第3章 程序设计初步

结合C语言源程序的目标代码，本章介绍基于IA-32系列CPU的汇编语言程序设计，同时分析C语言部分语句的实现方法。在说明堆栈的作用和介绍算术逻辑运算指令后，分别介绍分支、循环和子程序的设计。

3.1 堆栈的作用

在2.7节介绍了堆栈，并提及了堆栈的主要作用。为了今后方便地阅读与C语言源程序对应的目标代码，本节将具体说明堆栈的主要用途：保存函数的返回地址；用于向函数传递参数；安排函数的局部变量。由于这些都与子程序（过程）有关，因此先介绍过程调用和返回指令。请注意，这里堆栈的概念对应高级语言中栈的概念。

3.1.1 过程调用和返回指令

在汇编语言中，常把子程序称为过程（Procedure）。C语言中的函数是子程序，也就是汇编语言中的过程。

调用子程序（过程、函数）在本质上是控制转移，它与无条件转移的区别是调用子程序要考虑返回。CPU提供专门的过程调用指令和过程返回指令。通常，过程调用指令用于由主程序转移到子程序，过程返回指令用于由子程序返回到主程序。

1. 示例分析

在介绍过程调用指令和返回指令之前，先介绍C语言源程序及其目标代码的示例。

【例3-1】 对比分析如下所示的C语言源程序dp31和相应汇编格式指令的目标代码。

在如下C语言程序dp31中，main函数先调用函数cf211计算一个表达式的值，然后打印输出结果。函数cf211与2.5节例2-45相同，计算一个示例表达式，其中的函数调用约定_fastcall表示希望通过寄存器来传递参数。

```
#include <stdio.h>
int  _fastcall  cf211(int x, int y)
{
    return  (2*x+5*y+100);
}
//作为示例的主程序
int  main()
{
    int  val;
    val=cf211(23, 456);
```

```
    printf("val=%d\n", val);
    return  0;
}
```

在项目属性中，采用配置属性选项“在静态库中使用 MFC”，同时采用编译优化选项“使速度最大化”，编译上述程序后，可得到如下所示的用汇编格式指令表示的目标代码（标号和名称稍有修饰），分号之后是添加的注释。

```
;函数 cf211 的目标代码
;通过寄存器 ECX 和 EDX 传递参数 x 和 y
cf211     PROC                              ;过程开始
    lea   eax, DWORD PTR [edx+edx*4+100]    ;EAX=5* y+100
    lea   eax, DWORD PTR [eax+ecx*2]        ;EAX=EAX+2* x
    ret                                     ;返回（返回值在 EAX 中）      //@r1
cf211     ENDP                              ;过程结束
;
;函数 main 的目标代码
_main     PROC                              ;过程开始
                                            ;val=cf211(23, 456)
    mov   edx, 456                          ;由寄存器 EDX 传参数 y        //@p1
    mov   ecx, 23                           ;由寄存器 ECX 传参数 x        //@p2
    call  cf211                             ;调用函数 cf211               //@c1
                                            ;printf("val=%d\n", val)
    push  eax                               ;把 val 值压入堆栈
    push  OFFSET FMTS                       ;把输出格式字符串首地址压入堆栈
    call  _printf                           ;调用库函数_printf            //@c2
    add   esp, 8                            ;平衡堆栈                      //@s
    ;
    xor   eax, eax                          ;由 eax 传递返回值
    ret                                     ;返回                         //@r2
_main     ENDP                              ;过程结束
```

上述目标代码中的“过程开始”和“过程结束”行，并非汇编格式指令，但可以认为这些是汇编语言的语句，在形式上表示函数（过程）的代码的开始和结束。

从上述 C 语言源程序和对应的汇编格式指令代码可以看到：main 函数和 cf211 函数分别对应一个过程（子程序）。在过程_main 中，通过指令 call 调用了过程 cf211 和过程_printf（库函数）。在调用过程 cf211 时，通过寄存器传递参数，比较简单。但在调用过程_printf 时，通过堆栈传递参数，先把两个参数压入了堆栈。在过程 cf211 和过程_main 中，通过返回指令 ret 返回到主程序。函数的返回值在寄存器 EAX 中。

在标识符_main 和_printf 中，为首的下画线是 VC 2010 编译器加上去的，以便符合编译和链接的相关约定。

2. 过程调用指令

主程序调用子程序，在执行完子程序后，再返回到主程序。因此，在从主程序转移到子程序时，首先要保存返回地址，以便返回，然后再转移到子程序。

过程调用指令实现由主程序转移到子程序。所以，过程调用指令首先要保存返回地址，然后再转移到子程序的入口地址。为了简化，可以先这样认为：所谓返回地址，就是紧随过程调

用指令的下一条指令的地址偏移(有效地址)；所谓转移到子程序的入口地址，就是使得指令指针寄存器 EIP 等于子程序开始处的地址偏移。保存返回地址的方法是把返回地址压入堆栈。例如，执行例 3-1 中调用指令“call　cf211”的操作，首先把返回地址偏移(紧随其后的指令“push　eax”的地址偏移)压入堆栈，然后使得 EIP 等于函数 cf211 的第一条指令“lea　eax，DWORD PTR [edx+edx*4+100]”的地址偏移。

过程调用指令有多种方式，这里先介绍最简单的段内直接调用指令。

段内直接调用指令的一般格式如下：

CALL　*LABEL*

标号 LABEL 可以是程序中的一个标号，也可以是一个过程名。

【例 3-2】 如下指令演示了段内直接调用指令的使用。

```
CALL    HTOASC                  ;HTOASC 是某个子程序开始处的标号
CALL    SUB1                    ;SUB1 是某个过程的名称
```

段内直接调用指令的具体操作：①把返回地址偏移压入堆栈；②使得 EIP 的内容为目标地址偏移，从而实现转移。第二步与 2.6.4 节介绍的无条件转移指令的操作是相同的。实际上，与无条件转移指令 JMP 相比，过程调用指令 CALL 只是多了第一步：保存返回地址。

执行段内直接调用指令 CALL 的堆栈变化如图 3.1(a)、(b)所示。在保护方式(32 位代码段)下，返回地址偏移占 4 个字节。

3. 过程返回指令

子程序在执行完后，要返回到主程序。利用保存在堆栈中的返回地址，子程序就能够方便地返回主程序。

过程返回指令用于从子程序返回到主程序。在执行该指令时，从堆栈顶弹出返回地址，并转移到所弹出的地址，这样就实现了返回。通常，这个返回地址就是在执行对应的调用指令时所压入堆栈的返回地址。

过程返回指令的使用应该与过程调用指令所对应。这里先介绍简单的段内返回指令。

简单的段内返回指令的格式如下：

RET

该指令从堆栈弹出地址偏移，送到指令指针寄存器 EIP。在保护方式(32 位代码段)下，地址偏移是一个双字(32 位，4 个字节)。

执行简单的段内返回指令 RET 的堆栈变化如图 3.1(b)、(c)所示。它与段内调用指令相对应。

在例 3-1 的目标代码中，注释带//@r1 和//@r2 的行，都是返回指令 RET 的具体应用。在执行标有//@r1 行的返回指令“ret”时，从堆栈顶弹出在执行调用指令“call　cf211”时压入堆栈的返回地址偏移，把弹出地址偏移送到 EIP，从而返回到主程序，继续从指令“push　eax”开始执行。

【例 3-3】 如下 C 程序 dp32 采用嵌入汇编代码方式，演示子程序的调用及其返回，说明 CALL 指令和 RET 指令的使用。其中，子程序 UPPER 实现把寄存器 AL 中的字符大写化(如果为小写字母，则转换成大写字母，否则不变)；子程序 TUPPER 把寄存器 AX 中的两个字符大写化。演示步骤为，两次调用子程序 TUPPER，还调用子程序 UPPER，把字符串 string 中的 5 个字母转换为大写。通过 C 语言的库函数 printf 实现显示输出。当然，这仅仅

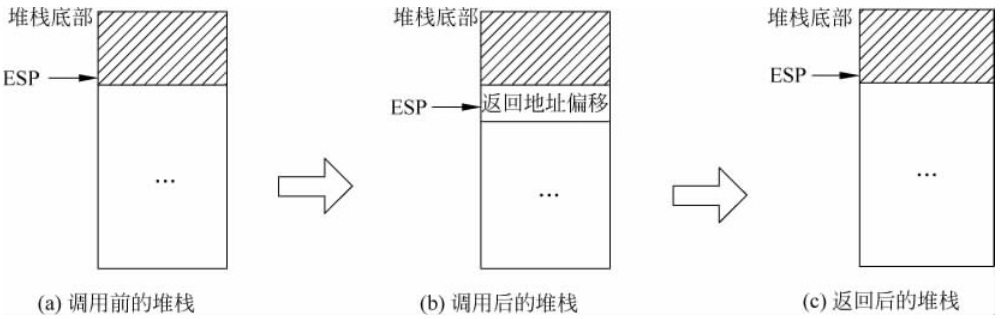

图 3.1　执行段内调用指令时的堆栈变化示意图

是个示例。

```
#include <stdio.h>
char  string[ ]="abcde";
int  main()
{
    _asm{                                  //嵌入汇编代码
        LEA   ESI, string                  //ESI 指向 string 首
        MOV   AX, [ESI]
        CALL  TUPPER                       //调用子程序 tupper
        MOV   [ESI], AX
        MOV   AX, [ESI+2]
        CALL  TUPPER                       //调用子程序 tupper
        MOV   [ESI+2], AX
        MOV   AL, [ESI+4]
        CALL  UPPER                        //调用子程序 upper
        MOV   [ESI+4], AL
    }
    printf("%s\n", string);                //显示为 ABCDE
    return  0;
    //嵌入汇编代码形式的子程序
    _asm{                                  //嵌入汇编代码
    UPPER:                                 //子程序入口标号
        CMP   AL, 'a'
        JB    UPPER2
        CMP   AL, 'z'
        JA    UPPER2
        SUB   AL, 20H                      //小写转大写
    UPPER2:
        RET                                //返回
    //
    TUPPER:                                //子程序入口标号
        CALL  UPPER                        //调用子程序
        XCHG  AH, AL
        CALL  UPPER                        //调用子程序
        XCHG  AH, AL
```

```
        RET                                    //返回
    }
}
```

从上述的嵌入汇编代码可知，子程序 TUPPER 又调用子程序 UPPER 来实现一个字符的大写化。子程序 TUPPER 和子程序 UPPER 都是通过寄存器传递参数。

需要特别注意，上述的程序组织方式是把子程序 UPPER 和 TUPPER 的嵌入汇编代码安排在 C 函数的 return 语句之后。这样能够避免不经过调用直接进入子程序。如果在有的编译器中不能通过，必须将 UPPER 和 TUPPER 的嵌入汇编代码安排在 return 语句之前才能通过，那么需要安排无条件转移指令或者 goto 语句跳过这些子程序的代码片段。

3.1.2 参数传递

主程序在调用子程序时，往往要向子程序传递一些参数；同样，子程序运行后也经常要把一些结果返回给主程序。主程序与子程序之间的这种信息传递被称为参数传递。通常把由主程序传给子程序的参数称为子程序的入口参数，把由子程序传给主程序的参数称为子程序的出口参数。一般而言，子程序既有入口参数又有出口参数。但有的子程序只有入口参数，而没有出口参数；少数子程序只有出口参数，而没有入口参数。

1. 参数传递方法

有多种传递参数的方法：寄存器传递法、堆栈传递法、约定内存单元传递法和 CALL 后续区传递法等。主程序与子程序之间传递参数的方法是根据具体情况而事先约定好的。有时可能同时采用多种方法。

利用寄存器传递参数就是把参数放在约定的寄存器中。例 3-3，就是通过寄存器传递参数。在本节的例 3-1 中，通过寄存器 ECX 和 EDX 给函数 cf211 传递入口参数，通过寄存器 EAX 传递出口参数。这种方法的优点是实现简单和调用方便。但由于寄存器的数量较少，并且寄存器往往还要存放其他数据，因此只适用于传递少量参数的情形。

堆栈可以用于传递参数，这也是堆栈的一个重要用途。

C 语言的函数通常利用堆栈传递入口参数，而利用寄存器传递出口参数。

如果使用堆栈传递入口参数，那么主程序在调用子程序之前，把需要传递的参数依次压入堆栈，然后子程序从堆栈中取入口参数。

利用堆栈传递参数可以不占用寄存器，也无须使用额外的存储单元。但由于参数和子程序的返回地址都一起在堆栈中，有时还要考虑保护寄存器，所以较为复杂。

2. 堆栈传递参数的示例一

为了说明如何通过堆栈传递参数，先介绍一个示例。

【例 3-4】 对比分析如下所示的 C 语言源程序 dp33 和相应汇编格式指令的目标代码。

```
#include <stdio.h>
int  cf34(int x, int y)
{
    return  (2*x+5*y+100);
}
//
int  main()
{
```

```
    int  val;
    val=cf34(23, 456);
    printf("val=%d\n", val);
    return  0;
}
```

上述C语言源程序与例3-1的源程序dp31几乎相同,除了自定义的函数名改为cf34和删掉了其调用约定_fastcall之外。这样,函数cf34就没有要求通过寄存器传递参数。

在项目属性中,采用配置属性选项"在静态库中使用MFC",同时采用编译优化选项"使速度最大化",编译上述程序后,可得到如下所示的用汇编格式指令表示的目标代码(标号和名称稍有修饰),其中"DWORD PTR"表示32位存储器操作数,分号之后是添加的注释。

```
;函数 cf34 的目标代码
;通过堆栈传递入口参数 x 和 y
cf34      PROC                                ;过程开始
    push  ebp                                 ;把 EBP 压入堆栈
    mov   ebp, esp                            ;使得 EBP 指向栈顶
    mov   eax, DWORD PTR [ebp+12]             ;从堆栈取参数 y              //@y
    mov   ecx, DWORD PTR [ebp+8]              ;从堆栈取参数 x              //@x
    lea   eax, DWORD PTR [eax+eax*4+100]      ;EAX=5* y+100
    lea   eax, DWORD PTR [eax+ecx*2]          ;EAX=EAX+2* x
    pop   ebp                                 ;恢复 EBP
    ret                                       ;返回(返回值在 EAX 中)
cf34      ENDP                                ;过程结束
;
;函数 main 的目标代码
_main     PROC                    ;过程开始
                                  ;val=cf34(23, 456)
    push  456                     ;把参数 y(000001c8H)压入堆栈          //@p1
    push  23                      ;把参数 x(00000017H)压入堆栈          //@p2
    call  cf34                    ;调用函数 cf34
                                  ;printf("val=%d\n", val)
    push  eax                     ;把 val 值压入堆栈
    push  OFFSET FMTS             ;把输出格式字符串首地址压入堆栈
    call  _printf                 ;调用库函数_printf
    add   esp, 16                 ;平衡堆栈                              //@s
    ;
    xor   eax, eax                ;由 eax 传递返回值
    ret                           ;返回
_main     ENDP                    ;过程结束
```

与例3-1中函数main的目标代码相比,这里函数main的目标代码有3处不一样,见注释带有//@p1、//@p2和//@s的行。由此可见,函数cf34采用堆栈传递入口参数。当然,库函数_printf也是通过堆栈传递入口参数。

与例3-1中函数cf211的目标代码相比,函数cf34的目标代码多了几行。因为没有通过寄存器传递参数,所以在开始计算表达式(2x+5y+100)之前,需要先从堆栈中取得入口参数 x 和 y 的值。

3. 堆栈传递参数分析

通过堆栈传递入口参数分为两个方面：一方面，主程序先把入口参数压入堆栈，然后利用 call 指令调用子程序；另一方面子程序(过程、函数)一般会先做必要的准备，然后从堆栈中取得参数。

图 3.2 给出了例 3-4 中调用函数 cf34 以及函数 cf34 执行前后的堆栈变化情况。

现在结合例 3-4 的目标代码和图 3.2，说明调用函数 cf34 前后以及它执行期间堆栈的变化情况。在调用子程序 cf34 之前，主程序 main 先把参数 y 和参数 x 压入堆栈，这时堆栈就由图 3.2(a)变化为图 3.2(b)。然后，执行 call 指令，调用子程序 cf34，调用指令 call 把返回地址偏移压入堆栈，这时堆栈变化如图 3.2(c)所示，现在进入到子程序 cf34。子程序 cf34 先把寄存器 EBP 压入堆栈，随即把堆栈指针寄存器 ESP 的值送到 EBP，这时堆栈变化如图 3.2(d)所示。现在子程序 cf34 就可以方便地从堆栈中取出相应的参数了。然后，子程序 cf34 执行相应的功能。在返回之前，从堆栈弹出刚才保存的 EBP 值，从而恢复 EBP，这时堆栈就回到如图 3.2(c)所示。最后，执行 ret 指令，返回到主程序 main，返回指令 ret 从堆栈弹出返回地址偏移，这时堆栈就回到如图 3.2(b)所示。现在回到了主程序 main。它接着执行其他的指令，包括调用子程序_printf 等。在从子程序_printf 返回后，主程序 main 执行注释带//@s 行的指令“add　esp，16”，在这之后，堆栈回到如图 3.2(a)所示。

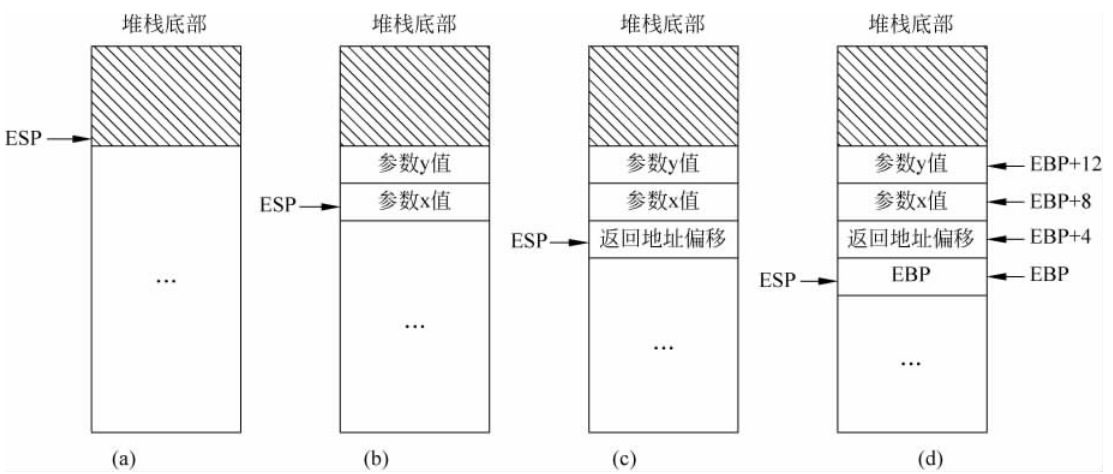

图 3.2　调用函数 cf34 前后堆栈变化示意图

在例 3-4 的目标代码中，注释带//@s 行指令的作用是平衡堆栈。因为主程序 main 在调用子程序 cf34 之前把两个参数压入堆栈，在调用子程序_printf 之前，又把两个参数压入堆栈，这样堆栈顶就有 4 个已经没有用的参数了。每个 32 位的参数，占用 4 个字节。所以利用指令“add　esp，16”一次性平衡堆栈。在例 3-1 的目标代码中，也有类似起平衡堆栈作用的指令“add　esp，8”，参见例 3-1 目标代码中标//@s 的行。在那里调用子程序 cf211 没有通过堆栈传递参数，当时堆栈顶只有两个在调用子程序_printf 之前压入的参数。实际上，在编译例 3-4 的源程序 dp33 时，采用了编译优化选项“使速度最大化”，这样所得的目标代码把两次平衡堆栈的操作合并到了一起。如果不采用编译优化，那么在从子程序 cf34 返回后，应该就会立即调整堆栈指针寄存器 ESP 的值，达到平衡堆栈的目的。

现在来看例 3-4 中子程序 cf34 如何从堆栈中取得由主程序压入的参数 x 和 y。结合图 3.2 可知，子程序 cf34 的目标代码中注释带//@x 和//@y 的行，分别从堆栈中把参数 x 和

y 取到了寄存器 ECX 和 EAX 中。为了通过寄存器 EBP 间接访问堆栈，先把堆栈指针寄存器 ESP 复制到 EBP，从而使得 EBP 指向当时的栈顶。当然这会破坏寄存器 EBP 中原有的内容，所以子程序一开始就把寄存器 EBP 的内容压入到堆栈中加以保护，在返回之前才恢复 EBP 原先的内容。在对应 C 语言源程序的目标代码中，通常会采用这种方式来访问堆栈，包括存取通过堆栈传递的参数。在 2.5.2 节介绍存储器寻址方式时，曾经指出过，如果基址寄存器是 EBP 或者 ESP，那么默认引用的段寄存器是 SS，这正是堆栈段。

在子程序开始时，首先把 EBP 压入到堆栈，然后复制 ESP 到 EBP，使得 EBP 指向堆栈顶，做好通过 EBP 访问堆栈的准备，常把这一步骤称为建立堆栈框架。在子程序返回之前，会从堆栈恢复 EBP，常把这一步骤称为撤销堆栈框架。

4. 堆栈传递参数的示例二

下面利用另一个示例来进一步观察堆栈传递参数的情况。

【例 3-5】 分析如下由 C 语言编写的函数 cf35 的目标代码。

```
int  cf35(int x, int y)
{
    if(x < y) x=y;
    return  x;
}
```

函数 cf35 有两个整型参数，返回值是两个数中的较大者。根据其功能，也许函数名为 max 更合适。当然，这仅仅是用于说明堆栈传递参数的示例。

采用编译优化选项“使速度最大化”，编译上述程序后，可得到如下所示的用汇编格式指令表示的目标代码(标号和名称稍有修饰)，其中“SHORT”表示转移目的地就在附近，分号之后是添加的注释。

```
;函数 cf35 的目标代码(速度最大化)
cf35      PROC                        ;表示过程(函数)开始
    push  ebp
    mov   ebp, esp                    ;建立堆栈框架
    mov   eax, DWORD PTR [ebp+8]      ;从堆栈取参数 x
    mov   ecx, DWORD PTR [ebp+12]     ;从堆栈取参数 y
    cmp   eax, ecx                    ;比较 x 和 y(EAX 代表 x,ECX 代表 y)
    jge   SHORT ln1cf35               ;如果 x 大于等于 y,就跳转
    mov   eax, ecx                    ;实现 x=y
ln1cf35:
    pop   ebp                         ;撤销堆栈框架
    ret                               ;返回
cf35      ENDP                        ;表示过程(函数)结束
```

