嵌入式系统原理及应用

第4章　ARM汇编语言程序设计

张帆

中南大学自动化学院

第4章 ARM汇编语言程序设计

- 4.1 GNU汇编器的平台无关伪指令
- 4.2 GNU汇编器支持的ARM伪指令
- 4.3 ARM 汇编语言的程序结构
- 4.4 汇编语言与C语言的混合编程
- 4.5 思考与练习

机器人

https://www.bilibili.com/video/BV1Uh41137Th/?spm_id_from=333.999.0.0

4.1 GNU汇编器的平台无关伪指令

GNU汇编器

GNU:维基百科上说:GNU，一个类UNIX的操作系统，由GNU计划推动，目标在于建立一个完全相容于UNIX的自由软件环境。这里GNU代表了一种平台，一类开发环境，更是一种汇编程序设计的语法格式。我们这里可以认为他代表一类组织，有自己的独特的程序设计规范，就是AT&T规范，一般称之为AT&T汇编，提到AT&T汇编，就应该知道这是另一种汇编语法格式，汇编指令是由处理器决定的。跟语法格式无关。

汇编 :指的是汇编语言。在汇编语言中，用助记符(Memoni)代替操作码，用地址符号(Symbol)或标号(Label)代替地址码。这样用符号代替机器语言的二进制码，就把机器语言变成了汇编语言。于是汇编语言亦称为符号语言。用汇编语言编写的程序，机器不能直接识别，要由一种程序将汇编语言翻译成机器语言，这种起翻译作用的程序叫汇编程序，汇编程序是系统软件中语言处理的系统软件。GNU有自己的一套独立的编译环境，所遵从的C语言、汇编语言等程序设计的语法格式也是不同的。

如，PC机上，对于x86系列处理器，有AT&T汇编和INTEL汇编两种语法格式。这两种格式的汇编，有指令大小写、操作数赋值方向、前缀、后缀、寻址方式等区别。因为是一类处理器，X86系列，指令是一样的。

4.1 GNU汇编器的平台无关伪指令

- 4.1.1　伪指令概念

没有对应的机器码，告诉汇编程序如何汇编的指令

所有汇编伪指令名称都是“.”开始，然后是字母（通常小写）

分为：符号定义伪指令，数据定义伪指令，汇编控制伪指令，杂项伪指令

4.1.2　符号定义伪指令

常见的符号定义伪指令有如下几种。

（1）用于定义全局变量的 .global和 .globl

（2）用于定义局部变量的.local

（3）用于对变量赋值的.set

（4）宏替换指令.equ

用于定义全局变量的 .global和 .globl

作用：定义一个全局符号（必须唯一），使得符号对连接器可见，为连接器ld使用，变为对整个工程可用的全局变量。

格式：.global symbol 等价于 .globl symbol。

写成两种形式是为了兼容其他的汇编器

用于定义局部变量的.local

作用：定义对外部不可见的局部符号，作用域仅在当前文件中

格式： .local symbol

用于对变量赋值的.set

作用：给一个全局变量或局部变量赋值

格式：.set symbol, expr

示例：

```
.set start, 0x40
.set start, 0x50
mov r1, #start ;  r1里面是0x50
```

宏替换指令.equ

和.equ的功能一样

格式：symbol .equ, expr

示例：

```
start .equ, 0x40
start .equ, 0x50
mov r1, #start ;r1里面是0x50
```

4.1.3　数据定义伪指令

一般用于为特定的数据分配存储单元，同时可完成已分配存储单元的初始化。

常见的数据定义伪指令有：

.byte,
.short,
.word,.long,
.quad,
.float,
.space,.skip
.string,.ascii,.asciz,
.rept

4.1.3　数据定义伪指令

①字节分配伪指令.byte

作用：在存储器中分配一个字节，用指定的数据对其初始化

格式：　label: .byte expr

label：程序标号
expr：可以是-128～255的数值，也可以是字符

示例：a: .byte #1　以C语言形式理解为char a=1;

4.1.3　数据定义伪指令

②双字节分配　-------》形式和字节分配相同只是把 .byte替换为 .short ；

③四字节分配　-------》换为.word或.long，四字节分配较为常见，因为32位系统较多。

④八字节分配　-------》换为.quad

4.1.3　数据定义伪指令

⑤单精度浮点变量：.float

作用：在存储器中分配四个字节，用指定的浮点数据对其初始化

格式：　label: .float expr

label：程序标号
expr：可以是4字节以内的浮点数值

示例：a: .float 1.11　以C语言形式理解为float a=1.11

4.1.3　数据定义伪指令

⑥连续存储区域 伪指令 .space　.skip

作用：分配一片连续的存储器区域并初始化,如没有填充值则用0填充

格式：label：　.space size, expr
或者　label：　.skip　size, expr

size指需要的连续内存大小（字节数）
expr为size指定字节位置的初始数值

示例：a: .space 8,0x1

以C语言形式为char a[8]={0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1};

4.1.3 数据定义伪指令

⑦字符串定义 .string .ascii .asciz

作用：都是定义一个字符串

格式：label: .string “str”
或 label: .ascii “str”
或 label: .asciz “str”

