译过程可以被细分为四个阶段：
  - 预处理（Pre-Processing）
  - 编译（Compiling）
  - 汇编（Assembling）
  - 链接（Linking）



- Linux 程序员可以根据自己的需要让 GCC 在编译的任何阶段结束，以便检查或使用编译器在该阶段的输出信息
- GCC 提供了 30 多条警告信息和三个警告级别，使用它们有助于增强程序的稳定性和可移植性。

Makefile概述

- make 命令对于构建具有多个源文件的程序有很大的帮助
- 只有 make 命令还是不够的，还必须用 Makefile 告诉它要做什么以及怎么做
- make 命令和 Makefile 配合使用，能给项目管理带来极大便利
- 一个 Makefile 由依赖关系和规则两部分内容组成



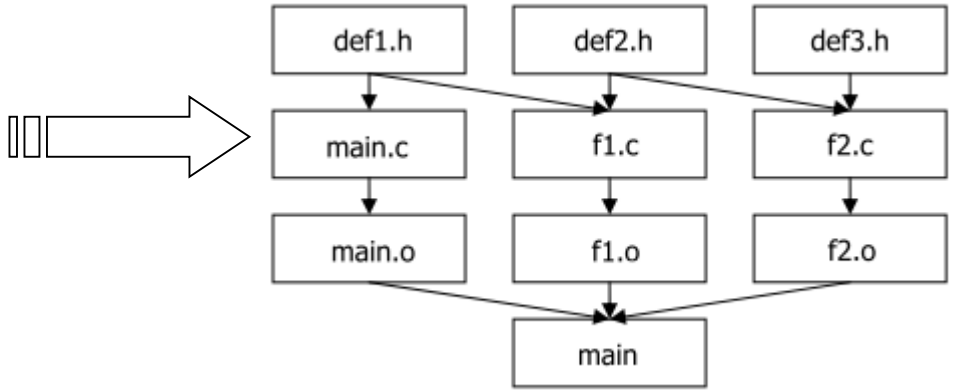
规则用来说明怎样使用所依赖的文件来建立目标文件

依赖关系由一个目标和一组该目标所依赖的源文件组成

Makefile 中的依赖关系

- make 程序自动生成和维护通常是可执行模块或应用程序的目标，目标的状态取决于它所依赖的那些模块的状态。
- make 的思想是为每一块模块都设置一个时间标记，然后根据时间标记和依赖关系来决定哪一些文件需要更新。一旦依赖模块的状态改变了，make 就会根据时间标记的新旧执行预先定义的一组命令来生成新的目标。

依赖关系规定了最终得到的应用程序跟生成它的各个源文件之间的关系



Makefile 中的规则

◆Makefile 规定相应的规则来描述如何生成目标，或者说使用哪些命令来根据依赖模块产生目标。

◆Makefile 是以相关行为基本单位的，相关行用来描述目标、模块及规则三者之间的关系。一个相关行格式通常为：冒号左边是目标名；冒号右边是目标所依赖的模块名；紧跟着的规则是由依赖模块产生目标所使用的命令。

相关行的格式为：

目标：[依赖模块][;命令]

习惯上写成多行形式

目标：[依赖模块]
命令
命令

Makefile

```
main.o: main.c
    gcc -c main.c
f1.o: f1.c
    gcc -c f1.c
f2.o: f2.c
    gcc -c f2.c

main: main.o f1.o f2.o
    gcc main.o f1.o f2.o –o main
```

