

第4章 ARM汇编程序设计

李灿 | 12#407A | lic@jsnu.edu.cn | https://sslic.cn/es

章节内容

- ARM汇编语言语句格式
- ARM汇编伪操作
- ARM汇编语言伪指令
- ARM汇编语言中的符号
- ARM汇编语言中的表达式
- ARM汇编语言程序结构
- 汇编语言子程序调用
- C语言与汇编语言混合编程
- ARM汇编语言设计实例

章节内容

- ARM汇编语言语句格式
- ARM汇编伪操作
- ARM汇编语言伪指令
- ARM汇编语言中的符号
- ARM汇编语言中的表达式
- ARM汇编语言程序结构
- 汇编语言子程序调用
- C语言与汇编语言混合编程
- ARM汇编语言设计实例

4.1 ARM汇编语言语句格式

- ARM/Thumb汇编源程序中的语句由**指令**、**伪操作**、**伪指令**组成

语句书写格式
{label} 指令 or 伪操作 or 伪指令 {; comments}

- ✧ { } 可选项
- ✧ Label 程序标号，代表该语句的地址
- ✧ comments 程序注释，便于程序的调试和理解

- **注意：**

① 程序标号要顶格写，无论有无程序标号，语句前必须有缩进

② 每一条指令的助记符可以全部大写/小写，但不允许大小写混用

③ 行的末尾用“\”（backslash）表示下一行与本行为同一条语句，且“\”后面不能加任何其他符号，包括空格和制表符

章节内容

- ARM汇编语言语句格式
- **ARM汇编伪操作**
- ARM汇编语言伪指令
- ARM汇编语言中的符号
- ARM汇编语言中的表达式
- ARM汇编语言程序结构
- 汇编语言子程序调用
- C语言与汇编语言混合编程
- ARM汇编语言设计实例

4.2 ARM汇编伪操作

- 伪操作并不像机器指令那样在CPU运行时执行，而是汇编程序在对源程序汇编期间进行处理
- 伪操作包括：

① 符号定义伪操作 Symbol Definition Directives
② 数据定义伪操作 Data Definition Directives
③ 汇编控制伪操作 Assembly Control Directives
④ 数据帧描述伪操作 Frame Description Directives
⑤ 信息报告伪操作 Report Directives
⑥ 其他杂项伪操作 Miscellaneous Directives

4.2.1 符号定义伪操作

- **作用：定义ARM汇编程序中的变量，对变量赋值以及为寄存器定义别名等**

伪操作	功能
GBLA | GBLL | GBLS	声明一个全局变量
LCLA | LCLL | LCLS	声明一个局部变量
SETA | SETL | SETS	为一个变量赋值
RLIST	为一个通用寄存器列表定义名称
CN	为一个协处理器的寄存器定义名称
CP	为一个协处理器定义名称
DN | SN	为一个VFP寄存器定义名称
FN	为一个FP浮点寄存器定义名称

4.2.1 符号定义伪操作

① GBLA | GBLL | GBLS

伪操作	功能
GBLA	声明一个全局算术变量并初始化为零
GBLL	声明一个全局逻辑变量并初始化为逻辑假，即{FALSE}
GBLS	声明一个全局字符串变量并初始化为空字符串，即“ ”

语法	<GBLx> variable

- 作用范围为包含该变量的整个源文件，且在该作用范围内其必须唯一，已经声明过的变量不能重新声明为其他类型
- 使用示例：

```
GBLA    A           ; 定义一个全局的数字变量，变量名为A
MOV     R1, #A      ; 将此变量值赋给R1，执行后R1的值将变为零
```

4.2.1 符号定义伪操作

② LCLA | LCLL | LCLS

伪操作	功能
LCLA	声明一个局部算术变量并初始化为零
LCLL	声明一个局部逻辑变量并初始化为逻辑假，即 {FALSE}
LCLS	声明一个局部字符串变量并初始化为空字符串，即“ ”

语法	<LCLx> variable

- 作用范围为包含该变量的宏代码的一个实例源文件，且在该作用范围内其必须唯一，已经声明过的变量不能重新声明为其他类型
- 使用示例（宏内声明）：

```
MACRO          ; 声明一个宏 ……
LCLS   err     ; 声明一个局部串变量err ……
```

4.2.1 符号定义伪操作

③ SETA | SETL | SETS

伪操作	功能
SETA	为一个算术变量赋值
SETL	为一个逻辑变量赋值，{TRUE}或者{FALSE}
SETS	为一个字符串变量赋值

语法	variable <SETx> expression

- 变量赋值之前必须用前面的伪操作进行声明
  或者在开发环境的命令行中定义
- ARM汇编器中定义了一些内置变量，它们不能用伪操作设置

4.2.1 符号定义伪操作

③ SETA | SETL | SETS

语法	variable <SETx> expression

✧ 使用示例：

```
        GBLA   A                ; 声明一个全局算术变量A
A       SETA   0xFF             ; 将变量A赋值为0xFF

        LCLL   L                ; 声明一个局部逻辑变量L
L       SETL   {TRUE}           ; 将变量L赋值为"真"

        GBLS   INST             ; 声明一个全局字符串变量INST
INST    SETS   "MOV R0, #0"     ; 将变量赋值为为"MOV R0,#0"
        $INST                   ; 此语句将被汇编成MOV R0,#0代码
```

表4.1 内置变量表

内置变量	含义
{PC} 或 .	当前指令的地址
{VAR} 或 @	存储区位置计数器的当前值
{TRUE}	逻辑真
{FALSE}	逻辑假
{OPT}	当前设置列表选项值，OPT用来保存当前列表选项 改变选项值，恢复它的原始值
{CONFIG}	如果汇编器汇编ARM代码，则值为32 如果汇编器汇编Thumb代码，则值为16
{ENDIAN}	如果汇编器在大端模式下，则值为big 如果汇编器在小端模式下，则值为little
{CODESIZE}	如果汇编器汇编ARM代码，则值为32 如果汇编器汇编Thumb代码，则值为16，与{CONFIG}同义
{CPU}	选定的CPU名，缺省时为ARMTDMI
{FPU}	选定的FPU名，缺省时为SoftVFP
{ARCHITECTURE}	STR pc，[…]或STM Rb，[…PC]指令的 地址和PC存储值之间的偏移量
{PCSTOREOFFSET}	ARM汇编器的版本号，为整数 同：| ads $ version	

4.2.1 符号定义伪操作

④ RLIST

伪操作	功能
RLIST	为一个通用寄存器列表定义名称

语法	name RLIST {registers list}

- name：寄存器列表的名称，不能与汇编器预定义的寄存器和协处理器的名称相冲突（预定义的寄存器均大小写敏感）
- 寄存器列表访问次序根据寄存器编号由低到高
  与列表中的排列顺序无关
- 使用示例：

```
RegList  RLIST  {R0-R5,R8,R10}
        ; 寄存器列表名称定义为RegList
        ; 可在ARM指令LDM/STM中通过该名称访问寄存器列表
```

表4.2 ARM汇编器预定义寄存器和协处理器名称

类别	预定义寄存器/协处理器，包括
通用寄存器	R0~R15和r0~r15 (16个通用寄存器)
	a1~a4(参数、结果或临时寄存器，与r0~r3同义)
	v1~v8(变量寄存器，与r4~r11同义)
	sb和SB(静态基址寄存器，与r9同义)
	sl和SL(堆栈限制寄存器，与r10同义)
	fp和FP(帧指针，与r11同义)
	ip和IP(过程调用中的临时寄存器，与r12同义)
	sp和SP(堆栈指针，与r13同义)
	lr和LR(链接寄存器，与r14同义)
	pc和PC(程序计数器，与r15同义)
程序状态寄存器	cpsr和CPSR，spsr和SPSR
浮点数寄存器	f0~f7和F0~F7(FPA寄存器)
	s0~s31和S0~S31(VFP单精度寄存器)
	d0~d15和D0~D15(VFP双精度寄存器)
协处理器及其寄存器	p0~p15(协处理器0~15)
	c0~c15(协处理器寄存器0~15)

4.2.1 符号定义伪操作

⑤ CN

伪操作	功能
CN	为一个协处理器的寄存器定义名称

语法	name CN expr

- name：寄存器的名称，不能与汇编器预定义的寄存器和协处理器的名称相冲突（预定义的寄存器均大小写敏感）
- expr：协处理器的寄存器编号，数值范围为0~15
  预定义的协处理器寄存器名称为c0~c15
- 使用示例：

```
power   CN   6          ; 为协处理器的寄存器6定义名称为power
```

4.2.1 符号定义伪操作

⑥ CP

伪操作	功能
CP	为一个协处理器定义名称

语法	name CP expr

- name：协处理器的名称，不能与汇编器预定义的寄存器和协处理器的名称相冲突（预定义的寄存器均大小写敏感）
- expr：协处理器的编号，数值范围为0~15
  预定义的协处理器名称为p0~p15
- 使用示例：

```
dmu   CP   6   ; 将协处理器6名称定义为dmu
```

4.2.1 符号定义伪操作

⑦ DN | SN

伪操作	功能
DN	为一个VFP双精度寄存器定义名称
SN	为一个VFP单精度寄存器定义名称

语法	name DN expr name SN expr

- name：寄存器的名称，不能与汇编器预定义的寄存器和协处理器的名称相冲突（预定义的寄存器均大小写敏感）
- expr：单精度VFP寄存器编号0~31或双精度的编号0~15
- 使用示例：

```
energy   DN   6    ; 将双精度VFP寄存器6名称定义为energy
mass     SN   16   ; 将单精度VFP寄存器16名称定义为mass
```

4.2.1 符号定义伪操作

⑧ FN

伪操作	功能
FN	为一个浮点FPA寄存器定义名称

语法	name FN expr

- name：寄存器的名称，不能与汇编器预定义的寄存器和协处理器的名称相冲突（预定义的寄存器均大小写敏感）
- expr：浮点FPA寄存器的编号，数值范围为0~7
- 使用示例：

```
energy   FN   6      ; 将浮点FPA寄存器6名称定义为energy
```

4.2.2 数据定义伪操作

- **作用：一般用于为特定的数据分配存储单元，如数据缓冲池定义、数据表定义、数据空间分配等，同时可完成已分配存储单元的初始化**

4.2.2 数据定义伪操作

伪操作	功能
LTORG	声明一个数据（文字）缓冲池（literal pool）的开始
MAP	定义一个结构化的内存表的首地址
FIELD	定义结构化的内存表中的一个数据域
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 | 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 并将每个单元的内容初始化成该单元相对于静态基址寄存器R9的偏移量
DCFD | 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 | 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	分配一段字的内存单元，用指定的数据初始化 并标识内存单元中存放的是指令，而不是数据
DCQ | 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 | 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，该伪代码会被编译器忽略

4.2.2 数据定义伪操作

① LTORG

伪操作	功能
LTORG	指示汇编器在该指令所在处立即建立数据缓冲池（文字池）

语法	LTORG

- ARM汇编器把默认数据缓冲池位置放在代码段的最后面，即下一个代码段开始之前或者END伪操作之前，但有时这个位置可能超出LDR等伪指令的作用范围（4KB），此时可以LTORG伪操作解决
- 比较大的程序可以放置多个LTORG伪操作，通常放在无条件跳转指令或子程序返回指令之后，以避免处理器错误地将缓冲池中的常量当作指令执行

4.2.2 数据定义伪操作

① LTORG

✧ 使用示例：

```
        ARER    Example, CODE, READONLY
start   BL      func
        B       .
func    LDR     r0, = 0x12345678 ; 将r0赋值为0x12345678
        MOV     pc, lr
        LTORG                    ; 建立包含0x12345678的数据缓冲池
data    SPACE   5000             ; 开辟一段长度大于4K的空间
                                 ; 如果没有上面的LTORG伪操作
                                 ; 汇编器将在此处建立数据缓冲池
                                 ; 与LDR伪指令位置间的距离超过4K
                                 ; 编译时将报错
        END
```

4.2.2 数据定义伪操作

② MAP

伪操作	功能
MAP	定义一个结构化的内存表的首地址 此时，内存表的位置计数器{VAR}设置为该地址值，MAP可用“^”代替

语法	MAP expr{, base-register}

✧ expr：数字表达式或者是程序中的标号

- 不使用base-register时，内存表首地址为expr
  此时，内存表的位置计数器{VAR}设置成该值
- 当使用base-register时，内存表首地址为expr与base-register的值之和

✧ 使用示例：

```
MAP    0x10, R9      ; 内存表首地址 = 0x10 + R9的值
```

4.2.2 数据定义伪操作

③ FIELD

伪操作	功能
FIELD	定义一个结构化内存表中的数据域 FIELD可用“#”代替，每个数据域可指定一个标号，供其他指令引用

语法	{label} FIELD expr

- {label}：可选项，表示当前内存表位置计数器{VAR}的值
- expr：该数据域的字节大小
- 汇编器处理完该条语句后，{VAR}的值将增加expr
- 使用说明：用MAP和FIELD伪操作结合使用（首地址+数据域）来描述一个内存结构表（并不能实际分配内存，不同结构表可以描述同一块内存），可多次使用MAP来建立赋值的内存结构表

4.2.2 数据定义伪操作

③ FIELD

✧ 使用示例（MAP+FIELD）：

```
        MAP    0        ; 定义一个内存结构表，首地址为0
a       FIELD  4        ; 定义一个4字节数据域，并命名为a
b       FIELD  4        ; 定义一个4字节数据域，并命名为b
        MAP    0, r9    ; 定义一个内存结构表，首地址为r9的值
        FIELD  4        ; 定义一个4字节数据域，但未命名
x       FIELD  4        ; 定义一个4字节数据域，并命名为x
y       FIELD  4        ; 定义一个4字节数据域，并命名为y
        ...
        LDR    r9, = sdat
        LDR    r0, [r9, #b]  ; 相当于"LDR r0, [r9, #4]"
        STR    r0, y         ; 相当于"STR r0, [r9, #8]"
        ...
        AREA   dat, DATA, READWRITE
sdat    SPACE  16
```

4.2.2 数据定义伪操作

④ SPACE

伪操作	功能
SPACE	分配一片连续的存储区域并初始化为0 SPACE也可用“%”代替

语法	{label} SPACE expr

- {label}：可选项，标号
- expr：要分配的字节数
- 使用示例：

```
DataSpace   SPACE   100
                    ;分配连续100字节的存储单元并初始化为0
```

4.2.2 数据定义伪操作

⑤ DCB

伪操作	功能
DCB	分配一段连续的字节存储单元并用伪操作中指定的表达式初始化 DCB也可用“=”代替

语法	{label} DCB expr{, expr}{, expr}...