示例如下

a: .ascii “abc” //连续地址，分配了3字节
a: .asciz “abc” //字符串后自动加0，分配了4字节
a: .string “abc” //与.asciz相同，可变种
.string16，.string32，.string64）

4.1.3 数据定义伪指令

⑧重复 .rept

作用：重复执行后面的命令，以.rept开始，并以.endr结束

格式：

```
.rept count        /*count 表示重复次数*/
commands           /*重复的commands命令体*/
.endr
```

示例：

```
.rept 3
.long 0
.endr
```

相当于.long 0 这条指令执行3次

4.1.4　汇编控制伪指令

汇编控制伪操作用于控制汇编程序的执行流程，类似于C语言的控制语句，常用的汇编控制伪操作包括以下几条。

（1）.if
　　.else
　　.endif

（2）.macro
　　.endm
　　.exitm

（1） .if .else .endif

语法格式：根据条件的成立与否决定是否执行某个指令序列。

- 当.if后面的逻辑表达式为真，则执行.if后的指令序列，否则执行.else后的指令序列。
- .else及其后指令序列可以没有，此时，当IF后面的逻辑表达式为真，则执行指令序列，否则继续执行后面的指令。
- 使用.endif对控制语句结束

.if、.else、.endif 伪指令可以嵌套使用。

语法格式如下：

```
.if  logical-expressing
…
.else
…
.endif
```

logical-expression：用于决定指令执行流程的逻辑表达式。

（1） .if .else .endif

使用说明：当程序中有一段指令需要在满足一定条件时执行，使用该指令。

该操作还有另一种形式。

```
.if logical-expression
    Instruction
.else logical-expression2
    Instructions
.else logical-expression3
    Instructions
.endif
```

.else形式避免了if-else形式的嵌套，使程序结构更加清晰、易读。

（2）　.macro　.endm　.exitm

- .macro　.endm伪指令可以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；而.exitm用于退出当前宏指令。
- 宏指令可以使用一个或多个参数，当宏操作被展开时，这些参数被相应的值替换。
- 类似于子程序，但使用子程序需要保护现场，增加了系统的开销。所以代码较短且需要传递的参数较多时，可以用宏代替子程序
- 包含在.macro和.endm之间的指令序列称为宏定义体。在宏定义体的第一行应该声明宏的原型（包含宏名、所需要的参数），然后就可以在汇编程序中通过宏名来调用该指令序列。在源程序被编译时，编译器将宏调用展开，用宏定义中的指令序列代替程序中的宏调用，并将实际参数值传递给宏定义中的形式参数。

（2）　.macro　.endm　.exitm

语法格式：

.macro macroname macargs …
…code…
.endm

macroname：所定义的宏的名称
macargs：宏指令的参数，类似于函数传参

（2）　.macro　.endm　.exitm

示例：

```
.macro sum from=0, to=5
.long   \from
.if     \to-\from
sum     (\from+1),\to
.endif
.endm
```

程序中使用“sum 0，5”进行宏的使用，上述示例要实现的功能是

```
.long 0
.long 1
.long 2
.long 3
.long 4
.long 5
```

4.1.5 杂项伪操作

（1）.align 用于使程序当前位置满足一定的对齐方式。

（2）.section 用来定义一个段的伪指令

（3）.data 用来定义一个数据段

（4）.text 用来定义一个代码段

（5）.include 用于包含一个头文件

（6）.arm定义以下代码使用ARM指令集编译。

（7）.code32作用同.arm

（8）.code16作用同.thumb

（9）.thumb定义以下代码使用Thumb指令集编译。

（10）.extern用于定义一个外部符号，用于兼容性其他汇编，一般被忽略

（11）.weak用于定义一个符号是弱符号。如果没有定义，则忽略，不会报错

（12）.end代表汇编程序的结束

（1）.align

.align可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。

语法格式：.align abs-expr

abs-expr是对齐表达式。表达式的值用于指定对齐方式，可能的取值为2的幂，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。

示例：

.align 2　　；4字节对齐，为什么是4？2^2

.string “abcde”

声明后面的字符串的对齐方式为4字节倍数对齐，这个字符串会占用8字节的存储空间

（2）.section （3） .data （4）.text

（2）.section用于定义一个段

一个GNU的源程序至少需要一个代码段，大的程序可以包含多个代码段和数据段

语法格式：.section sectionname

sectionname是所定义段的段名

（3）.data用于定义一个数据段

语法格式：.data subsectionname

subsectionname是所定义数据段的段名

（4）.text用于定义一个指令段

语法格式：.text subsectionname

subsectionname是所定义指令段的段名

（5）.include用于包含一个头文件
语法格式：.include “file”
示例：.include “s5pc100.h”
类似于C语言中的#include的功能