从上述目标代码可知，子程序 cf35 首先设置 EBP 做好访问堆栈中参数的准备；接着从堆栈中取出参数 x 和 y，分别送到寄存器 EAX 和 ECX，于是 EAX 就相当于形参 x，ECX 相当于形参 y；然后根据比较进行分支，使得 EAX 含有较大者；最后恢复 EBP 并返回。

假设不要求编译优化，即采用编译优化选项“已禁用”，编译上述 C 语言函数 cf35 后，可得到如下所示的目标代码，分号之后是添加的注释。

```
;函数 cf35 的目标代码(禁用优化)
cf35      PROC                        ;表示过程(函数)开始
```

```
    push  ebp
    mov   ebp, esp                    ;建立堆栈框架
    mov   eax, DWORD PTR [ebp+8]
    cmp   eax, DWORD PTR [ebp+12]
    jge   SHORT ln1cf35
    mov   ecx, DWORD PTR [ebp+12]
    mov   DWORD PTR [ebp+8], ecx
ln1cf35:
    mov   eax, DWORD PTR [ebp+8]
    pop   ebp                         ;撤销堆栈框架
    ret
cf35      ENDP                        ;表示过程(函数)结束
```

从上述目标代码可知,由于没有采用“使速度最大化”的编译,因此把堆栈中的[ebp+8]单元作为形参 x,[ebp+12]单元作为形参 y。子程序 cf35 首先做好访问堆栈中参数的准备;接着根据比较进行分支,使得形参 x 含有较大者;然后,从形参 x 中取出较大者到 EAX,作为返回值;最后恢复 EBP 并返回。这样处理,效率相对较低。

为了进一步提高效率,采用编译优化选项“使速度最大化”,同时优化项中还要求“省略帧指针”(实际上就是不建立堆栈框架),编译上述 C 语言函数 cf35,可得到如下所示的目标代码:

```
;函数 cf35 的目标代码(速度最大化,且省略帧指针)
cf35      PROC                        ;表示过程(函数)开始
    mov   eax, DWORD PTR [esp+4]      ;从堆栈取参数 x
    mov   ecx, DWORD PTR [esp+8]      ;从堆栈取参数 y
    cmp   eax, ecx                    ;比较之
    jge   SHORT ln1cf35               ;大于等于则跳转
    mov   eax, ecx                    ;使得 EAX 含较大者
ln1cf35:
    ret
cf35      ENDP                        ;表示过程(函数)结束
```

从上述目标代码可知,由于编译时要求“省略帧指针”,因此干脆以堆栈指针寄存器 ESP 作为基址寄存器,来访问堆栈中的参数。这样也就不需要保护寄存器 EBP 了。如图 3.2(c)所示,位于堆栈中的[esp+4]单元含有参数 x,[esp+8]单元含有参数 y。子程序 cf35 迅速从堆栈中取得参数 x 和 y,分别送到寄存器 EAX 和 ECX,并将寄存器 EAX 当作形参 x,寄存器 ECX 当作形参 y。这样做,效率应该是最高的。

3.1.3　局部变量

局部变量是高级语言中的概念。所谓局部变量,是指对其访问仅限于某个局部范围。在 C 语言中,局部的范围可能是函数或复合语句。局部变量还有动态和静态之分。

堆栈可以用于安排动态局部变量。

1. 局部变量示例一

【例 3-6】 分析如下由 C 语言编写的函数 cf36 的目标代码。

```
int  cf36(int x, int y)
{
```

```
    int  z;
    z=x;
    if(x<y) z=y;
    return  z;
}
```

函数 cf36 的功能与例 3-5 的函数 cf35 一样，返回较大者。作为示例，特意安排了一个局部变量 z。

如果采用编译优化选项“使速度最大化”，编译函数 cf36 后，所得目标代码与例 3-5 的函数 cf35 的目标代码完全一样。由于要求“使速度最大化”，因此局部变量 z 被“消解”掉了。

为了演示如何安排局部变量，假设不要求编译优化，即采用编译优化选项“已禁用”，编译上述 C 语言函数 cf36。编译后可得到如下所示的用汇编格式指令表示的目标代码，其中的“DWORD PTR”表示双字存储单元，“SHORT”表示转移目的地就在附近，分号之后是添加的注释。

```
cf36      PROC                        ;表示过程(函数)开始
    push  ebp
    mov   ebp, esp                    ;建立堆栈框架
    push  ecx                         ;在堆栈中安排局部变量 z
                                      ;z=x;
    mov   eax, DWORD PTR [ebp+8]      ;取得形参 x
    mov   DWORD PTR [ebp-4], eax      ;送到变量 z
                                      ;if(x<y)z=y;
    mov   ecx, DWORD PTR [ebp+8]      ;取得形参 x
    cmp   ecx, DWORD PTR [ebp+12]     ;比较 x 与 y
    jge   SHORT LN1cf36               ;如果 x 大于 y,则跳转
    mov   edx, DWORD PTR [ebp+12]     ;取得形参 y
    mov   DWORD PTR [ebp-4], edx      ;送到变量 z
LN1cf36:
                                      ;return  z;
    mov   eax, DWORD PTR [ebp-4]      ;把 z 送到 EAX
    mov   esp, ebp                    ;撤销局部变量 z
    pop   ebp                         ;撤销堆栈框架
    ret                               ;返回
cf36      ENDP                        ;表示过程(函数)结束
```

在没有采用编译优化的情况下，所得上述目标代码很清晰地对应 C 语言源程序的语句。

2. 堆栈中的局部变量

图 3.3 给出了例 3-6 中函数 cf36 执行阶段堆栈的变化情况。现在来分析对局部变量 z 的安排及访问方法。

从例 3-6 的目标代码可知，首先建立堆栈框架，这时堆栈就由图 3.3(a)变化为图 3.3(b)，现在可以基于 EBP 访问堆栈了。接着利用一条 push 指令，使得 EBP 减去 4，这时堆栈变化如图 3.3(c)所示。这是关键所在，其实质是为局部变量 z 在堆栈中安排存储单元，并非要在堆栈中保存某个寄存器的值。现在局部变量 z 出现了，可以基于 EBP 访问局部变量 z。然后，实现函数 cf36 的功能。在邻近返回前，把 EBP 送回到堆栈指针寄存器 ESP，这时堆栈由图 3.3(c)回到图 3.3(b)，这意味着从堆栈中撤销局部变量 z。在返回前，撤销堆栈框架，这时堆栈由

图 3.3(b)变化为图 3.3(a)。最后,利用 ret 指令,返回到主调程序。

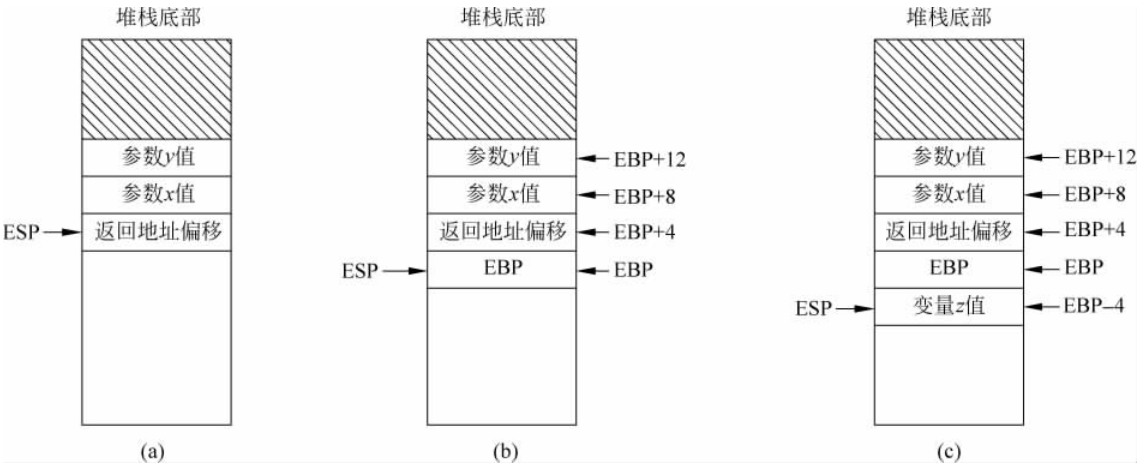

图 3.3　函数 cf36 执行期间堆栈变化示意图

由于局部变量 z 在堆栈中,且 EBP 指向堆栈,如图 3.3 所示,因此能够以 EBP 作为基址寄存器来访问局部变量 z。这种访问局部变量的方法,还是很方便的。这也是在子程序开始时先建立堆栈框架的原因。

3. 局部变量示例二

为了进一步说明 C 语言局部变量的安排,再介绍一个示例。

【例 3-7】 分析如下由 C 语言编写的函数 cf37 的目标代码。

```
int  cf37(int n)
{
    int  i, sum;
    sum=0;
    for(i=1; i<=n; i++)
        sum+=i;
    return  sum;
}
```

函数 cf37 的功能是计算 1 到 n 之间的整数之和。作为示例,有意采用了一个循环来求和,而且安排了两个局部变量 i 和 sum。

假设不要求编译优化,即采用编译优化选项“已禁用”,编译上述 C 语言函数 cf37 后,可得到如下所示的目标代码。

```
cf37      PROC                        ;表示过程(函数)开始
    push  ebp
    mov   ebp, esp                    ;建立堆栈框架
    sub   esp, 8                      ;安排局部变量 i 和 sum
    mov   DWORD PTR [ebp-8], 0        ;sum=0;
    mov   DWORD PTR [ebp-4], 1        ;i=1;
    jmp   SHORT LN3cf37               ;//@j1
                                      ;
LN2cf37:                              ;i++
    mov   eax, DWORD PTR [ebp-4]      ;取出 i
```

```
    add   eax, 1
    mov   DWORD PTR [ebp-4], eax    ;送回 i
LN3cf37:                            ;比较 i 和 n
    mov   ecx, DWORD PTR [ebp-4]
    cmp   ecx, DWORD PTR [ebp+8]
    jg    SHORT LN1cf37             ;如果 i 大于 n,则跳转
                                    ;sum+=i;
    mov   edx, DWORD PTR [ebp-8]
    add   edx, DWORD PTR [ebp-4]
    mov   DWORD PTR [ebp-8], edx
    jmp   SHORT LN2cf37             ;//@j2
                                    ;
LN1cf37:
    mov   eax, DWORD PTR [ebp-8]    ;准备返回参数
    mov   esp, ebp                  ;撤销局部变量 i 和 sum
    pop   ebp                       ;撤销堆栈框架
    ret
cf37    ENDP                        ;表示过程( 函数 )结束
```

从上述目标代码可知,对应源程序中的局部变量 *i* 和 sum 在堆栈中,分别是[EBP－4]和[EBP－8]所指定的存储单元。函数 cf37 执行阶段堆栈的变化情况也类似于图 3.3,不同之处是减少一个参数,增加一个局部变量。读者可以自行画出堆栈变化示意图。

还请留意上述目标代码中注释带有//@j1 和//@j2 行的无条件转移指令。在//@j1 行的无条件转移指令,跳过了调整循环变量的代码。在//@j2 行的无条件转移指令,转移到调整循环变量处。这与循环的具体实现有关。

3.2 算术逻辑运算指令

对数据的处理往往包含大量的算术和逻辑运算,因此处理器通常都会提供算术运算指令和逻辑运算指令,利用这些指令可以进行各种算术、逻辑运算操作。在第 2 章中已经介绍过作为算术运算指令的加减运算指令,本节先介绍乘除运算指令,然后再介绍逻辑运算指令和移位指令。

3.2.1 乘除运算指令

乘除运算指令分为无符号数运算指令和有符号数运算指令,这与加减运算指令不同。乘除运算指令对状态标志的影响,也没有加减运算指令那样自然。

在乘除运算指令中,有时操作数是隐含的。

1. 使用乘除运算指令的示例

在具体介绍乘除运算指令之前,先介绍使用乘除运算指令的示例。

【例 3-8】 观察如下由 C 语言编写的函数 cf38 的目标代码。这仅仅是个示例,其中两个参数都是整型。

```
int  cf38( int x, int y)
{
    return  ( x*x+3)/( 168*y) ;
}
```

采用编译优化选项“使速度最大化”，在编译后可得到如下所示的用汇编格式指令表示的目标代码，分号之后是添加的注释。

```
push  ebp
mov   ebp, esp                    ;建立堆栈框架
mov   eax, DWORD PTR [ebp+8]     ;取出参数 x
mov   ecx, DWORD PTR [ebp+12]    ;取出参数 y
imul  eax, eax                   ;实现 x*x,保存在 EAX
imul  ecx, 168                   ;实现 168*y,保存在 ECX
add   eax, 3                     ;实现 x*x+3
cdq                              ;把 EAX 符号扩展到 EDX(形成 64 位被除数)
idiv  ecx                        ;除运算,EDX 及 EAX 是 64 位被除数,ECX 是除数
pop   ebp                        ;撤销堆栈框架
ret                              ;返回
```

从上述目标代码可知，函数 cf38 采用 3.1.2 节介绍的堆栈传递参数方法。在计算表达式(x * x+3)/(168 * y)之前，先从堆栈中取得参数 x 和 y。然后，利用乘法和除法等指令计算该表达式的值。最后通过寄存器 EAX 返回结果。

2. 无符号数乘法指令(Unsigned Multiply，MUL)

无符号数乘法指令的格式如下：

```
MUL   OPRD
```

这条指令实现两个无符号操作数的乘法运算。指令看似只有一个操作数 OPRD，实际上，另一个操作数是隐含的，位于寄存器 AL、AX 或者 EAX 中(这取决于操作数 OPRD 的尺寸)。如果 OPRD 是字节操作数，则把 AL 中的无符号数与 OPRD 相乘，16 位结果送到 AX 中；如果 OPRD 是字操作数，则把 AX 中的无符号数与 OPRD 相乘，32 位结果送到寄存器对 DX：AX 中，DX 含高 16 位，AX 含低 16 位。如果 OPRD 是双字操作数，则把 EAX 中的无符号数与 OPRD 相乘，64 位结果送到寄存器对 EDX：EAX 中，EDX 含高 32 位，EAX 含低 32 位。所以由操作数 OPRD 决定是字节相乘，或是字相乘，还是双字相乘。

操作数 OPRD 可以是通用寄存器，也可以是存储单元，但不能是立即数。

【例 3-9】 如下指令演示无符号乘法指令的使用。

```
MUL   BL
MUL   ECX
```

如果乘积的高半部分(字节相乘时为 AH，字相乘时为 DX，双字相乘时为 EDX)不等于零，则标志 CF=1，OF=1；否则 CF=0，OF=0。所以，如果 CF=1 和 OF=1 表示在 AH、DX 或者 EDX 中含有结果的有效数。该指令对其他状态标志无定义。

3. 有符号数乘法指令(sIgned MULtiply，IMUL)

有符号数乘法指令能够实现两个有符号操作数的乘法运算。根据指令中显式的操作数的个数，它有如下 3 种格式：

```
IMUL   OPRD
IMUL   DEST, SRC
IMUL   DEST, SRC1, SRC2
```

(1) 单操作数形式。单操作数乘法指令实际上有一个隐含的操作数，位于寄存器 AL、AX 或者 EAX 中(这取决于操作数 OPRD 的尺寸)。它把被乘数和乘数均作为有符号数，此外与

无符号乘法指令 MUL 完全类似。

【例 3-10】 如下指令演示单操作数形式的有符号乘法指令的使用。

```
IMUL  CL
IMUL  DWORD PTR [EBP+12]          ;双字存储单元
```

(2) 双操作数形式。双操作数乘法指令实现两个操作数相乘,把乘积送到目的操作数 DEST,即:

```
DEST ⇐ DEST*SRC
```

目的操作数 DEST 只能是 16 位或者 32 位通用寄存器。但源操作数 SRC 不仅可以是通用寄存器或存储单元(须与目的操作数尺寸一致),还可以是一个立即数(尺寸不能超过目的操作数)。

在例 3-8 中就利用了双操作数有符号乘法指令计算表达 x * x 和表达式 168 * y。

(3) 三操作数形式。三操作数乘法指令实现操作数 SRC1 与操作数 SRC2 相乘,把乘积送到目的操作数 DEST,即:

```
DEST ⇐ SRC1*SRC2
```

目的操作数 DEST 只能是 16 位或者 32 位通用寄存器。源操作数 SRC1 可以是通用寄存器或存储单元(须与目的操作数尺寸一致),但不能是立即数。源操作数 SRC2 只能是一个立即数(尺寸不能超过目的操作数)。

【例 3-11】 如下指令演示三操作数形式的有符号乘法指令的使用。

```
IMUL  AX, CX, 3
IMUL  EDX, DWORD PTR [ESI], 5
```

可以把双操作数乘法指令理解为三操作数乘法指令的特殊情形。例如:

```
IMUL  AX, 7
IMUL  AX, AX, 7
```

对于双操作数或者三操作数乘法指令而言,由于存放乘积的目的操作数的尺寸与被乘数(或乘数)的尺寸相同,因此乘积有可能溢出。如果乘积溢出,那么高位部分将被截掉。这也解释了作为源操作数的立即数,其尺寸不能超过目的操作数尺寸的原因。

虽然双操作数或者三操作数乘法指令是有符号数乘法指令,但在不考虑溢出的情况下,也可以用于无符号数乘法操作。因为无论操作数是有符号数还是无符号数,乘积的低位部分是相同的。这也解释了无符号乘法指令没有双操作数形式和三操作数形式的原因。

对于单操作数乘法指令,如果乘积的高半部分含有有效位,则标志 CF=1,OF=1; 否则 CF=0,OF=0。对于双操作数或者三操作数乘法指令,如果因为溢出而将乘积的高位部分截掉,则标志 CF=1,OF=1; 否则 CF=0,OF=0。因此,在这样的乘法指令后可安排检测 OF 的条件转移指令,用于处理乘积溢出的情况。有符号数乘法指令对其他状态标志无定义。

4. 无符号数除法指令(DIVide,DIV)

无符号数除法指令的格式如下:

```
DIV   OPRD
```

这条指令实现两个无符号操作数的除法运算。指令看似只有一个操作数 OPRD(作为除数),实际上,另一个操作数(作为被除数)是隐含的,位于寄存器 AX、寄存器对 DX:AX 或者寄

存器对 EDX:EAX 中(DX 含有被除数的高 16 位,或者 EDX 含有被除数的高 32 位)。如果 OPRD 是字节操作数,则把 AX 中的无符号数除以 OPRD,所得商送到 AL 中,余数送到 AH 中;如果 OPRD 是字操作数,则把寄存器对 DX:AX 中的无符号数除以 OPRD,所得商送到 AX,余数送到 DX 中。如果 OPRD 是双字操作数,则把寄存器对 EDX:EAX 中的无符号数除以 OPRD,所得商送到 EAX 中,余数送到 EDX 中。所以由操作数 OPRD 决定是字节除,或是字除,还是双字除。

操作数 OPRD 可以是通用寄存器,也可以是存储单元,但不能是立即数。

无符号数除法指令对状态标志的影响无定义。

【例 3-12】 如下指令演示无符号数除法指令的使用。

```
DIV   BL
DIV   ESI
```

除法操作有个特殊情况需要注意。如果除数为 0 或者商太大,则将引起除法出错异常(或者中断)。所谓商太大,是指除法操作后所得商超出了用于存放商的寄存器的范围,即在字节除时商超过字节,或者在字除时商超过字,或者双字除时商超过双字。在第 8 章将介绍实方式下中断的相关概念,在第 9 章将介绍保护方式下异常的相关概念。

【例 3-13】 如下指令演示除法指令因商太大导致除法出错异常。

```
MOV   AX, 600
MOV   BL, 2
DIV   BL
```

如果执行上述代码片段,所得商应该是 300,超出了 AL 的表示范围(无符号数最大 255),将引起除法出错异常。

5. 有符号数除法指令(sIgned DIVide,IDIV)

有符号数除法指令的格式如下:

IDIV *OPRD*

这条指令实现两个有符号操作数的除法运算。指令看似只有一个作为除数的操作数 OPRD,实际上,作为被除数的另一个操作数是隐含的,位于寄存器 AX、寄存器对 DX:AX 或者寄存器对 EDX:EAX 中(取决于操作数 OPRD 的尺寸)。它把被除数和除数均作为有符号数,用于保存商和余数的寄存器与无符号数除法指令 DIV 一样。

如果不能整除,余数的符号与被除数一致,而且余数的绝对值小于除数的绝对值。

如果除数为 0,或者商太大(正数)或者太小(负数),则将引起除法出错异常。

有符号数除法指令对状态标志的影响无定义。

【例 3-14】 如下指令演示有符号数除法指令的使用。

```
IDIV  CL                ;被除数在 AX 中,所得商在 AL,余数在 AH
IDIV  EBX               ;被除数在 EDX:EAX 中,所得商在 EAX,余数在 EDX
```

6. 符号扩展指令

由于除法指令隐含使用字被除数、双字被除数或者四字被除数(寄存器对 EDX:EAX),因此有时需要在除操作之前扩展被除数,也就是增大操作数的尺寸,以便符合要求。另外,为了避免因商太大或者太小引起除法出错异常,需要扩展除数和被除数。IA-32 系列 CPU 专门提供了符号扩展指令。

(1) 字节转换为字指令(Convert Byte to Word,CBW)。字节转换为字指令的格式如下:

```
CBW
```

这条指令把寄存器 AL 中的符号扩展到寄存器 AH。即若 AL 的最高有效位为 0,则 AH=0;若 AL 的最高有效位为 1,则 AH=0FFH,即 AH 的 8 位全都为 1。数值后缀的符号 H 表示十六进制。

【例 3-15】 如下指令演示 CBW 指令的使用,注释部分给出了指令执行后寄存器的内容。

```
MOV   AX, 3487H          ;AX=3487H,即 AH=34H,AL=87H
CBW                      ;AH=0FFH,AL=87H,即 AX=0FF87H
```

假设被除数在 AL 中,CBW 指令使得被除数扩展到 AH,从而产生一个字长度的被除数。

(2) 字转换为双字指令(Convert Word to Double word,CWD)。字转换为双字指令的格式如下:

```
CWD
```

这条指令把寄存器 AX 中的符号扩展到寄存器 DX。即若 AX 的最高有效位为 0,则 DX=0;若 AX 的最高有效位为 1,则 DX=0FFFFH,即 DX 的 16 位全都为 1。