- {label}：可选项，标号
- expr：可为0～255的数字或字符串，可列一个或多个
- 使用示例：

```
String  DCB "string", 0   ; 定义一个以NULL结尾的字符串
```

4.2.2 数据定义伪操作

⑥ DCD | DCDU

伪操作	功能
DCD{U}	分配一段连续的字存储单元并用伪操作中指定的表达式初始化 DCD也可用“&”代替，DCD分配的字存储单元是字对齐的，DCDU则任意

语法	{label} DCD{U} expr{, expr}{, expr}...

- {label}：可选项，标号
- expr：程序标号或数字表达式，可列一个或多个
- **使用说明**：DCD伪操作定义第一个内存单元时，可能在之前填充0~3的零值字节（padding）以确保字对齐，DCDU则无此功能
- 使用示例：

```
Data   DCD   4, 5, 6 ; 分配一段连续字单元，初始化为4,5,6
```

4.2.2 数据定义伪操作

⑦ DCDO

伪操作	功能
DCDO	分配一段连续的字存储单元并将每个单元的内容初始化成该单元相对于静态基址寄存器SB(R9)的偏移量

语法	{label} DCDO expr{, expr}{, expr}...

- {label}：可选项，标号
- expr：基于静态基址寄存器的表达式或程序标号，一个或多个
- 使用示例：

```
IMPORT   symbol ; 引入一个外部标号，参见IMPORT伪操作
DCDO     symbol ; 分配一个字，值为symbol相对SB的偏移值
```

4.2.2 数据定义伪操作

⑧ DCFD | DCFDU

伪操作	功能
DCDF{U}	分配一段连续的双字的存储单元，并用双精度的浮点数初始化 DCFD分配的字存储单元是字对齐的，DCFDU则任意

语法	{label} DCFD{U} fpl{, fpl}{, fpl}...

- {label}：可选项，标号
- fpl：双精度浮点数
- 使用说明：DCFD伪操作定义第一个内存单元时，可能在之前填充0~3的零值字节以确保字对齐，DCDFU无此功能。fpl转换成内存单元的表示形式由选择的浮点运算单元FPU决定。若汇编器命令行选项“-FPU”为NONE，则不能用上述伪操作。
- 使用示例（双精度数初始化）：

```
FDataTest   DCFD   2E11,-5E7   ;  分配一片连续的字存储单元并初始化
```

4.2.2 数据定义伪操作

⑨ DCFS | DCFSU

伪操作	功能
DCFS{U}	分配一段连续的双字的存储单元，并用单精度的浮点数初始化 DCFS分配的字存储单元是字对齐的，DCFSU则任意

语法	{label} DCFS{U} fpl{, fpl}{, fpl}...

- {label}：可选项，标号
- fpl：单精度浮点数
- 使用说明：DCFS伪操作定义第一个内存单元时，可能在之前填充0~3的零值字节以确保字对齐，DCDFS无此功能。fpl转换成内存单元的表示形式由选择的浮点运算单元FPU决定。若汇编器命令行选项“-FPU”为NONE，则不能用上述伪操作。
- 使用示例（单精度数初始化）：

```
FDataTest   DCFS   2E5,-5E-7  ;  分配一片连续的字存储单元并初始化
```

4.2.2 数据定义伪操作

⑩ DCI

伪操作	功能
DCI	ARM中，分配一段字的内存单元（字对齐），并初始化
	Thumb中，分配一段半字的内存单元（半字对齐），并初始化

语法	{label} DCI expr{, expr}{, expr}...

- {label}：可选项，标号
- expr：数字表达式
- **使用说明**：
  - DCI和DCD在内存分配上相似，区别在于DCI分配的内存单元的数据被标记为代码，可以用于通过宏来定义处理器不支持的指令
  - 在ARM中，使用DCI伪操作定义第一个内存单元时，可能在之前填充0~3的零值字节以确保字对齐，在Thumb中，DCI将确保半字对齐

4.2.2 数据定义伪操作

⑩ DCI

伪操作	功能
DCI	ARM中，分配一段字的内存单元（字对齐），并初始化
	Thumb中，分配一段半字的内存单元（半字对齐），并初始化

语法	{label} DCI expr{, expr}{, expr}...

- {label}：可选项，标号
- expr：数字表达式
- 使用示例：

```
; 该宏将一个新指令newinstr $Rd, $Rm定义称合适的机器指令
MACRO      symbol
newinstr $Rd, $Rm ; 分配一个字，值为symbol相对SB的
DCI        0xE16F0F10 :OR: ($Rd :SHL: 12) :OR: $Rm
MEND
```

4.2.2 数据定义伪操作

⑪ DCQ | DCQU

伪操作	功能
DCQ{U}	分配一段连续的双字内存单元，并用64位整数初始化 DCQ分配的存储单元是字对齐的，DCQU则任意

语法	{label} DCQ{U} {-}lit{, {-}lit}...

- {label}：可选项，标号
- lit：64位数字表达式，取值范围为0~(2^64)-1
  - -lit的取值范围为-(2^63)~(-1)
  - -n与(2^64)-n的意义相同
- 使用示例：

```
DataTest    DCQ    -255, 2_101 ; 2_101表示二进制101
```

4.2.2 数据定义伪操作

⑫ DCW | DCWU

伪操作	功能
DCW{U}	分配一段连续的半字内存单元，并用16位整数初始化 DCW分配的存储单元是半字对齐的，DCWU则任意

语法	{label} DCW{U} expr{, expr}{, expr}...

- {label}：可选项，标号
- expr：64位数字表达式，取值范围为-2^{15}~$2^{16}-1$
- 使用说明：DCW伪操作定义第一个内存单元时，可能在之前填充0~1个零值字节，以确保半字对齐
- 使用示例：

```
data   DCW    4, 5, 6
```

4.2.3 汇编控制伪操作

- **作用：控制汇编程序的执行流程**

类型	伪操作		
选择	IF	ELSE	ENDIF
循环	WHILE	WEND	
宏	MACRO	MEND	MEXIT

4.2.3 汇编控制伪操作

① 选择：IF, ELSE, ENDIF

语法

```
IF   logical_expression          ; 逻辑表达式
instructions or directives       ; 指令/伪操作序列
{                                ; {}内为可选项
ELSE
instructions or directives       ; 指令/伪操作序列
}
ENDIF
```

语法

```
[    logical_expression          ; 逻辑表达式
instructions or directives       ; 指令/伪操作序列
{                                ; {}内为可选项
|
instructions or directives       ; 指令/伪操作序列
}
]
```

4.2.3 汇编控制伪操作

① 选择：IF，ELSE，ENDIF

✧ 使用示例：

```
        GBLL NewVer          ; 定义一个变量NewVer
NewVer  SETL {FLASE}
        IF :DEF: NewVer      ; 如果定义了NewVer
            IF NewVersion = {TRUE}
            ...              ; 此处代码不会被汇编
            ELSE
            ...              ; 此处代码将被汇编
            ENDIF
        ENDIF
```

4.2.3 汇编控制伪操作

② 选择：WHILE, WEND

语法	WHILE logical_expression ; 逻辑表达式 instructions or directives ; 指令/伪操作序列 WEND

✧ 使用示例：

```
        GBLA   count            ; 定义一个变量count
count   SETA   1
        WHILE  count <= 4       ; 进入循环的条件
count   SETA   count+1
        ...                     ; 此处代码将会重复4次
        WEND
```

4.2.3 汇编控制伪操作

③ 宏：MACRO，MEDN，MEXIT

- ✧ MACRO,MEND：将一段代码定义为一个整体，称为宏指令
  然后就可以在程序中通过宏指令多次调用该段代码
- ✧ MEXIT：从宏定义中跳转出去

语法

```
            MACRO       ; 宏定义开始
{$label}    maconame {$par {, $par}...}
            instructions or directives
            MEND
```

$label	可选项，宏指令展开时，label将被替换成相应的符号 通常是一个标号
name	宏的名称
$par	宏指令的参数，宏指令展开时被替换成相应值，类似函数中的形参 可以在定义时为参数指定相应的默认值，形式如下： $par = default-value $par = “default value”

4.2.3 汇编控制伪操作

③ 宏：MACRO，MEDN，MEXIT

✧ 使用示例：

```
        MACRO
$label  TestAndBranch     $dest, $reg, $cc
$label  CMP               $reg,  #0
        B                 $cc    $dest
        MEND
test    TestAndBranch     NonZero, r0, NE ; 调用宏
        ...
NonZero
        ...
```

```
test    CMP     r0, #0                    ; 宏展开
        BNE     NonZero
        ...
NonZero
        ...
```

4.2.4 数据帧描述伪操作

- 作用：用于汇编程序调试

① FRAME ADDRESS

② FRAME POP

③ FRAME PUSH

④ FRAME REGISTER

⑤ FRAME RESTORE

⑥ FRAME SAVE

⑦ FRAME STATE REMEMBER

⑧ FRAME STATE RESTORE

⑨ FUNCTION或PROC

⑩ ENDFUNC或ENDP

4.2.5 信息报告伪操作

伪操作	功能
ASSERT	在编译过程中产生一个错误信息报告
INFO	在编译过程中产生一个诊断信息
OPT	设置列表选项
TTL | SUBT	列表中插入标题或子标题

4.2.5 信息报告伪操作

① ASSERT

伪操作	功能
ASSERT	保证源程序被汇编时满足的相关条件，如果条件不满足将在编译时报告错误信息

语法	ASSERT logical_expression

✧ 使用示例：

```
; 检查是否为老版本
ASSERT  OldVersion = {TRUE}

; 检查是否满足 (label1的地址) <= (label2的地址)
ASSERT  label1 <= label2
```

4.2.5 信息报告伪操作

② INFO

伪操作	功能
INFO	在编译过程中打印提示信息（诊断信息）

语法	INFO numeric-expr, string-expr

- numeric-expr：数字表达式
- string-expr：要打印的提示信息
- 当numeric-expr为零时，编译过程中将打印string-expr
  否则编译过程中报错并打印string-expr后退出编译
- **使用说明**：可用于自定义编译错误

4.2.5 信息报告伪操作

② INFO

伪操作	功能
INFO	在编译过程中打印提示信息（诊断信息）

语法	INFO numeric-expr, string-expr

- 当numeric-expr为零时，编译过程中将打印string-expr
  否则编译过程中报错并打印string-expr后退出编译
- 使用示例：

```
INFO   0, "version4.0"                    ; 无错提示

IF     endofdata <= label
INFO   1, "data overrun at label"         ; 编译错误提示
ENDIF
```

4.2.5 信息报告伪操作

③ OPT

伪操作	功能
OPT	在程序中修改编译器列表选项（-list）的默认配置

语法	OPT n

- n：OPT伪操作设置编号，合理的值参见表4.3
- 使用示例：

```
start ...
      BL    func1
      ...
      OPT   4       ; 在func1之前设置分页符
func1 ...
```

表4.3 OPT伪操作设置编号

编号 n	作用
1	设置常规列表选项
2	关闭常规列表选项
4	设置分页符，在新的一页开始显示
8	将行号重新设置为0
16	设置选项，显示SET、GBL、LCL伪操作
32	设置选项，不显示SET、GBL、LCL伪操作
64	设置选项，显示宏展开
128	设置选项，不显示宏展开
256	设置选项，显示宏调用
512	设置选项，不显示宏调用
1024	设置选项，显示第一遍扫描列表
2048	设置选项，不显示第一遍扫描列表
4096	设置选项，显示条件汇编伪操作
8192	设置选项，不显示条件汇编伪操作
16384	设置选项，显示MEND伪操作
32768	设置选项，不显示MEND伪操作

4.2.4 信息报告伪操作

④ TTL | SUBT

伪操作	功能
TTL	在列表文件的每一页的开头插入一个标题 作用范围是其后面的每一页，直到遇到新的TTL伪操作
SUBT	在列表文件的每一页的开头插入一个子标题 作用范围是其后面的每一页，直到遇到新的SUBT伪操作

语法	
	TTL title SUBT subtitle

4.2.6 其他杂项伪操作

- 18条：
  - AREA
  - ALIGN
  - CODE16 | CODE32
  - END
  - ENTRY
  - EQU
  - EXPORT | GLOBAL
  - EXPORTAS
  - IMPORT
  - EXTERN
  - GET | INCLUDE
  - INCBIN
  - KEEP
  - NOFP
  - REQUIRE
  - REQUIRE8 | PRESERVE8
  - RN
  - ROUT

4.2.6 其他杂项伪操作

① AREA

伪操作	功能
AREA	定义一个代码段或数据段

语法	AREA sectionname{, attr}{, attr}...