（6）.arm定义以下代码使用ARM指令集编译，等价于.code32
语法格式：
.arm
;code

（9）.thumb定义以下代码使用Thumb指令集编译，等价于.code16
语法格式：
.thumb
;code

（10）.extern用于定义一个外部符号，用于兼容性其他汇编，一般被忽略

语法格式：.extern symbol

symbol：要引用的符号名称，区分大小写

（11）.weak用于定义一个符号是弱符号。如果没有定义，则忽略，不会报错

语法格式：.weak symbol

symbol：要声明的符号名称

初始化了的全局变量为强符号
未初始化的全局变量为弱符号

1、不允许强符号被多次定义
2、如果一个符号有文件定义弱符号，也有文件定义强符号，那么选择强符号
3、如果一个符号在所有文件中都是弱符号，那么选择占用空间最大的一个。

（12）.end代表汇编程序的结束

语法格式：.end

表示汇编的结束

每一个汇编源文件必须以END结束

4.1 GNU汇编器支持的ARM伪指令

- GNU汇编器支持ARM伪指令，这些伪指令在汇编阶段被翻译成ARM或者Thumb（或Thumb-2）指令（或指令序列）。
- ARM伪指令包含ADR、ADRL、LDR等。

4.2.1　ADR伪指令

（1）语法格式

ADR伪指令为小范围地址读取伪指令。ADR伪指令将基于PC相对偏移地址或基于寄存器相对偏移地址值读取到寄存器中，当地址值是字节对齐时，取值范围为-255～255，当地址值是字对齐时，取值范围为-1020～1020。当地址值是16字节对齐时其取值范围更大。

语法格式：ADR <register> , <label or exper>

register，目标寄存器

label，基于PC或具有寄存器的表达式。

在编译源程序时，汇编器首先计算出当前PC到exper的偏移值#offset_to_exper,然后会用一条ADD或者SUB指令来替换这条伪指令,

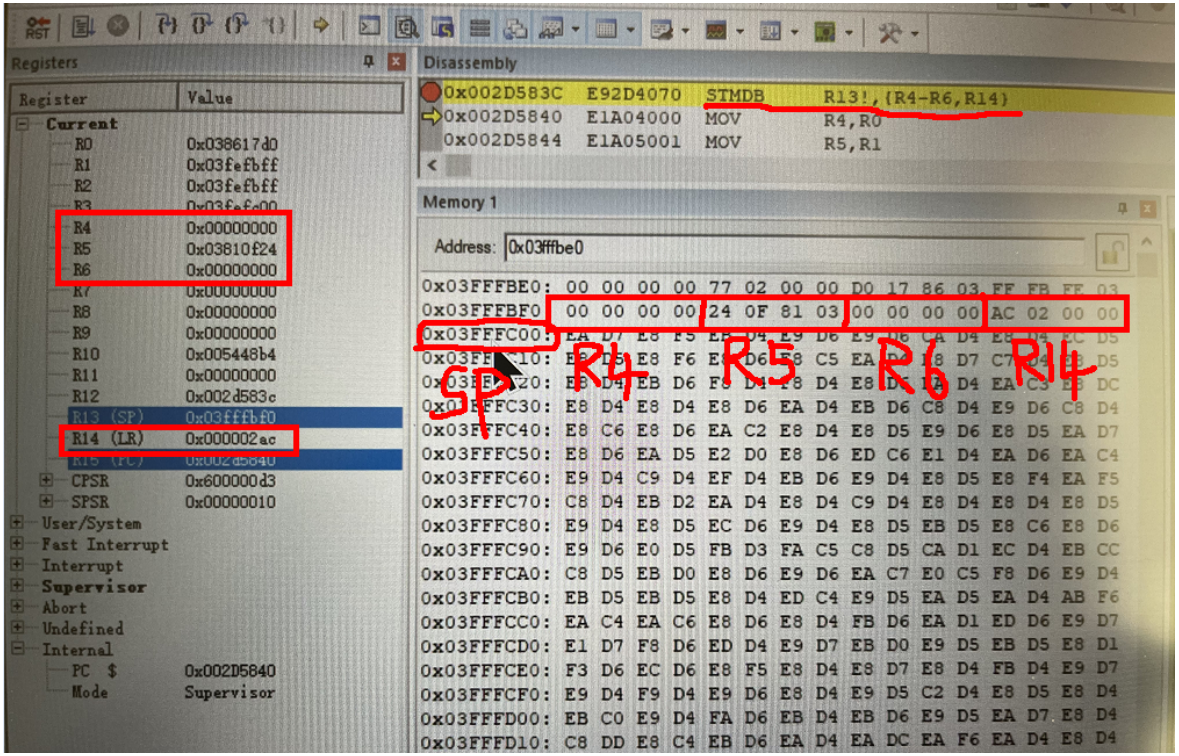
例如: ADD register,PC,#offset_to_exper。

注意，标号exper与指令必须在同一代码段。

4.2.1　ADR伪指令

示例

```
start MOV r0,#10 ;
ADR r4, start ;
```



4.2.2　ADRL伪指令

（1）语法格式

- ADRL伪指令为中等范围地址读取伪指令。ADRL伪指令将基于PC相对偏移的地址或基于寄存器相对偏移的地址值读取到寄存器中，当地址值是字节对齐时，取值范围为-64K～64KB；当地址值是字对齐时，取值范围为-256K～256KB。当地址值是16字节对齐时，其取值范围更大。在32位的Thumb-2指令中，地址取值范围到达-1～1MB。