这条指令能在两个字操作数相除以前,产生一个双字长度的被除数。

(3) 双字转换为四字指令(Convert Doubleword to Quadword,CDQ)。双字转换为四字指令的格式如下:

```
CDQ
```

这条指令把寄存器 EAX 中的符号扩展到寄存器 EDX。即若 EAX 的最高有效位为 0,则 EDX=0;若 EAX 的最高有效位为 1,则 EDX=0FFFFFFFFH,即 EDX 的 32 位全都为 1。在例 3-8 的目标代码中,有该指令的运用。

(4) 另一条字转换为双字指令(Convert Word to Double Word,CWDE)。另一条字转换为双字指令的格式如下:

```
CWDE
```

这条指令把寄存器 AX 中的符号扩展到寄存器 EAX 的高 16 位。即若 AX 的最高有效位为 0,则 EAX 的高 16 位都为 0;若 AX 的最高有效位为 1,则 EAX 的高 16 位都为 1。

这条指令类似于 CBW,把 AL 中的符号位扩展到 AX;但不同于 CWD,不是把 AX 的符号位扩展到 DX,而是扩展到 EAX 的高 16 位。

这四条符号扩展指令,不影响各状态标志。

【例 3-16】 如下 C 程序 dp39 及其嵌入汇编代码,演示除法指令和符号扩展指令的使用。

```
#include <stdio.h>
int  main()
{
    int  quotient, remainder;                    //为了输出结果,安排两个变量
    _asm{
        MOV   AX,- 601
        MOV   BL,10
        IDIV  BL                                 //除数是 BL,被除数是 AX
        MOV   BL,AH                              //先临时保存余数
        ;
```

```
        CBW                                    //商在 AL,符号扩展到 AX
        CWDE                                   //AX 符号扩展到 EAX
        MOV  quotient, EAX
        ;
        MOV  AL,BL                             //余数送到 AL
        CBW                                    //AL 符号扩展到 AX
        CWDE                                   //AX 符号扩展到 EAX
        MOV  remainder, EAX
    }
    printf("quotient=%d\n", quotient);     //显示为-60
    printf("remainder=%d\n", remainder);   //显示为-1
    printf("\n");
    //
    _asm{
        MOV  AX,- 601
        CWD                                    //AX 符号扩展到 DX
        MOV  BX,3
        IDIV BX                                //除数是 BX,被除数是 DX:AX
        ;
        CWDE                                   //商在 AX,符号扩展到 EAX
        MOV  quotient,EAX
        ;
        MOV  AX,DX                             //余数 DX 送到 AX
        CWDE                                   //符号扩展到 EAX
        MOV  remainder,EAX
    }
    printf("quotient=%d\n", quotient);     //显示为-200
    printf("remainder=%d\n", remainder);   //显示为-1
    return  0;
}
```

上述演示程序分两部分,第一部分演示 16 位被除数和 8 位除数的情形,在实施除法操作后,利用符号扩展指令把商和余数分别扩展成 32 位,并送到对应变量中。第二部分演示 32 位被除数和 16 位除数的情形。这时虽然实际的被除数可以用 16 位表示,但由于除数太小,实施除法操作会导致溢出,因此必须先把被除数扩展到 32 位表示。当然,也可以把被除数扩展到 64 位,这是解决除法溢出的方法。

注意,在无符号数除操作之前,不宜利用 CBW、CWD 或者 CDQ 指令来扩展符号位,一般采用逻辑异或 XOR 指令把高 8 位、高 16 位或者高 32 位直接清 0,即进行无符号扩展。

7. 扩展传送指令

为了更加高效,IA-32 系列 CPU 还提供了两条扩展传送指令,利用它们可以在传送数据的过程中完成符号扩展或者零扩展(无符号扩展)。

(1) 符号扩展传送指令(Move with Sign-Extension,MOVSX)。符号扩展传送指令的格式如下:

MOVSX ***DEST***, ***SRC***

此指令把源操作数 SRC 符号扩展后送至目的操作数 DEST。

源操作数 SRC 可以是通用寄存器或存储单元，而目的操作数 DEST 只能是通用寄存器。与普通传送指令不同，目的操作数的尺寸必须大于源操作数的尺寸。源操作数的尺寸可以是 8 位或者 16 位；目的操作数的尺寸可以是 16 位或者 32 位。

符号扩展传送指令不会改变源操作数，也不影响标志寄存器中的状态标志。

【例 3-17】 如下指令演示 MOVSX 指令的使用，注释部分给出了指令执行后寄存器的内容，数据末的后缀 H 表示十六进制。

```
MOV    AL, 85H                    ;AL=85H
MOVSX  EDX, AL                    ;EDX=FFFFFF85H
MOVSX  CX, AL                     ;CX=FF85H
MOV    AL, 75H                    ;AL=75H
MOVSX  EAX, AL                    ;EAX=00000075H
```

(2) 零扩展传送指令(Move with Zero-Extend，MOVZX)。零扩展(无符号扩展)传送指令的格式如下：

MOVZX DEST, SRC

此指令把源操作数 SRC 零扩展后送至目的操作数 DEST。

源操作数 SRC 可以是通用寄存器或存储单元，而目的操作数 DEST 只能是通用寄存器。源操作数的尺寸可以是 8 位或者 16 位；目的操作数的尺寸只可以是 16 位或者 32 位。

零扩展(无符号扩展)传送指令不会改变源操作数，也不影响标志寄存器中的状态标志。

【例 3-18】 如下指令演示 MOVZX 指令的使用，注释部分给出了指令执行后寄存器的内容，数据末的后缀 H 表示十六进制。

```
MOV    DX, 8885H                  ;DX=8885H
MOVZX  ECX, DL                    ;ECX=00000085H
MOVZX  EAX, DX                    ;EAX=00008885H
```

【例 3-19】 比较分析如下由 C 语言编写的函数 cf310 和 cf311 的目标代码。这仅仅是示例，请注意参数的类型和函数返回值的类型。

```
int  cf310(char x, char y)
{
    return  (x+22)/y;
}
//
unsigned int  cf311(unsigned char x, unsigned char y)
{
    return  (unsigned)(x+22)/y ;
}
```

在编译后可得到如下所示的用汇编格式指令表示的目标代码，其中“BYTE PTR”表示字节存储单元，分号之后是添加的注释。

```
;函数 cf310 的目标代码
push   ebp
mov    ebp, esp                   ;建立堆栈框架
movsx  eax, BYTE PTR [ebp+8]      ;把参数 x 符号扩展后送到 eax
add    eax, 22
movsx  ecx, BYTE PTR [ebp+12]     ;把参数 y 符号扩展后送到 ecx
```

```
    cdq                                    ;符号扩展,形成 64 位的被除数
    idiv   ecx
    pop    ebp                             ;撤销堆栈框架
    ret
    ;
    ;函数 cf311 的目标代码
    push   ebp
    mov    ebp, esp                        ;建立堆栈框架
    movzx  eax, BYTE PTR [ebp+8]           ;把参数 x 零扩展后送到 eax
    add    eax, 22
    movzx  ecx, BYTE PTR [ebp+12]          ;把参数 y 零扩展后送到 ecx
    xor    edx, edx                        ;零扩展,形成 64 位的被除数 //@z
    div    ecx
    pop    ebp                             ;撤销堆栈框架
    ret
```

从上述目标代码可以清楚地看到符号扩展传送指令 MOVSX 和零扩展传送指令 MOVZX 的作用,还可以看到对于有符号数,采用符号扩展指令 cdq 来形成 64 位的被除数(存放在 EDX:EAX 寄存器对中),对于无符号数,采用逻辑异或指令 xor 直接把 edx 清 0,从而形成 64 位的被除数。

细心的读者也许会发现在 C 函数 cf311 中安排了强制类型转换。这是为了演示相关指令有意安排的。如果不做这样的安排,目标代码又会怎么样?

3.2.2 逻辑运算指令

在 C 语言中有一组按位逻辑运算符,它们是按位取反运算符(~)、按位与运算符(&)、按位或运算符(|)、按位异或运算符(^)。利用这些运算符,可以方便地进行各种位运算。

IA-32 系列 CPU 提供一组逻辑运算指令,包括否(NOT)、与(AND)、或(OR)、异或(XOR)等。这些逻辑运算指令实际的操作就是按位运算,确切地说是按位进行的逻辑运算。

1. 使用逻辑运算指令的示例

【例 3-20】 观察如下由 C 语言编写的函数 cf312 的目标代码。这仅仅是个示例,其中两个参数都是无符号整型。

```
unsigned  int  cf312(unsigned int x, unsigned int y)
{
    int  z=0;
    if((x & 3)||(x-5)|~y))
    {
        z=x ^ 255;
    }
    return  z;
}
```

假设不要求编译优化,即采用编译优化选项"已禁用",编译上述程序后,可得到如下所示的用汇编格式指令表示的目标代码(标号稍有修饰),分号之后是添加的注释。

```
    ;函数 cf312 的目标代码(无编译优化)
    push  ebp
```

```
    mov   ebp, esp                    ;建立堆栈框架
    push  ecx                         ;安排局部变量 z
    mov   DWORD PTR [ebp-4], 0        ;z=0;
                                      ;
    mov   eax, DWORD PTR [ebp+8]      ;取出参数 x
    and   eax, 3                      ;x & 3
    jne   SHORT LN1cf312              ;如结果不为 0,表示条件成立,跳转
                                      ;
    mov   ecx, DWORD PTR [ebp+8]      ;又取出参数 x
    sub   ecx, 5                      ;x-5
    mov   edx, DWORD PTR [ebp+12]     ;取出参数 y
    not   edx                         ;~ y
    or    ecx, edx                    ;(x-5) | ~ y
    je    SHORT LN2cf312              ;如结果为 0,表示条件不成立,跳转
                                      ;
LN1cf312:                             ;条件成立
    mov   eax, DWORD PTR [ebp+8]      ;又取出参数 x
    xor   eax, 255                    ;x ^ 255
    mov   DWORD PTR [ebp-4], eax      ;保存 x
                                      ;
LN2cf312:                             ;准备返回
    mov   eax, DWORD PTR [ebp-4]      ;准备返回值
    mov   esp, ebp                    ;撤销局部变量 z
    pop   ebp                         ;撤销堆栈框架
    ret
```

从上述目标代码可知,函数 cf312 不仅采用 3.1.2 节介绍的堆栈传递参数方法,而且还采用 3.1.3 节介绍的在堆栈中安排局部变量的方法。在计算表达式时,利用了逻辑运算指令,最后通过寄存器 EAX 返回结果。

2. 关于逻辑运算指令的通用说明

IA-32 系列 CPU 提供的逻辑运算指令,除了上述的否(NOT)、与(AND)、或(OR)、异或(XOR)外,还有测试指令 TEST,而且除了指令 NOT 外,均有两个操作数。关于这组指令有如下几点通用说明。

(1) 只有通用寄存器或存储单元可作为目的操作数,用于存放运算结果。

(2) 如果只有一个操作数,则该操作数既是源又是目的。

(3) 如果有两个操作数,那么最多只能有一个是存储单元,源操作数可以是立即数。

(4) 存储单元可采用 2.5 节中介绍的各种存储器操作数寻址方式。

(5) 操作数可以是字节、字或者双字。但如果有两个操作数,则它们的尺寸必须一致。

3. 否运算指令(NOT)

NOT 运算指令的格式如下:

```
NOT   OPRD
```

这条指令把操作数 OPRD 按位“取反”,然后送回 OPRD。按位“取反”是指把为 0 的位设置成 1,把为 1 的位清成 0。

【例 3-21】 如下指令演示 NOT 运算指令的使用。

```
NO    CL
NOT   EAX
```

该指令对标志没有影响。

4. 与运算指令(AND)

AND 运算指令的格式如下：

```
AND   DEST, SRC
```

这条指令对两个操作数进行按位的逻辑“与”运算，结果送到目的操作数 DEST。按位“与”是指当两个操作数对应位都为 1 时，把结果的对应位设置成 1，否则清成 0。

该指令使得标志 CF＝0，标志 OF＝0，标志 PF、ZF、SF 反映运算结果，标志 AF 未定义。

【例 3-22】 如下指令演示“与”运算指令的使用，注释是执行指令后有关寄存器的内容。

```
AND   ECX, ESI
MOV   AX, 3437H          ;AX=3437H
AND   AX, 0F0FH          ;AX=0407H
```

某个操作数自己与自己相“与”，则值不变，但可使进位标志 CF 清为 0。“与”运算指令主要用在使一个操作数中的若干位维持不变，而另外若干位清为 0 的场合。把要维持不变的这些位与“1”相“与”，而把要清为 0 的这些位与“0”相“与”就能达到此目的。

5. 或运算指令(OR)

OR 运算指令的格式如下：

```
OR    DEST, SRC
```

这条指令对两个操作数进行按位的逻辑“或”运算，结果送到目的操作数 DEST。按位“或”是指当两个操作数对应位都为 0 时，把结果的对应位清成 0，否则设置成 1。

【例 3-23】 如下指令演示“或”运算指令的使用。

```
OR    CL, CH
OR    EBX, EAX
```

该指令使标志 CF＝0，标志 OF＝0，标志 PF、ZF、SF 反映运算结果，标志 AF 未定义。

某个操作数自己与自己相“或”，则值不变，但可使进位标志 CF 清为 0。“或”运算指令主要用在使一个操作数中的若干位维持不变，而另外若干位置为 1 的场合。把要维持不变的这些位与“0”相“或”，而把要设置为 1 的这些位与“1”相“或”就能达到此目的。

【例 3-24】 如下指令演示“或”运算指令的使用，注释是执行指令后有关寄存器的内容。

```
MOV   AL, 41H            ;AL=01000001B,后缀 B 表示二进制
OR    AL, 20H            ;AL=01100001B
```

6. 异或运算指令(XOR)

XOR 运算指令的格式如下：

```
XOR   DEST, SRC
```

这条指令对两个操作数进行按位的逻辑“异或”运算，结果送到目的操作数 DEST。按位“异或”是指当两个操作数对应位不同时，把结果的对应位设置成 1，当两个操作数对应位相同时，把结果的对应位清成 0。

该指令使标志 CF＝0，标志 OF＝0，标志 PF、ZF、SF 反映运算结果，标志 AF 未定义。

【例 3-25】 如下指令演示“异或”运算指令的使用，注释是执行指令后有关寄存器的

内容。

```
XOR   ECX, ECX              ;ECX=0,CF=0
MOV   AL, 34H               ;AL=00110100B,后缀 B 表示二进制
MOV   BL, 0FH               ;BL=00001111B
XOR   AL, BL                ;AL=00111011B
```

某个操作数自己与自己相“异或”,则结果总是为 0。这是因为每一个对应位肯定都相同,所以结果的每一位都为 0。因此,上述第一条指令执行后,ECX 为 0。经常会采用异或指令把寄存器的内容清零。例 3-19 函数 cf311 的目标代码中,就利用“xor edx edx”指令将寄存器 EDX 清为 0,实现无符号数的扩展。

“异或”操作指令主要用在使一个操作数中的若干位维持不变,而另外若干位设置取反的场合。把要维持不变的这些位与“0”相“异或”,而把要取反的这些位与“1”相“异或”就能达到此目的。

7. 测试指令(TEST)

测试指令的格式如下:

TEST *DEST, SRC*

这条指令和 AND 指令类似,也是把两个操作数进行按位“与”,但结果不送到目的操作数 DEST,仅仅影响状态标志。该指令执行以后,标志 ZF、PF 和 SF 反映运算结果,标志 CF 和 OF 被清为 0。

该指令通常用于检测某些位是否为 1,但又不希望改变操作数值的场合。就像比较指令 CMP,能够影响状态标志,但不影响操作数的值。

【例 3-26】 如下指令检查 AL 中的位 6 和位 2 是否有一位为 1。

```
TEST  AL, 01000100B         ;后缀 B 表示二进制
```

如果位 6 和位 2 全为 0,那么在执行上面的指令后,ZF 被设置为 1,否则 ZF 被清 0。在程序中,随后的指令可以根据 ZF 进行分支转移。

【例 3-27】 比较分析例 3-20 中函数 cf312 的另一个版本的目标代码。

采用编译优化选项“使速度最大化”,重新编译函数 cf312,可得到如下所示的用汇编格式指令表示的目标代码。

```
;函数 cf312 的目标代码(使速度最大化)
    push  ebp
    mov   ebp, esp                  ;建立堆栈框架
    mov   ecx, DWORD PTR [ebp+8]    ;取出参数 x
    xor   eax, eax                  ;z=0
    test  cl, 3                     ;测试 x 的低 8 位(x & 3)
    jne   SHORT LN1cf312
    push  esi                       ;临时保存 ESI
    mov   esi, DWORD PTR [ebp+12]   ;取出参数 y
    not   esi                       ;~y
    lea   edx, DWORD PTR [ecx- 5]   ;x-5
    or    edx, esi                  ;(x-5)|~y
    pop   esi                       ;恢复 ESI
    je    SHORT LN2cf312
LN1cf312:
```

```
    xor   ecx, 255                ;x ^ 255
    mov   eax, ecx                ;准备返回值
LN2cf312:
    pop   ebp                     ;撤销堆栈框架
    ret
```

从上述目标代码可知，由于采用了“使速度最大化”的编译选项，因此就没有把局部变量安排在堆栈中，而是把 EAX 作为局部变量 z。此外，还使用 TEST 指令代替了 AND 指令。

3.2.3 移位指令

IA-32 系列 CPU 有三大类移位指令：一般移位指令、循环移位指令和双精度移位指令。按移位的方向，这三大类移位指令又可分为左移移位指令和右移移位指令。

移位指令实现把某个通用寄存器或者某个存储单元的内容，按某种方式向左或者向右移动一位或者 m 位。移位指令中要标明需要移位的操作数，这个操作数既是源操作数又是目的操作数；移位指令还要标明需要移动的位数，可以是一个 8 位的立即数，或者是寄存器 CL。

1. 一般移位指令

一般移位指令包括算术左移指令(Shift Arithmetic Left，SAL)、逻辑左移指令(SHift logic Left，SHL)、算术右移指令(Shift Arithmetic Right，SAR)、逻辑右移指令(SHift logic Right，SHR)。这些指令的格式如下：

```
SAL   OPRD, count          ;算术左移指令(同逻辑左移指令)
SHL   OPRD, count          ;逻辑左移指令(同算术左移指令)
SAR   OPRD, count          ;算术右移指令
SHR   OPRD, count          ;逻辑右移指令
```

操作数 OPRD 可以是通用寄存器，也可以是存储器单元，其尺寸可以是字节、字或者双字，如果是存储单元，可以采用各种存储器操作数寻址方式。count 表示移位的位数，可以是一个 8 位立即数，也可以是寄存器 CL。寄存器 CL 表示移位数由 CL 的值决定。通过截取 count 的低 5 位，实际的移位数被限于 0～31 之间。

这些指令执行后，标志 CF 受影响；标志 SF、ZF 和 PF 反映移位后的结果；标志 OF 受影响情况较复杂；标志 AF 未定义。

(1) 算术左移或逻辑左移指令(Shift Arithmetic Left 或 SHift logic Left，SAL/SHL)。算术左移和逻辑左移进行相同的动作，是一样的，为了方便理解，有两个助记符，但只有一条机器指令。

算术左移指令 SAL/逻辑左移指令 SHL 把操作数 OPRD 左移 count 位，同时每向左移动一位，右边用 0 补足一位，移出的最高位进入标志位 CF，如图 3.4(a)所示。

【例 3-28】 如下的代码片段用于说明 SHL 指令的使用，注释给出了指令执行后的操作数值和部分标志的变化情况。

```
MOV   EBX, 7400EF9CH          ;EBX=7400EF9CH
ADD   EBX, 0                  ;EBX=7400EF9CH,CF=0,SF=0,ZF=0,PF=1
SHL   EBX, 1                  ;EBX=E801DF38H,CF=0,SF=1,ZF=0,PF=0
MOV   CL, 3                   ;CL=3
SHL   EBX, CL                 ;EBX=400EF9C0H,CF=1,SF=0,ZF=0,PF=1
SHL   EBX, 16                 ;EBX=F9C00000H,CF=0,SF=1,ZF=0,PF=1
```

```
SHL    EBX, 12                    ;EBX=00000000H,CF=0,SF=0,ZF=1,PF=1
```

只要左移之后的结果未超出一个字节、一个字或者一个双字的表达范围，那么每左移一次，原操作数每一位的权增加了一倍，即相当于原数乘以 2。

【例 3-29】 如下的代码片段实现把寄存器 AL 中的内容（设为无符号数）乘以 10，结果存放在 AX 中。

```
XOR    AH, AH                     ;AH=0
SHL    AX, 1                      ;2*X
MOV    BX, AX                     ;暂存 2*X
SHL    AX, 2                      ;8*X
ADD    AX, BX                     ;8*X+2*X
```

（2）算术右移指令（Shift Arithmetic Right，SAR）。算术右移指令 SAR 把操作数 OPRD 右移 count 位，同时每向右移一位，左边的符号位保持不变，移出的最低位进入标志位 CF，如图 3.4(b)所示。

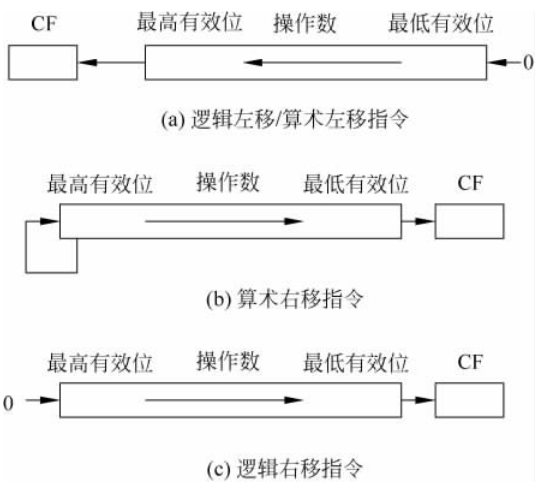

图 3.4 一般移位指令执行示意图

【例 3-30】 如下的程序片段用于说明 SAR 指令的使用，注释给出了指令执行后的操作数值和部分标志的变化情况，请注意符号位（最高有效位）的变化。

```
MOV    DX, 82C3H                  ;DX=82C3H
SAR    DX, 1                      ;DX=C161H,CF=1,SF=1,ZF=0,PF=0
MOV    CL, 3                      ;CL=3
SAR    DX, CL                     ;DX=F82CH,CF=0,SF=1,ZF=0,PF=0
SAR    DX, 4                      ;DX=FF82H,CF=1,SF=1,ZF=0,PF=1
```