- sectionname：段名，若以数字开头，则该段名需用“|”括起来，如|1_test|，还有一些具有约定名称的段，如|.text|表示C编译器产生的代码段或与C语言库相关的代码段
- attr：该代码段/数据段的相关属性，多个属性用逗号分隔
- 常用属性见下表：

4.2.6 其他杂项伪操作

① AREA

属性	含义
ALIGN = expr	默认情况下ELF的代码段和数据段字对齐 expr取值为0~31，对齐方式为2^{expr}字节对齐，如expr=3时8字节对齐 此与ALIGN伪操作的定义不同，参见ALIGN伪操作
ASSOC = section	制定与本段相关的ELF段，任何包含section段的链接必须包含本段
CODE	定义代码段，默认为READONLY
DATA	定义数据段，默认为READWRITE
COMDEF	定义一个通用的段，该段可以包含代码或者数据 链接器为在各源文件中同名的COMDEF段分配相同的内存单元 同名COMDEF段内容必须相同。
COMMON	定义一个通用数据段，不能人为在段中定义代码或数据，链接器将该段初始化为零，链接器为在各源文件中同名的COMMON段分配共用的内存单元，并为其分配合适的足够大的尺寸
NOINIT	定义本数据段在汇编时不初始化或者初始化为零
READONLY	指定本段为只读，代码段默认为READONLY
READWRITE	指定本段为可读可写，数据段的默认属性为READWRITE

4.2.6 其他杂项伪操作

① AREA

伪操作	功能
AREA	定义一个代码段或数据段

语法	AREA sectionname{, attr}{, attr}...

✧ **使用说明：**

- 可以用AREA伪操作将程序分为多个ELF格式的段，且段名称可以相同，这时这些同名的段被放在同一个ELF段中
- 一个汇编语言程序至少要包含一个段，当程序太长时，也可以将程序分为多个代码段和数据段

✧ **使用示例：**

```
AREA   DatName, DATA, READONLY  ;  定义一个只读数据段
AREA   CodName, CODE, READONLY  ;  定义一个只读代码段
```

4.2.6 其他杂项伪操作

② ALIGN

伪操作	功能
ALIGN	通过添加填充字节的方式，使当前位置满足一定的对齐方式

语法	ALIGN {expr{, offset}}

- expr：指定对齐方式，取值为2的幂次，如1、2、4、8、16等
  若未指定表达式，则将当前位置对齐到下一个字的位置
- offset：偏移量，也是数字表达式，若使用该字段，则当前位置对齐到下一个具有 offset + n*expr 形式的地址

4.2.6 其他杂项伪操作

② ALIGN

语法	ALIGN {expr{, offset}}

✧ 使用示例：

```
                AREA   Example, CODE, READONLY
start           ...
                BL     subroutine
                ...
                B      {PC}
somedata        DCB    1
                ALIGN          ; 确保下面的代码字对齐
subroutine      ...
                MOV    pc,    lr
                END
```

4.2.6 其他杂项伪操作

③ CODE16 | CODE32

伪操作	功能
CODE16	告诉汇编器后面的指令序列为Thumb指令
CODE32	告诉汇编器后面的指令序列为ARM指令

语法	CODEx

✧ 使用说明：

- CODEx伪操作告诉汇编器后面的指令序列的类型，以在必要的情况下插入补丁字节（padding）保证ARM代码字对齐和Thumb代码半字对齐，而并不实际切换程序状态

4.2.6 其他杂项伪操作

③ CODE16 | CODE32

语法	CODEx

✧ 使用示例：

```
        AREA   ChangeState, CODE
        CODE32                  ; 指示下面为ARM代码
        LDR    r0, =start+1     ; 获得start地址并将第零位置1
        BX     r0               ; 跳转并切换程序状态
        CODE16                  ; 指示下面为Thumb代码
start   ...
```

4.2.6 其他杂项伪操作

④ END

伪操作	功能
END	通知编译器已经到了源程序的结尾

语法	END

✧ **使用说明**：任何汇编源文件必须以END单独一行作为结尾

✧ 使用示例：

```
        AREA   Example, CODE, READONLY
start   ...
        END    ; 这是源程序的结尾
```

4.2.6 其他杂项伪操作

⑤ ENTRY

伪操作	功能
ENTRY	指定汇编程序的入口点

语法	ENTRY

✧ **使用说明：**

- 在一个完整的汇编程序中至少要有一个ENTRY
- 当有多个ENTRY时，程序的真正入口点由链接器指定
- 在一个源文件里最多只能有一个ENTRY（可以没有）

✧ **使用示例：**

```
AREA   Vect, CODE, READONLY
ENTRY          ; 这是汇编程序的入口点
```

4.2.6 其他杂项伪操作

⑥ EQU

伪操作	功能
EQU	为数字常量、基于寄存器的值或程序中的标号定义一个名称 EQU可用“*”代替

语法	name EQU expr{, type}

- ✧ name：要定义的常量的名称
- ✧ expr：基于寄存器的地址值、程序中的标号
  32位绝对地址常量或32位整型常量
  - 当且仅当expr为32位绝对地址常量时，可用type指示expr表示的数据类型，如：CODE16，CODE32，DATA

4.2.6 其他杂项伪操作

⑥ EQU

伪操作	功能
EQU	为数字常量、基于寄存器的值或程序中的标号定义一个名称 EQU可用“*”代替

语法	name EQU expr{, type}

✧ **使用说明：**

- EQU的作用类似于C语言中的#define
- 用于为一个常量定义名称

✧ **使用示例：**

```
abc     EQU     2               ; 常量2定义为abc
xyz     EQU     label+8         ; label是程序中的标号
```

4.2.6 其他杂项伪操作

⑦ EXPORT | GLOBAL

伪操作	功能
EXPORT	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 参见IMPORT和EXTERN伪操作，EXPORT可用GLOBAL代替

语法	EXPORT {symbol}{[WEAK]}

- ✧ symbol：要声明的标号，大小写敏感，若省略则该文件全部标号均被声明为全局标号
- ✧ [WEAK]：声明其他的同名标号优先于该标号被引用
- ✧ 使用示例：

```
        AREA   Example, CODE, READONLY
        EXPORT   DoADD          ; 声明全局标号DoADD
DoADD   ADD      r0, r0, r1
```

4.2.6 其他杂项伪操作

⑧ EXPORTAS

伪操作	功能
EXPORTAS	在目标文件中创建一个符号，来替换源文件中与之对应的符号

语法	EXPORTAS symbol1, symbol2

- symbol1：源文件中的符号
- symbol2：目标文件(object file)中相应的符号
  取代目标文件中的symbol1
- **使用说明**：可以更改目标文件中的一个符号而不必更改源文件中所有与之对应的符号

```
two  EQU      2              ; 常量2定义为two
     EXPORTAS  two, one      ; 目标文件符号表中将出现one，值为2
     EXPORT    two           ; 源文件中符号two不变，值为2
```

4.2.6 其他杂项伪操作

⑨ IMPORT

伪操作	功能
IMPORT	通知编译器要使用的标号在其他的源文件中定义 无论当前源文件引用与否，该标号均放到当前源文件符号表

语法	IMPORT symbol{[WEAK]}

- symbol：要引入的标号，大小写敏感
- [WEAK]：当所有的源文件都没有定义这样一个标号时，编译器也不给出错误信息，在多数情况下将该标号置为0，若该标号为B或BL指令引用，则将B或BL指令置为NOP操作
- 使用示例：

```
IMPORT      __main      ; 初始化
B           __main      ; 执行
```

4.2.6 其他杂项伪操作

⑩ EXTERN

伪操作	功能
EXTERN	通知编译器要使用的标号在其他的源文件中定义 只有当前源文件中引用时，该标号才放到当前源文件符号表（区别IMPORT）

语法	EXTERN symbol{[WEAK]}

- symbol：要引入的标号，大小写敏感
- [WEAK]：当所有的源文件都没有定义这样一个标号时，编译器也不给出错误信息，在多数情况下将该标号置为0，若该标号为B或BL指令引用，则将B或BL指令置为NOP操作
- 使用示例：

```
EXTERN      __main      ; 初始化
B           __main      ; 执行
```

4.2.6 其他杂项伪操作

⑪ GET | INCLUDE

伪操作	功能
GET	将一个源文件包含到当前的源文件中，并将被包含的源文件在当前位置进行汇编处理 可以使用INCLUDE代替GET

语法	GET filename

- 类似C语言中的“include”
- 使用示例：

```
GET     a.s
GET     D:\program\ex.s
```

4.2.6 其他杂项伪操作

⑫ INCBIN

伪操作	功能
INCBIN	将一个目标文件或任何其他类型的数据文件包含到当前的源文件中被包含的文件不被编译，不作任何变动地存放在当前文件中，编译器从其后开始继续处理（区别GET）

语法	INCBIN filename

✧ 使用示例：

```
AREA     Init, CODE, READONLY
INCBIN   a1.dat       ; 通知编译器当前源文件包含文件a1.dat
INCBIN   D: \a2.txt   ; ~~当前源文件包含文件C: \a2.txt
...
END
```

4.2.6 其他杂项伪操作

⑬ KEEP

伪操作	功能
KEEP	告诉编译器将局部符号包含在目标文件的符号表中

语法	KEEP {symbol}

- ✧ symbol：可选，若不指定，则除了寄存器相关符号之外的所有局部变量均被保留
- ✧ **使用说明：**
  - 默认情况下，编译器仅将下列符号包含到目标文件的符号表中：
    - 被输出的符号
    - 被重定义的符号
- ✧ **使用示例：**

```
label ADC   r2, r3, r4
      KEEP  label ; label将在调试器中可见
      ADD   r2, r2, r5
```

4.2.6 其他杂项伪操作

⑭ NOFP

伪操作	功能
NOFP	禁止源程序中包含浮点运算指令

语法	NOFP

✧ **使用说明：**

- 对于没有硬件或软件仿真支持浮点运算时，使用NOFP伪操作，代码中出现浮点运算指令时编译器将报错
- 如果浮点运算指令出现在NOFP之前编译器也会报错

4.2.6 其他杂项伪操作

⑮ REQUIRE

伪操作	功能
REQUIRE	指定段之间的相互依赖关系

语法	REQUIRE label

- ✧ label：所需要的标号的名称
- ✧ **使用说明**：
  - 当进行链接处理时包含了使用REQUIRE的段，则定义label的段也将被包含在连接器中

4.2.6 其他杂项伪操作

⑯ REQUIRE8 | PRESERVE8

伪操作	功能
REQUIRE8	指示链接器当前代码中要求数据栈为8字节对齐
PRESERVE8	指示链接器当前代码中数据栈是8字节对齐的

语法	REQUIRE8 PRESERVE8

✧ **使用说明：**

- 用LDRD和STRD指令加载或存储双字时要求内存单元地址8字节对齐
- 用这些指令操作数据栈时，要求数据栈8字节对齐
- 链接器可以保证要求数据栈8字节对齐的代码只能直接或间接地被“声明数据栈是8字节对齐的代码”调用

4.2.6 其他杂项伪操作

⑰ RN

伪操作	功能
RN	给一个寄存器定义一个别名

语法	name RN expr

- name：不能与表4.2中预定义寄存器和协处理器名称冲突
- expr：寄存器编号r0~r15
- **使用说明**：方便程序员记忆该寄存器的功能
- 使用示例：

```
age     RN      r10
```

4.2.6 其他杂项伪操作

⑱ ROUT

伪操作	功能
ROUT	限制局部标号的作用范围

语法	{routname}　ROUT

- routname：可选，代表区间的名称
- **使用说明**：
  - 在程序中未使用ROUT时，局部标号的作用范围为所在的AREA
  - 使用ROUT后，局部标号的作为范围为当前ROUT和下一个ROUT之间从而可以防止错误地引用局部标号

章节内容

- ARM汇编语言语句格式
- ARM汇编伪操作
- **ARM汇编语言伪指令**
- ARM汇编语言中的符号
- ARM汇编语言中的表达式
- ARM汇编语言程序结构
- 汇编语言子程序调用
- C语言与汇编语言混合编程
- ARM汇编语言设计实例

4.3 ARM汇编语言伪指令

- **伪指令**不是真正的ARM指令或Thumb指令，而是在**编译时**由**汇编器**替换成合适的ARM或Thumb指令序列
  - ARM伪指令包括：
    - ADR
    - ADRL
    - LDR
    - LDFD
    - LDFS
    - NOP
  - 除NOP之外，上述伪指令不能出现在ARM C和C++嵌入汇编中

4.3 ARM汇编语言伪指令：ADR

- ADR：小范围的地址读取伪指令

伪指令格式	操作
ADR{cond}　register, expr	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加载到目标寄存器中

- register　加载的目标寄存器
- expr　地址表达式（基于PC的地址值或基于寄存器的表达式）

✧ 地址表达式expr的取指范围：

- 当地址值是字节对齐时，其取指范围为 -255～255
- 当地址值是字对齐时，其取指范围为 -1020～1020

✧ 注意：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

4.3 ARM汇编语言伪指令：ADR

- ADR：小范围的地址读取伪指令

伪指令格式	操作
ADR{cond}　register, expr	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加载到目标寄存器中

- register　加载的目标寄存器
- expr　地址表达式（基于PC的地址值或基于寄存器的表达式）

✧ 使用示例：

```
start    MOV r0, #0
         ADR r1, start    ; 将标号start的地址存入r1
         ; 该指令将被替换为SUB r1, pc, #0xC
```

4.3 ARM汇编语言伪指令：ADRL

- **ADRL：中等范围的地址读取伪指令**

伪指令格式	操作
ADRL{cond} register, expr	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加载到目标寄存器中

- register　加载的目标寄存器
- expr　地址表达式（基于PC的地址值或基于寄存器的表达式）

✧ 地址表达式expr的取指范围：

- 当地址值是字节对齐时，取指范围为-64K～64K （-65536~65535）
- 当地址值是字对齐时，取指范围为-256K～256K （-262144~262143）

✧ 注意：通常，编译器用两条ADD指令或SUB指令来实现该ADRL伪指令的功能，若不能用两条指令实现，则产生错误，编译失败

4.3 ARM汇编语言伪指令：ADRL

- **ADRL：中等范围的地址读取伪指令**

伪指令格式	操作
ADRL{cond} register, expr	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加载到目标寄存器中