语法格式：ADRL <register> , <label>

register，目标寄存器

label，基于PC或具有寄存器的表达式。

示例

```
start MOV r0,#10 ;
ADRL r4,start+60000 ;
```

4.2.2 ADRL伪指令

- ADRL伪指令与ADR伪指令相似，用于将基于PC相对偏移的地址或基于寄存器相对偏移的地址值读取到寄存器中。
- 所不同的是，ADRL伪指令比ADR伪指令可以读取更大范围的地址。这是因为在编译阶段，ADRL伪指令被编译器换成两条指令。即使一条指令可以完成该操作，编译器也将产生两条指令，其中一条为多余指令。
- 如果汇编器不能在两条指令内完成操作，将报告错误，中止编译
- 在编译源程序时，汇编器会用两条合适的指令来替换这条伪指令。

例如：
```
ADD register,PC,offset1
ADD register,register,offset2
```

- 与ADR相比，它能读取更大范围的地址。
- 注意，标号exper与指令必须在同一代码段。

4.2.3　LDR伪指令

（1）语法格式

- LDR伪指令为大范围的地址读取,用于加载32位的立即数或一个地址值到指定寄存器。

语法格式：LDR〈register〉，〈=expr〉

register，目标寄存器

=expr，32位常量表达式

在汇编编译源程序时，LDR伪指令被编译器替换成一条合适的指令。若加载的常数未超出MOV或MVN的范围，则使用MOV或MVN指令代替该LDR伪指令，否则汇编器将常量放入文字池，并使用一条程序相对偏移的LDR指令从文字池读出常量。

4.2.3 LDR伪指令

（2）示例

① 将常数0xff0读到R3中。

```
LDR R3,=0xff0 ;
相当于下面的ARM指令：
MOV R3,#0xff0
```

② 将常数0xfff读到R1中。

```
LDR R1,=0xfff ;
相当于下面的ARM指令：
LDR R1,[pc,offset_to_litpool]
…
litpool DCD 0xfff
```

③ 将place标号地址读入R1中。

```
LDR R2,=place ;
相当于下面的ARM指令：
LDR R2,[pc,offset_to_litpool]
…
litpool DCD place
```

DCD：数据定义（ Data Definition ）伪指令一般用于为特定的数据分配存储单元，同时可完成已分配存储单元的初始化。

4.3 ARM 汇编语言的语句格式

在汇编语言程序设计中，经常使用各种符号代替地址（addresses）、变量（variables）和常量（constants）等，以增加程序的灵活性和可读性。尽管符号的命名由编程者决定，但并不是任意的，必须遵循以下的约定。

1. 符号区分大小写，同名的大、小写符号会被编译器认为是两个不同的符号。
2. 符号在其作用范围内必须惟一。
3. 自定义的符号名不能与系统的保留字相同。其中保留字包括系统内部变量（built in variable）和系统预定义（predefined symbol）的符号。
4. 符号名不应与指令或伪指令同名。如果要使用和指令或伪指令同名的符号要用双斜杠“//”将其括起来，如“//ASSERT//”。
   （注意：//并不是符号名的一部分）
1. 局部标号以数字开头，其他的符号都不能以数字开头。

4.3 ARM 汇编语言的语句格式

1. 变量（variable）

程序中的变量是指其值在程序的运行过程中可以改变的量。ARM（Thumb）汇编程序所支持的变量有三种。

- 数字变量（numeric）。
- 逻辑变量（logical）。
- 字符串变量（string）。

数字变量用于在程序的运行中保存数字值，但注意数字值的大小不应超出数字变量所能表示的范围。

逻辑变量用于在程序的运行中保存逻辑值，逻辑值只有两种取值情况：真（{TURE}）和假（{FALSE}）。

字符串变量用于在程序的运行中保存一个字符串，注意字符串的长度不应超出字符串变量所能表示的范围。

在ARM（Thumb）汇编语言程序设计中，可使用.global伪指令声明全局变量，使用.local伪指令声明局部变量，可使用.set对其进行初始化。

4.3 ARM 汇编语言的语句格式

2．常量（constants）

程序中的常量是指其值在程序的运行过程中不能被改变的量。ARM（Thumb）汇编程序所支持的常量有数字常量、逻辑常量和字符串常量。

数字常量一般为32位的整数，当作为无符号数时，其取值范围为0～2^32−1，当作为有符号数时，其取值范围为−2^31～2^31−1。汇编器认为−n和2^32−n是相等的。

逻辑常量只有两种取值情况，真或假。

字符串常量为一个固定的字符串，一般用于程序运行时的信息提示。

4.3 ARM 汇编语言的语句格式

3. 程序中的变量代换

汇编语言中的变量可以作为一整行出现在汇编程序中，也可以作为行的一部分使用。

如果在数字变量前面有一个代换操作符“$”，编译器会将该数字变量的值转换为十六进制的字符串，并将该十六进制的字符串代换“$”后的数字变量。

如果在逻辑变量前面有一个代换操作符“$”，编译器会将该逻辑变量代换为它的取值（真或假）

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。

如果程序中需要字符“$”，则可以用“$$”来表示。汇编器将不进行变量替换，而是将“$$”作为“$”。

4.3 ARM 汇编语言的语句格式

4. 程序标号（label）

在ARM汇编中，标号代表一个地址，段内标号的地址在汇编时确定，而段外标号地址值在链接时确定。

根据标号的生成方式，程序标号分为以下三种。

- 程序相关标号（Program-relative labels）。
- 寄存器相关标号（Register-relative labels）。
- 绝对地址（Absolute address）。