对于有符号数或无符号数而言，算术右移一位相当于除以 2。但在非整除的情况下与使用 IDIV 指令不完全一样。

（3）逻辑右移指令（SHift logic Right，SHR）。逻辑右移指令使操作数 OPRD 右移 count 位，同时每向右移一位，左边用 0 补足，移出的最低位进入标志位 CF，如图 3.4(c)所示。

【例 3-31】 如下的代码片段用于说明 SHR 指令的使用，注释给出了指令执行后的操作数值和部分标志的变化情况。

```
MOV    DX, 82C3H                  ;DX=82C3H
```

```
SHR   DX, 1                    ;DX=4161H,CF=1,SF=0,ZF=0,PF=0
MOV   CL, 3                    ;CL=3
SHR   DX, CL                   ;DX=082CH,CF=0,SF=0,ZF=0,PF=0
SHR   DX, 12                   ;DX=0000H,CF=1,SF=0,ZF=1,PF=1
```

对于无符号数而言，逻辑右移一位相当于除以 2。

【例 3-32】 分析如下由 C 语言编写的函数 cf313 的目标代码。这仅仅是个示例，其中两个参数都是无符号整型。

```
unsigned  cf313(unsigned x, unsigned y)
{
    return  (x << 2) - (y >> 4) - (x / 32) - (y * 8);
}
```

假设不要求编译优化，即采用编译优化选项“已禁用”，编译上述程序后，可得到如下所示的用汇编格式指令表示的目标代码，分号之后是添加的注释。

```
;函数 cf313 的目标代码(不要求编译优化)
push  ebp
mov   ebp, esp                 ;建立堆栈框架
mov   eax, DWORD PTR [ebp+8]   ;取得参数 x
shl   eax, 2                   ;把 x 向左移 2 位
mov   ecx, DWORD PTR [ebp+12]  ;取得参数 y
shr   ecx, 4                   ;把 y 向右移 4 位
sub   eax, ecx
mov   edx, DWORD PTR [ebp+8]   ;取得参数 x
shr   edx, 5                   ;无符号整数除以 32
sub   eax, edx
mov   ecx, DWORD PTR [ebp+12]  ;取得参数 y
shl   ecx, 3                   ;乘以 8
sub   eax, ecx                 ;返回结果存放在 EAX 中
pop   ebp                      ;撤销堆栈框架
ret
```

从上述目标代码可知，C 语言中移位运算符的实现；同时还可以看到，2 次幂运算的实现方法。

2. 循环移位指令

循环移位指令包括左循环移位指令(ROtate Left，ROL)、右循环移位指令(ROtate Right，ROR)、带进位左循环移位指令(Rotate through CF Left，RCL)，带进位右循环移位指令(Rotate through CF Right，RCR)。这些指令的格式如下：

```
ROL   OPRD, count              ;左循环移位指令
ROR   OPRD, count              ;右循环移位指令
RCL   OPRD, count              ;带进位左循环移位指令
RCR   OPRD, count              ;带进位右循环移位指令
```

操作数 OPRD 可以是通用寄存器，也可以是存储器单元，其尺寸可以是字节、字或者双字。count 表示移位的位数，可以是一个 8 位立即数，也可以是寄存器 CL。寄存器 CL 表示移位数由 CL 的值决定。通过截取 count 的低 5 位，移位数被限于 0～31 之间，而且为了更加有效，真正的移位数还受到操作数 OPRD 尺寸长度的限制。

这些指令执行后，标志 CF 受影响；标志 OF 受影响情况稍复杂；其他状态标志不受影响。

前两条循环移位指令没有把进位标志位 CF 包含在循环的环中；后两条循环移位指令把进位标志 CF 包含在循环的环中，即作为整个循环的一部分。这 4 条循环移位指令的操作如图 3.5 所示。

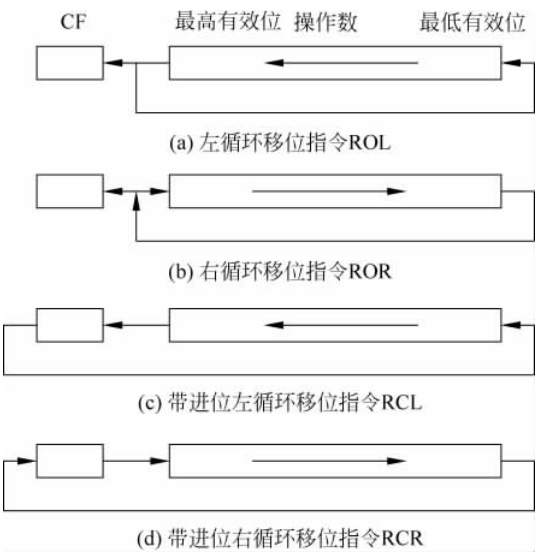

图 3.5　循环移位指令执行示意图

左循环移位指令 ROL 是指把操作数 OPRD 循环左移 count 位。每向左移一位，操作数的最高位移入最低位，同时最高位也移入进位标志 CF。

右循环移位指令 ROR 是指把操作数 OPRD 循环右移 count 位。每向右移一位，操作数的最低位移入最高位，同时最低位也移入进位标志 CF。

带进位左循环移位指令 RCL 是指把操作数 OPRD 连同 CF 循环左移 count 位。每向左移一位，操作数的最高位移入进位标志 CF，CF 移入操作数的最低位，这也被称为大循环左移。

带进位右循环移位指令 RCR 是指把操作数 OPRD 连同 CF 循环右移 count 位。每向右移一位，操作数的最低位移入进位标志 CF，CF 移入操作数的最高位，这也被称为大循环右移。

【例 3-33】 如下的程序片段用于说明循环移位指令的使用，注释给出了指令执行后的操作数值和标志 CF 的情况。

```
MOV   DX, 82C3H              ;DX=82C3H
ROL   DX, 1                  ;DX=0587H, CF=1
MOV   CL, 3                  ;CL=3
ROL   DX, CL                 ;DX=2C38H, CF=0
MOV   EBX, 8A2035F7H         ;EBX=8A2035F7H
ROR   EBX, 4                 ;EBX=78A2035FH, CF=0
STC                          ;CF=1(设置进位标志)
RCL   EBX, 1                 ;EBX=F14406BFH, CF=0
RCR   EBX, CL                ;EBX=DE2880D7H, CF=1
```

利用循环移位指令，能够方便地实现在一个操作数内部的移位。利用带进位循环移位指令，能够实现跨操作数之间的移位。

【例 3-34】　下面的代码片段实现把 AL 的最低位送入 BL 的最低位，仍保持 AL 不变。

```
ROR   BL, 1                    ;BL 循环右移一位
ROR   AL, 1                    ;AL 循环右移一位，最低位到 CF
RCL   BL, 1                    ;BL 带进位左移，带进了来自 AL 的最低位
ROL   AL, 1                    ;恢复 AL
```

3. 双精度移位指令

为了方便地把一个操作数中的部分内容通过移位方式复制到另一个操作数，IA-32 系列 CPU 提供了双精度移位指令。按移动的方向，分为左移和右移，即双精度左移指令 SHLD 和双精度右移指令 SHRD。

双精度移位指令的一般格式如下：

```
SHLD  OPRD1, OPRD2, count
SHRD  OPRD1, OPRD2, count
```

操作数 OPRD1 作为目的操作数可以是通用寄存器，也可以是存储器单元，其尺寸是字或者双字。操作数 OPRD2 相当于源操作数，只能是寄存器，其尺寸必须与操作数 OPRD1 一致。count 表示移位的位数，可以是一个 8 位立即数，也可以是寄存器 CL。寄存器 CL 表示移位数由 CL 的值决定。通过截取 count 的低 5 位，移位数被限于 0～31 之间。

双精度左移指令 SHLD 是指把目的操作数 OPRD1 左移指定的 count 位，在低端空出的位用操作数 OPRD2 高端的 count 位填补，但操作数 OPRD2 的内容保持不变。操作数 OPRD1 中最后移出的位保留在进位标志 CF 中。

双精度右移指令 SHRD 是指把目的操作数 OPRD1 右移指定的 count 位，在高端空出的位用操作数 OPRD2 低端的 count 位填补，但操作数 OPRD2 的内容保持不变。操作数 OPRD1 中最后移出的位保留在进位标志 CF 中。

在截取低 5 位之后，如果 count 为 0，不进行任何操作；如果 count 仍然超过操作数的尺寸，那么目的操作数的结果无定义，各状态标志也无定义。

在仅仅移动一位的情况下，当符号位发生变化，则置溢出标志 OF，否则清 OF。双精度移位指令还影响标志 SF、ZF 和 PF，对 AF 无定义。

【例 3-35】　下面的代码片段用于说明循环移位指令的使用，注释给出了指令执行后的操作数值和标志 CF 的情况。

```
MOV   AX, 8321H
MOV   DX, 5678H
SHLD  AX, DX, 1                ;AX=0642H,DX=5678H,CF=1,OF=1
SHLD  AX, DX, 2                ;AX=1909H,DX=5678H,CF=0,OF=0
;
MOV   EAX, 01234867H
MOV   EDX, 5ABCDEF9H
SHRD  EAX, EDX, 4              ;EAX=90123486H,CF=0,OF=1
MOV   CL, 8
SHRD  EAX, EDX, CL             ;EAX=F9901234H,CF=1,OF=0
```

利用双精度移位指令，能够比较方便地实现跨操作数之间的移位。

【例 3-36】　下面的程序片段实现例 3-34 的要求。

```
SHRD  BX, AX, 1
```

```
ROL   BX, 1
```

【例 3-37】 下面的指令可实现把 EAX 中的 32 位数保存到寄存器对 DX:AX 中。

```
SHLD  EDX, EAX, 16
```

3.3 分支程序设计

几乎所有的程序都不是从头顺序地执行到尾,而是在处理中经常存在着判断,并根据某种条件的判定结果转向不同的处理。这样程序就不再是简单地顺序执行,而是分成两个或多个分支。在 2.6 节介绍条件转移指令和无条件转移指令的基础上,本节先结合 C 语言实例函数的目标代码,介绍分析实现分支的基本方法,然后进一步说明无条件转移指令和条件转移指令。

3.3.1 分支程序设计示例

分支程序的两种基本结构如图 3.6 所示,这两种结构分别对应 C 语言中的 if 语句和 if-else 语句。在汇编语言中,利用条件转移指令和无条件转移指令实现分支。

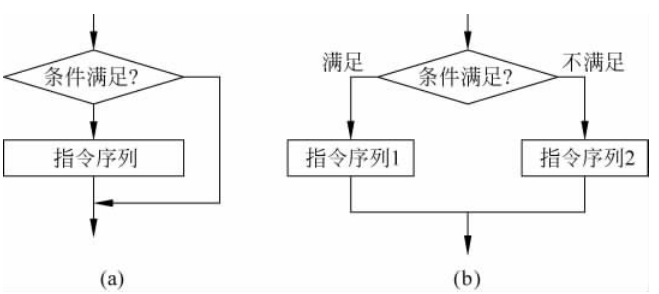

图 3.6 分支程序的结构示意图

1. 简单分支示例

下面先来看两个 C 语言函数的目标代码,从中了解简单分支的实现方法。

【例 3-38】 分析如下 C 语言编写的函数 cf315 的目标代码。

```
int  cf315(int ch)
{
    if(ch >='A' && ch <='Z')
        ch+=0x20;          //小写字母与对应大写字母 ASCII 码相差 32
    return  ch;
}
```

函数 cf315 的功能是字符小写化(如果为大写字母,则转换成小写字母,否则不变)。考虑到通用性和效率,所以参数和返回值都是整型。

假设不要求编译优化,即采用编译优化选项“已禁用”,编译上述程序后,可得到如下所示的用汇编格式指令表示的目标代码(标号稍有修饰)。其中,“DWORD PTR”表示双字存储单元,“SHORT”表示转移目的地就在附近,分号之后是添加的注释。

```
;函数 cf315 的目标代码(不采用编译优化)
push  ebp
```

```
    mov   ebp, esp                    ;建立堆栈框架
                                      ;if( ch >='A' && ch <='Z')
    cmp   DWORD PTR [ebp+8], 65
    jl    SHORT LN1cf315              ;小于,则跳转
    cmp   DWORD PTR [ebp+8], 90
    jg    SHORT LN1cf315              ;大于,则跳转
                                      ;ch+=0x20;
    mov   eax, DWORD PTR [ebp+8]
    add   eax, 32
    mov   DWORD PTR [ebp+8], eax
LN1cf315:                             ;return  ch;
    mov   eax, DWORD PTR [ebp+8]      ;返回值
    pop   ebp                         ;撤销堆栈框架
    ret                               ;返回
```

从上述目标代码可知,在建立堆栈框架后,堆栈中的存储单元[EBP+8]就是入口参数ch。两处分支转移都是属于图3.6(a)的结构,都采用条件转移指令实现。

另外,在例3-3的程序dp32中,采用嵌入汇编代码形式的子程序UPPER实现类似的功能(字符大写化)。但是,它通过寄存器AL传递参数,所以显得简单些。

如果采用编译优化选项"使速度最大化",编译上述程序后,可得到如下所示的目标代码(标号稍有修饰)。

```
    ;函数 cf315 的目标代码(使速度最大化)
    push  ebp
    mov   ebp, esp                    ;建立堆栈框架
                                      ;if( ch >='A' && ch <='Z')
    mov   eax, DWORD PTR [ebp+8]
    lea   ecx, DWORD PTR [eax-65]
    cmp   ecx, 25
    ja    SHORT LN1cf315
                                      ;ch+=0x20;
    add   eax, 32
LN1cf315:                             ;return  ch;
    pop   ebp                         ;撤销堆栈框架
    ret
```

从上述目标代码可知,确实进行了很好的优化处理。首先,减少了一次分支转移。这是很有价值的。源程序中的条件虽然由两个关系表达式组成,但实质是判断是否属于某个区域范围$[x,y]$,因此就被巧妙地合并到一起,成为判断是否属于$[0,y-x]$。减少转移,有助于提高执行效率。因此尽量减少转移,是优化的目标之一。其次,充分利用寄存器,这也是提高执行效率的手段。

【例3-39】 分析把一位十六进制数转换为对应ASCII码的目标代码。

十六进制数需要用16个符号来表示。除了0～9外,还用了6个字母,也就是大写的字母A～F,或者小写的字母a～f。为了输出数据,通常需要把数据转换成对应的字符串,也就是由数据的每一位所对应字符构成的字符串,而字符一般用ASCII码表示,这样也就形成了ASCII码串。

假定十六进制数用0～9和大写字母A～F来表示,那么一位十六进制数与对应ASCII码

的关系如表 3.1 所示。

表 3.1　一位十六进制数与对应 ASCUII 码的关系

十六进制数	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
ASCII 码	30H	31H	32H	33H	34H	35H	36H	37H	38H	39H	41H	42H	43H	44H	45H	46H

这种对应关系可表示为一个分段函数：

$$y=\begin{cases} x+30\text{H} & (0\leqslant x\leqslant 9) \\ x+37\text{H} & (0\text{AH}\leqslant x\leqslant 0\text{FH}) \end{cases}$$

如下所示 C 语言编写的函数 cf316，能够实现上述的转换功能。

```
int  cf316(int m)                    //入口参数为一位十六进制数
{
    m=m & 0x0f;                      //确保一位十六进制数(在 0~15 之间)
    if(m < 10)
        m+=0x30;                     //数字符 0~ 9
    else
        m+=0x37;                     //字母 A~ F
    return  m;
}
```

假设不要求编译优化，编译上述程序后，可得到如下所示的目标代码(标号稍有修饰)。其中，“SHORT”表示转移目的地就在附近。

```
    ;cf316 目标代码(禁止编译优化)
    push  ebp
    mov   ebp, esp                   ;建立堆栈框架
                                     ;m=m & 0x0f;
    mov   eax, DWORD PTR [ebp+8]
    and   eax, 15
    mov   DWORD PTR [ebp+8], eax
                                     ;if(m < 10)
    cmp   DWORD PTR [ebp+8], 10
    jge   SHORT LN2cf316
                                     ;m+=0x30;
    mov   ecx, DWORD PTR [ebp+8]
    add   ecx, 48
    mov   DWORD PTR [ebp+8], ecx
    jmp   SHORT LN1cf316             ; //@j
LN2cf316:                            ;m+=0x37;
    mov   edx, DWORD PTR [ebp+8]
    add   edx, 55
    mov   DWORD PTR [ebp+8], edx
LN1cf316:                            ;return  m;
    mov   eax, DWORD PTR [ebp+8]     ;EAX 含返回值
    pop   ebp                        ;撤销堆栈框架
    ret
```

从上述目标代码可知，在堆栈中的存储单元[EBP+8]就是参数 m。由于没有要求编译优

化，因此每次操作形参 m 后，都又保存到存储单元中，这样整个代码显得比较冗长。代码的分支结构如图 3.6(b)所示，其中利用无条件转移指令 jmp 跳过对应“else”部分的代码，这样在完成分支之后，两部分又合并到了一起，参见代码中注释带//@j 的行。

采用编译优化选项“使速度最大化”，编译上述程序后，可得到如下所示的目标代码：

```
    ;cf316 目标代码(使速度最大化)
    push  ebp
    mov   ebp, esp                    ;建立堆栈框架
                                      ;m=m & 0x0f;
    mov   eax, DWORD PTR [ebp+8]
    and   eax, 15
                                      ;if(m < 10)
    cmp   eax, 10
    jge   SHORT LN2cf316
                                      ;m+=0x30;
    add   eax, 48
    pop   ebp
    ret
LN2cf316:                             ;m+=0x37;
    add   eax, 55
    pop   ebp
    ret
```

从上述目标代码可知，在两个方面进行了优化处理。一方面，把寄存器 EAX 作为形参变量。在从堆栈中取得参数 *m* 后，对形参 *m* 的操作就在寄存器 EAX 中进行了。另一方面，避免了无条件转移指令 jmp。分支的两部分，各自完成后不再合并到一起，而是直接返回。虽然经过了编译优化，但似乎还可以进一步优化。

2. 分支的优化

减少转移是优化的目标之一。从上述两个经过编译优化的目标代码都可以清楚地看到这一点。虽然现代的编译器能够实施优化，但源程序自身结构的优化仍然很重要。

一般情况下，如果分支结构为图 3.6(b)，并且其中一个分支比较简单时，可考虑把它改变转换为图 3.6(a)的结构。具体方法为：在判断之前先假设满足简单的情形。

【例 3-40】 优化例 3-39 的源程序。

对例 3-39 程序中分支的一边稍作变形，可使其包含分支的另一边，分支结构就从图 3.6(b)退化为图 3.6(a)。如下所示的函数 cf317 是优化后的源程序。如此调整分支后，使得处理既简单又高效。

```
int  cf317(int m)
{
    m=m & 0x0f;
    m+=0x30;
    if(m > '9')
        m+=7;
    return  m;
}
```

采用编译优化选项“使速度最大化”，编译上述程序后，可得到如下所示的目标代码：

```
    ;cf317 目标代码(使速度最大化)
    push  ebp
    mov   ebp, esp
    mov   eax, DWORD PTR [ebp+8]
    and   eax, 15                    ;m=m & 0x0f;
    add   eax, 48                    ;m+=0x30;
    cmp   eax, 57                    ;if(m > '9')
    jle   SHORT LN1cf317
    add   eax, 7                     ;m+=7;
LN1cf317:
    pop   ebp
    ret
```

从上述目标代码可知,只含一个条件转移指令。相比例 3-39 两个版本的目标代码,它确实既简单又高效。

3.3.2 无条件和条件转移指令

在 2.6.4 节和 2.6.2 节介绍过无条件转移指令和条件转移指令,为了更好地应用这些指令实现分支程序设计,本节作更进一步的介绍。

1. 相关基本概念

存储器的分段管理使得转移稍显复杂。由于程序代码可分为多个段,因此根据转移时是否重置代码段寄存器 CS 的内容,转移又可分为段内转移和段间转移两大类。段内转移是指仅仅重新设置指令指针寄存器 EIP 的转移,由于没有重置 CS,因此转移后继续执行的指令仍在同一个代码段中。段间转移是指不仅重新设置 EIP,而且重新设置代码段寄存器 CS 的转移,由于重置 CS,因此转移后继续执行的指令在另一个代码段中。段内转移也称为近转移,段间转移也称为远转移。

条件转移指令和循环指令(将在 3.4 节介绍)只能实现段内转移。无条件转移指令和过程调用及返回指令既可以是段内转移,也可以是段间转移。软中断指令和中断返回指令一定是段间转移,将在后面的章节中介绍。

对无条件转移指令和过程调用指令而言,按给出转移目的地址的方式不同,还可分为直接转移和间接转移两种。在转移指令中直接给出转移目的地址的转移称为直接转移。在转移指令中没有直接给出转移目的地址,但给出了包含转移目的地址的寄存器或者存储单元,这样的转移称为间接转移。

2. 无条件转移指令

如上所述,无条件转移指令可分为 4 种:段内直接转移、段内间接转移、段间直接转移和段间间接转移。无条件转移指令均不影响标志寄存器中的状态标志。

(1) 无条件段内直接转移指令。在 2.6.4 节介绍的无条件转移指令就是段内直接转移指令。这是用得最多的无条件转移指令,在前面章节的示例中,已经多次用到。该指令的书写格式如下:

```
JMP   LABEL
```

标号 LABEL 用于表示转移的目标位置,或者说转移目的地。

无条件段内直接转移指令的对应机器指令由操作码和地址差值两部分组成,其格式如

图 3.7 所示。开始部分是无条件段内直接转移指令的操作码，随后有若干个字节表示的地址差值。

图 3.7　相对转移指令机器码的格式

这里的地址差 rel，是转移目标地址偏移（标号 LABEL 所指定指令的地址偏移）与紧随 JMP 指令的下一条指令的地址偏移之间的差值。汇编器在汇编过程中可以计算出地址差 rel。因此，在执行无条件段内直接转移指令时，实际的动作是把指令中的地址差 rel 加到指令指针寄存器 EIP 上，使 EIP 的内容为转移目标地址偏移，从而实现转移。

