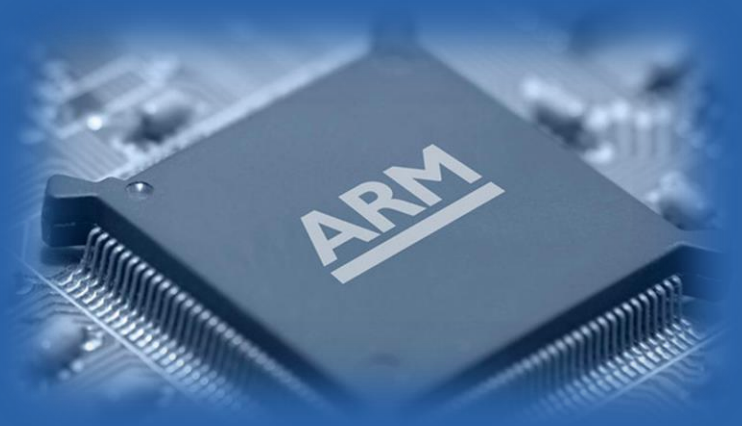
仪器科学与光电工程学院
School of Instrumentation Science and Opto-electronics Engineering

微机原理与接口技术

第八讲 ARM汇编程序设计




```
bl   __isoc99_scanf
ldr r2, [sp, #12]
ldr r3, [sp, #8]
cmp r2, r3
bgt .L7
ldrlt   r1, [sp, #4]
ldrge   r1, [sp, #4]
sublt   r1, r1, #1
strlt   r1, [sp, #4]
```

4.11 伪指令——LDR

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

理论上这里加载的地址为任意32位数。但也必须注意，文字池中存放该32位立即数的地址，与LDR伪指令的地址偏差必须；（1）ARM或32位Thumb指令±4kB；（2）16位thumb指令的寻址范围：0~1kB；

LDR　目标寄存器, = 表达式/标号

目标寄存器 ← 表达式

地址表达式expr的取指范围为任意值

4.11 伪指令——LDR

应用示例（源程序）：

```
190             LDR     R0, =SystemInit
191             BLX     R0
192             LDR     R0, =__main
193             BX      R0
194             ENDP
195
196 ; Dummy Exception Handlers (infinite loops which can be m
197
```

```
55  SystemInit PROC
56
57      ; Enable GPIO clock
58      LDR     R1, =RCC_AHB1ENR
```

编译后的反汇编代码：

地址　　程序代码　　；目标地址

```
190:                LDR     R0, =SystemInit
0x08000188 4809     LDR     r0,[pc,#36]  ; @0x080001B0
191:                BLX     R0
```

SystemInit入口地址

```
0x080001B0 01D1     DCW     0x01D1
0x080001B2 0800     DCW     0x0800
```

```
58:         LDR     R1, =RCC_AHB1ENR    ;Pseudo-load address in R1
0x080001D0 4917     LDR     r1,[pc,#92]  ; @0x08000230
```

SystemInit入口地址

使用伪指令将程序标号SystemInit的地址存入R0

必须加入”=”

第4章 ARM汇编语言程序设计

4.1 ARM汇编语言语句格式

4.2 复位加载与调试

4.3 结构化程序设计方法

4.4 C语言与汇编的相互调用

简单的ARM程序

```
;文件名：TEST1.S
;功能：实现两个寄存器相加
;说明：使用“；”进行注释，调试
        AREA    init,CODE,READONLY   ;声明代码段Example1
        Reset_Handler proc
;implemet code here.
START   MOV     R0,#0                ;设置参数
        MOV     R1,#10
LOOP    BL      ADD_SUB              ;调用子程序ADD_SUB
        B       LOOP                 ;跳转到LOOP
ADD_SUB
        ADDS    R0,R0,R1             ;R0 = R0 + R1
        MOV     PC,LR                ;子程序返回
        END                          ;文件结束
```

标号顶格写

实际代码段

声明文件结束

4.1 ARM汇编语言程序格式

完整的arm汇编程序包含若干汇编语言语句

[标号|常量名] <指令|伪指令|汇编指示指令 操作数1，2，…> [；注释]