4.3 ARM 汇编语言的语句格式

（1）程序相关标号

程序相关标号指位于目标指令前的标号或程序中的数据定义伪操作前的标号。这种标号在汇编时将被处理成PC值加上或减去一个数字常量。它常用于表示跳转指令的目标地址或代码段中所嵌入的少量数据。

（2）寄存器相关地址

这种标号在汇编时将被处理成寄存器的值加上或减去一个数字常量。它常被用于访问数据段中的数据。

（3）绝对地址

绝对地址是一个32位的数字量，使用它可以直接寻址整个内存空间。

4.3 ARM 汇编语言的语句格式

5. 局部标号

局部标号是一个0～99之间的十进制数字，可重复定义。局部标号后面可以紧接一个通常表示该局部变量作用范围的符号。局部变量的作用范围为当前段。

局部标号在子程序或程序循环中常被用到，也可以配合宏定义伪操作（.MACRO和.ENDM）来使程序结构更加合理。

在同一个段中，可以使用相同的数字命名不同的局部变量。默认情况下，汇编器会寻址最近的变量。也可以通过汇编器命令选项来改变搜索顺序。

4.4 ARM 汇编语言的程序结构

4.4.1 汇编语言的程序格式

在 ARM（Thumb）汇编语言程序中可以使用.section 来进行分段，其中每一个段用段名或者文件结尾为结束，这些段使用默认的标志，如 a 为允许段，w 为可写 段，x 为执行段。

在一个段中，可以定义下列的子段：

.text

.data

.bss

.sdata

.sbss

4.4 ARM 汇编语言的程序结构

由此我们可知道，段可以分为代码段、数据段及其他存储用的段，

.text（正文 段）包含程序的指令代码；

.data(数据段)包含固定的数据，如常量、字符串；

.bss （未初始化数据段）包含未初始化的变量、数组等，

当程序较长时，可以分割为多个代码段和数据段，多个段在程序编译链接时最终形成一个可执行的映像文件。

汇编语言的程序格式的例子如下：

```
 .section.data
< initialized data here>
.section .bss
< uninitialized data here>
.section .text
.globl _start
_start:
<instruction code goes here>
```

```
.text
.global _start
.set k,0x40
_start:
mov r0,#k
ldr r1,=0x20009000
str r0, [r1]
mov pc, lr
.end
```

4.4 ARM 汇编语言的程序结构

4.4.2 汇编语言子程序调用

- 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。在程序中，使用指令“BL子程序”名即可完成子程序的调用。

- 该指令在执行时完成如下操作：将子程序的返回地址存放在连接寄存器LR中，同时将程序计数器PC指向子程序的入口点。当子程序执行完毕需要返回调用处时，只需要将存放在LR中的返回地址重新拷贝给PC即可。

- 在调用子程序的同时，也可以完成参数的传递和从子程序返回运算的结果，通常可以使用寄存器R0～R3完成。

4.4 ARM 汇编语言的程序结构

以下是使用BL指令调用子程序的汇编语言源程序的基本结构：

```
.text
.global _start
_start:
ldr r0，=0x3ff5000
ldr r1，0xff
str r1，[r0]
ldr r0，=0x3ff5008
ldr r1，0x01
str r1，[r0]
bl print_text
…
print_text：
…
mov pc，lr
…
```

4.4 ARM 汇编语言的程序结构

4.4.3 过程调用标准AAPCS

- 为了使不同编译器编译的程序之间能够相互调用，必须为子程序间的调用规定一定的规则。AAPCS就是这样一个标准。
- 所谓AAPCS，其英文全称为Procedure Call Standard for the ARM Architecture (AAPCS)，即ARM体系结构过程调用标准。
- 它是ABI（Application Binary Interface (ABI) for the ARM Architecture (base standard) [BSABI]）标准的一部分。

4.4 ARM 汇编语言的程序结构

ARM寄存器使用规则（AAPCS）

- 子程序间通过寄存器R0、R1、R2、 R3来传递参数。如果参数多于4个，则多出的部分用堆栈传递。被调用的子程序在返回前无需恢复寄存器R0-R3的内容。

- 在子程序中，使用寄存器R4-R11来保存局部变量.