无条件段内直接转移指令中的地址差 rel 可用 8 位（一个字节）表示，也可用 32 位（4 个字节）或者 16 位（2 个字节）来表示。如果只用 8 位表示地址差，就称为短（short）转移；否则就称为近（near）转移。由于差值是一个有符号数，因此无条件转移指令可以实现向前方（未来）转移，也可以实现向后方（过往）转移。8 位表示的地址差的范围为－128～＋127，转移的范围比较有限。在保护方式下（32 位代码段），地址差也可以用 32 位来表示，这样就可以转移到段内的任何有效目标地址。

这种利用目标地址与转移指令所处地址之间的差值来表示转移目标地址的转移方式，称为相对转移。相对转移有利于程序的浮动。

如果当汇编器汇编到某条转移指令时能够正确地计算出地址差 rel，那么汇编器就根据地址差的大小，决定采用 8 位表示，还是采用 32 位（或者 16 位）表示。否则，汇编器可能会用较多的位数来表示地址差。如果程序员在写程序时能估计出用 8 位就可以表示地址差，那么可在标号前加一个汇编器操作符“SHORT”。在前面许多示例的目标代码中，经常出现的符号“SHORT”，就是表示转移目的地就在附近，用一个字节就可以表示地址差。

（2）无条件段内间接转移指令。无条件段内间接转移指令的使用格式如下：

```
JMP    OPRD
```

该指令使控制无条件地转移到由操作数 OPRD 的内容给定的目标地址处。在保护方式下（32 位代码段），OPRD 是 32 位通用寄存器或者双字存储单元，其内容直接被装入指令指针寄存器 EIP，从而实现转移。

【例 3-41】 如下指令演示了无条件段内间接转移指令的使用，这仅仅是个示例。

```
JMP    ECX                    ;目标地址是 ECX 寄存器的内容
JMP    DWORD PTR [EBX]        ;目标地址是由 EBX 所指向的双字存储单元内容
```

【例 3-42】 如下 C 语言程序 dp318 演示无条件段内转移指令的使用。

```
#include <stdio.h>
    char  flag1='0', flag2='0', flag3='0';
    int  ptonext;                    //存放转移地址
int  main()
{
    _asm{
        LEA   EAX, STEP2             //取得第二步的开始地址
        MOV   ptonext, EAX           //保存到存储单元
        ;
```

```
        LEA   EDX, STEP1            //取得第一步的开始地址
        JMP   EDX                   //转移到第一步(段内间接转移)
        ;
    STEP2:
        MOV   flag2, 'B'
        JMP   STEP3                 //转移到第三步(段内直接转移)
        ;
    STEP1:
        MOV   flag1, 'A'
        JMP   ptonext               //转移到第二步(段内间接转移)
        ;
    STEP3:
        MOV   flag3, 'C'
    }
    printf("%c,%c,%c\n",flag1,flag2,flag3) ; //显示为 A,B,C
    return  0;
}
```

上述程序 dp318 的嵌入汇编代码部分,演示了通过寄存器或存储单元实现无条件段内间接转移,也再次演示了无条件段内直接转移。

(3) 无条件段间转移指令。无条件段间直接转移指令的使用格式与上述的无条件段内直接转移指令相类似,无条件段间间接转移指令的使用格式和上述的无条件段内间接转移指令相类似。但由于涉及改变代码段寄存器 CS 的内容,因此较为复杂,将在第 6 章和第 9 章中介绍。

3. 条件转移指令

在 2.6.2 节中对条件转移指令进行过介绍,列出了所有条件转移指令。

虽然条件转移指令通常根据标志寄存器中的状态标志来判断条件是否满足,但条件转移指令本身的执行不影响状态标志。条件转移指令只局限于段内转移。

条件转移指令也采用相对转移方式。条件转移指令机器码的格式如图 3.7 所示。首先是表示各种条件转移指令的操作码部分,随后有若干个字节表示 EIP 的当前值与转移目的地偏移之间的差值。当条件满足时,就把这个差值加到 EIP 上,从而使 EIP 等于标号所代表的偏移,这样,取下一条指令时就取出标号处的指令了,也就实现了转移。由于差值是一个有符号数,因此条件转移指令可以实现向前方(未来)转移,也可以实现向后方(过往)转移。

条件转移指令的地址差 rel 可用 8 位(1 个字节)表示,也可用 32 位(4 个字节)表示,或者用 16 位(2 个字节)表示。当用 8 位表示地址差时,表示的范围为－128～＋127,因此转移的范围比较有限。在保护方式下(32 位代码段),地址差也可以用 32 位来表示,这样条件转移指令就可以转移到段内的任何有效目标地址。

在前面的示例中,已经多次使用了多种条件转移指令。

3.3.3 多路分支的实现

当根据某个变量的值,进行多种不同处理时,就产生了多路分支。多路分支的结构如图 3.8 所示。虽然任何复杂的多路分支总可分解成多个简单分支,但基于简单分支实现多路分支效率不高。在 C 语言中,常用 switch 语句实现多路分支。在汇编语言中,如何实现多路

分支呢？通过无条件间接转移指令和目标地址表来实现多路分支。

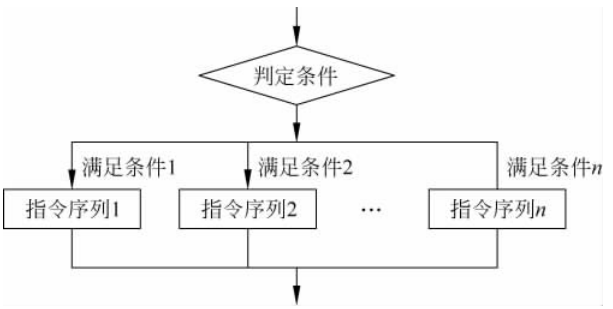

图 3.8　多路分支结构示意图

【例 3-43】 分析 C 语言中 switch 语句的实现。下面的函数 cf319 根据参数 operation 的值进行不同的处理，利用 switch 语句实现。

```
int  cf319(int x, int operation)
{
    int  y;
    //多路分支
    switch(operation) {
        case 1:
            y=3*x;
            break;
        case 2:
            y=5*x+6;
            break;
        case 4:
        case 5:
            y=x*x ;
            break;
        case 8:
            y=x*x+4*x;
            break;
        default:
            y=x ;
    }
    if(y>1000)
        y=1000;
    return  y;
}
```

为了演示 switch 语句的实现，刻意没有安排连续的 case 值，虽然从 1 开始到 8，但中间缺少了 3、6 和 7 的情形。

如果采用编译优化选项“使速度最大化”，编译上述程序后，可得到如下所示的目标代码，其中“DWORD　PTR”表示双字存储单元，“SHORT”表示转移目的地址就在附近，只需用一个字节就可以表示地址差。

```
;函数 cf319 目标代码
```

```
    push  ebp
    mov   ebp, esp                          ;建立堆栈框架
                                            ;switch( operation) {
    mov   eax, DWORD PTR [ebp+12]           ;取得参数 operation( case 值)
    dec   eax                               ;从 0 开始计算,所以先减去 1
    cmp   eax, 7                            ;从 0 开始计算,最多就是 7
    ja    SHORT LN2cf319                    ;超过,则转 default
    ;
    jmp   DWORD PTR LN12cf319[eax*4]        ;实施多路分支 //@j
    ;
LN6cf319:                                   ;case 1:
                                            ;y=3* x;
    mov   eax, DWORD PTR [ebp+8]
    lea   eax, DWORD PTR [eax+ eax*2]
    jmp   SHORT LN7cf319                    ;break;
    ;
LN5cf319:                                   ;case 2:
                                            ;y=5* x+6;
    mov   eax, DWORD PTR [ebp+8]
    lea   eax, DWORD PTR [eax+ eax*4+6]
    jmp   SHORT LN7cf319                    ;break;
    ;
LN4cf319:                                   ;case 4:
                                            ;y=x* x ;
    mov   eax, DWORD PTR [ebp+8]
    imul  eax, eax
    jmp   SHORT LN7cf319                    ;break;
    ;
LN3cf319:                                   ;case 8:
                                            ;y=x* x+4* x;
    mov   ecx, DWORD PTR [ebp+8]
    lea   eax, DWORD PTR [ecx+4]
    imul  eax, ecx
    jmp   SHORT LN7cf319                    ;break;
    ;
LN2cf319:                                   ;default:
                                            ;y=x ;
    mov   eax, DWORD PTR [ebp+8]
                                            ;}
LN7cf319:                                   ;if( y > 1000)
    cmp   eax, 1000
    jle   SHORT LN1cf319
                                            ;y=1000;
    mov   eax, 1000
LN1cf319:                                   ;return  y;
    pop   ebp                               ;撤销堆栈框架
    ret
    ;
```

```
LN12cf319:                              ;多向分支目标地址表
    DD    LN6cf319                      ;case 1
    DD    LN5cf319                      ;case 2
    DD    LN2cf319                      ;default
    DD    LN4cf319                      ;case 4
    DD    LN4cf319                      ;case 5
    DD    LN2cf319                      ;default
    DD    LN2cf319                      ;default
    DD    LN3cf319                      ;case 8
```

把上述目标代码与源程序对比分析，可以很清楚地看到实现各路分支的具体指令。与break语句对应的是一条无条件转移指令，由它跳转到switch语句结束处。

在上述目标代码中，指令“jmp DWORD PTR LN12cf319[eax * 4]”是关键的一条指令。它实现无条件段内间接转移，根据case值，转移到对应的分支处。在目标代码的尾部有一张目标地址表，每项(4个字节)存放一路分支的入口地址。对应跳空case值的“不存在”分支，安排了default分支的入口地址。从目标代码可见，标号LN12cf319是该目标地址表的起始地址偏移，于是有效地址表达式LN12cf319[eax * 4]，表示目标地址表中的某一项的有效地址。所以，上述无条件段内间接转移指令jmp的执行，就是从目标地址表中取得一路分支的入口地址送到EIP，从而实现转移。获取目标地址表中的哪一项由寄存器EAX决定，也就是由case值决定。

当多路分支在5路以上时，可以考虑利用无条件间接转移指令和目标地址表来实现多路分支，这种实现方法既方便又高效。

3.4 循环程序设计

当需要重复某些操作时，就应该考虑使用循环方式。循环结构是程序的基本结构之一，通常，一个程序总会包含循环。本节先结合C语言实例函数的目标代码介绍实现循环的基本方法，然后介绍专门的循环指令，最后介绍多重循环的实现。

3.4.1 循环程序设计示例

循环通常由4个部分组成：初始化部分、循环体部分、调整部分、控制部分。各部分之间的关系如图3.9所示。图3.9(a)是先执行后判断的结构，图3.9(b)是先判断后执行的结构。有时这4个部分可以简化，形成互相包含交叉的情况，不一定能明确分成4个部分。

有多种方法可实现循环的控制，常用的有计数控制法和条件控制法等。

1. 先执行后判断的示例

在C语言中，do-while循环控制语句是先执行后判断的。

【例3-44】 分析如下C语言编写的函数cf320的目标代码。

```
int  cf320(unsigned int n)
{
    int  len=0;
    do{
        len++;
        n=n/10;
```

```
    } while( n !=0) ;
    return  len ;
}
```

函数 cf320 的功能是统计无符号整数 n 作为十进制数时的位数。局部变量 len 用于存放整数的位数。由于至少一位，因此采用了 do-while 循环结构。每次把整数 n 除以 10，直到 n 为 0 结束循环。

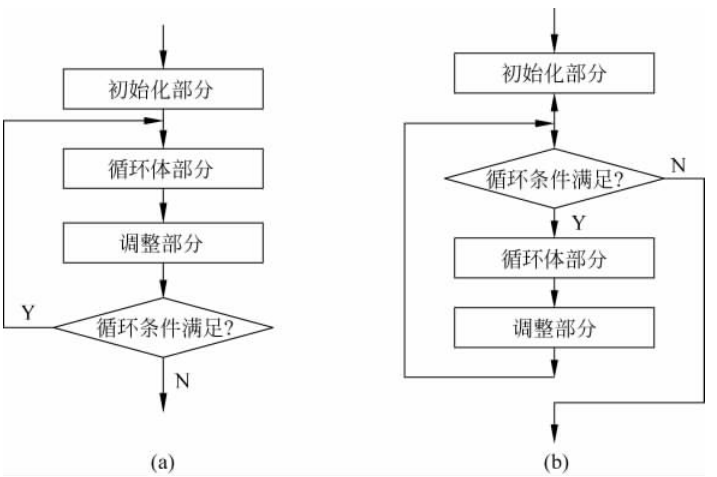

图 3.9　循环结构示意图

假设不要求编译优化，即采用编译优化选项“已禁用”，编译上述程序后，可得到如下所示的用汇编格式指令表示的目标代码。其中，“DWORD PTR”表示双字存储单元，“SHORT”表示转移目的地就在附近，分号之后是添加的注释。

```
    ;函数 cf320 的目标代码( 不采用编译优化 )
    push  ebp
    mov   ebp, esp                    ;建立堆栈框架
    push  ecx                         ;使堆栈指针 ESP-4,安排局部变量 len
    mov   DWORD PTR [ebp-4], 0        ;len=0;
LN3cf320:                             ;do{
                                      ;len++;
    mov   eax, DWORD PTR [ebp-4]
    add   eax, 1
    mov   DWORD PTR [ebp-4], eax
                                      ;n=n/10;
    mov   eax, DWORD PTR [ebp+8]
    xor   edx, edx                    ;因 n 是无符号数,用 XOR 指令清 0
    mov   ecx, 10
    div   ecx
    mov   DWORD PTR [ebp+8], eax
                                      ;} while( n !=0) ;
    cmp   DWORD PTR [ebp+8], 0
    jne   SHORT LN3cf320
                                      ;return  len ;
    mov   eax, DWORD PTR [ebp-4]      ;准备返回值
```

```
                                        ;}
    mov   esp, ebp                      ;撤销局部变量 len
    pop   ebp                           ;撤销堆栈框架
    ret
```

从上述目标代码可知，函数 cf320 不仅通过堆栈传递参数，而且还在堆栈中安排了局部变量 len。如同在 3.1.3 中的介绍，在建立堆栈框架后，通过一条 push 指令安排局部变量 len 的存储单元。这样存储单元[ebp－4]就是 len，存储单元[ebp＋8]是形参 n。

上述目标代码采用如图 3.9(a)所示的循环结构。这个循环的 4 个部分俱全：在初始化部分设置 len 的初值；循环体部分比较简单，只是增加计数；可以认为计算 $n/10$ 是循环调整部分的内容；根据 n 是否为 0，控制循环。

如果采用编译优化选项“使大小最小化”，编译上述程序后，可得到如下所示的目标代码(标号稍有修饰)。

```
    ;函数 cf320 的目标代码(采用" 使大小最小化 " 编译优化选项)
    push  ebp
    mov   ebp, esp
                                        ;ECX 作为 len
    xor   ecx, ecx                      ;len=0;
    push  esi                           ;在使用 ESI 之前,保护之
LL3cf320:
                                        ;do{
                                        ;len++;
                                        ;n=n/10;
    mov   eax, DWORD PTR [ebp+8]
    push  10                            ;准备借助堆栈送到 ESI
    xor   edx, edx                      ;使得 EDX=0
    pop   esi                           ;使得 ESI=10
    div   esi
    inc   ecx
    mov   DWORD PTR [ebp+8], eax
                                        ;} while(n !=0) ;
    test  eax, eax                      ;测试 n 是否为 0?
    jne   SHORT LL3cf320
                                        ;return len ;
    mov   eax, ecx                      ;准备返回值
    pop   esi                           ;恢复 ESI
                                        ;}
    pop   ebp
    ret
```

从上述目标代码可知，由于采用了“使大小最小化”的编译优化选项，因此把寄存器 ECX 作为局部变量 len。此外，还利用指令“push　10”和“pop　esi”使得 ESI 为 10，虽然有两条指令，但机器码的字节数却要少。由此可见，编译器是“费尽心机”。

2. 先判断后执行示例一

在 C 语言中，while 循环控制语句是先判断后执行的。

【例 3-45】 分析如下 C 语言编写的函数 cf321 的目标代码。

```
int  cf321(char * str)
{
    char *pc=str;
    while(*pc)
        pc++;
    return  (pc-str);
}
```

函数 cf321 的功能是测量字符串 str 的长度。局部变量 pc 是字符型指针变量,指向字符串中的字符。字符串可能为空串,所以采用 while 循环结构。依次逐一检查字符串中的字符,如果没有遇到结束标记(0),就调整指针,否则结束循环。

假设不要求编译优化,即采用编译优化选项“已禁用”,编译上述程序后,可得到如下所示的目标代码。

```
    ;函数 cf321 的目标代码(不采用编译优化)
    push  ebp
    mov   ebp, esp                    ;建立堆栈框架
    push  ecx                         ;安排局部变量 pc
                                      ;pc=str;
    mov   eax, DWORD PTR [ebp+8]      ;取得 str
    mov   DWORD PTR [ebp- 4], eax     ;送到 pc
LN2cf321:
                                      ;while(*pc)
    mov   ecx, DWORD PTR [ebp- 4]     ;取得 pc
    movsx edx, BYTE PTR [ecx]         ;EDX=*pc
    test  edx, edx                    ;判断所指向的字符是否为结束标记
    je    SHORT LN1cf321              ;遇到结束标记,则跳转
                                      ;pc++;
    mov   eax, DWORD PTR [ebp- 4]
    add   eax, 1
    mov   DWORD PTR [ebp- 4], eax
    jmp   SHORT LN2cf321              ;无条件跳转 //@j
LN1cf321:
                                      ;return(pc-str);
   mov    eax, DWORD PTR [ebp- 4]
   sub    eax, DWORD PTR [ebp+8]
                                      ;}
    mov   esp, ebp                    ;撤销局部变量 pc
    pop   ebp                         ;撤销堆栈框架
    ret
```

在上述目标代码中,采用与例 3-44 目标代码中相同的方法,在堆栈中安排了局部变量 pc。这样存储单元[ebp-4]就是 pc,存储单元[ebp+8]是形参 str。值得指出,由于 pc 是指针变量,因此把其值作为地址,才能取得它所指向的字符。采用了符号扩展传送指令 movsx,既取出 pc 所指向的字节值,又进行符号扩展。随后的指令“test edx,edx”作用是测试寄存器 EDX 是否为 0,它通过自身相与操作来影响状态标志。

上述目标代码采用如图 3.9(b)所示的循环结构。在注释带有//@j 行中的无条件转移指令很重要,在执行完循环体后,跳转上去,判断循环条件是否满足,是否需要继续循环。这里的

循环结束条件是遇到字符串尾(字符串结束标记 0)。

类似于例 3-44,如果采用编译优化选项“使大小最小化”,编译上述程序后,可得到如下所示的目标代码。

```
    ;函数 cf321 的目标代码(采用" 使大小最小化 " 编译优化选项)
    push  ebp
    mov   ebp, esp                      ;建立堆栈框架
    mov   ecx, DWORD PTR [ebp+8]        ;取出 str 存放到 ECX
                                        ;while(*pc)
    cmp   BYTE PTR [ecx], 0             ;判断首字符是否为结束标记
    mov   eax, ecx                      ;pc=str;
    je    SHORT LN1cf321                ;如果遇结束标记,结束循环
LL2cf321:
    inc   eax                           ;pc++;
    cmp   BYTE PTR [eax], 0             ;while(*pc)
    jne   SHORT LL2cf321                ;如果未遇结束标记,继续循环
LN1cf321:                               ;return(pc-str);
    sub   eax, ecx
    pop   ebp                           ;}
    ret
```

从上述目标代码可知,由于采用了“使大小最小化”的编译选项,cf321 的目标代码没有在堆栈中安排局部变量,而直接采用寄存器 EAX 作为局部变量 pc。特别是有两处判断循环条件是否满足,第一处的判断仅进行一次,只是判断首字符是否为字符串结束标记。这样做可以避免使用一条无条件转移指令(见上述无编译优化生成的目标代码中注释带//@j 标记的行)。

3. 先判断后执行示例二

在 C 语言中,for 循环控制语句也是先判断后执行的。

【例 3-46】 分析如下 C 语言编写的函数 cf322 的目标代码。

```
int  cf322(int arr[],int n)
{
    int  i,sum=0;
    for(i=0; i<n; i++)
        sum+=arr[i];
    return  sum/n ;
}
```

函数 cf322 的功能是计算一个整型数组中元素的平均值。数组和元素的个数作为入口参数。安排了局部变量 i 和 sum,其中 i 是循环控制变量,sum 用于统计元素值之和。

假设不要求编译优化,即采用编译优化选项“已禁用”,编译上述程序后,可得到如下所示的目标代码。

```
    ;函数 cf322 的目标代码(不采用编译优化)
    push  ebp
    mov   ebp, esp                      ;建立堆栈框架
    sub   esp, 8                        ;安排局部变量
    mov   DWORD PTR [ebp-8], 0          ;sum=0;
                                        ;for(i=0; i<n; i++)
    mov   DWORD PTR [ebp-4], 0
```

```
    jmp   SHORT LN3cf322              ;//@j1
LN2cf322:                             ;调整循环变量 i
    mov   eax, DWORD PTR [ebp-4]
    add   eax, 1
    mov   DWORD PTR [ebp-4], eax
LN3cf322:                             ;比较 i 与 n
    mov   ecx, DWORD PTR [ebp-4]
    cmp   ecx, DWORD PTR [ebp+12]
    jge   SHORT LN1cf322              ;如果 i 不小于 n,则结束循环
                                      ;sum+=arr[i];
    mov   edx, DWORD PTR [ebp-4]      ;取得变量 i 值
    mov   eax, DWORD PTR [ebp+8]      ;取得参数(数组首元素地址)
    mov   ecx, DWORD PTR [ebp-8]      ;取得变量 sum 值
    add   ecx, DWORD PTR [eax+edx*4]  ;加
    mov   DWORD PTR [ebp-8], ecx      ;保存到变量 sum
    jmp   SHORT LN2cf322              ;//@j2
LN1cf322:                             ;return sum/n ;
    mov   eax, DWORD PTR [ebp-8]
    cdq                               ;符号扩展到 EDX(64 位被除数)
    idiv  DWORD PTR [ebp+12]          ;除,所得商在 EAX 中
                                      ;
    mov   esp, ebp                    ;撤销局部变量
    pop   ebp                         ;恢复 EBP
    ret                               ;返回
```