嵌入式软件编程技术

- 嵌入式编程基本概念
- 嵌入式汇编编程技术
- 嵌入式高级编程技术
- 高级语言与低级语言混合编程

嵌入式汇编编程技术

- 基本语法
- 程序编译
- 程序例子

汇编语言程序-基本语法（1）

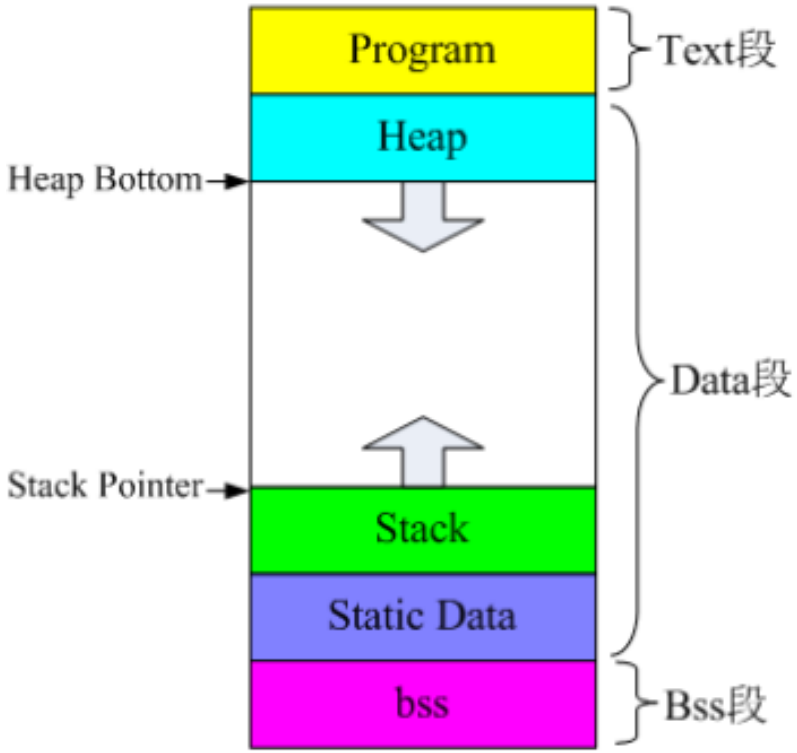
- ARM汇编语句格式
  - {label:}　　{instruction}　　{@comment}
  - {label:}　　{directive}　　{@comment}
  - {label:}　　{pseudo-instruction}　{@comment}

汇编语言程序-基本语法（2）

- 常用的预定义寄存器名称
  - R0-R15，通用寄存器
  - A1-A4，入口参数、处理结果、暂存，同R0-R3
  - V1-V8，变量寄存器，同R4-R11
  - IP，保存栈指针SP，同R12；SP，栈指针，同R13
  - LR，链接寄存器，同R14；PC，同R15
  - CPSR，当前程序状态寄存器
  - SPSR，程序状态备份寄存器
  - F0-F7，浮点运算加速寄存器
  - S0-S31，单精度浮点寄存器
  - D0-D15，双精度浮点寄存器

汇编语言程序-基本语法（3）

- 汇编程序段
  - text section
    代码段，只读
  - data section
    数据段，存放已初始化的全局变量、静态变量和常量等
  - bss section
    block started by symbol，存放未初始化全局和静态变量



汇编语言程序-基本语法（4）

- 汇编程序设计规范
  - 符号命名规则
  - 注释
  - 其他规则

汇编语言程序-编译

- 预处理
- 汇编器
- 链接器

汇编语言程序-例子：求一个数的阶乘

求一个数的阶乘

用ARM汇编语言实现20！，并将64位结果放在R9:R8 中（R9中存放高32位）

分析：64位结果的乘法指令通过两个32位的寄存器相乘，可以得到64位结果，在每次循环相乘中，可以将存放64位结果的两个32位寄存器分别与递增量相乘，最后将得到的高32位结果相加。

使用的寄存器说明：

R8:存放阶乘结果低32位

R9:存放阶乘结果高32位

R0:计数器

R1:计算过程中暂存R9（R9值使用前会被覆盖）

汇编语言程序-例子：求一个数的阶乘

```
.global _start
.text
_start:
        Mov R8, #20      @低32位初始化为20
        Mov R9,#0        @高32位初始化为0
        Sub R0,R8,#1     @初始化计数器
Loop:
        MOV R1,R9        @暂存高位值
        UMULL R8,R9,R0,R8        @[R9:R8]=R0*R8
        MLA R9,R1,R0,R9          @R9=R1*R0+R9
        SUBS R0, R0, #1          @计数器递减
        BNE Loop         @计数器不为0时继续循环
.Stop:
        B Stop
.end                     @文件结束
```

R8=0x82B40000

R9=0x21C3677C

嵌入式软件编程技术

- 嵌入式编程基本概念
- 嵌入式汇编编程技术
- 嵌入式高级编程技术
- 高级语言与低级语言混合编程

函数可重入问题（1）

```
static int tmp;
void swap(int *a, int *b)
{
   tmp=*a;
   *a = *b;
   *b = tmp;
}
```

全局变量 -> 局部变量

```
void task1(void)
{
   …
   swap(a, b);
}

void task2(void)
{
   …
   swap(c,d);
}
```

函数可重入问题（2）

```
static int tmp;
void swap(int *a, int *b)
{
   [申请信号量操作]
   tmp=*a;
   *a = *b;
   *b = tmp;
   [释放信号量操作]
}
```

```
void task1(void)
{
   …
   swap(a, b);
}

void task2(void)
{
   …
   swap(c,d);
}
```