  如果在子程序中使用到了寄存器R4-R11中的某些寄存器，子程序进入时必须保存这些寄存器的值，在返回前必须恢复这些寄存器的值；

  对于子程序中没有用到的寄存器则不必进行这些操作。在Thumb程序中，通常只能使用寄存器R4-R7来保存局部变量。

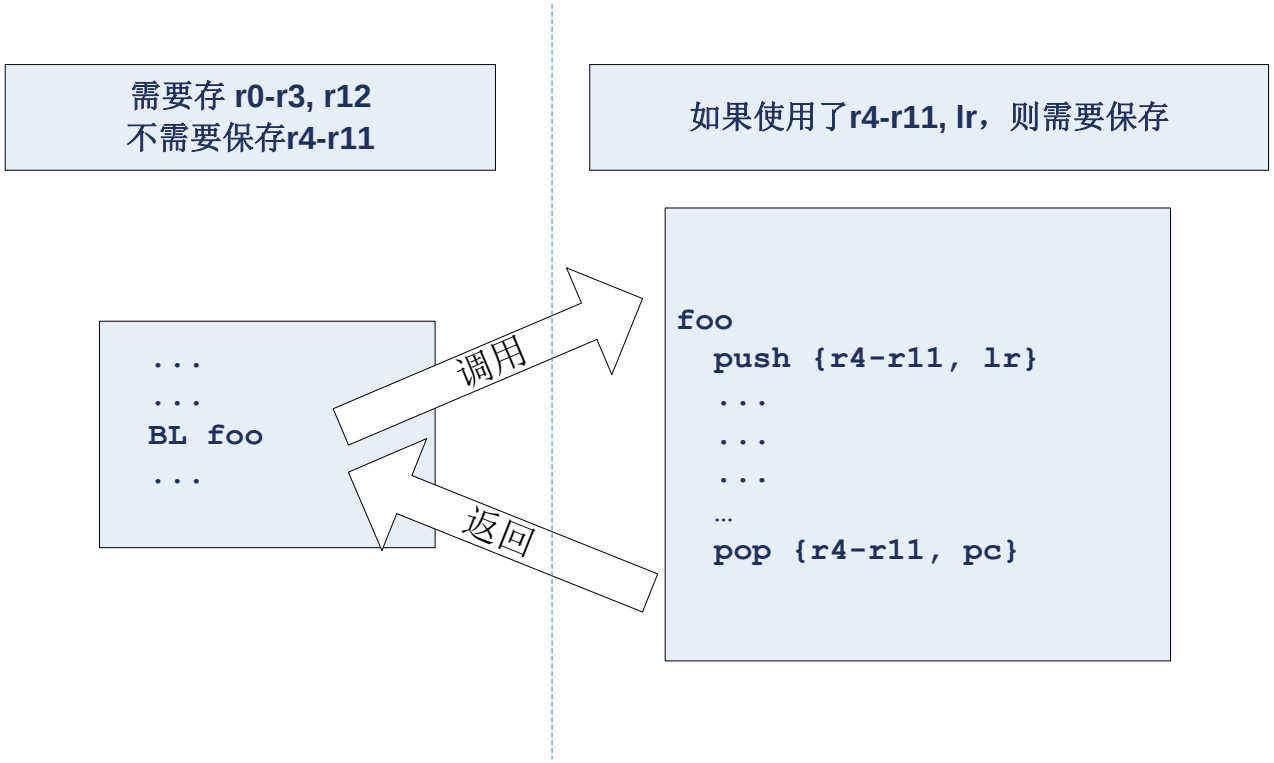
4.4 ARM 汇编语言的程序结构

ARM寄存器使用规则（AAPCS）

- 寄存器R12用作子程序间scratch寄存器（常用于保存SP，在函数返回时使用该寄存器出栈），记作ip。在子程序间的连接代码段中常有这种使用规则。
- 寄存器R13用作数据栈指针，记作sp。在子程序中寄存器R13不能用作其他用途。寄存器sp在进入子程序时的值和退出子程序时的值必须相等。
- 寄存器R14称为连接寄存器，记作lr。它用于保存子程序的返回地址。如果在子程序中保存了返回地址，寄存器R14则可以用作其他用途。
- 寄存器R15是程序计数器，记作pc。它不能用作其他用途。

4.4 ARM 汇编语言的程序结构

ARM寄存器在函数调用过程中的保护规则，如图所示：



4.4 ARM 汇编语言的程序结构

4.4.4 汇编语言程序设计举例

通过组合使用条件执行和条件标志设置，可以简单地实现分支语句的功能，但不需要任何分支指令。这样可以改善性能，因为分支指令会占用较多的周期数；同时这样做也可以减小代码尺寸，提高代码密度。

下面是一段C语言程序，该程序实现了著名的Euclid最大公约数算法。

```
int gcd(int a, int b)
{
      while (a != b)
        {
           if (a > b)
                   a = a - b;
           else
                   b = b - a;
        }
      return a;
}
```

4.4 ARM 汇编语言的程序结构

BEQ:branch if equal
BLT:branch if less than
BGT:branch if greater than
BNE:branch not equal

用ARM汇编语言重写来重写这个例子，如下所示。

程序1

```
gcd     CMP     r0, r1
        BEQ     end
        BLT     less
        SUB     r0, r0, r1
        B       gcd
less
        SUB     r1, r1, r0
        B       gcd
End
```

充分地利用条件执行修改左面的例子，得到程序2

【程序2】

```
gcd
        CMP     r0, r1
        SUBGT   r0, r0, r1
        SUBLT   r1, r1, r0
        BNE     gcd
```