从上述目标代码可知,它也是先判断后执行,代码的组织结构如图 3.10 所示,与图 3.9(b)有所不同。只安排了一处循环条件的判断,所以先利用无条件转移指令(见注释带//@j1 的行)直接跳转到循环条件判断处。如果需要循环,则执行循环体,之后利用无条件转移指令(见注释带//@j2 的行)跳转,调整循环变量。

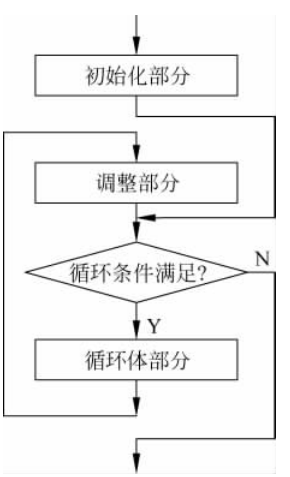

图 3.10 先判断后执行的另一种组织形式

类似于例 3-44 和例 3-45,也可以采用编译优化选项“使大小最小化”,编译上述程序后,可得到如下所示的目标代码。

;函数 cf322 的目标代码(采用" 使大小最小化 " 编译优化选项)

```
    push  ebp
    mov   ebp, esp                    ;建立堆栈框架
    xor   ecx, ecx                    ;i=0;
    xor   eax, eax                    ;sum=0;
                                      ;i < n ?
    cmp   DWORD PTR [ebp+12], ecx     ;比较 n 与 i
```

```
    jle    SHORT LN1cf322                ;如果 n<=i,则结束循环
LL3cf322:
                                         ;sum+=arr[i];
    mov    edx, DWORD PTR [ebp+8]        ;EDX 指向数组首元素
    add    eax, DWORD PTR [edx+ecx*4]    ;EDX+ ECX*4 指向第 i 个元素
                                         ;i++
    inc    ecx
                                         ;i < n ?
    cmp    ecx, DWORD PTR [ebp+12]
    jl     SHORT LL3cf322                ;如果 i < n,则继续循环
LN1cf322:
                                         ;return sum/n ;
    cdq
    idiv   DWORD PTR [ebp+12]
    pop    ebp                           ;恢复 EBP
    ret                                  ;返回
```

从上述目标代码可知,由于采用了“使大小最小化”的编译选项,cf322 的目标代码没有在堆栈中安排局部变量,而直接采用寄存器 ECX 和 EAX 分别作为局部变量 *i* 和 sum。此外,通过安排两处判断循环条件是否满足,避免了两条无条件转移指令。

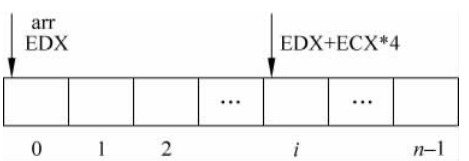

图 3.11　访问数组元素示意图

从上述目标代码还可知,虽然源程序中函数 cf322 的一个形参是数组,但调用过程中实际传递的就是指针,也就是地址。在堆栈的[EBP+8]单元中,存放的是由主调函数在调用前压入堆栈的数组首元素的地址。图 3.11 给出了存取数组元素的示意图,每一个存储单元为 4 个字节,因为整型数占用 4 个字节。

3.4.2　循环指令

利用条件转移指令和无条件转移指令可以实现循环,但为了更加方便地实现循环,IA-32 系列 CPU 还专门提供了循环指令。

1. 循环指令

循环指令类似于条件转移指令,不仅属于段内转移,而且还采用相对转移的方式,即通过在指令指针寄存器 EIP 上加一个地址差的方式实现转移。需要注意,循环指令中只用一个字节(8 位)表示地址差,所以循环转移的范围仅为－128～＋127。

在保护方式(32 位代码段)下,循环指令将自动以寄存器 ECX 作为循环计数器。

循环指令不影响各标志。

(1) 计数循环指令(LOOP)。计数循环指令的格式如下：

LOOP *LABEL*

这条指令使寄存器 ECX 的值减 1，如果结果不等于 0，则转移到标号 LABEL 处，否则顺序执行 LOOP 指令后的指令。

这条指令类似于如下的两条指令：

```
DEC    ECX
JNZ    LABEL
```

通常在利用 LOOP 指令构成循环时，先要设置好计数器 ECX 的初值，即循环次数。由于首先进行寄存器 ECX 减 1 操作，再判断结果是否为 0，因此循环最多可进行 2 的 32 次方次。

【例 3-47】 如下代码片段统计寄存器 EAX 中位是 1 的个数，统计结果存放在寄存器 EDX 中。假设 EAX=000023F6H，那么有 9 个位是 1：

```
    XOR   EDX, EDX            ;清 EDX
    MOV   ECX, 32             ;设置循环计数
LAB1:
    SHR   EAX, 1              ;右移一位(最低位进入进位标志 CF)
    ADC   DL, 0               ;统计(实际是加 CF)
    LOOP  LAB1                ;循环
```

对于循环次数已知的循环，利用循环指令，能够更加简明高效。

(2) 等于/全零循环指令(LOOPE/LOOPZ)。等于/全零循环指令有两个助记符，格式如下：

LOOPE *LABEL*
LOOPZ *LABEL*

这条指令使寄存器 ECX 的值减 1，如果结果不等于 0，并且零标志 ZF 等于 1(表示相等)，那么就转移到标号 LABEL 处，否则顺序执行。注意指令本身实施的寄存器 ECX 减 1 操作并不影响标志。

【例 3-48】 如下代码片段实现在一个字符数组中查找第一个非空格字符，假设字符数组 buff 的长度为 100。

```
    LEA   EDX, buff           ;指向字符数组首
    MOV   ECX, 100            ;
    MOV   AL, 20H             ;空格字符
    DEC   EDX                 ;为了简化循环,先减 1
LAB2:
    INC   EDX                 ;调整到指向当前字符
    CMP   AL, [EDX]           ;比较
    LOOPE LAB2                ;判断和循环计数同时进行
```

(3) 不等于/非零循环指令(LOOPNE/LOOPNZ)。不等于/非零循环指令也有两个助记符，格式如下：

LOOPNE *LABEL*
LOOPNZ *LABEL*

这条指令使寄存器 ECX 的值减 1，如果结果不等于 0，并且零标志 ZF 等于 0(表示不相

等),那么就转移到标号 LABEL 处,否则顺序执行。注意指令本身实施的寄存器 ECX 减 1 操作不影响标志。

【例 3-49】 如下 C 程序 dp323 及其嵌入汇编代码,演示 LOOPNE 指令的使用。利用嵌入汇编代码测量由用户输入的字符串的长度。

```
#include <stdio.h>
int  main()
{
    char string[100];                  //用于存放字符串
    int len;                           //用于存放字符串长度
    printf("Input string:");
    scanf("%s",string);                //由用户输入一个字符串
    //嵌入汇编代码
    _asm{
        LEA   EDI, string              //使得 EDI 指向字符串
        XOR   ECX, ECX                 //假设字符串"无限长"
        XOR   AL, AL                   //使 AL=0(字符串结束标记)
        DEC   EDI                      //为了简化循环,先减 1
    LAB3:
        INC   EDI                      //指向待判断字符
        CMP   AL, [EDI]                //是否为结束标记
        LOOPNE  LAB3                   //如果不是结束标记,继续循环
        ;
        NOT   ECX                      //根据 ECX 推断字符串长度
        MOV   len, ECX
    }
    printf("len=%d\n",len);            //显示输入字符串的长度
    return  0;
}
```

在上述演示程序 dp323 的嵌入式汇编代码中,利用 LOOPNE 指令控制循环,每次循环调整指针和判断是否遇到字符串结束标记。在循环开始前,把循环计数器 ECX 清为 0,相当于假设字符串“无限长”。由于 LOOPNE 指令每次减 1,因此 ECX 就隐含了循环的次数,这样就可以根据 ECX 的值简单推算出字符串的长度。

2. 计数器转移指令

如上所述,循环指令把寄存器 ECX 作为循环计数器,如果循环计数器的初值为 0,意味着要进行 2 的 32 次方的循环。但普通程序中,如果循环次数为 0,往往表示不进行循环。为此,IA-32 系列 CPU 还提供了一条把 ECX 是否为 0 作为判断条件的条件转移指令,称为计数器转移指令。

计数器转移指令的格式如下:

JECXZ LABEL

该指令实现当寄存器 ECX 的值等于 0 时转移到标号 LABEL 处,否则顺序执行。

通常在循环开始之前使用该指令,所以当循环次数为 0 时,就可以跳过循环体。

【例 3-50】 如下 C 程序 dp324 演示通过堆栈传递参数调用子程序和指令 JECXZ 的使用。主程序的功能是计算由用户输入的若干成绩的平均值。它调用子程序完成平均值的计算。子程序的功能及原型与例 3-46 中函数 cf322 相同。

```
#include <stdio.h>
# define  COUNT  5                    //假设成绩项数
int  main()
{
    int  score[COUNT];                //用于存放由用户输入的成绩
    int  i, average;
    for( i=0; i < COUNT; i++) {       //由用户从键盘输入成绩
        printf(" score[%d]=", i) ;
        scanf(" %d", &score[i]) ;
    }
    //调用子程序计算成绩平均值
    _asm{
        LEA   EAX, score
        PUSH  COUNT                   //把数组长度压入堆栈
        PUSH  EAX                     //把数组起始地址压入堆栈
        CALL  AVER                    //调用子程序
        ADD   ESP, 8                  //平衡堆栈
        MOV   average, EAX
    }
    //
    printf(" average=%d\n",average);//显示所得平均值
    return  0;
    //子程序 AVER
    _asm{
    AVER:                             //子程序入口
        PUSH  EBP
        MOV   EBP, ESP                //建立堆栈框架
        MOV   ECX, [EBP+12]           //取得数组长度
        MOV   EDX, [EBP+8]            //取得数组起始地址
        XOR   EAX, EAX                //将 EAX 作为和 sum
        XOR   EBX, EBX                //将 EBX 作为下标 i
        JECXZ OVER                    //如数组长度为 0,不循环累加
    NEXT:
        ADD   EAX, [EDX+ EBX*4]       //累加
        INC   EBX                     //调整下标 i
        LOOP  NEXT                    //减计数方式控制循环
        ;
        CDQ                           //计算平均值
        IDIV  DWORD PTR [EBP+12]
    OVER:
        POP   EBP                     //撤销堆栈框架
```

```
        RET                                    //返回
    }
}
```

从上述代码可知，子程序在建立堆栈框架后，从堆栈中取得数组长度并送到寄存器 ECX，随后采用计数器转移指令 JECXZ 检查数组元素个数是否为 0，如果 ECX 为 0 就不实施循环累加。与函数 cf322 的目标代码比较可知，这里的代码更可靠，同时效率也稍高。

3. 应用示例

【例 3-51】 编写一个代码片段，把 32 位二进制数转换为 10 位十进制数的 ASCII 码串。为了简单化，设二进制数是无符号的。

在 C 语言中，如果希望采用十进制数的形式输出一个无符号整型变量 uintx 的值，可以利用如下的语句：

```
printf("%u\n", uintx);
```

事实上，由 C 语言的库函数 printf 实现了二进制数到十进制数的转换。假设不准使用输出无符号整型的格式符"%u"，只准使用输出字符串的格式符"%s"，那么怎么办？本例的要求是直接用汇编代码来实现转换。设已有如下的 C 程序 dp325 框架，现在需要编写中间部分的汇编代码片段：

```
//演示程序 dp325 框架
#include <stdio.h>
int  main()
{
    unsigned uintx=56789123;          //无符号整型变量
    char buffer[11];                  //用于存放 ASCII 码串的缓冲区
    _asm{
        …                             //实现转换的汇编代码片段
    }
    printf("%s\n", buffer);           //输出字符串
    return  0;
}
```

把一个整数除以 10，所得的余数就是个位数。如果把所得的商再除以 10，所得的余数就是十位数。如果继续把所得的商除以 10，所得的余数就是百位数。以此类推，就可以得到一个整数的各位十进制数了。32 位二进制数能表示的最大十进制数只有 10 位，所以循环地除上 10 次，就可以得到各位十进制数。

为了把一位十进制数转换为对应的 ASCII 码，只要加上数字符'0'的 ASCII 码。

由于先得到个位数，然后得到十位数，再得到百位数，因此在把所得的各位十进制数的 ASCII 码存放到字符串中时，要从字符串的尾部开始。图 3.12 给出了 ASCII 码串的存储示意图，其中每一字节的数据是十六进制表示的 ASCII 码。

对应的代码片段 as326 如下：

```
;汇编代码片段 as326
_asm{
    LEA   ESI, buffer                 ;获取存放字符串的缓冲区首地址
    MOV   EAX, uintx                  ;取得待转换的数据
```

```
    MOV   ECX, 10                    ;循环次数(十进制数的位数)
    MOV   EBX, 10                    ;十进制的基数是 10
NEXT:
    XOR   EDX, EDX                   ;形成 64 位的被除数(无符号数除)
    DIV   EBX                        ;除以 10,EAX 含商,EDX 含余数
    ADD   DL, '0'                    ;把十进制位转成对应的 ASCII 码
    MOV   [ESI+ ECX-1], DL           ;保存到缓冲区
    LOOP  NEXT                       ;计数循环
    ;
    MOV   BYTE PTR [ESI+10],0        ;设置字符串结束标志
}
```

在上述代码片段中,由于总是循环 10 次,因此转换所得的 ASCII 字符串可能在前端有若干个字符'0'。

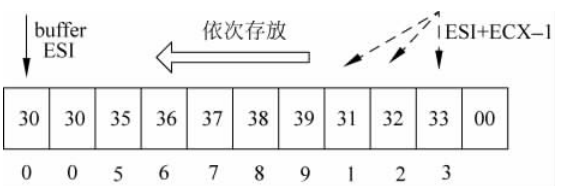

图 3.12 ASCII 码串的存储示意图

【例 3-52】 深化例 3-51。①设二进制数是有符号的,如果是负数,则所得字符串的第一个字符应该是负号;②不需要前端可能出现的'0'。

设变量 intx 和存放字符串的缓冲区定义如下:

```
    int   intx=- 57312;
    char  buffer[16];              //足够长
```

满足所需要求的代码片段 as327 如下所示:

```
;汇编代码片段 as327
_asm{
    LEA   ESI, buffer              ;置指针初值
    MOV   EAX, intx                ;取得待转换的数据
    CMP   EAX, 0                   ;判断待转换数据是否为负数
    JGE   LAB1                     ;非负数,跳转
    MOV   BYTE PTR [ESI], '- '     ;先保存一个负号
    INC   ESI                      ;调整指针
    NEG   EAX                      ;取相反数,得正数
LAB1:
    MOV   ECX, 10                  ;最多循环 10 次
    MOV   EBX, 10                  ;每次除以 10
    MOV   EDI, 0                   ;置有效位数的计数器初值
NEXT1:
    XOR   EDX, EDX
    DIV   EBX                      ;获得一位十进制数
    ;
    PUSH  EDX                      ;把所得一位十进制数压入堆栈//@1
```

```
    INC     EDI                     ;有效位数增加 1
    ;
    OR      EAX, EAX                ;测试结果(商)
    LOOPNE NEXT1                    ;如结果不为 0,考虑继续循环//@2
    ;----------------------------------------
    MOV     ECX, EDI                ;置下一个循环的计数
NEXT2:
    POP     EDX                     ;从堆栈弹出余数//@3
    ADD     DL, '0'                 ;转成对应的 ASCII 码
    MOV     [ESI], DL               ;依次存放到缓冲区
    INC     ESI
    LOOP    NEXT2                   ;循环处理下一位
    ;
    MOV     BYTE PTR [ESI], 0       ;设置字符串结束标志
}
```

从上述程序片段可知,在开始转换之前,先判断待转换的数据是否为负,如果是负数,先使得字符串的第一个字符为负号,同时取相反值,这样待转换数据就变成正数了。

代码片段含有两个循环。第一个循环通过循环除以 10 的方法,取得十进制数的各位数值,但是循环结束的方式有所改变(见带//@2 的行),不仅最多循环 10 次,而且只有结果(表示需要转换的数据)不为 0 时,才继续循环。因为如果需要继续转换的数据为 0,也就意味着产生前端没有意义的 0。例 3-44 的函数 cf320 就是这个思路。为了改变存放顺序,利用了堆栈的先进后出特点,把所得的各位十进制数依次压入堆栈(见带//@1 的行),同时利用寄存器 EDI 统计压入堆栈的次数,为弹出操作做好准备。图 3.13 给出了压入堆栈操作后的堆栈局部示意图,其中堆栈的每一项是 32 位(4 个字节)。

第二个循环比较简单,依次进行弹出操作,每弹出一位十进制数,在转换成对应的 ASCII 后,就依次存入字符串。第二个循环的循环次数是压入堆栈操作的次数,也就是十进制数的有效位数。其实,可以根据第一个循环结束时寄存器 ECX 的值,来得到压入堆栈操作的次数,这样就不需要专门用寄存器 EDI 来统计了。

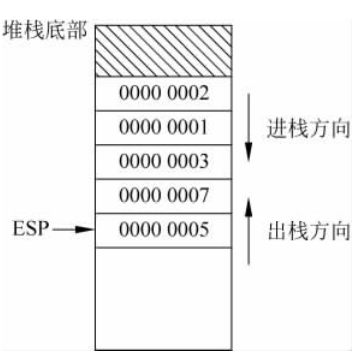

图 3.13　把各位十进制数压入堆栈后的示意图

3.4.3　多重循环设计举例

多重循环就是循环之中还有循环。

【例 3-53】 如下 C 函数 cf328 的功能是,在某个无符号整型数组中查找第一个特征数据,

如果找到返回下标值(索引号),否则返回-1。这里的特征是指在用二进制表示该数据时,其中1的个数超过20。

```
int  cf328(unsigned arr[], int n)
{
    int  i;                    //循环变量
    unsigned  value;           //用于判断特征值
    int  count=0;              //统计数据中1的个数
    for(i=0; i<n; i++)         //循环遍历数组中的每个数据
    {
        value=arr[i];
        if(value<=0xfffff)     //0xfffff是20位1的最小值
            continue;          //如不超过0xfffff,肯定不是特征数据
        count=0;
        while(value!=0)        //统计1的个数
        {
            if(value & 1==1)   //测最低位是否为1
                count++;
            value=value>>1;    //向右移一位
        }
        if(count>20)           //找到第一个,跳出循环
            break;
    }
    if(count<=20)              //如没有找到特征数据,返回-1
        i=-1;
    return  i;
}
```

采用编译优化选项"使大小最小化",编译上述程序后,可得到如下所示的目标代码(标号稍有修饰),分号之后是添加的注释。

```
    ;函数cf328的目标代码(使大小最小化)
    push  ebp
    mov   ebp, esp                      ;建立堆栈框架
                                        ;for语句
    xor   eax, eax                      ;eax作为变量i
    xor   edx, edx                      ;edx作为变量count
    cmp   DWORD PTR [ebp+12], eax       ;比较n与i
    jle   SHORT LN17cf328               ;当n<=i时,跳过外循环
LL9cf328:                               ;外循环体开始
                                        ;value=arr[i];
    mov   ecx, DWORD PTR [ebp+8]        ;取得作为参数的数组起始地址
    mov   ecx, DWORD PTR [ecx+ eax* 4]  ;取第i项数据
                                        ;if(value<=0xfffff) continue;
    cmp   ecx, 1048575
    jbe   SHORT LN8cf328
                                        ;
    xor   edx, edx                      ;count=0;
```

```
                                        ;while 语句
    test  ecx, ecx                      ;判断 value 是否为 0
    je    SHORT LN8cf328                ;value 为 0,跳过内循环
LL5cf328:                               ;内循环体开始
    test  cl, 1                         ;value & 1==1 ?
    je    SHORT LN3cf328
    inc   edx                           ;count++;
LN3cf328:
    shr   ecx, 1                        ;value=value >> 1;
                                        ;内循环体结束
    jne   SHORT LL5cf328                ;如果 value 不为 0,继续内循环
                                        ;if( count > 20) break;
    cmp   edx, 20
    jg    SHORT LN1cf328
LN8cf328:                               ;外循环体结束
    inc   eax                           ;i++
    cmp   eax, DWORD PTR [ebp+12]       ;i < n ?
    jl    SHORT LL9cf328                ;如果 i < n,继续外循环
                                        ;
    cmp   edx, 20                       ;判断是否找到特征数据
    jg    SHORT LN1cf328
LN17cf328:
    or    eax, -1                       ;i=-1;
LN1cf328:
    pop   ebp                           ;撤销堆栈框架
    ret
```

由于只要求目标代码的尺寸最小化,因此从上述目标代码中,可以清楚地看到外循环 for 和内循环 while 的实现。

从上述目标代码中,还可以清楚地看到 break 语句和 continue 语句的具体实现。它们的本质就是转移。在这里它们从属于条件语句,所以体现为条件转移。

另外,请比较例 3-47 统计二进制数 1 的实现方法。

【例 3-54】 设 buffer 缓冲区中有 10 个整数,编写一个代码片段,将它们由小到大排序。

有各种各样的排序算法,这里为了方便地说明二重循环,采用选择排序,图 3.14 给出了排序算法流程图,其中 N 表示待排序的数据个数;I 和 J 分别表示从 0 开始的下标。

设 buffer 缓冲区定义如下:

```
int  buffer[10]={ 2222, 1, 3500, -300, 67, 100, 76, 8, 29, -17};
```

实现选择排序的代码片段 as329 如下。其中,ESI 相当于外层循环控制变量 I,EDI 相当于内层循环控制变量 J,寄存器 EBX 含有数据缓冲区开始地址。

```
# define  LEN  10
;汇编代码片段 as329
_asm{
    LEA   EBX, buffer                   ;设置缓冲区开始地址
```

```
    MOV   ESI, 0                    ;I=0
FORI:
    MOV   EDI, ESI
    INC   EDI                       ;J=I+1
FORJ:
    MOV   EAX, [EBX+4*ESI]
    CMP   EAX, [EBX+4*EDI]          ;A[I]与 A[J]比较
    JLE   NEXTJ                     ;A[I]小于等于 A[J]跳转
    XCHG  EAX, [EBX+4*EDI]          ;A[I]与 A[J]交换
    MOV   [EBX+4*ESI], EAX
NEXTJ:
    INC   EDI                       ;J=J+1
    CMP   EDI, LEN
    JB    FORJ                      ;J<N 时跳转
NEXTI:
    INC   ESI                       ;I=I+1
    CMP   ESI, LEN-1
    JB    FORI                      ;I<N-1 时跳转
}
```