- register　加载的目标寄存器
- expr　地址表达式（基于PC的地址值或基于寄存器的表达式）

✧ 使用示例：

```
start   MOV r0, #0
        ADRL    r1, start+6000
; 该ADRL伪指令被替换为如下两条指令：
ADD     r1, pc, 0xE800
ADD     r1, r1, 0x254
```

4.3 ARM汇编语言伪指令：LDR

- LDR：中等范围的地址读取伪指令

伪指令格式	操作
LDR{cond} register, = expr|label_expr	将任意一个32位常数或地址值加载到指定寄存器

- register 加载的目标寄存器
- expr 地址表达式（基于PC的地址值或外部表达式）

✧ 表达式expr为32位常量，编译器根据下列情况替换成相应语句：

① expr未超出MOV|MVN的范围，则用MOV|MVN指令代替LDR伪指令

② 否则汇编器将expr放入文字池，并用一条基于PC的LDR指令加载expr

✧ 注意：

- LDR伪指令被编译器替换成一条合适的指令
- 从指令位置到文字池的偏移量必须小于4KB

4.3 ARM汇编语言伪指令：LDR

- LDR：中等范围的地址读取伪指令

伪指令格式	操作
LDR{cond} register, = expr|label_expr	将任意一个32位常数或地址值加载到指定寄存器

- register 加载的目标寄存器
- expr 地址表达式（基于PC的地址值或外部表达式）

✧ 使用示例（加载常量）：

```
LDR    R2, =0xFF0          ; MOV  R2, #0xFF0
LDR    R0, =0xFF000000     ; MOV  R0, #0xFF000000
LDR    R1, =0xFFFFFFFE     ; MVN  R1, #0x1
```

4.3 ARM汇编语言伪指令：LDR

- LDR：中等范围的地址读取伪指令

伪指令格式	操作
LDR{cond} register, = expr|label_expr	将任意一个32位常数或地址值加载到指定寄存器

- register 加载的目标寄存器
- expr 地址表达式（基于PC的地址值或外部表达式）

✧ 使用示例（加载地址）：

```
                LDR pc, =ResetHandler
; 设ResetHandler为外部程序标号，地址为0x00002540

; 翻译后的反汇编代码为：
[0x00000000]   LDR pc, 0x0000011C
...                                  ; 其他代码
[0x00000000]   DCD 0x00002540        ; 地址数据缓冲池
```

4.3 ARM汇编语言伪指令：NOP

- LDR：中等范围的地址读取伪指令

伪指令格式	操作
NOP	汇编时被编译器替换为一条无用语（ARM中的空操作） 例如：MOV r0, r0

- 使用说明：NOP可用于延时操作，不会影响CPSR中的标志位
  - ✧ 使用示例（延时子程序）：

```
        MOV    R1, #1234
Delay
        NOP                    ; 空操作
        NOP                    ; 空操作
        SUBS   R1, R1, #1      ; 循环次数减
        BNE    Delay           ; 如果循环未结束，跳转Delay继续
        MOV    PC, LR          ; 子程序返回
```

章节内容

- ARM汇编语言语句格式
- ARM汇编伪操作
- ARM汇编语言伪指令
- **ARM汇编语言中的符号**
- ARM汇编语言中的表达式
- ARM汇编语言程序结构
- 汇编语言子程序调用
- C语言与汇编语言混合编程
- ARM汇编语言设计实例

4.4 ARM汇编语言中的符号

- 在ARM汇编语言中，可以使用符号（symbols）来代表变量，地址和数字常量
  - ✧ 代表地址的符号也称标号（labels）
- 本节内容
  - ① 符号命名规则
  - ② 变量（variables）
  - ③ 数字常量（numeric constants）
  - ④ 汇编时变量置换
  - ⑤ 标号(labels)
  - ⑥ 局部标号（local labels）

4.4 ARM汇编语言中的符号

① 符号命名规则

- 符号由字母（大小写敏感）、数字及下划线组成
- 除了局部标号外，其他标号不能以数字开头
- 符号在其作用范围内必须唯一
- 符号不能与系统内置变量（表4.1）或预定义符号（表4.2）同名
- 程序中的符号通常不要与指令助记符或者伪操作同名
  - 当程序中的符号与指令助记符或者伪操作同名时，用双竖线将符号括起来，如||ASSERT||，此处双竖线并不是符号的组成部分
- 当应用更为广义的符号时，比如编译器产生的标号通常用单竖线括起来，例如：|IMAGEZIBASE|（其中IMAGEZIBASE是ZI区基址，由编译器产生）

4.4 ARM汇编语言中的符号

② 变量

- ✧ 汇编程序中的变量在汇编处理过程中其值可能发生变化
- ✧ 这里变量有数字变量，逻辑变量和字符串变量3种类型，且一旦定义则类型不能改变
- ✧ 可用GBLx定义全局变量，LCLx定义局部变量，SETx对其初始化

4.4 ARM汇编语言中的符号

③ 数字常量

- ✧ 一般为32位的整数
- ✧ 可用EQU伪操作定义数字常量

4.4 ARM汇编语言中的符号

④ 汇编时变量置换

- 程序中的变量可通过代换操作取得一个常量，**代换操作符为“$”**
  - 如果在字符串变量前面有一个代换操作符“$”，编译器会将该字符串变量的值代换“$”后的字符串变量
  - 当程序中需要字符$时，用$$来表示，编译器将$$当作$
- **用“.”指示变量名结束**
- 使用示例：

```
        LCLS   S1          ; 定义局部字符串变量S1
        LCLS   S2          ; 定义局部字符串变量S2
S1      SETS   "Test"      ; 初始化变量S1为“Test”
S2      SETS   "$$This is a $S1.!"
                           ; 初始化变量S2为“$This is a Test!”
```

4.4 ARM汇编语言中的符号

⑤ 标号

✧ 表示程序中的指令或者数据地址的符号

- 基于PC的标号：PC值加/减一个数字常量
  常用于程序跳转、代码段中插入少量数据等
- 基于寄存器的标号：寄存器加/减一个数字常量
  可用MAP/FIELD/EQU定义，常用于访问数据段中的数据
- 绝对地址：32位的数字常量，寻址范围0~$2^{32}-1$

⑥ 局部标号

✧ 是一个数字值（0~99），可以紧随其后加上表示该局部变量范围的符号（参见ROUT伪操作）

- 如果不加该符号，其作用范围为当前段

✧ 局部标号的典型应用是在一个短例程里面应用循环或条件转移，在宏定义里面很有用

章节内容

- ARM汇编语言语句格式
- ARM汇编伪操作
- ARM汇编语言伪指令
- ARM汇编语言中的符号
- **ARM汇编语言中的表达式**
- ARM汇编语言程序结构
- 汇编语言子程序调用
- C语言与汇编语言混合编程
- ARM汇编语言设计实例

4.5 ARM汇编语言中的表达式

- 本节内容：
  - ① 字符串的格式和字符串表达式
  - ② 整型数字的格式和数字表达式
  - ③ 浮点数的格式
  - ④ 基于寄存器的表达式，基于PC的表达式
  - ⑤ 逻辑值的格式，逻辑表达式以及关系表达式
  - ⑥ 操作符和操作符优先级

4.5 ARM汇编语言中的表达式

① 字符串的格式和字符串表达式

- ✧ 一系列双引号括起来的字符，长度为0~512
  - 当字符串中有 " 时，用 "" 表示
  - 当字符串中有 $ 时，用 $$ 表示
- ✧ 字符串变量用GBLS|LCLS伪操作定义，用SETS设置
- ✧ 与字符串有关的操作符有LEN、CHR、STR、LEFT等

```
str1   SETS   "it is a ""string"""
              ; str1的值为“it is a "string"”
str2   SETS   "$str1.$$"
              ; str2的值为“it is a "string" $”
```

4.5 ARM汇编语言中的表达式

② 整型数字的格式和数字表达式

- 整型数字有以下几种格式
  - 十进制数（默认）
  - 十六进制数：如0xFF、&3A
  - n进制数：如2_110010
  - 字符：如：'A'、'\'
- DCQ(U)伪操作中用到的整型数字范围为$0\sim2^{64}-1$
  - 其他伪操作用到的整型数字范围为$0\sim2^{32}-1$
- 数字常量由EQU定义，数字变量由GBLA|LCLA定义，由SETS设置
- 与数字有关的操作符有NOT、AND、加/减/乘法等

```
a       SETA    256*256          ; 256*256为数字表达式
        MOV     r1, #a
```

4.5 ARM汇编语言中的表达式

③ 浮点数的格式

✧ 浮点数有以下几种格式

- {-} digitsE {-} digits
- {-} {digits} .digits {E {-} digits}
- 0x or &

✧ 单精度浮点数的范围为

1.17549435e-38 ~ 3.40282347e+38

✧ 双精度浮点数的范围为

2.2250738585072014e-308 ~ 1.7976931348623157e+308

```
DCFD   1E308, -4E-100      ; 1×10^308, -4×10^-100
DCFS   1.0
```

4.5 ARM汇编语言中的表达式

④ 基于寄存器的表达式，基于PC的表达式

- ✧ 表示一个寄存器加/减一个数字常量，通常用MAP和FIELD定义
- ✧ 相关操作符有BASE等

4.5 ARM汇编语言中的表达式

⑤ 逻辑值的格式，逻辑表达式以及关系表达式

- 逻辑值表示的格式：逻辑真{TRUE}、逻辑假{FALSE}
- 逻辑变量由GBLL或LCLL定义，由SETL赋值
- 相关操作符有LAND、LNOT等

4.5 ARM汇编语言中的表达式

⑥ 操作符和操作符优先级

✧ 单目操作符

- 位于操作数的左边

✧ 双目操作符

- 包括：乘法、字符串、移位、加减法和位操作、关系操作、逻辑操作

✧ 操作符优先级

4.5 ARM汇编语言中的表达式

表4.4 单目操作符及用法

操作符	用法	说明
?	? A	得到定义标号A的代码行所生成的代码所占字节数
BASE	:BASE:A	得到A的基址寄存器标号，宏定义中很有用
INDEX	:INDEX:A	得到A相对于其基址寄存器的偏移量
+ 和 -	+A 和 -A	正/负号
LEN	:LEN:A	返回字符串A的长度
CHR	:CHR:A	得到ASCII码为A的单字符的字符串
STR	:STR:A	将数字表达式转化为其十六进制的字符串形式，将逻辑表达式转化为"T"或"F"
NOT	:NOT:A	将32位数字量A按位取反
LNOT	:LNOT:A	将逻辑表达式取反
DEF	:DEF:A	判断标号A是否已定义
SB_OFFSET_19_12	:SB_OFFSET_19_12:A	返回A与静态基址寄存器sb(r9)的偏移值的bit[19:12] 仅见于ADD和LDR指令
SB_OFFSET_11_0	:SB_OFFSET_11_0:A	返回A与静态基址寄存器sb(r9)的偏移值的bit[11:0] 仅见于ADD和LDR指令

4.5 ARM汇编语言中的表达式

表4.5~4.7 双目操作符及用法（乘法、字符串、移位操作）

操作符	用法	说明
*	A*B	A乘以B
/	A/B	A除以B的商
MOD	A:MOD:B	A除以B的余数
LEFT	A:LEFT:B	取A(字符串)左边的B(数字)个字符
RIGHT	A:RIGHT:B	取A(字符串)右边的B(数字)个字符
CC	A:CC:B	取A(字符串)结合B(字符串)
ROL	A:ROL:B	取值为：A循环左移B位
ROR	A:ROR:B	取值为：A循环右移B位
SHL	A:SHL:B	取值为：A左移B位
SHR	A:SHR:B	取值为：A右移B位

4.5 ARM汇编语言中的表达式

表4.8~4.10 双目操作符及用法（加/减/位操作、关系操作、逻辑操作）

操作符	用法	说明
+	A+B	加法
-	A-B	减法
AND	A:AND:B	A和B按位取与
OR	A:OR:B	A和B按位取或
EOR	A:EOR:B	A和B按位取异或
=	A=B	A和B相等
>	A>B	A大于B
>=	A>=B	A大于或等于B
<	A<B	A小于B
<=	A<=B	A小于或等于B
/= 或 <>	A/=B 或 A<>B	A不等于B
LAND	A:LAND:B	A和B逻辑与
LOR	A:LOR:B	A和B逻辑或
LEOR	A:LEOR:B	A和B逻辑异或

4.5 ARM汇编语言中的表达式

表4.11 汇编与C语言操作符优先级的排列对比

汇编语言操作符优先级（从大到小）	与汇编等效或相似的标准C语言操作符	C语言操作符优先级（从大到小）
括号	括号	括号
单目操作符	单目操作符	单目操作符
* / MOD	* / %	* / %
字符串操作	无	+ -
SHL SHR ROR ROL	<< >>	<< >>
+ - AND OR EOR	+ - & | ^	< <= > >=
= > >= < <= /= <>	== > >= < <= !=	== !=
LAND LOR LEOR	&& ||	&
		^
		|
		&&
		||