4.1 ARM汇编语言程序格式

以程序段为单位组织代码。

段：相对独立的指令或数据序列

The AREA directive instructs the assembler to assemble a new code or data section。

```
AREA Example,CODE,READONLY ; An example code section.
    ; code
```

例 4-1 将数值 N，N-1，…，2，1 相加，结果保存在 R1 中。

```
; asm4-1.s
N               EQU         10          ; 定义常量 N 值为 10
; 栈配置
Stack_Size      EQU         0x00000400      ; 定义栈空间大小
                AREA        MyStack，  NOINIT，  READWRITE，  ALIGN=3; 声明栈段
Stack_Mem       SPACE       Stack_Size      ; 分配内存空间


__initial_sp
; 异常 / 中断向量表（复位后，向量表位于地址 0 处）
                AREA    Reset，  DATA，  READONLY    ; 声明 Reset 数据段
__Vectors       DCD     __initial_sp            ; 栈顶地址（MSP 初始值）
                DCD  Reset_Handler              ;“复位”异常处理代码的起始地址

                THUMB           ; 表示接下来的代码为 THUMB 指令集
                PRESERVE8       ; 表示接下来的代码保持 8 字节栈对齐
                AREA    Init，  CODE，  READONLY    ; 声明代码段
                ENTRY
```

```
;“复位”异常处理代码
Reset_Handler
        ;初始化寄存器
        LDR    R0,    =N          ;初始化循环计数值
        MOV    R1,    #0          ;初始化计算结果
        ;将数值 N，N-1，…，2，1 相加，计算结果在 R1 中
loop
        ADD    r1,    r0          ;R1 = R1 + R0
        SUBS   r0,    #1          ;减小 R0，更新标志位（'S' 后缀）
        BNE    loop     ;若上一条 SUBS 指令计算结果非 0，则跳到 loop
deadloop
        B    deadloop       ;无限循环
        NOP
        END
```

```
stack_size      equ     0x00000400
                area    stack,noinit,readwrite,align=3
stack_mem       space   stack_size
__initial_sp
                preserve8
                thumb

; Vector Table Mapped to Address 0 at Reset
                area    reset,data,readonly
                export  __Vectors
                export  __Vectors_End
                export  __Vectors_Size
                export  Reset_Handler
                export  __initial_sp
__Vectors       dcd     __initial_sp
                dcd     Reset_Handler
__Vectors_End
__Vectors_Size  equ  __Vectors_End - __Vectors
```

```
        area    init,code,readonly

n       equ 10
; Reset handler
Reset_Handler   proc

        ldr r0,=n
        mov r1,#0
loop
        add r1,r0
        subs r0,#1
        bne loop
here
      b here
       endp
       align
       end

   END
```

The END directive informs the assembler that it has reached the end of a source file.

AREA

The AREA directive instructs the assembler to assemble a new code or data section.

Syntax

`AREA` *`sectionname`*`{,`*`attr`*`}{,`*`attr`*`}...`

`ALIGN=`*`expression`*

By default, ELF sections are aligned on a four-byte boundary. *`expression`* can have any integer value from 0 to 31. The section is aligned on a $2^{expression}$-byte boundary. For example, if *expression* is 10, the section is aligned on a 1KB boundary.
This is not the same as the way that the `ALIGN` directive is specified.

REQUIRE8 and PRESERVE8

The `REQUIRE8` and `PRESERVE8` directives specify that the current file requires or preserves eight-byte alignment of the stack.

EXPORT or GLOBAL

The `EXPORT` directive declares a symbol that can be used by the linker to resolve symbol references in separate object and library files. `GLOBAL` is a synonym for `EXPORT`.

指示命令

- 符号定义
- 数据定义
- 汇编控制
- 其他指示命令

符号定义指示命令

- 定义全局变量：　　**GBLA**、**GBLL**和**GBLS**
- 定义局部变量：　　**LCLA**、**LCLL**和**LCLS**
- 变量赋值：　　**SETA**、**SETL**、**SETS**
- 通用寄存器列表定义名称:**RLIST**

GBLA、GBLL和GBLS

- **GBLA、GBLL和GBLS**

  语法格式：

  GBLA（GBLL或GBLS）　　全局变量名

```
            GBLA   objectsize    ; 全局的数字变量objectsize ，为0
Objectsize  SETA    0xff         ; 将该变量赋值为0xff
            SPACE  objectsize    ; 引用该变量

            GBLL   statusB       ; 全局的逻辑变量statusB ，为{False}
statusB     SETL   {TRUE}        ; 将该变量赋值为真
```