函数可重入问题（3）

```
static int tmp;
void swap(int *a, int *b)
{
   Disalble_IRQ();
   tmp=*a;
   *a = *b;
   *b = tmp;
   Enable_IRQ()
}
```

```
void task1(void)
{
   …
   swap(a, b);
}

void task2(void)
{
   …
   swap(c,d);
}
```

函数可重入问题（3）

```
static int tmp;
void swap(int *a, int *b)
{
   Disalble_IRQ();
   tmp=*a;
   *a = *b;
   *b = tmp;
   Enable_IRQ()
}
```

```
void task1(void)
{
   …
   swap(a, b);
}

void task2(void)
{
   …
   swap(c,d);
}
```

中断处理过程

- 硬件部分
  - 复制CPSR到SPSR_<mode>
  - 设置正确的CPSR位
  - 切换到<mode>
  - 保存返回地址到LR_<mode>
  - 设置PC跳转到相应的异常向量表入口

中断处理过程

- 软件部分
  - 把SPSR和LR压栈
  - 把中断服务程序的寄存器压栈
  - 开中断，允许嵌套中断
  - 中断服务程序执行完后，恢复寄存器
  - 弹出SPSR和PC，恢复执行

中断处理程序和汇编

```
__irq void IRQHandler(void)
{
    volatile unsigned int *source =
(unsigned int*)0x80000000;
    if(*source ==1)
        int_hander_1();
    *source = 0;
}
```

```
STMFD SP!, {r0-r4,r12,lr}
MOV r4, #0x80000000
LDR r0, [r4,#0]
CMP r0, #1
BLEQ int_hander_1
MOV r0, #0
STR r0, [r4,#0]
LDMFD SP!,{r0-r4,r12,lr}
```

嵌入式软件编程技术

- 嵌入式编程基本概念
- 嵌入式汇编编程技术
- 嵌入式高级编程技术
- 高级语言与低级语言混合编程

高级语言与低级语言混合编程

- 若汇编代码较为简洁，则可使用直接内嵌汇编的方法
- 否则要将汇编程序以文件的形式加入到项目中，按照ATPCS(ARM/Thumb过程调用标准，ARM/Thumb Procedure Call Standard)的规定与C程序相互调用与访问

一些规则

堆栈使用规则

采用满递减类型(FD,Full Descending)，即堆栈通过减小存储器地址而向下增长

参数传递规则

整数参数的前4个使用R0-R3传递，其他参数使用堆栈传递；浮点参数使用编号最小且能够满足需要的一组连续的FP寄存器传递

汇编程序调用C程序的方法

- 汇编程序编写要遵循ATPCS规则，以保证程序调用时参数正确传递
- 首先在汇编程序中使用IMPORT伪指令事先声明将要调用的C语言函数
- 然后通过BL指令来调用C函数

例子

```
int add(int x,int y)
{
   return(x+y);
}
```

```
IMPORT add;声明要调用的C函数
……
MOV r0, 1
MOV r1, 2
BL add          ;调用C函数add
```

注：使用r0和r1实现参数传递，返回结果由r0带回

C程序调用汇编程序

- 汇编程序编写也要遵循ATPCS规则，以保证程序调用时参数正确传递
- 首先在汇编程序中使用EXPORT伪指令声明被调用的子程序，表示该子程序将在其他文件中被调用
- 然后在C程序中使用extern关键字声明要调用的汇编子程序为外部函数

例子

```
EXPORT add ;声明add子程序将被外部函数调用
 ……
 add ;求和子程序add
 ADD r0,r0,r1
 MOV pc,lr


extern int add (int x,int y); //声明add为外部函数
void main()
{
   int a=1,b=2,c;
   c=add(a,b); //调用add子程序
   ……
 }
```

C程序中内嵌汇编语句

- 在C语言中内嵌汇编语句可以实现一些高级语言不能实现或者不容易实现的功能
- 对于时间紧迫的功能也可以通过在C语言中内嵌汇编语句来实现
- 内嵌的汇编器支持大部分ARM指令和Thumb指令，但不支持底层功能

例子

```
__asm __ (“instruction
            …
            instruction”);   //Linux gcc中支持
```

```
asm(
  汇编语句模板:
  输出部分:
  输入部分:
  修改部分)
```

```
asm("mov %0, %1, ror #1" : "=r" (result) : "r" (value));
```

例子2

```
#include <stdio.h>
int main(void)
{
    int result ,value;
    value=1;
    printf("old value is %x",value);
    __asm("mov %0,%1,ror #1": "=r"(result):"r"(value));
    printf("new result is %x\n",result)  ;

    return 1;
}
```

□谢　谢！