单目>双目
相邻单目，从右至左执行
相邻双目，从左至右执行

注意：汇编和C中操作符优先级有区别
例如：1+2:SHR:3与1+(2:SHR:3)相同，而1+2>>3与(1+2)>>3相同

章节内容

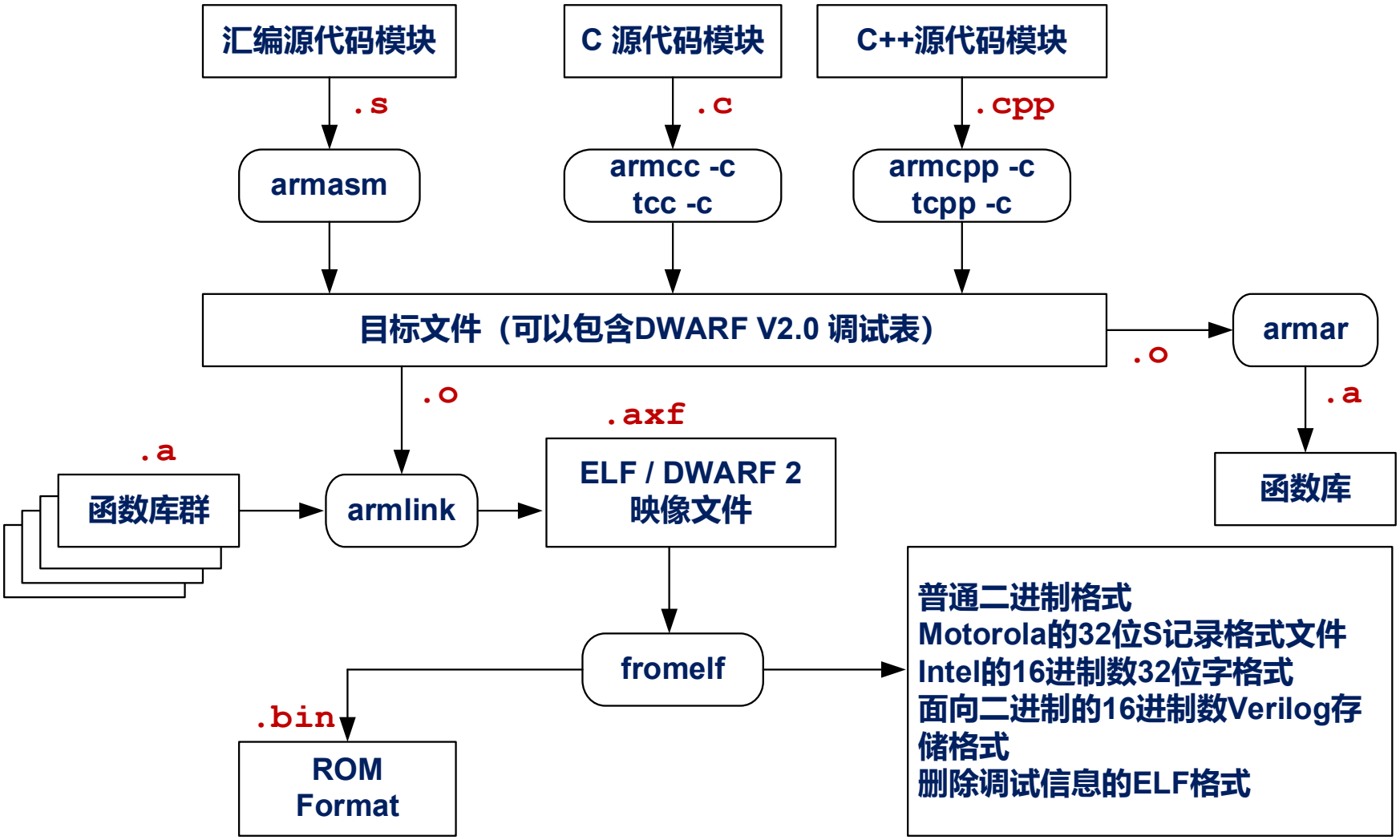
- ARM汇编语言语句格式
- ARM汇编伪操作
- ARM汇编语言伪指令
- ARM汇编语言中的符号
- ARM汇编语言中的表达式
- **ARM汇编语言程序结构**
- 汇编语言子程序调用
- C语言与汇编语言混合编程
- ARM汇编语言设计实例

- ARM映射文件结构
- ARM映射文件各组成部分地址映射关系
- Scatter文件的应用

4.6 ARM汇编语言程序结构

- ARM中各种源文件（包括汇编、C/C++程序）经过ARM编译器编译后生成ELF格式的目标文件
- 这些目标文件和相应的C/C++运行库经过ARM链接器处理后，生成ELF格式的映像文件（image）
  - ELF(Executable and Linking Format)
    可执行链接文件格式
  - ELF格式的映像文件可以被写入嵌入式设备的ROM中
  - ELF目标文件的文件名后缀是“.o”

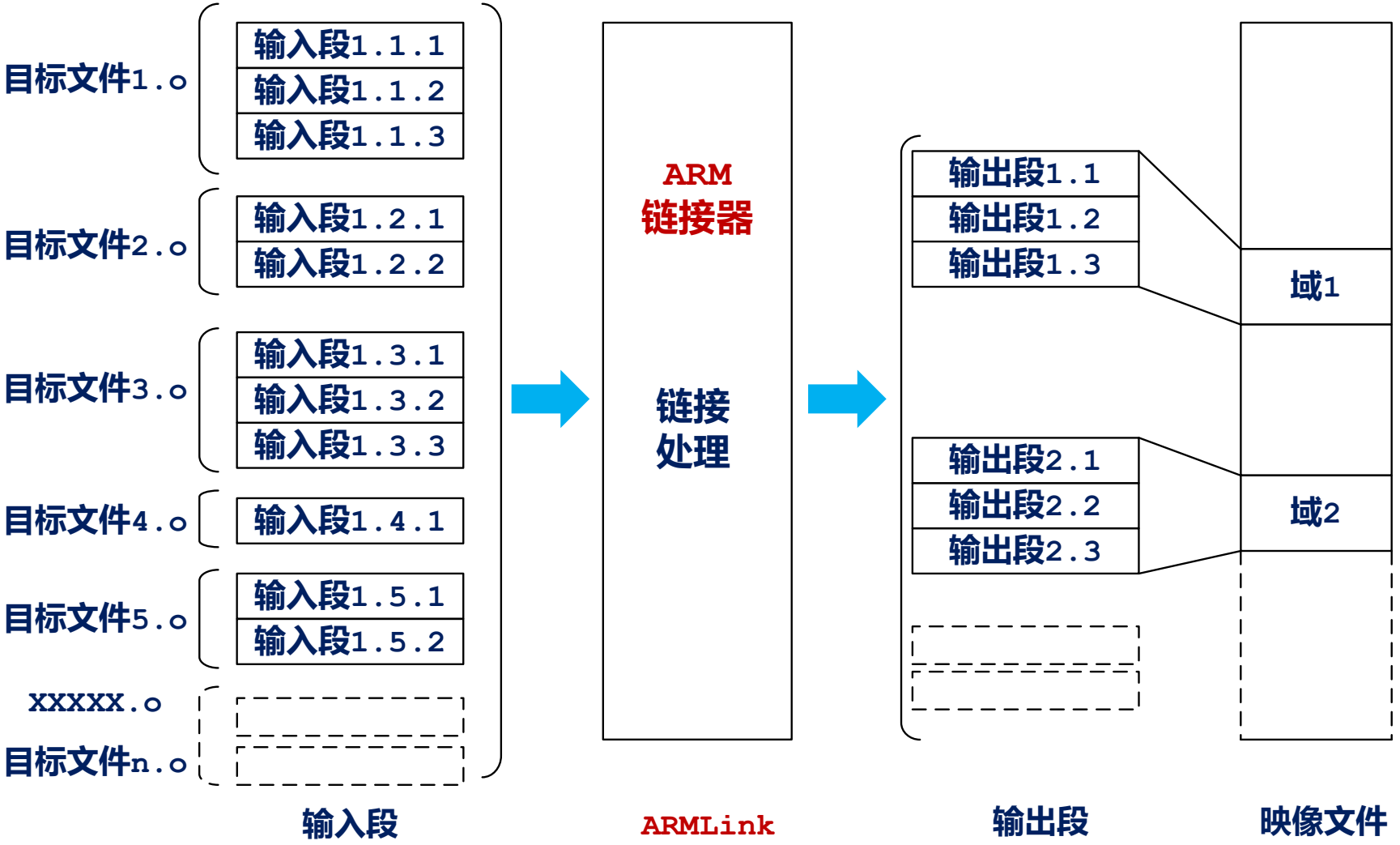
4.6 ARM汇编语言程序结构



4.6.1 ARM映像文件的结构

- ARM映像文件是个层次性结构的文件，其中包含了域、输出段和输入段，各部分关系如下：
  - ✧ 一个映像文件由一个或多个域组成
  - ✧ 每个域包含一个或多个输出段
  - ✧ 每个输出段包含一个或多个输入段
  - ✧ 各输入段包含了目标文件中的代码和数据

4.6.1 ARM映像文件的结构



4.6.1 ARM映像文件的结构

- **输入段**中包含4 类内容：
  - ① 代码
  - ② 已经初始化的数据
  - ③ 未经过初始化的存储区域
  - ④ 内容初始化成0的存储区域
- 每个输入段有相应的属性：
  - ✧ 只读（RO ）
  - ✧ 可读写（RW ）
  - ✧ 初始化成0（ZI）
- ARM 链接器**根据**各输入段的**属性**将这些输入段**分组，再组成不同的输出段以及域**

4.6.1 ARM映像文件的结构

- **输出段**中包含一系列具有相同RO、RW和ZI属性的输入段
  - 输出段的属性与其中包含的输入段的属性相同
  - 在一个输出段内部，各输入段按照一定的规则排序
- **域**中包含1~3个输出段，其中**各输出段的属性各不相同**
  - 各输出段的排列顺序由其属性决定
    1. RO属性的输出段排在最前面
    2. 其次是RW 属性的输出段
    3. 最后是ZI 属性的输出段
  - 一个域通常映射到一个物理存储器上，如ROM和RAM等

4.6.2 映像文件各组成部分的地址映射关系

- ARM映像文件的各组成部分在存储系统中的地址有两种：
  - 加载时地址：映像文件位于存储器中时的初始地址，也就是系统未启动运行时对应的地址
  - 运行时地址：映像文件运行时的地址，就是系统启动运行后对应的地址
- 举例：
  - 已初始化的RW属性数据，在加载时可能保存在ROM中，但运行时被移动到RAM中

4.6.2 映像文件各组成部分的地址映射关系

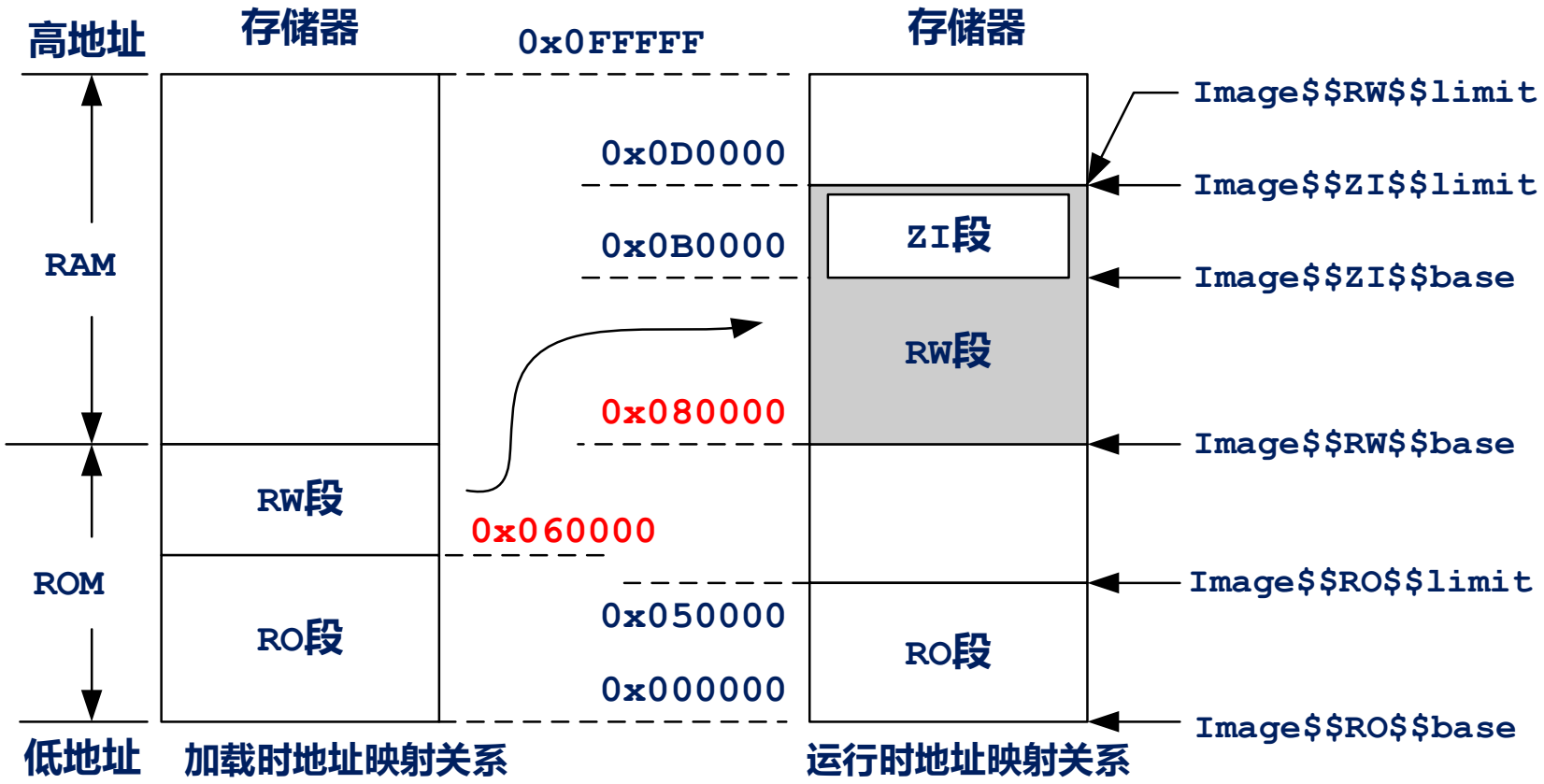


图4.2 映射文件的地址映射关系

4.6.2 映像文件各组成部分的地址映射关系

- **说明：**
  - Image$$RO$$base　　RO输出段运行时起始地址
  - Image$$RO$$limit　　RO输出段运行时存储区界限
  - Image$$RW$$base　　RW输出段运行时起始地址
  - Image$$RW$$limit　　RW输出段运行时存储区界限
  - Image$$ZI$$base　　ZI输出段运行时起始地址
  - Image$$ZI$$limit　　ZI输出段运行时存储区界限
- **三个段空间计算公式：**
  - RO段大小 = Image$$RO$$limit - Image$$RO$$base
  - RW段大小 = Image$$RW$$limit - Image$$RW$$base
  - ZI段大小 = Image$$ZI$$limit - Image$$ZI$$base