全局：作用范围为包含该变量的源程序

LCLA、LCLL和LCLS

- **LCLA、LCLL和LCLS**

  语法格式：

  LCLA（LCLL或LCLS）　　局部变量名

```
        LCLA objectsize    ；局部的数字变量objectsize ，为0
Objectsize SETA   0xff          ；将该变量赋值为0xff
        SPACE objectsize    ；引用该变量

        LCLL statusB     ；局部的逻辑变量statusB ，为{False}
statusB    SETL   {TRUE}    ；将该变量赋值为真
```

局部：作用范围为包含该局部变量的宏代码的一个实例

SETA、SETL、SETS

- **SETA、SETL和SETS**

语法格式：

变量名　SETA（SETL或SETS）　　表达式

在向变量赋值前，必须先声明该变量

```
    ; straightforward substitution
            GBLS    add4ff
            ;
add4ff      SETS    "ADD  r4,r4,#0xFF"    ; set up add4ff
            $add4ff.00                    ; invoke add4ff
            ; this produces
            ADD  r4,r4,#0xFF00
    ; elaborate substitution
            GBLS    s1
            GBLS    s2
            GBLS    fixup
            GBLA    count
            ;
count       SETA    14
s1          SETS    "a$$b$count" ; s1 now has value a$b0000000E
s2          SETS    "abc"
fixup       SETS    "|xy$s2.z|"  ; fixup now has value |xyabcz|
|C$$code|   MOV     r4,#16       ; but the label here is C$$code
```

RLIST

- **RLIST**

  语法格式:

  名称　　RLIST　　{寄存器列表}

```
Context   RLIST   {r0-r6, r8, r10-r12,r15}
```

; 将寄存器列表名称定义为Context ，可在ARM指令LDM/STM中通过该名称访问寄存器列表。排列顺序无关

数据定义指示命令

- DCB
- DCW（DCWU）
- DCD（DCDU）
- DCFD（DCFDU）
- DCFS（DCFSU）
- SPACE
- MAP
- FIELD

DCB

语法格式：

{标号}　DCB　表达式

表达式取值范围：**－128～255**的数字或字符串。

DCB：　**“=”**

```
Nullstring    DCB   “Null string”,0
; 构造一个以0结尾的字符串
```

DCW（或DCWU）

语法格式：

{标号}　　DCW（或DCWU）　　表达式

DCW：半字对齐

DCWU：不严格半字对齐。

表达式取值范围：**－32768～65535**

```
data1    DCW   －128，num1+8
；num1必须是已经定义过的
```

DCD（或DCDU）

语法格式：

{标号}　　DCD（或DCDU）　　表达式

DCD：“ **&** ”

DCD：字对齐

DCDU：不严格字对齐。

```
data1    DCD   1,5,20       ;其值为1，5，20
data2    DCD   memaddr+4
```

；分配一个字单元，其值为程序中标号memaddr加4个字节

DCFD（或DCFDU）

语法格式：

{标号} DCFD（或DCFDU） 表达式

每个双精度的浮点数占据两个字单元。

DCFD：字对齐

DCFDU：不严格字对齐

```
DCFD        1E308, -4E—100
DCFDU       100000
```

DCFS（或DCFSU）

语法格式：

{标号}　DCFS（或DCFSU）　表达式

每个单精度的浮点数占据一个字单元。

DCFS：字对齐

DCFSU：不严格字对齐

```
        DCFS        1E3,-4E-9
```

SPACE

语法格式：

{标号}　　SPACE　　表达式

分配一片连续的存储区域并初始化为0。其中，表达式为要分配的字节数。

SPACE：“ % ”

```
Datastruc   SPACE   280
; 分配连续280字节的存储单元并初始化为0
```

MAP

语法格式：

MAP　表达式　{，基址寄存器}

用于定义一个结构化的内存表的首地址。

MAP：“　^　”

通常与FIELD伪指令配合使用来定义结构化的内存表。

```
MAP    0x80，R9
；定义结构化内存表首地址的值为0x80＋R9
```

FIELD

语法格式:

{标号}　FIELD　表达式

定义一个结构化内存表中的数据域。FILED也可用“#”代替。

```
            MAP    0          ; 定义结构化内存表首地址为0
consta      FIELD  4          ; consta的长度为4字节，相对位置为0
constb      FIELD  4          ; constb的长度为4字节，相对位置为4
x           FIELD  8          ; x的长度为8字节，相对位置为0x8
y           FIELD  8          ; y的长度为8字节，相对位置为0x10
string      FIELD  256        ; y的长度为256字节，相对位置为0x18
```

汇编控制指示命令

汇编控制（Assembly Control）指示命令用于控制汇编程序的执行流程，常用的汇编控制指示命令包括以下几条：

- **IF、ELSE、ENDIF**
- **WHILE、WEND**
- **MACRO、MEND**
- **MEXIT**

IF、ELSE、ENDIF

语法格式:

```
IF      逻辑表达式
指令序列1
ELSE
指令序列2
ENDIF
```

示例:

```
IF Version="1.0"
;指令
; 伪指令
ELSE
;指令
; 伪指令
ENDIF
```

WHILE、WEND

语法格式：

```
WHILE    逻辑表达式
         指令序列
WEND
```

示例：

```
count    SETA     1
         WHILE    count<=4
count    SETA     count+1
         ;code
         WEND
```

MACRO、MEND

语法格式：

```
        MACRO
$标号   宏名   $参数1，$参数2，……
        指令序列
        MEND
```

示例：在ARM中完成测试—跳转操作需要两条指令，定义一条宏指令完成测试—跳转操作

```
            MACRO
$label      TestAndBranch   $dest, $reg, $cc
$label      CMP     $reg,  #0
            B$cc    $dest
            MEND
```

MACRO、MEND

```
; 在程序中调用该宏
  test   TestAndBranch   NonZero, r0, NE
         …
         …
```

```
         MACRO
$label   TestAndBranch   $dest, $reg, $cc
$label   CMP      $reg,   #0
         B$cc     $dest
         MEND
```

```
; 程序被汇编后，宏展开的结果
  test   CMP   r0, #0
         BNE   NonZero
         …
         …
```

MEXIT

语法格式：

MEXIT

MEXIT用于从宏定义中跳转出去。

```
MACRO
.....
IF condition
MEXIT
ELSE
.....
ENDIF
MEND
```

其它指示命令

- ALIGN
- AREA
- CODE16/CODE32
- END
- ENTRY
- EQU
- EXPORT/GLOBAL
- IMPORT/EXTERN
- GET/INCLUDE
- INCBIN

4.2 复位启动和调试

* 复位过程
* 启动模式
* 存储器重映射

4.2 复位启动和调试

复位过程:

MSP取值、复位向量

启动模式选择

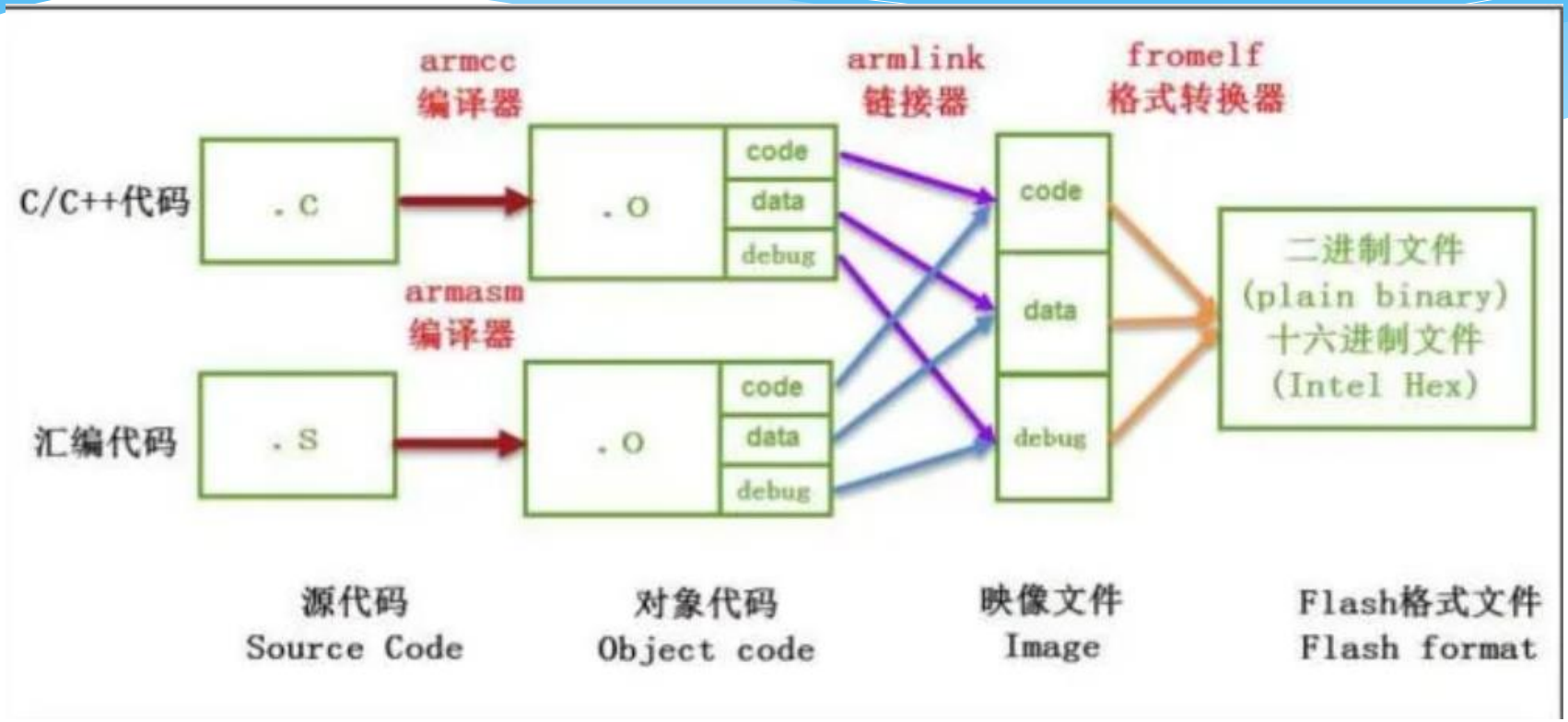
地址空间	地址范围	大小
主 flash	0x08000000~0x080FFFFF	1 MB
系统存储器	0x1FFF0000~0x1FFF77FF	30 KB
SRAM1	0x20000000~0x2001BFFF	112 KB

4.2 复位启动和调试

启动模式选择

地址空间	地址范围	大小
主 flash	0x08000000~0x080FFFFF	1 MB
系统存储器	0x1FFF0000~0x1FFF77FF	30 KB
SRAM1	0x20000000~0x2001BFFF	112 KB

程序编译连接过程


armcc
编译器
armlink
链接器
fromelf
格式转换器
C/C++代码
.c
.o
code
data
debug
armasm
编译器
汇编代码
.s
.o
code
data
debug
code
data
debug
二进制文件
(plain binary)
十六进制文件
(Intel Hex)
源代码
Source Code
对象代码
Object code
映像文件
Image
Flash格式文件
Flash format

4.3 ARM汇编语言的基本结构程序设计

① 顺序结构

② 选择结构

③ 循环结构

- 子程序

;例子4-2

```
1  stack_size equ 0x00000400
2      area    stack,noinit,readwrite,align=3
3  stack_mem space stack_size
4  __initial_sp
5      preserve8
6      thumb
7  ; Vector Table Mapped to Address 0 at Reset
8      area    reset,data,readonly
9      export  __Vectors
10     export  __Vectors_End
11     export  __Vectors_Size
12     export  Reset_Handler
13     export  __initial_sp
14 __Vectors dcd __initial_sp
15           dcd Reset_Handler
16 __Vectors_End
17
18 __Vectors_Size equ  __Vectors_End - __Vectors
19
```

;例子4-2

```
20      area init,code,readonly
21  Reset_Handler proc
22  ;implemet code here.
23
24
25  ;example 4.2[p92]
26      mov r3,#10
27      cmp r3,#9
28      ble next
29      add r3,r3,#7
30  next
31      add r3,r3,#'0'
32  here
33      b here
34      align
35      endp
36      end
```