可以把上述代码片段 as329 嵌入到某个 C 程序中,实现排序的功能。

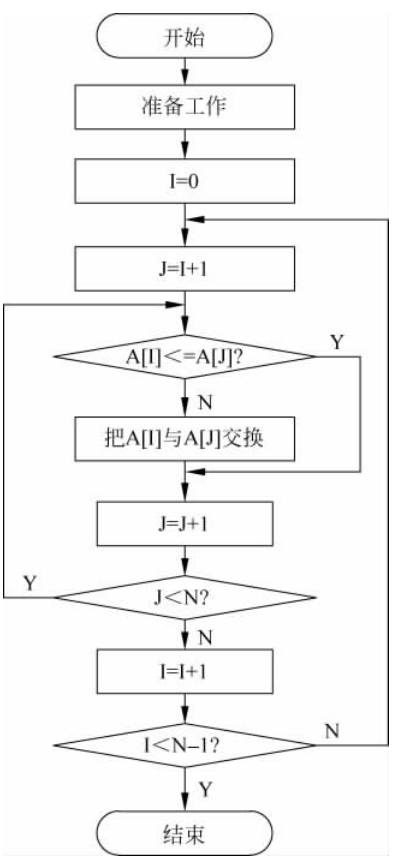

图 3.14　排序算法流程图

3.5　子程序设计

如果某个程序片段将反复在程序中出现，就把它设计成子程序。这样能有效地缩短程序长度、节约存储空间。如果某个程序片段具有通用性，可供许多程序共享，也把它设计成子程序。这样能大大减轻程序设计的工作量，如库函数程序。此外，当某个程序片段的功能相对独立时，也可把它设计成子程序，这样便于模块化，也便于程序的阅读、调试和修改。

在汇编语言中，子程序常常以过程(Procedure)的形式出现。在 3.1 节介绍过程调用和返回指令的基础上，本节首先举例说明设计子程序的方法和规范，然后进一步介绍调用过程的方法。

3.5.1　子程序设计要点

子程序是供主程序调用的，所以在设计子程序时，必须遵循与主程序的约定。除了高效地完成子程序相应的功能外，在设计子程序时，需要注意，传递参数的方法；安排局部变量的方法；保护寄存器的约定；描述子程序的说明。

1. 传递参数的方法

在前面的章节中已经介绍了一些子程序示例，从中可以看到传递参数的方法。从 C 函数的目标代码中可知，如果源程序中默认调用约定，那么利用堆栈传递入口参数(函数参数)，利用寄存器传递出口参数(函数返回值)。为了提高效率，可以在 C 源程序中明确_fastcall 的调用约定，这表示希望利用寄存器传递入口参数，在采用编译优化的情形下，所得目标代码利用寄存器传递入口参数。

在采用汇编语言编写子程序时，采用哪种方法传递出入口参数，取决于约定。在例 3-3 的演示程序 dp32 中，子程序 TUPPER 和 UPPER 就是利用寄存器传递出入口参数。在例 3-50 的演示程序 dp324 中，子程序 AVER 就是利用堆栈传递入口参数，利用寄存器 EAX 传递出口参数。

利用寄存器传递参数，虽然简单并且效率比较高，但由于寄存器较少，可传递参数比较少。利用堆栈传递参数比较复杂并且效率较低，但可以传递足够多的参数。

【例 3-55】 分析如下所列 C 函数 cf330 的目标代码。该函数的功能是把一个无符号整数(32 位二进制数)转换为 8 位十六进制数的 ASCII 码串。函数有两个入口参数，分别是整数 m 和存放 ASCII 码串的首地址，希望采用寄存器传递入口参数。

```
void  _fastcall  cf330(unsigned m, char * buffer)
{
    int  i;
    char  val;
    for(i=1; i<=8; i++)                    //循环 8 次
    {
        val=(m>>(32-i*4)) & 0x0f;          //先向右移,再截取 4 位
        val+='0';                          //转成对应 ASCII 码
        if(val>'9') val+=7;
        * buffer++=val;                    //依次保存
    }
    * buffer='\0';                         //ASCII 码串结束标记
```

```
    return ;
}
```

由于4个二进制位对应一个十六进制位,因此直接采用移位的方式来得到各个十六进制位。无符号整数是32位二进制数,对应8个十六进制位,因此循环8次。

采用编译优化选项"使大小最小化",编译上述程序后,可得到如下所示的目标代码,分号之后是添加的注释。

```
    ;cf330的目标代码(大小最小化)
    push  esi                          ;保护寄存器esi
    xor   esi, esi                     ;esi作为局部变量i //@o
    push  edi                          ;保护寄存器edi
    mov   edi, ecx                     ;edi=m
    inc   esi                          ;i=1
LL4cf330:
                                       ;val=(m>>(32-i*4)) & 0x0f;
    push  8
    pop   ecx                          ;ecx=8 //@o
    sub   ecx, esi
    shl   ecx, 2                       ;ecx=(8-i)*4
    mov   eax, edi                     ;eax=m
    shr   eax, cl                      ;eax=m>>(32-i*4)
    and   al, 15                       ;& 0x0f
    add   al, 48                       ;val+='0';
    cmp   al, 57                       ;if(val>'9') val+=7;
    jle   SHORT LN1cf330
    add   al, 7
LN1cf330:
    mov   BYTE PTR [edx], al           ;* buffer++=val;
    inc   edx
                                       ;
    inc   esi                          ;i++
    cmp   esi, 8                       ;i<=8吗?
    jle   SHORT LL4cf330               ;是,则跳转
    pop   edi                          ;恢复edi
    mov   BYTE PTR [edx], 0            ;* buffer='\0';
    pop   esi                          ;恢复esi
    ret
```

由于源程序中采用了调用约定_fastcall,而且又选择了编译优化,因此通过寄存器向函数传递入口参数。从上述目标代码可知,利用寄存器ecx传递参数*m*,利用寄存器edx传递参数buffer。

由于选择了编译优化,从上述目标代码还可知,给动态局部变量*i*分配了寄存器esi,给动态局部变量val分配了寄存器eax(或者al)。这样能够有效地提高执行效率。同时,还可以看到,由于使用了寄存器esi和edi,因此在一开始就把它们压入堆栈保护,在子程序返回之前从堆栈中将其恢复。

在上述目标代码中,还有一些优化措施,见注释中带//@o的行,将在第5章中介绍。

2. 安排局部变量的方法

为了实现自身的功能，子程序（函数）往往还需要定义一些局部变量，以便清楚地表达处理逻辑，临时保存中间结果。局部就是限于子程序（函数）或者限于代码片段（复合语句）。

例 3-55 中整型变量 *i* 和字符变量 val，都是动态局部变量。由于选择了编译优化，把这两个变量安排在寄存器中。虽然寄存器作为局部变量可以提高效率，但寄存器数量较少，所以一般不把局部变量安排在寄存器中。在 3.1 节中介绍了利用堆栈来安排局部变量，这个方法虽然较复杂，但可以安排足够多的局部变量。实际上，在 C 函数的目标代码绝大部分的局部变量被安排在堆栈中。在不要求编译优化时，通常局部变量都被安排在堆栈中。在前面多个没有经过编译优化的目标代码中，可以清楚地看到这一点。

下面是一个把局部变量安排在寄存器中的示例。

【例 3-56】 分析如下 C 函数 cf331 的目标代码。该函数的功能是统计整型数组中值为正数、负数和 0 的元素个数。函数有 4 个入口参数，其中，两个参数指定数组和数组长度，另两个参数是指针，指出存放正数和负数个数的所在。函数返回值是指定数组中值为 0 的元素个数。

```
int  cf331(int arr[], int n, int *pp, int *pn)
{
    int  i, pcount, ncount, zcount;
    pcount=ncount=zcount=0;
    for(i=0; i<n; i++)                    //循环,依次检查并统计
    {
        if(arr[i]>0)
            pcount++;
        else if(arr[i]<0)
            ncount++;
        else
            zcount++;
    }
    *pp=pcount;                           //送出正数的个数
    *pn=ncount;                           //送出负数的个数
    return  zcount;                       //返回 0 的个数
}
```

采用编译优化选项“使速度最达化”，编译上述程序后，可得到如下所示的目标代码，分号之后是添加的注释。

```
;cf331 的目标代码(使速度最大化)
;利用堆栈传递入口参数,利用寄存器 eax 传递出口参数
push  ebp
mov   ebp, esp                        ;建立堆栈框架
push  ebx                             ;保护 ebx
mov   ebx, DWORD PTR [ebp+12]         ;ebx=n
push  esi                             ;保护 esi、edi
push  edi
xor   eax, eax                        ;zcount=0
xor   esi, esi                        ;ncount=0
xor   edi, edi                        ;pcount=0
xor   edx, edx                        ;i=0
```

```
    test  ebx, ebx
    jle   SHORT LN5cf331
LL7cf331:
    mov   ecx, DWORD PTR [ebp+8]          ;ecx=arr
    mov   ecx, DWORD PTR [ecx+ edx* 4]    ;ecx=arr[i]
    test  ecx, ecx                        ;测 arr[i]的正负
    jle   SHORT LN12cf331                 ;arr[i] <=0,则跳转
    inc   edi                             ;pcount++;
    jmp   SHORT LN6cf331
LN12cf331:
    jns   SHORT LN2cf331
    inc   esi                             ;ncount++;
    jmp   SHORT LN6cf331
LN2cf331:
    inc   eax                             ;zcount++;
LN6cf331:
    inc   edx                             ;i++
    cmp   edx, ebx                        ;i < n 吗?
    jl    SHORT LL7cf331                  ;i < n,继续循环
LN5cf331:
    mov   edx, DWORD PTR [ebp+16]
    mov   ecx, DWORD PTR [ebp+20]
    mov   DWORD PTR [edx], edi            ;*pp=pcount;
    pop   edi
    mov   DWORD PTR [ecx], esi            ;*pn=ncount;
    pop   esi
    pop   ebx                             ;恢复被保护的寄存器
    pop   ebp                             ;撤销堆栈框架
    ret
```

从上述目标代码可知,由于要求了编译优化,因此 pcount 等 4 个局部变量都被安排在寄存器中。在开始部分,先利用堆栈保护了寄存器 ebx、esi 和 edi,在返回之前,从堆栈中弹出以恢复它们原先的值。

3. 保护寄存器的约定

子程序为了完成其功能,通常要临时利用一些寄存器存放内容。例如,作为局部变量使用。也就是说,在子程序运行时有时会破坏一些寄存器的原有内容。所以,如果不采取措施,那么在调用子程序后,主程序就无法再使用这些寄存器的原有内容了,这常常会导致主程序的错误。为此,要对有关寄存器的内容进行保护与恢复。

保护寄存器内容的简单方法是,在子程序一开始就把在子程序中会改变的寄存器内容压入堆栈,而在返回之前再恢复这些寄存器的内容。

由于子程序往往会用到多个寄存器,把这些用到的寄存器全部压入堆栈加以保护,过后再恢复,这样做会降低代码效率。实际上,这是主程序和子程序之间缺少“默契”。从前面多个 C 函数的目标代码可知,一方面,函数代码虽然使用寄存器 eax、ecx 和 edx,但并不事先保护它们;另一方面,函数代码只要使用了寄存器 ebx、esi、edi 和 ebp,总是先保护它们,过后再恢复。这体现了 C 语言中主程序与子程序的“默契”。还可以看到,函数为了通过 ebp 访问堆栈中的

可能存在的参数和局部变量，所以在建立堆栈框架时，先把 ebp 压入堆栈，然后把 esp 送到 ebp，最后通过弹出 ebp，恢复 ebp 原先内容，来撤销堆栈框架。当然，函数代码对堆栈指针寄存器 esp 的使用是极其谨慎的。

因此，保护寄存器的常用方法是子程序只保护主程序关心的那些寄存器。所谓关心的寄存器，是根据主程序与子程序的约定来确定的。这样处理，既达到了保护寄存器的目的，又减少效率损耗。例 3-55 和例 3-56 都采用这种常用的保护方法。当然，如果约定所有寄存器都是“关心”的，那么就退化为只要破坏，就加以保护。

值得指出，在利用堆栈进行寄存器的保护和恢复时，一定要注意堆栈的先进后出特性，一定要注意堆栈平衡。

至此可以看到，在堆栈中既要存放传递给子程序的入口参数，又要存放子程序的返回地址，还有可能利用堆栈来保护寄存器的内容，所以堆栈中的内容较为混杂。但是，绝对不能出现差错，绝对不能失去平衡，除非为了特殊目的而故意为之。在子程序开始之初，建立堆栈框架，在子程序返回之前，撤销堆栈框架，这是一个比较好的方法。

4. 描述子程序的说明

为了能正确地使用子程序，在给出子程序代码时还要给出子程序的说明信息。子程序说明信息一般由以下几部分组成，每一部分的表述应该简明确切。

(1) 子程序名(或者入口标号)。

(2) 子程序功能描述。

(3) 子程序的入口参数和出口参数。

(4) 所影响的寄存器等情况。

(5) 使用的算法和重要的性能指标。

(6) 其他调用注意事项和说明信息。

(7) 调用实例。

子程序的说明信息至少应该包含上述前三部分的内容。

3.5.2　子程序设计举例

下面采用嵌入汇编代码方式，举例说明子程序的设计。

【例 3-57】 编写把二进制数(32 位)转换为十六进制数(8 位)ASCII 码串的子程序。这个子程序的功能，与例 3-55 函数 cf330 的功能相同。

实现上述功能的子程序 as332 如下，以注释方式给出了子程序的相关说明信息。

```
//子程序名(入口标号) : BTOHS
//功      能: 把 32 位二进制数转换为 8 位十六进制数的 ASCII 码串
//入口参数:(1) 存放 ASCII 码串缓冲区的首地址(先压入堆栈)
//         (2) 二进制数据(后压入堆栈)
//出口参数: 无
//其他说明:(1) 缓冲区应该足够大(至少 9 个字节)
//         (2) ASCII 串以字节 0 为结束标记
//         (3) 影响寄存器 EAX、ECX、EDX 的值
__asm{
BTOHS:                                  ;子程序入口标号
    PUSH  EBP
    MOV   EBP, ESP                      ;建立堆栈框架
```

```
    PUSH  EDI                          ;保护寄存器 EDI
    MOV   EDI, [EBP+12]                ;取得存放 ASCII 码串的首地址
    MOV   EDX, [EBP+8]                 ;取得待转换数据
    MOV   ECX, 8                       ;设置循环次数
NEXT:
    ROL   EDX, 4                       ;循环左移 4 位,高 4 位移到低 4 位
    MOV   AL, DL                       ;待转换数据送到 AL
    AND   AL, 0FH                      ;把 AL 中低 4 位转换成对应的 ASCII 码
    ADD   AL, '0'
    CMP   AL, '9'
    JBE   LAB580
    ADD   AL, 7
LAB580:
    MOV   [EDI], AL                    ;把所得字符保存到缓冲区
    INC   EDI                          ;调整缓冲区指针
    LOOP  NEXT                         ;循环,转换下一个 4 位
    MOV   BYTE PTR [EDI], 0            ;置字符串结束标志
    POP   EDI                          ;恢复 EDI
    POP   EBP
    RET
}
```

与例 3-55 函数 cf330 的目标代码相比,上述子程序不仅利用指令 LOOP 进行循环控制,而且利用移位指令 ROL 进行循环移位,从而使得代码更简洁。由于使用了主程序可能"关心"的寄存器 EDI,因此一开始就压入堆栈保护。此外,还利用堆栈传递入口参数。

【例 3-58】 编写一个子程序,把由十进制数字符构成的字符串转换成对应的数值。

本例子程序实现例 3-51 代码片段相反的功能。

设十进制数字串中各位对应的值为 d_n、d_{n-1}、…、d_2、d_1,那么它所表示的二进制数可由下式计算得出:

$$Y=((((0\times10+d_n)\times10+d_{n-1})\times10+\cdots)\times10+d_2)\times10+d_1$$

可通过迭代的方法进行上式的计算,迭代公式如下,Y 的初值为 0:

$$Y=Y\times10+d_i(i=n,n-1,\cdots,1)$$

所以,当十进制数字串中数字符的个数为 n 时,那么只需进行 n 次迭代计算。

实现功能要求的子程序 as333 如下,以注释的方式给出了子程序的相关说明信息。

```
//子程序名(入口标号) : DSTOBV
//功     能: 把十进制数字串转换成对应的二进制数值
//入口参数: ESI=待转换数字串的起始地址偏移
//          ECX=待转换数字串的长度(十进制数字的位数)
//出口参数: EAX=转换所得数值
//说     明: (1) 不考虑数字串过长的情形
//          (2) 寄存器 EBX、ECX、EDX、ESI 受到影响
_asm{
DSTOBV:
    XOR   EDX, EDX                     ;EDX 作为 Y
    XOR   EAX, EAX
    JECXZ LAB2                         ;排除数字串为空的情形
```

```
LAB1:
    IMUL  EDX, 10                    ;Y*10
    MOV   AL, [ESI]                  ;取一位字符
    INC   ESI
    AND   AL, 0FH                    ;得到某一位十进制数值 di
    ADD   EDX, EAX                   ;Y=Y*10+di
    LOOP  LAB1                       ;迭代计算
LAB2:
    MOV   EAX, EDX                   ;准备返回值
    RET
}
```

上述子程序虽然考虑了数字串是空(ECX 为 0)的情形,但没有考虑数字串过长的情形。在实际应用中,不仅数字串的长度往往不能确定,而且可能还会夹杂非数字符号。

【例 3-59】 改写例 3-58 的子程序,使其更实用。

改写后的子程序 as334 如下所示,并以注释的方式给出了子程序的相关信息。

```
//子程序名(入口标号) : DSTOB
//功     能: 把十进制数字串转换成对应的二进制数值,遇到非数字符结束
//入口参数: ESI=待转换数字串的起始地址偏移
//出口参数: EAX=转换所得数值(空串时,返回值是 0)
//说     明: (1) 数字串以空(0)为结束标志,或者非数字符为结束标志
//          (2) 如果数字串太长,导致数值超过 32 位,高位被截掉
//          (3) 寄存器 EDX 受影响
//          (4) 调用子程序 ISDIGIT(判断是否是数字符)
_asm{
DSTOB:
    PUSH  ESI                        ;保护寄存器 ESI
    XOR   EDX, EDX                   ;EDX 作为 Y
    XOR   EAX, EAX
LAB1:
    MOV   AL, [ESI]                  ;取一个字符
    INC   ESI
    CALL  ISDIGIT                    ;判断字符是否有效
    OR    AL, AL
    JZ    LAB2                       ;无效,跳转返回
    IMUL  EDX, 10                    ;Y*10
    AND   AL, 0FH                    ;得到某一位十进制数值 di
    ADD   EDX, EAX                   ;Y=Y*10+di
    JMP   LAB1                       ;迭代计算
LAB2:
    MOV   EAX, EDX                   ;准备返回值
    POP   ESI                        ;恢复 ESI
    RET
}
```

上述子程序 as334(入口标号 DSTOB)调用了另一个子程序 as335(入口标号 ISDIGIT)来判断某个字符是否是数字符,子程序 as335 的代码如下所示:

```
//子程序名(入口标号) : ISDIGIT
//功能: 判断字符是否为十进制数字符
//入口参数: AL=字符
//出口参数: 如果是非数字符,AL=0; 否则 AL 保持不变
_asm{
ISDIGIT:
    CMP   AL,'0'                    ;与字符'0'比较
    JL    ISDIG1                    ;有效字符是'0'- '9'
    CMP   AL,'9'
    JA    ISDIG1
    RET
ISDIG1:                             ;非数字符
    XOR   AL,AL                     ;AL=0
    RET
}
```

下面通过另一个示例演示调用上述子程序。

【例 3-60】 演示调用上述子程序 as334(入口标号 DSTOB)和子程序 as335(入口标号 ISDIGIT)。

演示调用上述子程序的主程序 dp336 的框架如下所示:

```
#include <stdio.h>
int  main()
{
    char  buff1[16]="328";
    char  buff2[16]="1234024";
    unsigned  x1, x2;
    unsigned  sum;
    ;
    _asm{
        LEA    ESI, buff1           ;转换一个字符串
        CALL   DSTOB
        MOV    x1, EAX
        ;
        LEA    ESI, buff2           ;转换另一个字符串
        CALL   DSTOB
        MOV    x2, EAX
        ;
        MOV    EDX, x1              ;求和
        ADD    EDX, x2
        MOV    sum, EDX
        ;                           ;如这些代码位于前面
        JMP    OK                   ;通过该指令来跳过随后的子程序部分!//@1
    }
    //
    //在这里安排子程序 DSTOB 和 ISDIGIT 的代码
    //
OK:
```

```
    printf("%d\n", sum);
    return  0;                          ;//@2
}
```

需要特别指出，上述的程序组织方式在//@1 行的无条件转移指令“JMP　OK”很重要，利用该指令跳过随后的子程序的代码。如果把子程序 as334 和 as335 的代码安排在最后(//@2 之后)，在利用较低版本的编译器编译时，不能选择“使速度最大化”编译优化选项。

【例 3-61】 采用汇编语言编写一个子程序 as337，实现 C 语言库函数 strstr()的功能。库函数 strstr()的原型如下：

```
char  *strstr(char *str1, char *str2);
```

其功能是，在字符串 str1 中查找字符串 str2 中第一次出现的位置。如果在字符串 str1 中找到字符串 str2，返回该位置的指针；如果没有找到，返回空指针。