章节内容

- ARM汇编语言语句格式
- ARM汇编伪操作
- ARM汇编语言伪指令
- ARM汇编语言中的符号
- ARM汇编语言中的表达式
- ARM汇编语言程序结构
- **汇编语言子程序调用**
- C语言与汇编语言混合编程
- ARM汇编语言设计实例

4.7 汇编语言子程序调用

- 使用子程序可以进行**模块化**编程，实现较复杂的功能，子程序间调用要**符合ATPCS标准**
- **本节内容：**
  - ✧ 子程序的调用
  - ✧ ATPCS（ARM-Thumb Procedure Call Standard）

4.7.1 子程序调用

- 在ARM汇编语言程序中，子程序的调用一般是通过BL指令来实现的

语法格式	BL subroutine

- 该指令在执行时完成如下操作：
  - ① 将子程序的返回地址存放在链接寄存器LR中
  - ② 同时将程序计数器PC指向子程序的入口点
  - ③ 当子程序执行完毕需返回调用处时，只需要将存放在LR中的返回地址重新拷贝给程序计数器PC
  - ✧ 在调用子程序的同时，也可以完成参数传递和从子程序返回运算结果，通常可以使用寄存器R0～R3完成
  - ✧ 独立编译的子程序间的互访要遵守ATPCS标准

4.7.1 子程序调用

- 子程序调用示例：

```
        AREA   Init, CODE, READONLY
        ENTRY
start
        LDR    R0, =0x3FF5000
        LDR    R1, =0xFF
        STR    R1, [R0]
        LDR    R0, =0x3FF5008
        LDR    R1, =0x01
        STR    R1, [R0]
        BL     PRINT_TEXT    ; 调用PRINT_TEXT子程序
        ...
; 子程序
PRINT_TEXT
        ...
        MOV    PC, LR        ; 返回主调程序
        ...
        END
```

4.7.2 ATPCS准则

- 独立汇编和编译的子程序间的互访要遵守ATPCS标准
- ATPCS规定了子程序间互访的一些基本规则：
  - 寄存器的使用和命名
  - 数据栈的使用
  - 参数的传递和结果返回

4.7.2 ATPCS准则

■ 寄存器的使用和命名

① 用r0~r3传递参数或返回结果，其中r0~r3可记作a1~a4
被调子程序返回前无需恢复r0~r3的值
主调程序若要在调用其他程序后继续使用原来r0~r3值则必须保存

② 子程序中用r4~r11保存局部变量（Thumb中只能用r4~r7），其中r4~r11可以记作v1~v8，如果子程序使用到其中某些寄存器，则必须在使用前保存这些寄存器的值，并在返回前予以恢复

③ 寄存器r12为调用过程中临时寄存器（intra-process-scratch），记作ip，用于保存SP，在函数返回时使用该寄存器出栈

④ 寄存器r13用作数据栈指针，记作sp
不能用于其他目的，子程序调用前后其值必须保持一致

⑤ 寄存器r14为链接寄存器，记作lr，用于保存子程序的返回地址
如果在程序中保存了r14，则其可用于其他目的

⑥ 寄存器r15为程序计数器，记作pc，不能用于其他目的

4.7.2 ATPCS准则

■ 寄存器的使用和命名

寄存器	别名	专称	使用规则
r15	-	pc	程序计数器
r14	-	lr	链接寄存器
r13	-	sp	数据栈指针
r12	-	ip	程序调用中间临时寄存器
r11	v8	-	ARM状态局部变量寄存器8
r10	v7	sl	ARM状态局部变量寄存器7或在支持数据栈检查的ATPCS中为堆栈限制指针
r9	v6	sb	ARM状态局部变量寄存器6或静态基址寄存器
r8	v5	-	ARM状态局部变量寄存器5
r7	v4	wr	局部变量寄存器4或thumb状态局部变量寄存器
r6~r4	v3~v1	-	局部变量寄存器3~1
r3~r0	a4~a1	-	参数，结果或临时寄存器

4.7.2 ATPCS准则

- **数据栈的使用规则**
  - ✧ 在ATPCS中规定数据栈为**满递减型**
  - ✧ 外部接口的数据栈必须**8字节对齐**

4.7.2 ATPCS准则

■ 参数的传递和结果返回

✧ 子程序的参数个数可以是固定和非固定的两种情况参数传递规则不同

✧ 参数个数不固定

① 当参数不超过4个时，依次用寄存器r0~r3保存参数

② 超过4个时剩余的参数用数据栈保存，入栈顺序与参数顺序相反

- 这样，对于超过4字节的参数（如双精度浮点数，结构体），可能一部分保存于寄存器，另一部分保存于数据栈，或者全部压栈

✧ 参数个数固定

- 对于包含浮点运算的硬件的系统，在浮点参数的传递方面与参数个数不固定情况不同

  规则：浮点参数按顺序处理，为每个浮点参数分配满足其需要的且编号最小的一组连续的FP寄存器

4.7.2 ATPCS准则

■ 参数的传递和结果返回

- ✧ 子程序的参数个数可以是固定和非固定的两种情况参数传递规则不同
- ✧ 参数个数不固定
- ✧ 参数个数固定
- ✧ 关于子程序结果返回的规则为：
  - ① 结果为一个字的整数时，用r0返回
  - ② 结果为2～4字的整数时，用r0~r1，r0~r2，r0~r3返回
  - ③ 结果为浮点数，用浮点寄存器f0，d0或s0返回
  - ④ 结果为复合型的浮点数，用f0~fn或d0~dn返回
  - ⑤ 对于更多位数的结果，间接用内存返回

章节内容

- ARM汇编语言语句格式
- ARM汇编伪操作
- ARM汇编语言伪指令
- ARM汇编语言中的符号
- ARM汇编语言中的表达式
- ARM汇编语言程序结构
- 汇编语言子程序调用
- **C语言与汇编语言混合编程**
- ARM汇编语言设计实例

- ✧ 内嵌汇编
- ✧ C和汇编互相调用
- ✧ 编译器特定关键字

4.8.1 内嵌汇编

- C和C++编译器中内置了**内嵌汇编器**，可以用其实现C语言不能或不易完成的操作
- 本节内容：
  ① 内嵌汇编指令的用法
  ② 内嵌汇编器与ARM ASM汇编器的区别
  ③ 内嵌汇编在C和C++程序中的使用格式

4.8.1 内嵌汇编

■ 内嵌汇编指令的用法

内嵌的汇编指令支持大部分ARM和Thumb指令

① 操作数

✧ 内嵌指令的操作数**可以是C或C++表达式**，这些表达式的值均按**无符号数**处理

- 当指令中同时使用寄存器和C或C++表达式时，表达式不要过于复杂，以免编译器在计算表达式时用到过多的寄存器以至于与指令中用的寄存器冲突

4.8.1 内嵌汇编

② 寄存器

- ✧ **不推荐直接使用**，可能影响编译器寄存器分配，从而影响程序效率
- ✧ 如果必须使用要注意：
  - 不要向PC赋值，只能利用B或BL指令实现跳转
  - 需要注意编译器可能会使用r12和r13存储临时变量，在计算表达式的值时可能会把r0~r3，r12，r14用于函数调用
  - 如果C变量用到了指令中用到的物理寄存器，编译器一般会在必要时用栈保存或恢复这些寄存器，但排除sp，sl，fp和sb

③ 常量

- ✧ 定值表达式前的“#”可以省略
- ✧ 如果用“#”，则其后面必须是常量

4.8.1 内嵌汇编

④ 标号

- ✧ 可以利用B指令（不能用BL）跳转到C或C++中的标号

⑤ 指令展开

- ✧ 除了与协处理器相关的指令，大多数ARM或Thumb指令对常量的操作会被展开成多条指令
- ✧ 各指令的展开对标志位的影响情况：
  - 算术指令可以正确设置NZCV标志位
  - 逻辑指令可以正确设置NZ标志位，不影响V位，破坏C位

⑥ 内存分配

- ✧ 所有内存分配都在C/C++程序中声明，用标号在内嵌汇编中引用
- ✧ 不要在内嵌汇编中用伪操作分配内存

4.8.1 内嵌汇编

⑦ SWI和BL指令的使用

- 在内嵌的SWI和BL指令中，除了正常的操作数域外，还必须增加3个可选的寄存器列表：
  - 第1个寄存器列表中的寄存器用于存放输入的参数
  - 第2个寄存器列表中的寄存器用于存放返回的结果
  - 第3个寄存器列表中的寄存器的内容可能被调用的子程序破坏

4.8.1 内嵌汇编

- **内嵌汇编器与ARM ASM汇编器的区别**

使用内嵌汇编还应注意以下几点：

① 不能通过`(.)`或`{PC}`获得当前指令地址

② 不能用`LDR Rn, =expr`伪指令，可以用`MOV Rn, expr`替代（可生成从数据缓冲池中加载数据的汇编指令）

③ 不支持标号表达式

④ 不支持`ADR`和`ADRL`伪指令

⑤ 表示十六进制数只能用`0x`，不能用`&`

⑥ 编译器可能使用寄存器`r0~r3`、`ip`、`lr`存放中间结果
在使用这些寄存器时要注意

⑦ 编译器在计算C表达式时，可能改变CPSR寄存器的NZCV标志位
在指令中使用这些标志位时要注意

4.8.1 内嵌汇编

- **内嵌汇编器与ARM ASM汇编器的区别**

使用内嵌汇编还应注意以下几点：

⑧ 指令中使用的C变量不要与ARM物理寄存器同名

⑨ LDM与STM指令的寄存器列表中只能使用物理寄存器
不能使用C表达式

⑩ 不能写寄存器PC，不支持BX和BLX指令

⑪ 用户不需要维护数据栈，因为编译器会根据需要自动保存或恢复工作寄存器的值

⑫ 用户可以改变处理器模式，修改ATPCS寄存器sb、sl、fp，改变协处理器状态，但这并不为编译器所知

- 所以：如果用户改变了处理器的模式，不要使用原来的C表达式，直至重新恢复到原来的处理器模式后，方可使用这些C表达式

4.8.1 内嵌汇编

■ 内嵌汇编在C/C++程序中的使用格式

✧ 在标准C中可以使用__asm关键字来声明内嵌汇编语句

语法格式	__asm
	{
	asm_instruction [; asm_instruction]
	[asm_instruction]
	}

✧ 说明：

- 一条指令占多行用“\”续行
- 一行多条指令用“;”分隔
- 不能用“;”注释，可用C语言的注释方法

4.8.1 内嵌汇编

- 内嵌汇编在C/C++程序中的使用格式
  - ✧ 在标准C中可以使用__asm关键字来声明内嵌汇编语句

语法格式	__asm
	{
	asm_instruction [; asm_instruction]
	[asm_instruction]
	}

  - ✧ 在C++中，除了以上方法外还可用asm关键字

语法格式	asm("asm_instruction[; asm_instruction]");

    - 括号内必须是一个指令序列构成的字符串

4.8.1 内嵌汇编

- 使用示例：

```
void strcopy(const char *src, char *dst)
{       //本程序实现将字符串src拷贝到dst
        int  ch;
        __asm
        {
        loop:
            LDRB ch, [src], #1   //每次读取字符串中一个字符
            STRB ch, [dst], #1   //将读取的字符存入dst所指
            CMP  ch, 0           //比较字符串结尾标志位"0"
            BNE  loop            //如果ch不为0，跳至loop
        }
}
```

4.8.2 C语言和汇编语言互相调用

1. 汇编中访问C程序变量

- ✧ 在C程序中声明的**全局变量**可以被汇编程序**通过地址**间接访问
- ✧ 访问方法如下：
  - ① 使用IMPORT伪操作**声明**该全局变量
  - ② 使用LDR伪指令**读取**该全局变量的**内存地址**
    通常该全局变量的内存地址值存放在程序的数据缓冲池中
  - ③ 根据该数据的类型：
    - 使用相应的LDR指令**读取**该全局变量的**值**
    - 使用相应的STR指令**修改**该全局变量的**值**

各数据类型及对应的LDR/STR指令如下：

(1) 对于无符号变量，使用LDRB/STRB访问char

(2) 使用LDRH/STRH访问short

(3) 使用LDR/STR访问integer

(4) 对于有符号数，使用LDRSB和LDRSH

4.8.2 C语言和汇编语言互相调用

- 使用示例（汇编中访问C程序变量）

```
        AREA    globals, CODE
        EXPORT  asmsubroutine  ;汇编子程序实现外部C变量加2
        IMPORT  cgblvar        ;声明外部C变量
asmsubroutine
        LDR     r1, =cgblvar   ;用伪指令LDR读取该变量的地址
        LDR     r0, [r1]       ;读取r1指向的值，即cgbvar值
        ADD     r0, r0, #2     ;将外部C变量加2
        STR     r0, [r1]       ;结果存入r1，即存入cgblvar
        MOV     pc, lr
        END
```

4.8.2 C语言和汇编语言互相调用

2. 汇编语言与C语言程序互调：C调用汇编程序

- ✧ 汇编程序设计遵守ATPCS准则，保证程序调用时参数正确传递
- ✧ 汇编程序中使用EXPORT伪操作声明某程序
  使该程序可以被其他程序调用
- ✧ C语言程序中使用extern关键词声明被调用的汇编程序