第四节 汇编和C程序相互调用

* 在C程序中使用汇编指令
* Inline assembler support
* Embedded assembler

The inline assembler

以_asm标识

```
__asm("ADD x, x, #1\n"
      "MOV y, x\n");
```

```
__asm
{
    ...
    instruction
    ...
}
```

```
int f(int x)
{
    int r0;
    __asm
    {
        ADD r0, x, 1
        EOR x, r0, x
    }
    return x;
}
```

第四节 汇编和C程序相互调用

* Embedded assembler

以_asm 标识开始的汇编语句形成的函数

一个嵌入式汇编语言函数形式如下

```
__asm return-type function-name(parameter-list)
{
  // ARM/Thumb assembly code
  instruction{;comment is optional}
  ...
  instruction
}
```

第四节 汇编和C程序相互调用

* Embedded assembler

例子

```
#include <stdio.h>
__asm void my_strcpy(const char *src, char *dst)
{
loop
        LDRB  r2, [r0], #1
        STRB  r2, [r1], #1
        CMP   r2, #0
        BNE   loop
        BX    lr
}
int main(void)
{
    const char *a = "Hello world!";
    char b[20];
    my_strcpy (a, b);
    printf("Original string: '%s'\n", a);
    printf("Copied   string: '%s'\n", b);
    return 0;
}
```

第四节 汇编和C程序相互调用

* Embedded assembler的限制
  * 无符号
  * 不能直接向PC寄存器赋值
  * 通常内联的汇编指令中不要指定物理寄存器，
  * 只有指令B可以使用C程序中的标号

第四节 汇编和C程序相互调用

参数传递的AAPCS规则：

* 函数的最前面4个整型参数通过ARM的寄存器r0---r3来传递，多于4个的参数用满递减堆栈来传递
* 不多于4个整型返回值可以用寄存器r0---r3来传递
* 双字类型的参数通过一对连续的寄存器传递

Procedure Call Standard for the ARM® Architecture

Document number: ARM IHI 0042F, current through ABI release 2.10
Date of Issue: 24th November 2015

* **AAPCS**

Procedure Call Standard for the ARM Architecture. Governs the exchange of control and data between functions at runtime. There is a variant of the AAPCS for each of the major execution environment types supported by the toolchain.

第四节 汇编和C程序相互调用

在C程序中使用extern关键字声明要调用的汇编程序函数名称。

```
#include<stdio.h>
    extern void strcopy(char *d, const char *s)
    int main(void)
    {
        const char *srcstr="First string-source";
       char dstsrt[ ]="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);
    }
```

第四节 汇编和C程序相互调用

在汇编程序中使用EXPORT伪指令声明本子程序，使其它程序可以调用此子程序

```
  AREA    SCopy,      CODE, READONLY
  EXPORT strcopy
strcopy
  ;寄存器R0存放第一个参数，即目标字符串的地址
  ;寄存器R1存放第二个参数，即源字符串的地址
  LDRB      R2,[R1],#1   ;读取字节数据，源地址加1
  STRB      R2,[R0],#1   ;保存读取的字节数据,目标地址加1
  CMP R2,#0                    ;判断字符串是否复制完毕
  BNE strcopy                  ;没有复制完毕，继续循环
  MOV PC,LR                    ;返回
  END
```

第四节 汇编和C程序相互调用

* 汇编调用C

```
/*函数sum5()返回5个整数的和*/
int sum5(int a, int b, int c, int d, int e)
{
  return(a+b+c+d+e);   //返回5个变量的和
}
```

* 5个参数，分别使用寄存器R0存储第1个参数，R1存储第2个数，R2存储第3个参数，R3存储第4个参数，第5个参数利用堆栈传送。

第四节 汇编和C程序相互调用

```
    EXPORT   example
  AREA  example, CODE, READONLY
IMPORT sum5        ;声明外部标号sum5，即C函数sum5()
STR          LR, [SP, # -4]!
ADD          R1, R0, R0
ADD          R2, R1, R0
ADD          R3, R1, R2
STR          R3, [SP,# -4]!
ADD          R3, R1, R1
BL           sum5
ADD          SP, SP, #4
LDR          PC, [SP], #4
END
```