【程序1】仅使用了分支指令，【程序2】充分利用了ARM指令条件执行的特点，仅使用了4条指令就完成了全部算法。这对提供程序的代码密度和执行速度十分有帮助。

事实上，分支指令十分影响处理器的速度。每次执行分支指令，处理器都会排空流水线，重新装载指令。

4.5 汇编语言与C语言的混合编程

在C代码中实现汇编语言的方法有内联汇编和内嵌汇编两种，使用它们可以在C程序中实现C语言不能完成的一些工作。

内联汇编：在C程序中直接编写汇编程序段而形成一个语句块

内嵌汇编：编写在C程序外的单独汇编程序，可以像函数那样被C程序调用。

例如，在下面几种情况中必须使用内联汇编或嵌入型汇编。

（1）程序中使用饱和算术运算（Saturating Arithmetic），如SSAT16和USAT16指令。

（2）程序中需要对协处理器进行操作。

（3）在C程序中完成对程序状态寄存器的操作。

一个完整的ARM应用程序通常是C/C++语言和汇编程序的混合。

4.5 汇编语言与C语言的混合编程

一个ARM工程应由多个文件组成，包括汇编语言源文件(.s)、C/C++语言源文件(.c)和C/C++头文件(.h)等。

编译器：汇编文件 .s

汇编器：目标文件 .o

链接器：可执行文件.elf

编程过程中，汇编与C的衔接？

（AAPCS，内联汇编，互相调用（内嵌））

4.5.1 GNU内联汇编

“会被修改的部分”表示你已对其中列出的寄存器中的值进行了改动。因此我们需要把那些没有在输出/输入寄存器中的部分列出，但是在汇编语句中明确使用到或隐含使用到的寄存器明列在这个部分。

1．内联汇编语法

（1）格式。 GNU 风格的 ARM 内联汇编语言的格式如下：

```
asm volatile ("asm code"
              : output
              : input
              : changed );
```

举例
https://blog.csdn.net/qq_44823010/article/details/126198592

必须以“;”结尾，不管有多长对 C 都只是一条语句。

volatile是一个特征修饰符（type specifier）.volatile的作用是作为指令关键字，确保本条指令不会因编译器的优化而省略，且要求每次直接读值。

4.5.1 GNU内联汇编

（2）asm 内嵌汇编关键字。

volatile：告诉编译器不要优化内嵌汇编，如果想优化可以不加。

（3）ANSI C 规范的关键字。

__asm__
__volatile__ //前面和后面都有两个下画线，它们之间没有空格

如果后面部分没有内容，“：”可以省略，前面或中间的不能省略“：”
没有 asm code 也不可以省略“”””
没有 changed 必须省略“：”

```
__asm__ __volatile__("cli": : :"memory")
```

4.5.1 GNU内联汇编

2．汇编代码

汇编代码必须放在一个字符串内，但字符串中间不能按回车键换行，可以写成多个字符串，只要字符串之间不加任何符号，编译完后就会变成一个字符串。

```
"mov r0,r0\n\t" //指令之间必须要换行，\t 可以不加，只是为了在汇编文件中
                  的指令格式对齐
"mov r1,r1\n\t"
"mov r2,r2"
```

字符串内不是只能放指令，可以放一些标签、变量、循环、宏等，还可以把内嵌汇编放在 C 函数外面，用内嵌汇编定义函数、变量、段等，总之就跟直接在写汇编文件一样。

编译器不检查 asm code 的内容是否合法，直接交给汇编器

4.5.1 GNU内联汇编

3. output（ASM --> C）和 input（C --> ASM） （传参）

（1）指定输出值。

```
__asm__ __volatile__ (
        "asm code"
        :“constraint”（variable）
);
```

① constraint 定义 variable 的存放位置：

r——使用任何可用的通用寄存器；

m——使用变量的内存地址。

② output 修饰符：

+——可读可写；

=——只写；

&——该输出操作数不能使用输入部分使用过的寄存器，只能用“+&”或“=&”的方式使用。

4.5.1 GNU内联汇编

3. output（ASM --> C）和 input（C --> ASM） （传参）
（2）指定输入值。

```
__asm__ __volatile__ (
        "asm code"
        :
        :“constraint”（variable / immediate）
);
```

constraint 定义 variable / immediate 的存放位置：

r——使用任何可用的通用寄存器（变量和立即数都可以）；
m——使用变量的内存地址（不能用立即数）；
i——使用立即数（不能用变量）；

4.5.1 GNU内联汇编

（3）使用占位符。

在汇编语句中，数字加前缀 %，如 %0、%1 等，表示需要使用寄存器的样板操作数。

```
int a = 100,b = 200;
int result;
__asm__ __volatile__ (
        “mov %0,%3\n\t”  //mov r3,#123  %0 代表 result，%3 代表 123（编译器会自动加 # 号）
        “ldr r0,%1\n\t”  //ldr r0,[fp, #-12] %1 代表 a 的地址
        “ldr r1,%2\n\t”  //ldr r1,[fp, #-16] %2 代表 b 的地址
        “str r0,%2\n\t”  //str r0,[fp, #-16]因为%1 和%2 是地址，所以只能用 ldr 或 str 指令
        “str r1,%1\n\t”  //str r1,[fp, #-12]如果用错指令，编译时不会报错，要到汇编时才会报错
      : “=r”(result), “+m”(a), “+m”(b) //out1 是%0，out2 是%1，... , outN 是%N-1
      : “i”(123)       //in1 是%N，in2 是%N+1，... );
```