4.8.2 C语言和汇编语言互相调用

- 使用示例（C调用汇编程序）

```
    AREA    scopy, CODE
    EXPORT  strcopy  ; 声明外部可调用
strcopy              ; r0中为目标字符串地址，r1中为源字符串地址
    LDRB    r2, [r1], #1
    STRB    r2, [r0], #1
    CMP     r2, #0
    BNE     strcopy
    MOV     pc, lr
    END
```

```
extern void strcopy(char *dest, const char *src);
int main(void)
{
    char *str1 = "string";
    char str2[10];
    strcopy((char*) str2, str1); // 调用汇编程序strcopy
    ...
    return 0;
}
```

4.8.2 C语言和汇编语言互相调用

2. 汇编语言与C语言程序互调：汇编调用C程序

- ✧ 汇编程序设计遵守ATPCS准则，保证程序调用时参数正确传递
- ✧ 汇编程序中使用IMPORT伪操作声明要调用外部程序

4.8.2 C语言和汇编语言互相调用

■ 使用示例（汇编调用C程序）f调用g，计算：$n+2n+3n+4n+5n$

```
    EXPORT   f                    ; 输出函数f，参数n用r0传递
    AREA     func, CODE
    IMPORT   g                    ; 声明引入外部函数g
f   STR      lr, [sp, #-4]!       ; 因该子程序要调用另一子程序，保存返回地址
                                  ; 直接用r0作为函数g的参数a
    ADD      r1, r0, r0           ; 计算2*n并送入r1，作为函数g的参数b
    ADD      r2, r1, r0           ; 计算3*n并送入r2，作为函数g的参数c
    ADD      r3, r1, r2           ; 计算5*n
    STR      r3, [sp, #-4]!       ; 5*n用栈传递给函数函数g的参数e
    ADD      r3, r1, r1           ; 计算4*n并送入r3，作为函数g的参数d
    BL       g                    ; 调用函数g，结果将返回到r0
    ADD      sp, sp, #4           ; 使sp指向已存的lr
    LDR      pc, [sp], #4         ; 汇编子程序返回
    END
```

```
int  g(int a, int b, int c, int d, int e)
{
    return  a + b + c +d +e; //实现5个整数相加
}
```

4.8.3 ARM C编译器的特定关键字

- ARM C编译器支持一些对ANSI C进行扩展的关键字，用于声明特定的函数或数据类型

① `__value_in_regs`

✧ 声明一个函数将通过整型寄存器（或浮点寄存器）以结构体的形式返回2~4个整型（或浮点）结果

语法格式	`__value_in_regs 返回类型 function-name([参数列表]);`

```
typedef struct
{
    int     a;
    int     b;
    int     c;
    int     d;
}   four_results;
__value_in_regs  four_results  mfun(int e, int f, int g, int h)
{...}
```

4.8.3 ARM C编译器的特定关键字

② __swi

✧ 使用该关键字声明的函数，可以有最多4个参数和4个返回值

```
void __swi(0)  myswi(void);
```

4.8.3 ARM C编译器的特定关键字

③ __volatile

- 用该关键字限定一个对象，告诉编译器该对象可能在程序之外被修改，这样编译器在编译时将不会优化程序中对其进行的操作
- 通常使用该关键字定义系统中的I/O寄存器

```
#define vu16  volatile unsigned short
typedef struct
{
    vu16   DR; // typedef volatile unsigned short vu16
    vu16   unused1;
    vu16   DIR;
    vu16   unused2;
    ...
}  GPIO_Type;
```

4.8.3 ARM C编译器的特定关键字

④ __inline

✧ 使用该关键字声明的C函数要求：编译器**将该函数在其被调用的地方展开**

- 如果该函数展开过长可能影响代码的紧凑性和性能时，该函数可能会被当成一般函数处理
- 所以一般利用此关键字声明**代码长度比较短且调用比较频繁**的函数

```
__inline U8 GPIO_ReadByte(GPIO_Type* GPIOx)
{
        return (U8) (GPIOx->DR & 0x00FF);
}
```

4.8.3 ARM C编译器的特定关键字

⑤ __irq

✧ 使用该关键字声明一个函数，告诉编译器该函数是一个IRQ或FIQ中断处理函数

- 这时该函数将在进入时保存默认的ATPCS标准要求保存的寄存器以及除了浮点寄存器外的被该函数破坏的寄存器
- 函数返回前恢复这些寄存器的值，将SPSR的值赋予CPSR，并通过将LR-4的值赋予PC实现返回
- 由于SPSR不被保存，所以不可以用该关键字定义一个可重入中断处理函数，即该中断处理函数运行过程不可以被中断
- 注意：该函数不能有参数且返回类型必须为void

```
void __irq IRQhandler(void)
{
        ...
}
```

4.8.3 ARM C编译器的特定关键字

⑥ __pure

✧ 使用该关键字声明一个函数，其返回结果仅依赖于其输入参数，不会访问除了数据栈之外其他任何存储单元

```
int __pure mfun(int a, int b)
{
        return a + b;
}
```

章节内容

- ARM汇编语言语句格式
- ARM汇编伪操作
- ARM汇编语言伪指令
- ARM汇编语言中的符号
- ARM汇编语言中的表达式
- ARM汇编语言程序结构
- 汇编语言子程序调用
- C语言与汇编语言混合编程
- **ARM汇编语言设计实例**

4.9 ARM汇编语言设计实例

- ARM嵌入式系统开发的过程中
  - 一般都采用C/C++语言来编写程序 （应用程序）
  - 系统的中断向量表、引导装载程序（Bootloader）、对实时性要求较高的运算程序等，一般都采用汇编语言编写
- 汇编语言中最基本程序结构
  - 分支结构
  - 循环结构

4.9.1 分支结构

- 通过**条件**来决定执行程序不同的分支
  - 等同于高级语言中的`if…else…`或者`switch…case`结构
  - **使用方法：**
    - 条件：通过**指令+后缀S**，或用**CMP、CMN、TST、TEQ**等指令
    - 分支：通过**指令+条件**，或用**B**指令

- **例4.1：简单分支程序**
  - 寄存器r0和r1中有两个正整数，求这两个数的最大公约数，结果存储在r0中
  - **分析：**程序所需条件判断很少，用**CPSR寄存器的标志位**即可实现程序的分支和跳转

4.9.1 分支结构

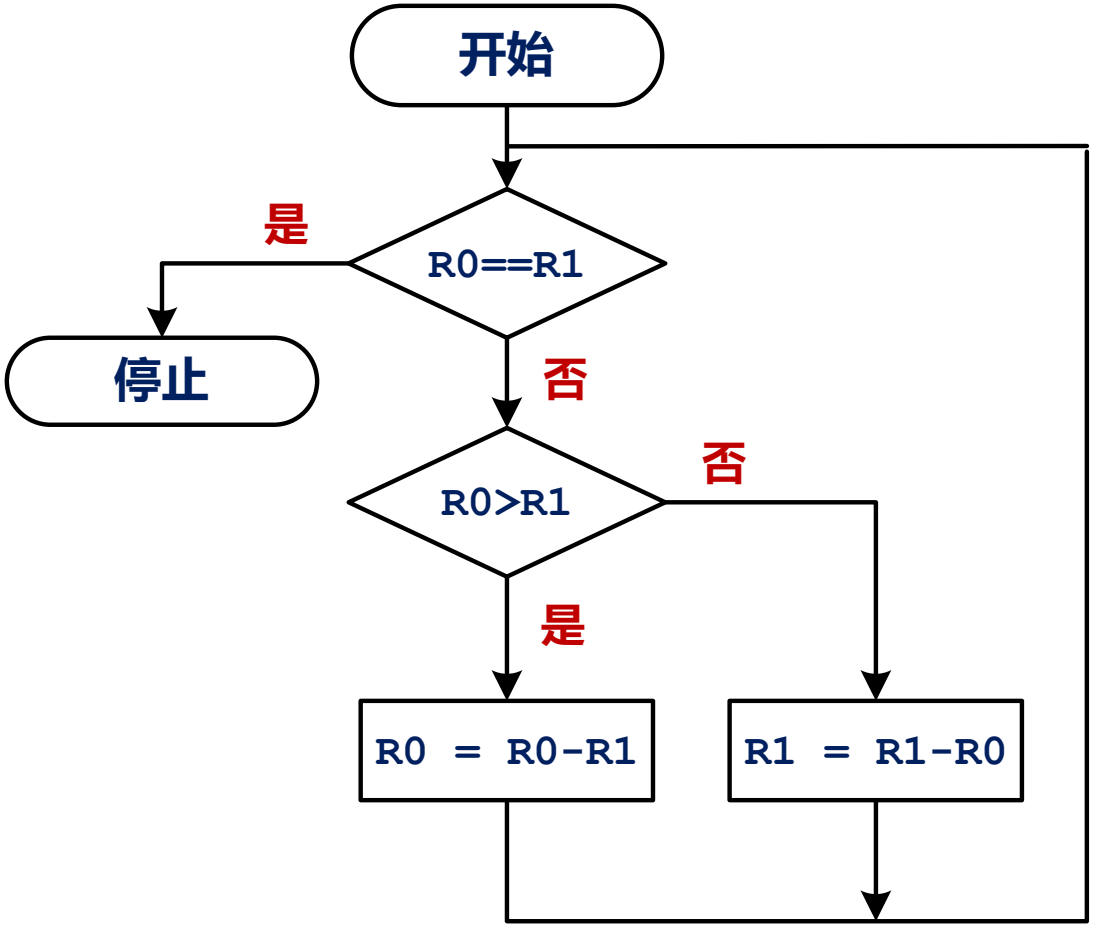


图4.6 简单分支流程图

4.9.1 分支结构

```
        AREA    Exma1, CODE    ; 程序代码段开始
        ENTRY                  ; 程序入口
        MOV     r0, #15        ; r0 = 15
        MOV     r1, #9         ; r1 = 9
start                          ; 程序标号start
        CMP     r0, r1         ; 比较r0和r1，更新标志位
        SUBLT   r1, r1, r0     ; 如果r0<r1，则r1 = r1-r0
        SUBGT   r0, r0, r1     ; 如果r0>r1，则r0 = r0-r1
        BNE     start          ; 如果r0!=r1，则跳转至start
stop                           ; 程序标号stop
        B       stop           ; 跳转到stop，进入死循环
        END                    ; 程序结束
```

- 程序结束时，r0和r1均为3

4.9.1 分支结构

- **例4.2：复杂分支程序**
  - 寄存器r0、r1和r2中有三个数，求其中的最大的一个，结果存储在r3中
  - **分析：**
    - 程序所需条件判断较多，且条件嵌套，而CPSR寄存器只有一个，因此指令不能够根据CPSR寄存器的标志位来进行正确的分支
    - 要实现正确的程序分支，可以通过跳转语句来跳到相应的程序段

4.9.1 分支结构

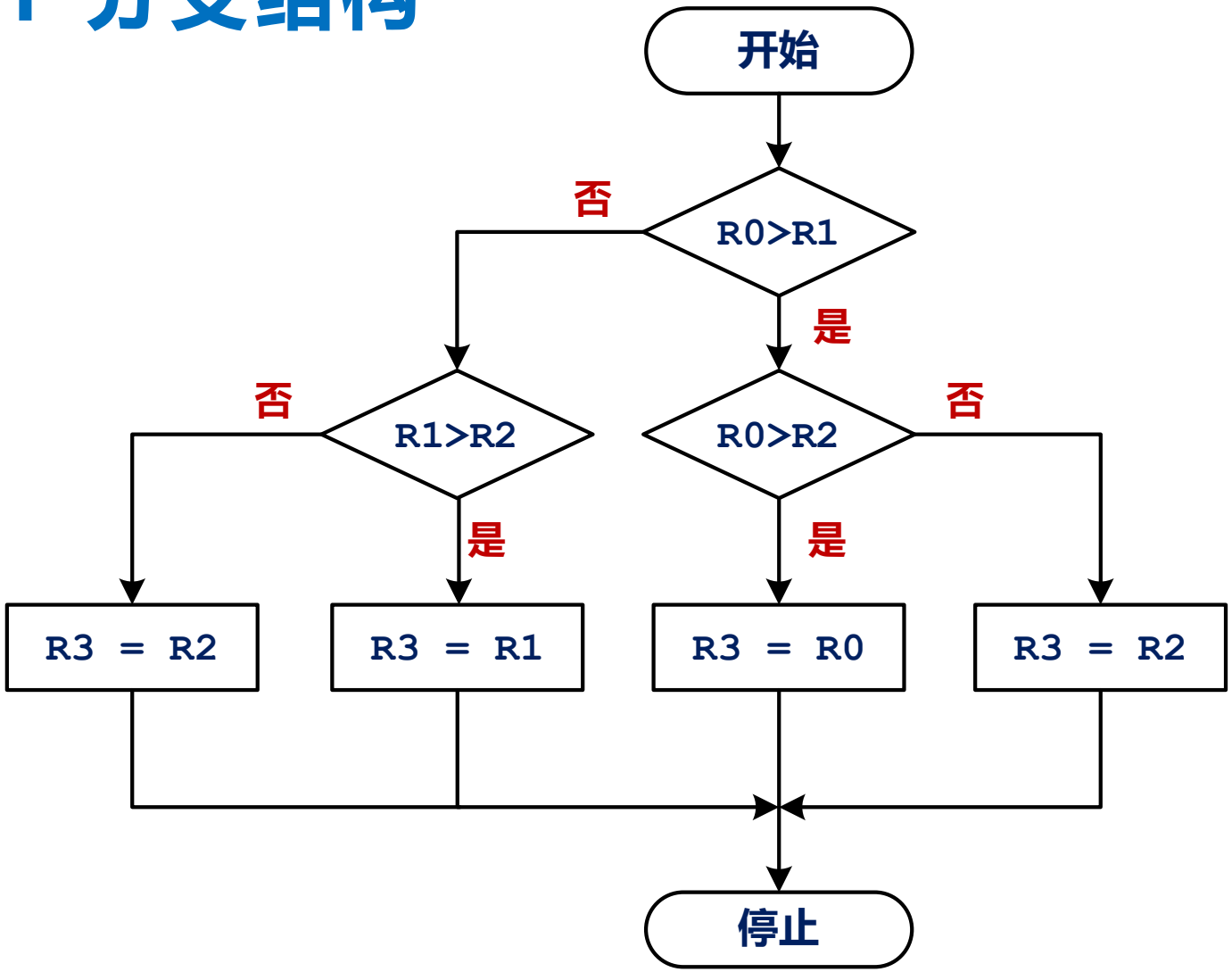


图4.7 复杂分支流程图

4.9.1 分支结构

```
        AREA    Exam2, CODE     ; 程序代码段开始
        ENTRY                   ; 程序入口
        MOV     R0, #10         ; r0 = 10
        MOV     R1, #30         ; r1 = 30
        MOV     R2, #20         ; r2 = 20
start                           ; 程序标号start
        CMP     R0, R1          ; 比较r0和r1
        BLE     lbl_a           ; 如果r0<=r1, 则跳转到lbl_a分支
        CMP     R0, R2          ; 比较r0和r2
        MOVGT   R3, R0          ; 如果r0>r2, 则r3 = r0
        MOVLE   R3, R2          ; 如果r0<=r2,则r3 = r2
        B       lbl_b           ; 跳转到分支结尾
lbl_a                           ; 程序标号lbl_a
        CMP     R1, R2          ; 比较r1和r2
        MOVGT   R3, R1          ; 如果r1>r2,则r3 = r1
        MOVLE   R3, R2          ; 如果r1<=r2,则r3 = r2
lbl_b                           ; 分支结尾, 程序标号lbl_b
        B       .               ; 原地跳转, 进入死循环
        END                     ; 程序结束
```