判断一个字符串是否是另一字符串的子串的方法很多，现选取实现较简单的一种算法。子程序 as337 如下所示：

```
//子程序名(入口标号) : STRSTR
//功能: 在字符串 str1 中查找第一次出现的字符串 str2
//说明: 符合 C 函数的调用约定
_asm{
STRSTR:
    PUSH  EBP
    MOV   EBP, ESP                  ;建立堆栈框架
    ;
    PUSH  EBX                       ;保护相关寄存器
    PUSH  ESI
    PUSH  EDI
                                    ;测字符串 2 的长度
    MOV   EDI, [EBP+12]
    DEC   EDI
NEXT:
    INC   EDI
    CMP   BYTE PTR [EDI],0          ;从串 2 取一个字符
    JNZ   NEXT                      ;如果串 2 没有结束,继续
                                    ;
    MOV   ECX, EDI                  ;ECX 指向串 2 结束标记处
    MOV   EAX, [EBP+12]
    SUB   ECX, EAX                  ;地址差是串 2 长度
                                    ;
    JECXZ OVER1                     ;如果串 2 为空,不需要搜索
                                    ;在串 1 中,搜索串 2
    MOV   EDX, ECX                  ;保存串 2 长度
    MOV   EBX, [EBP+8]              ;取串 1 首地址
FORI:
    MOV   ESI, EBX                  ;ESI=开始搜索串 1 的起始地址
    MOV   EDI, [EBP+12]             ;EDI=串 2 首地址
    MOV   ECX, EDX                  ;ECX=串 2 长度
FORJ:
```

```
    MOV   AL, [ESI]
    CMP   AL, [EDI]                ;比较一个字符
    JNZ   NEXTI                    ;不等,从串 1 下一个字符重新搜索
NEXTJ:
    INC   EDI
    INC   ESI
    LOOP  FORJ                     ;继续比较下一字符
    JMP   OVER2                    ;在串 1 中,搜索到串 2
NEXTI:
    INC   EBX                      ;从串 1 的下一个字符开始
    OR    AL, AL                   ;判断串 1 是否结束
    JNZ   FORI                     ;没有结束,重新开始搜索
                                   ;至此,串 1 已经结束
OVER1:
    XOR   EBX, EBX
OVER2:
    MOV   EAX, EBX                 ;准备出口参数
    POP   EDI                      ;恢复寄存器
    POP   ESI
    POP   EBX
    POP   EBP                      ;撤销堆栈框架
    RET
}
```

在上述程序片段中,首先是采用一个循环测量字符串 2 的长度。测量方法是结束地址减去起始地址,这与例 3-45 的方法相同。然后,采用一个二重循环,判断字符串 2 是否为字符串 1 的子串,外层循环控制依次遍历字符串 1 中的字符,内层循环控制在字符串 1 的某个位置开始,依次与字符串 2 中的字符比较。

3.5.3 子程序调用方法

在 3.1 节简单介绍了过程调用和返回指令,为了更好地应用这些指令调用子程序,本节作进一步的介绍。

1. 过程调用指令

与无条件转移指令 JMP 相比,过程调用指令 CALL 会把返回地址压入堆栈,其他方面都是相似的。过程调用指令实施转移,在转移的范围和转移的方式上,与无条件转移指令是一样的。在机器码的格式上,过程调用指令与无条件转移指令也是一样的。

在 3.3.2 节介绍过段内转移和段间转移的概念,也介绍过直接转移和间接转移的概念。像无条件转移指令一样,过程调用指令有段内调用和段间调用之分,有时也称为近调用和远调用。按照给出过程入口地址的方式来分,过程调用指令分为直接调用和间接调用。这样,过程调用指令可分为 4 种:段内直接调用、段内间接调用、段间直接调用和段间间接调用。

虽然有 4 种过程调用指令,但在汇编语言中,均用指令助记符 CALL 表示。

过程调用指令不影响状态标志。

(1) 段内直接调用指令。在 3.1 节介绍的过程调用指令,就是段内直接调用指令。在前面的示例中,已经多次用到。这里可以简单地认为,除了保存返回地址外,段内直接调用指令

CALL 与无条件段内直接转移指令 JMP 是一样的。

(2) 段内间接调用指令。段内间接调用指令的使用格式如下：

CALL ***OPRD***

该指令调用由操作数 OPRD 给出入口地址偏移的子程序。执行如下具体操作：

① 把返回地址偏移压入堆栈保存。

② 把 OPRD 的内容(目标地址偏移)送到 EIP,从而实现转移。在保护方式下(32 位代码段),OPRD 是 32 位通用寄存器或者双字存储单元。

同样可以简单地认为,除了保存返回地址外,段内间接调用指令 CALL 与无条件段内间接转移指令 JMP 是一样的。

【例 3-62】 如下指令演示了段内间接调用指令的使用。

```
CALL  ECX                ;子程序入口地址是寄存器 ECX 的内容
CALL  DWORD PTR [EDX]    ;入口地址是由 EDX 给定的双字存储单元内容
```

(3) 段间调用指令。段间直接调用指令的使用格式与上述的段内直接调用指令相似,段间间接调用指令的使用格式和上述的段内间接调用指令相类似。同样地,这些指令分别与段间直接转移指令和段间间接转移指令也相似。在执行段间调用指令时,首先要把返回地址压入堆栈,然后再转到子程序入口地址处。但由于段间调用要改变代码段寄存器 CS,因此在把返回地址压入堆栈时,先要把 CS 压入堆栈,再把 EIP 压入堆栈。由于涉及改变代码段寄存器 CS 的内容,因此较为复杂,将在第 6 章和第 9 章作进一步介绍。

2. 间接调用子程序示例

下面通过示例来说明间接调用子程序的方法。

【例 3-63】 如下 C 语言程序 dp338 演示段内间接调用指令的使用。

```
#include <stdio.h>
int  subr_addr;                    //用于存放子程序入口地址
int  valu;                         //用于保存结果
int  main()
{
    _asm{
        LEA   EDX, SUBR2           //取得子程序 2 的入口地址
        MOV   subr_addr, EDX       //保存到存储单元
        ;
        LEA   EDX, SUBR1           //取得子程序 1 的入口地址
        XOR   EAX, EAX             //入口参数 EAX=0
        CALL  EDX                  //调用子程序 1(段内间接调用)
        ;
        CALL  subr_addr            //调用子程序 2(段内间接调用)
        ;
        MOV   valu, EAX
    }
    printf("valu=%d\n",valu);      //显示为 valu=28
    return  0;
    //
    _asm{                          //嵌入汇编代码
    SUBR1:                         //示例子程序 1
```

```
        ADD   EAX, 8
        RET                          //返回
        ;
    SUBR2:                           //示例子程序 2
        ADD   EAX, 20
        RET                          //返回
    }
}
```

上述演示程序 dp338 的嵌入汇编代码部分,先后演示了通过寄存器和存储单元间接调用子程序。作为演示,所以两个子程序都很简单,通过寄存器传递出入口参数。注意,包含两个子程序的嵌入汇编代码,被安排 return 语句之后。

【例 3-64】 如下 C 程序 dp339 演示了指向函数指针的使用,分析对应的目标代码,观察指向函数指针的具体实现细节。

```
#include <stdio.h>
int  max(int x, int y);              //声明函数原型
int  min(int x, int y);              //声明函数原型
//
int  main()
{
    int  (*pf)(int,int);             //定义指向函数的指针变量
    int  val1, val2;                 //存放结果的变量

    pf=max;                          //使得 pf 指向函数 max
    val1=(*pf)(13,15);               //调用由 pf 指向的函数

    pf=min;                          //使得 pf 指向函数 min
    val2=(*pf)(23,25);               //调用由 pf 指向的函数

    printf("%d,%d\n",val1,val2);     //显示为 15,23
    return  0;
}
```

为了更清楚地观察到指向函数的指针变量的具体实现,补充函数 max 和函数 min 的源代码,不采用编译优化,编译上述源程序后,可得如下的目标代码(标号稍有修饰)。

```
;程序 dp339 的目标代码(不采用编译优化)
;标号 max_YAHHH、标号 min_YAHHH,分别表示函数的入口地址
push  ebp
mov   ebp, esp                       ;建立堆栈框架
sub   esp, 12                        ;安排 3 个局部变量 pf、val1 和 val2
                                     ;pf=max;
mov   DWORD PTR [ebp-4], OFFSET  max_YAHHH
                                     ;val1=(*pf)(13,15);
push  15
push  13
call  DWORD PTR [ebp-4]              ;间接调用指针所指的函数 max
add   esp, 8
```

```
                                        ;val1=返回结果
mov   DWORD PTR [ebp-12], eax
                                        ;pf=min;
mov   DWORD PTR [ebp- 4], OFFSET  min_YAHHH
                                        ;val2=(*pf)(23,25);
push  25
push  23
call  DWORD PTR [ebp- 4]                ;间接调用指针所指的函数 min
add   esp, 8
                                        ;val2=返回结果
mov   DWORD PTR [ebp- 8], eax
                                        ;printf("%d,%d\n",val1,val2);
mov   eax, DWORD PTR [ebp- 8]           ;eax=val2
push  eax
mov   ecx, DWORD PTR [ebp-12]           ;ecx=val1
push  ecx
push  OFFSET  FORMTS                    ;格式字符串
call  _printf                           ;段内直接调用 //@c
add   esp, 12                           ;平衡堆栈
                                        ;
xor   eax, eax                          ;准备返回值
mov   esp, ebp                          ;撤销局部变量
pop   ebp                               ;撤销堆栈框架
ret
```

从上述目标代码可知,依靠间接调用指令,实现指向函数的指针变量的功效。函数的 3 个局部变量被安排在堆栈中;通过堆栈向子程序传递参数;在调用结束后,及时平衡堆栈。

如果在项目的常规配置属性“MFC 的使用”选项中,采用“使用标准 Windows 库”,编译上述源程序 dp339 后,所得目标代码会采用间接调用方式调用库函数,而不是现在的段内直接调用方式,见注释带//@c 的行。

3. 过程返回指令

过程返回指令用于从子程序返回到主程序。在执行该指令时,从堆栈顶弹出返回地址,并转移到所弹出的地址,这样就实现了返回。通常,这个返回地址就是在执行对应的调用指令时所压入堆栈的返回地址。

过程返回指令的使用应该与过程调用指令所对应。如上所述,过程调用分为段内调用和段间调用,所以过程返回指令也分为段内返回和段间返回。过程返回指令没有直接与间接之分。但过程返回指令却可选带一个立即数,以便在返回的同时撤销在堆栈中的参数。

过程返回指令不影响标志寄存器中的状态标志。

(1) 段内返回指令。在前面示例中所用的返回指令,都是段内返回指令。

(2) 段内带立即数返回指令。段内带立即数返回指令的格式如下,其中 count 是一个 16 位的立即数:

RET　*count*

该指令在实现段内返回的同时,再额外根据 count 值调整堆栈指针。具体操作是,先从堆栈弹出返回地址偏移(当然会调整 ESP),再把 count 加到堆栈指针 ESP 上。

(3) 段间返回指令。段间返回指令与段间调用指令相对应,它从堆栈顶弹出的返回地址不仅包含返回地址偏移,还包括返回地址的段号(段值或者段选择子)。在第 6 章将进一步介绍段间返回指令。

4. 带立即数返回指令应用示例

下面通过示例来说明带立即数返回指令的应用。

【例 3-65】 对比分析如下所示的 C 语言源程序 dp340 及其目标代码。

```
#include<stdio.h>
int  _stdcall  cf341(int x, int y)
{
    return  (2*x+5*y+100);
}
//
int  main()
{
    int  val;
    val=cf341(23, 456);
    printf("val=%d\n", val);
    return  0;
}
```

除了自定义的函数名为 cf341 和调用约定_stdcall 之外,上述 C 语言源程序与例 3-1 的源程序 dp31 和例 3-4 的源程序 dp33 几乎相同。由于采用了调用约定_stdcall,因此由函数(子程序)在返回的同时,撤销调用时压入堆栈的参数。

在项目属性中,采用配置属性选项“在静态库中使用 MFC”,同时采用编译优化选项“使速度最大化”,编译上述程序后,可得到如下所示的目标代码(标号和名称稍有修饰)。

```
;函数 main 的目标代码
push  456                          ;val=cf341(23, 456);
push  23
call  cf341YGHHH                   ;直接调用函数 cf341
                                   ;printf("val=%d\n", val);
push  eax
push  OFFSET  FORMS
call  DWORD PTR __imp__printf      ;间接调用函数_printf
add   esp, 8
xor   eax, eax                     ;准备返回值
ret
```

从上述 main 函数的目标代码可知,在调用函数 cf341 之前,把参数压入堆栈,但从函数 cf341 返回之后,并没有平衡堆栈,撤销堆栈中的参数。由于配置属性中“在静态库中使用 MFC”,因此采用间接调用指令调用库函数 printf,而且随后就平衡堆栈。此外,由于编译优化选项“使速度最大化”,因此用寄存器 eax 充当局部变量 val。

```
;函数 cf341 的目标代码
;通过堆栈传递入口参数 x 和 y
;返回时撤销由主程序压入堆栈的参数 x 和 y
push  ebp
```

```
mov   ebp, esp                          ;建立堆栈框架
mov   eax, DWORD PTR [ebp+12]           ;取得参数 y
mov   ecx, DWORD PTR [ebp+8]            ;取得参数 x
lea   eax, DWORD PTR [eax+ eax*4+100]
lea   eax, DWORD PTR [eax+ ecx*2]
pop   ebp                               ;撤销堆栈框架
ret   8                                 ; //@!
```

从上述目标代码可知,最后的返回指令 ret 是带立即数 8 的返回指令。在返回的同时,还使得堆栈指针寄存器 ESP 加 8,这样就撤销了主程序调用它时压入堆栈的参数。这种处理方式,更加需要主程序和子程序的配合协调。

习　　题

1. 请指出下列指令的错误所在。

```
(1) MUL   AX, 5          (2) DIV   13
(3) AND   BX, AL         (4) OR    AL, EDX
(5) XOR   0FH, AL        (6) TEST  3, DX
(7) SHL   EAX, CX        (8) ROR   BX, DL
```

2. 什么是除法溢出？如何解决 32 位被除数 16 位除数可能产生的溢出问题？

3. 说明符号扩展和零扩展操作的异同,并列举相关的操作指令。

4. 写出以下程序片段中每条逻辑运算指令执行后标志 ZF、SF 和 PF 的状态。

```
MOV   AL, 45H
AND   AL, 0FH
OR    AL, 0C3H
XOR   AL, AL
```

5. 写出以下程序片段中每条移位指令执行后标志 CF、ZF、SF 和 PF 的状态。

```
MOV   AL, 84H
SAR   AL, 1
SHR   AL, 1
ROR   AL, 1
RCL   AL, 1
SHL   AL, 1
ROL   AL, 1
```

6. 请说明下列指令片段的功能。

```
xor   eax,eax
mov   al, [esp+4]
mov   ecx, eax
shl   eax, 8
add   eax, ecx
mov   ecx, eax
shl   eax, 10h
add   eax, ecx
```

7. 哪些指令把寄存器 ECX 作为计数器？哪些指令把寄存器 CL 作为计数器？

8. 指令“MOV AL,0”使寄存器 AL 清 0。写出至少另外 4 条可使寄存器 AL 清 0 的指令。

9. 假设寄存器 EBX 内容为 2。至少写出 4 条使寄存器 EBX 内容为 1 的指令。

10. 指令“MOV EDX,1”使寄存器 EDX 内容为 1。另外写出 3 个使寄存器 EDX 内容为 1 的指令片段,每个指令片段最多含有两条指令,且指令各不相同(不同助记符)。

11. 假设寄存器中 AL 含有一个无符号整数,请给出判断该数据为奇数或者偶数的多种方法。

12. 利用算术左移指令或算术右移指令可以实现乘或除计算操作,这与使用乘法指令或除法指令进行类似计算操作有何区别?

13. 与利用条件转移指令实现循环操作相比,采用专门的循环指令实现循环操作有何优点?

14. 实现同一功能,往往有多种方法。在选择方法时,要考虑哪些因素?

15. 请举例说明如何利用段内无条件转移指令 JMP 调用子程序。

16. 请举例说明如何利用返回指令 RET 调用子程序。

17. 具有何种特点的程序片段应该设计成子程序或者过程?

18. 设计子程序时,需要注意哪些问题?子程序说明信息应包含哪些内容?

19. 主程序与子程序之间如何传递参数?请举例说明每种方法,并对这些方法作比较。

20. 如果利用堆栈传递参数,那么有两种平衡堆栈的方法,请比较这两种方法。

21. 请举例说明堆栈的 4 种主要用途。列举一个能够同时反映这些用途的示例。

22. 请简要说明 C 语言中未初始化局部变量的初值是随机值的原因。

23. 相对转移和绝对转移的区别是什么?相对转移有何优点?

24. 直接转移和间接转移各自有何特点?分别适用于哪些场合?

25. 请画出 3.1 节的例 3-7 中调用函数 cf37 期间堆栈的变化示意图。

26. 参考 3.2 节的例 3-16,修改被除数的尺寸,观察除法操作溢出的结果;尝试把被除数扩展到 64 位,观察除法操作的结果。

27. 参考 3.2 节的例 3-19,删除函数 cf311 中的强制类型转换,然后编译生成目标代码,并进行比较分析。

28. 参考 3.3 节的例 3-43,删除函数 cf319 中 case 4 及以上的分支情形,然后编译生成目标代码,并进行比较分析。

29. 参考 3.4 节的例 3-52,进一步优化嵌入汇编代码 as327。

30. 参考 3.4 节的例 3-54,修改嵌入汇编代码片段 as329,采用其他排序算法。

31. 参考 3.5 节的例 3-61,进一步优化嵌入汇编代码 as337。

32. 参考 3.5 节的例 3-64 演示程序 dp339,调整编译选项采用“使用标准 Windows 库”,分析编译后生成的目标代码。

33. 采用速度最大化的编译选项,编译生成如下函数的目标代码,并进行观察分析。

```
char *my( char *dst, char value, unsigned int count)
{
    char *start=dst;
    while  ( count-- )
        *dst++=value;
```

```
    return (start);
}
```

34. 在 VC 2010 环境下，编辑运行如下控制台程序，请给出运行结果。

```
#include <stdio.h>
int  varx=6;
int  buff[5]={1,2,3,4,5};
int  main()
{
    int  var_a, var_b, var_c;
    _asm{
        MOV   EAX, 6
        CALL  SUBR1
        MOV   var_a, EAX
        ;
        PUSH  DWORD PTR 5
        LEA   EAX, buff
        PUSH  EAX
        CALL  SUBR2
        ADD   ESP, 8
        MOV   var_b, EAX
        ;
        LEA   EAX, var_c
        PUSH  EAX
        MOV   EAX, varx
        PUSH  EAX
        CALL  SUBR3
    }
    printf("D=%d,E=%d,F=%d\n", var_a, var_b, var_c);
    return  0;
    _asm{
    SUBR1:
        ADD   EAX, 7
        RET
        ;
    SUBR2:
        PUSH  EBP
        MOV   EBP, ESP
        MOV   ECX, [EBP+12]
        MOV   EDX, [EBP+8]
        XOR   EAX, EAX
    SUBR2A:
        ADD   EAX, [EDX]
        ADD   EDX, 4
        LOOP  SUBR2A
        POP   EBP
        RET
        ;
```

```
    SUBR3:
        PUSH  EBP
        MOV   EBP, ESP
        MOV   EAX, [EBP+8]
        MOV   ECX, [EBP+12]
        ADD   EAX, 8
        MOV   [ECX], EAX
        POP   EBP
        RET   8
    }
}
```

下列编程题，除了输入和输出操作之外，请采用嵌入汇编的形式实现。

35. 由用户输入一个无符号整数(整型)；统计该 32 位整数中位值为 0 的个数；显示输出统计结果。

36. 由用户输入两个字符(字符型)；将它们视为两个 8 位数据，合并成一个 16 位无符号整数($a_7b_7a_6b_6a_5b_5a_4b_4a_3b_3a_2b_2a_1b_1a_0b_0$)；显示输出合并的数据。

37. 由用户输入两个整数(整型)；分别取得它们的绝对值，作为新数据；交换这两个 32 位数据的高 16 位，得到新的整数；显示输出这两个新的整数。

38. 由用户从键盘输入一个字符串；分别统计字符串中英文字母、十进制数字符和其他符号的个数；显示输出统计结果。

39. 由用户从键盘输入一个字符串；逆序排列字符串中的所有字符；显示输出逆序所得字符串。

40. 由用户从键盘输入一个字符串；删除字符串中的所有非英文字母，形成新的字符串；显示输出新的字符串。

41. 由用户从键盘输入一个字符串；将所有可能的小写字母转换为对应的大写字母；最后显示输出字符串。请采用子程序实现把可能的小写字母转换为大写字母。

42. 由用户输入一个十进制整数(整型)；将该整数转换为对应十进制数的 ASCII 码字符串；然后显示输出所得字符串。请采用子程序实现把整数转换为十进制数的字符串。

43. 请编写程序实现：由用户从键盘输入一个字符串；然后测量字符串长度；最后输出字符串长度。要求输出时，只能采用字符串格式(先把长度值转换为对应的十进制数字符串)。

44. 由用户输入一个无符号十进制整数(整型)；将该整数转换为对应十六进制数输出。要求输出时，只能采用字符串格式(先转换为对应的十六进制数字符串)。

45. 由用户输入一个字符，分别用十进制数形式、十六进制数形式和二进制数形式，显示输出其对应的 ASCII 码。请采用合适的子程序。要求输出时，只能采用字符串格式。

46. 由用户从键盘输入一个字符串；然后统计字符串中元音字母的个数；最后以八进制数形式输出统计结果。请采用合适的子程序。要求输出时，只能采用字符串格式。

47. 由用户从键盘输入一个十进制数字符串(假设不含其他字符)；然后把该十进制数字符串转换成对应的数值；接着把该数值转换成对应的十六进制数字符串；最后输出十六进制数字符串。请采用子程序实现把十进制数字符串转换为对应的数值。

48. 由用户以字符串形式输入一个十六进制数(假设其不含其他字符)，显示输出对应的十进制数。请采用合适的子程序。

49. 由用户输入一个十二进制数，将其转换为十三进制数输出。请采用合适的子程序。要求在输入和输出时，都只能采用字符串格式。

50. 由用户从键盘先后输入两个自然数；然后分别计算这两个数的和、差、积；分别显示输出结果。请采用合适的子程序。要求在输入和输出时，都只能采用字符串格式。

51. 由用户从键盘输入两个自然数；然后显示输出这两个数的商和余数。请采用合适的子程序。要求在输入和输出时，都只能采用字符串格式。

52. 由用户输入一个字符串，其中可能含有多个由十进制数字符构成的子串，将这些十进制数字符串作为数据值，显示输出这些数据之和。假设字符串为 A123We56st002345，其各数据之和为 2524。