4.5.1 GNU内联汇编

（4）引用占位符。

```
int num = 100;
__asm__ __volatile__ (
        "add %0,%1,#100\n\t"
        : "=r"(a)
        : "0"(a)          //"0"是零，即%0，引用时不可以加 %，只能 input 引
                            用 output
);                        //引用是为了更能分清输出、输入部分
```

4.5.1 GNU内联汇编

5）&修饰符。

```
int num;
__asm__ __volatile__ (    //mov r3, #123 //编译器自动加的指令
        "mov %0,%1\n\t"  //mov r3,r3 //输入和输出使用相同的寄存器
        : "=r"(num)
        : "r"(123)
);

int num;
__asm__ __volatile__ (  //mov r3, #123
        "mov %0,%1\n\t" //mov r2,r3 //加了&后输入和输出的寄存器不一样了
        : "=&r"(num)     //mov r3,r2，编译器自动加的指令
        : "r"(123)
);
```

4.5.1 GNU内联汇编

4. 内联编程示例

```
#include <stdio.h>
unsigned long ByteSwap(unsigned long val)
{ int ch;
asm volatile (
"eor r3, %1, %1, ror #16\n\t"
"bic r3, r3, #0x00ff0000\n\t"
"mov %0, %1, ror #8\n\t"
"eor %0, %0, r3, lsr #8"
: "=r" (val)
: "0"(val)
: "r3"
);
}
```

```
int main(void)
{
unsigned long test_a = 0x1234,result;
result = ByteSwap(test_a);
printf("Result:%d\r\n", result);
return 0;
}
```


result=0x41200000?

4.5.2 C和汇编的混合编程（内嵌）

相互调用时，注意遵守AAPCS规则。

1．从 C 程序中调用汇编语言 （例子）

该段代码实现了将一个字符串复制到另一个字符串。

```
#include <stdio.h>
extern void strcopy(char *d, const char *s);
int main()
{
const char *srcstr = "First string - source ";
char dststr[] = "Second string - destination ";
printf("Before copying:\n");
printf(" %s\n %s\n",srcstr,dststr);
strcopy(dststr,srcstr);
printf("After copying:\n");
printf(" %s\n %s\n",srcstr,dststr);
return(0);
}
```

4.5.2 C和汇编的混合编程（内嵌）

下面为调用的汇编程序。

```
.global strcopy
strcopy:            ; R0 指向目的字符串 ;R1 指向源字符串
LDRB R2, [R1],#1    ; 加载字节并更新源字符串指针地址
STRB R2, [R0],#1    ; 存储字节并更新目的字符串指针地址
CMP R2, #0          ; 判断是否为字符串结尾
BNE strcopy         ; 如果不是，程序跳转到 strcopy 继续复制
MOV pc,lr           ;程序返回
```

注意：R0,R1，R2，R3的传参作用

4.5.2 C和汇编的混合编程（内嵌）

2．从汇编语言调用 C 程序 （例子）

下面的子程序段定义了 C 语言函数。

```
int g(int a, int b, int c, int d, int e)
{
return a + b + c + d + e;
}
```

下面的程序段显示了汇编语言调用。假设程序进入 f 时，R0 中的值为 i。

```
int f(int i) { return g(i, 2*i, 3*i, 4*i, 5*i); }
```

4.5.2 C和汇编的混合编程（内嵌）

假设程序进入 f 时，R0 中的值为 i。

```
.text
.global _start
_start:
STR lr, [sp, #-4]!        // 保存返回地址lr
ADD R1, R0, R0            // 计算 2*i(第 2 个参数)
ADD R2, R1, R0            // 计算 3*i(第 3 个参数)
ADD R3, R1, R2            // 计算 5*i
STR R3, [sp, #-4]!        // 第 5 个参数通过堆栈传递
ADD R3, R1, R1            // 计算 4*i(第 4 个参数)
BL g                      // 调用 C 程序
ADD sp, sp, #4            // 从堆栈中删除第 5 个参数
.end
```

注意：R0,R1，R2，R3以及堆栈的传参作用

小结

本章介绍了ARM汇编程序设计的过程与方法，包括

GNU汇编器的平台无关伪指令
GNU汇编器支持的ARM伪指令
ARM 汇编语言的程序结构
汇编语言与C语言的混合编程

等内容。

这些内容是嵌入式编程的基础，希望读者掌握。

思考与练习

1. 在ARM汇编中如何定义一个全局的数字变量？
2. ADR和LDR的用法有什么区别？
3. AAPCS中规定的ARM寄存器的使用规则？
4. 什么是内联汇编？什么是嵌入型汇编？两者之间的区别是什么？
5. 汇编代码中如何调用C代码中定义的函数？

谢谢！

70