4.9.1 分支结构

- **例4.3：利用跳转表实现分支转移**
  - 寄存器R1、R2中有两个数：
    - 若R0为0，则求R1与R2的和
    - 若R0为1，则求R1与R2的差
  - 结果存储在R0中
  - 分析：
    - 跳转表实际上实现了高级语言的switch…case结构，在多case的情况下尤其有效

4.9.1 分支结构

- C语言代码

```
int DoAdd(int a, int b)
{
    return a + b;
}
int DoSub(int a, int b)
{
    return a - b;
}
```

```
void main()
{
    int R0 = 0;
    int R1 = 3;
    int R2 = 2;
    int (*arithfunc)();
    int switch(R0)
    {
        case 0:
            arithfunc = DoAdd;
            R0 = arithfunc(R1, R2);
            break;
        case 1:
            arithfunc = DoSub;
            R0 = arithfunc(R1, R2);
            break;
        default:
            break;
    }
    while(1);
}
```

4.9.1 分支结构

- 汇编语言实现方法：
  - 用汇编语言实现switch…case结构，需要一块代码区域专门存放跳转表，类似于C语言的函数指针
  - 根据R0的值，计算出函数指针应该指向跳转表的某一个地址，进而执行该函数

```
        AREA    Exam3, CODE     ; 程序代码段开始
num     EQU     2               ; 定义常量num = 2
        ENTRY                   ; 程序入口
start                           ; 程序标号start
        MOV     R0, #0          ; R0 = 0
        MOV     R1, #3          ; R1 = 3
        MOV     R2, #2          ; R2 = 2
        BL      subfun          ; 跳转到子程序subfun
        B       .               ; 原地跳转，进入死循环
subfun
        CMP     R0, #num        ; 比较R0和num
        MOVHS   PC, LR          ; 如果R0 >= num，则返回
        ADR     R3, JumpTable   ; R3 = JumpTable的基地址
        LDR     PC, [R3, R0, LSL #2]; PC←R3+R0*4，转到R0对应子函数
JumpTable
        DCD     FuncAdd         ; FuncAdd函数的地址
        DCD     FuncSub         ; FuncSub函数的地址
FuncAdd                         ; FuncAdd函数的标号
        ADD     R0, R1, R2      ; R0 = R1 + R2
        MOV     PC, LR          ; 返回
FuncSub                         ; FuncSub函数的标号
        SUB     R0, R1, R2      ; R0 = R1 - R2
        MOV     PC, LR          ; 返回
        END                     ; 程序结束
```

4.9.1 分支结构

■ **例4.4：中断向量表**

```
        AREA    Exam4, CODE, READONLY
        LDR     PC, Reset_Addr          ; 跳转到Reset_Addr对应的地址
        LDR     PC, Undefied_Addr       ; 类上
        LCR     PC, SWI_Addr            ; 类上
        LDR     PC, Prefetch_Addr       ; 类上
        LDR     PC, Abort_Add           ; 类上
        NOP                             ; 预留向量，什么都不做
        LDR     PC, IRQ_Addr            ; 类上
        LDR     PC, FIQ_Addr            ; 类上
; ************************************************
; 以下保存各个中断处理器程序的入口地址，即跳转表
Reset_Addr              DCD     Reset_Handler
Undefined_Addr          DCD     Undefined_Handler
SWI_Addr                DCD     SWI_Handler
Prefetch_Addr           DCD     Prefetch_Handler
Abort_Addr              DCD     Abort_Handler
                        DCD     0                       ; 预留向量
IRQ_Addr                DCD     IRQ_Handler
FIQ_Addr                DCD     FIQ_Handler
```

4.9.1 分支结构

■ **例4.4：中断向量表**

```
; 以下是中断处理程序，本例中不做任何处理，直接返回
; ************************************************
        IMPORT  __main ; 引入C运行时库的__main函数符号
Reset_Handler
        B  __main       ; 跳转到__main函数
Undefined_Handler
        MOV PC, LR      ; 不做处理，直接返回
SWI_Handler
        MOV PC, LR
Prefetch_Handler
        MOV PC, LR
Abort_Handler
        MOV PC, LR
IRQ_Handler
        MOV PC, LR
FIQ_Handler
        MOV PC, LR
        END
```

4.9.2 循环结构

- 高效的运算程序需要高效的循环结构，程序的复杂度就是用循环的次数来衡量的
  - ✧ 在嵌入式系统中，引导装载程序的作用之一就是把程序代码从扩展存储器复制到内存中
  - ✧ 要实现批量数据复制，就必须用到循环结构
  - ✧ 在循环中嵌套循环称为多重循环
  - ✧ 在一定条件下，为节省寄存器，可以将循环变量合并在一起使用

4.9.2 循环结构

- **例4.5：数据块复制**
  - ✧ 编程实现将从地址src开始的num个字的数据复制到地址dst去
  - ✧ 分析：
    - 首先将num个数按8个数一组进行块复制，然后将剩余的数逐个复制
    - 复制方法：设置循环变量，并令其递减，如果不为零，则跳转到循环体开头

```
        AREA    Exam5, CODE, READONLY
num     EQU     25                  ; 令num = 25
        ENTRY                       ; 程序入口
start                               ; 程序标号start
        LDR     R0, =src            ; R0 = src的地址
        LDR     R1, =dst            ; R1 = dst的地址
        MOV     R2, #num            ; R2 = num
Blockcopy                           ; 标号Blockcopy，循环体开始
        MOVS    R3, R2, LSR #3      ; R3 = R2>>3
        BEQ     wordcopy            ; 若R3 = 0，则跳转到字复制程序
        STMFD   SP!, {R4-R11}       ; 保存R4-R11到堆栈
copy                                ; 标号copy，块复制程序
        LDMIA   R0!,{R4-R11}        ; 将R0地址开始的8个字复制到寄存器R4-R11
        STMIA   R1!,{R4-R11}        ; 将R4-R11中的数复制到R1地址开始的8个字
        SUBS    R3, R3, #1          ; R3 = R3-1
        BNE     copy                ; 如果R3不为0，则跳转到标号copy
wordcopy                            ; 标号wordcopy，字复制程序
        ANDS    R2, R2, #7          ; R2 = R2&0x07，R2取末尾3位
        BEQ     Stop                ; 若R2 = 0，则跳转到标号Stop
CopyLoop                            ; 字复制程序循环体开始
        LDR     R3, [R0], #4        ; R3 = [R0], R0 = R0+4
        STR     R3, [R1], #4        ; [R1] = R3, R1 = R1+4
        SUBS    R2, R2, #1          ; R2 = R2-1
```

4.9.2 循环结构

- **例4.6：多重循环**

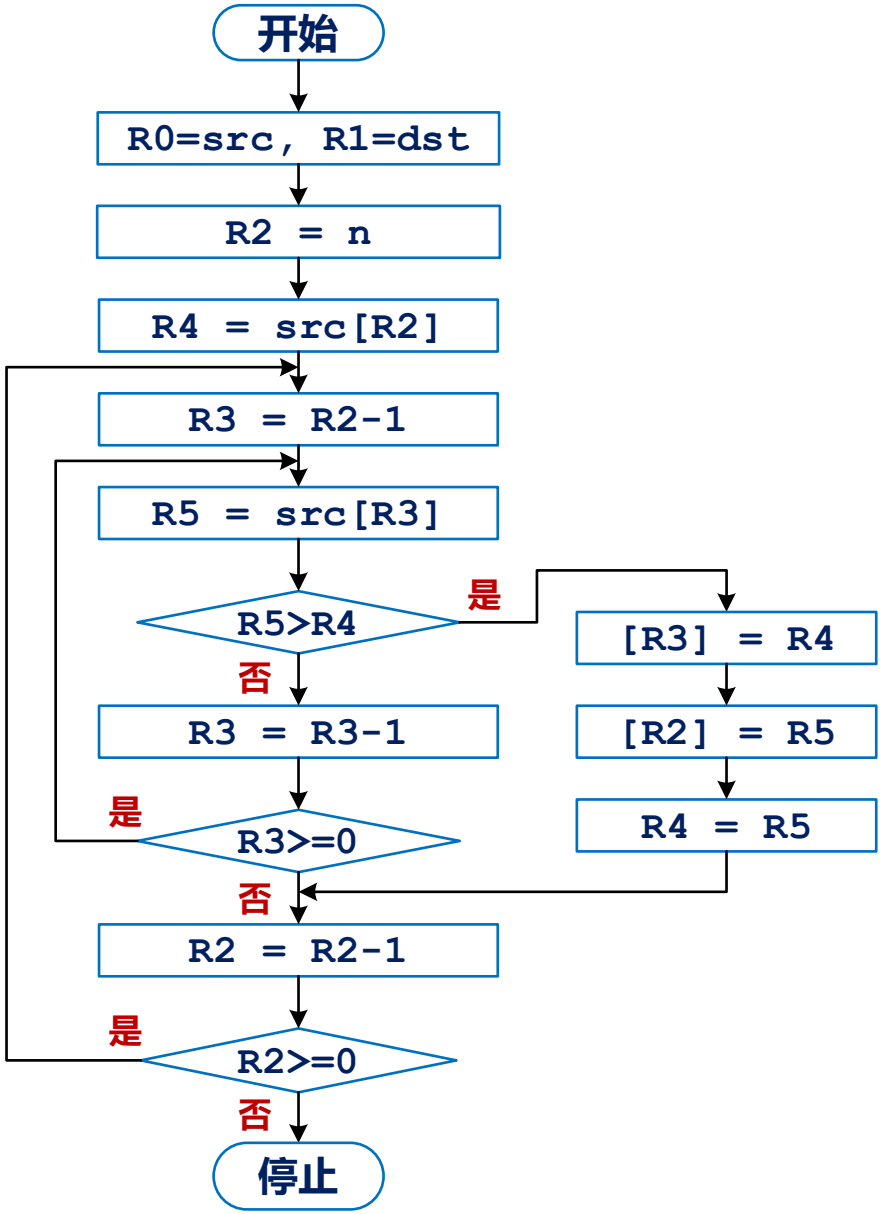
✧ 对n个数进行从小到大排序

✧ 分析：采用两重循环（冒泡法）

① 从第n个数开始，依次与前面的数比较，发现大的就与之交换

② 再用第n-1个数与后面的数进行比较，如此循环

4.9.2 循环结构


开始
R0=src, R1=dst
R2 = n
R4 = src[R2]
R3 = R2-1
R5 = src[R3]
R5>R4
是
[R3] = R4
否
R3 = R3-1
[R2] = R5
是
R3>=0
R4 = R5
否
R2 = R2-1
是
R2>=0
否
停止

```
        AREA    Exam6, CODE, READONLY
num     EQU     10                  ; 令num = 10
        ENTRY                       ; 程序入口
start                               ; 程序标号start
        LDR     R0, =src            ; R0 = src的地址
        MOV     R2, #num-1          ; R2 = num-1
LOOP1                       ; 标号Blockcopy，循环体开始
        LDR     R4, [R0,R2,LSL #2]; R4 = SRC [R2]
        SUBS    R3, R2, #1          ; R3 = R2-1
        LDRPL   R5, [R0,R3,LSL #2]   ; R5 = src [R3]
        CMP     R5, R4              ; 比较R5和R4
        STRGT   R4, [R0,R3,LSL #2]   ; 若R5>R4，则src [R3] = R4
        STRGT   R5, [R0,R2,LSL #2]   ; 若R5>R4，则src [R2] = R5
        MOVGT   R4, R5              ; 若R5>R4，则R4 = R5
        SUBS    R3, R3, #1          ; 若R3 = R3-1
        BPL     LOOP2               ; 若R3 >= 0，则跳转到LOOP2
        SUBS    R2, R2, #1          ; 若R2 = R2-1
        BHI     LOOP1               ; 若R2 >= 0，则跳转到LOOP1
        B       .                   ; 原地跳转，进入死循环
        AREA    DataArea, DATA, READWRITE   ; 数据段，可读写
src     DCD     6,1,9,8,4,5,0,7,3,2 ;预存数据，数据源
        END                         ; 程序结束
```

本章小结

ARM汇编伪操作

ARM汇编语言伪指令、符号、表达式

ARM汇编语言程序结构、子程序调用

C语言与汇编语言混合编程

ARM汇编语言设计实例