

仅仅表示地址的地址标号。

检测点 16.1

下面的程序将 code 段中 a 处的 8 个数据累加，结果存储到 b 处的双字中，补全程序。

```
assume cs:code
code segment
    a dw 1,2,3,4,5,6,7,8
    b dd 0

start:mov si,0
      mov cx,8
s:    mov ax,_____
      add _____,ax
      adc _____,0
      add si,_____
      loop s

      mov ax,4c00h
      int 21h

code ends
end start
```

16.2 在其他段中使用数据标号

一般来说，我们不在代码段中定义数据，而是将数据定义到其他段中。在其他段中，我们也可以使用数据标号来描述存储数据的单元的地址和长度。

注意，在后面加有“.”的地址标号，只能在代码段中使用，不能在其他段中使用。

下面的程序将 data 段中 a 标号处的 8 个数据累加，结果存储到 b 标号处的字中。

```
assume cs:code,ds:data
data segment
    a db 1,2,3,4,5,6,7,8
    b dw 0
data ends

code segment
start:  mov ax,data
        mov ds,ax

        mov si,0
        mov cx,8
```

```
s:  mov al,a[si]
    mov ah,0
    add b,ax
    inc si
    loop s

    mov ax,4c00h
    int 21h

code ends
end start
```

注意, 如果想在代码段中直接用数据标号访问数据, 则需要用伪指令 `assume` 将标号所在的段和一个段寄存器联系起来。否则编译器在编译的时候, 无法确定标号的段地址在哪一个寄存器中。当然, 这种联系是编译器需要的, 但绝对不是说, 我们因为编译器的工作需要, 用 `assume` 指令将段寄存器和某个段相联系, 段寄存器中就会真的存放该段的地址。我们在程序中还要使用指令对段寄存器进行设置。

比如, 在上面的程序中, 我们要在代码段 `code` 中用 `data` 段中的数据标号 `a`、`b` 访问数据, 则必须用 `assume` 将一个寄存器和 `data` 段相联。在程序中, 我们用 `ds` 寄存器和 `data` 段相联, 则编译器对相关指令的编译如下。

```
指令:  mov al,a[si]
编译为: mov al,[si+0]

指令:  add b,ax
编译为: add [8].ax
```

因为这些实际编译出的指令, 都默认所访问单元的段地址在 `ds` 中, 而实际要访问的段为 `data`, 所以若要访问正确, 在这些指令执行前, `ds` 中必须为 `data` 段的段地址。则我们在程序中使用指令:

```
mov ax,data
mov ds,ax
```

设置 `ds` 指向 `data` 段。

可以将标号当作数据来定义, 此时, 编译器将标号所表示的地址当作数据的值。

比如:

```
data segment
    a db 1,2,3,4,5,6,7,8
    b dw 0
    c dw a,b
data ends
```

数据标号 `c` 处存储的两个字型数据为标号 `a`、`b` 的偏移地址。相当于:

```
data segment
    a db 1,2,3,4,5,6,7,8
    b dw 0
    c dw offset a,offset b
data ends
```

再比如:

```
data segment
    a db 1,2,3,4,5,6,7,8
    b dw 0
    c dd a,b
data ends
```

数据标号 c 处存储的两个双字型数据为标号 a 的偏移地址和段地址、标号 b 的偏移地址和段地址。相当于:

```
data segment
    a db 1,2,3,4,5,6,7,8
    b dw 0
    c dw offset a, seg a, offset b, seg b
data ends
```

seg 操作符, 功能为取得某一标号的段地址。

检测点 16.2

下面的程序将 data 段中 a 处的 8 个数据累加, 结果存储到 b 处的字中, 补全程序。

```
assume cs:code,es:data

data segment
    a db 1,2,3,4,5,6,7,8
    b dw 0
data ends

code segment
start: _____
    _____
    mov si,0
    mov cx,8
s: mov al,a[si]
    mov ah,0
    add b,ax
    inc si
    loop s

    mov ax,4c00h
    int 21h
code ends
end start
```

16.3 直接定址表

现在,我们讨论用查表的方法编写相关程序的技巧。

编写子程序,以十六进制的形式在屏幕中间显示给定的字节型数据。

分析:一个字节需要用两个十六进制数码来表示,所以,子程序需要在屏幕上显示两个 ASCII 字符。我们当然要用“0”、“1”、“2”、“3”、“4”、“5”、“6”、“7”、“8”、“9”、“A”、“B”、“C”、“D”、“E”、“F”这 16 个字符来显示十六进制数码。

我们可以将一个字节的高 4 位和低 4 位分开,分别用它们的值得到对应的数码字符。比如 2Bh,可以得到高 4 位的值为 2,低 4 位的值为 11,那么如何用这两个数值得到对应的数码字符“2”和“B”呢?

最简单的办法就是一个一个地比较,如下:

如果数值为 0,则显示“0”;
如果数值为 1,则显示“1”;
⋮
如果数值为 11,则显示“B”;
⋮

我们可以看出,这样做,程序中要使用多条比较、转移指令。程序将比较长,混乱。

显然,我们希望能够在数值 0~15 和字符“0”~“F”之间找到一种映射关系。这样用 0~15 间的任何数值,都可以通过这种映射关系直接得到“0”~“F”中对应的字符。

数值 0~9 和字符“0”~“9”之间的映射关系是很明显的,即:

数值+30h=对应字符的 ASCII 值
0+30h=“0”的 ASCII 值
1+30h=“1”的 ASCII 值
2+30h=“2”的 ASCII 值
⋮

但是,10~15 和“A”~“F”之间的映射关系是:

数值+37h=对应字符的 ASCII 值
10+37h=“A”的 ASCII 值
11+37h=“B”的 ASCII 值
12+37h=“C”的 ASCII 值
⋮

可见，我们可以利用数值和字符之间的这种原本存在的映射关系，通过高 4 位和低 4 位值得到对应的字符码。但是由于映射关系的不同，我们在程序中必须进行一些比较，对于大于 9 的数值，我们要用不同的计算方法。

这样做，虽然使程序得到了简化。但是，如果我们希望用更简捷的算法，就要考虑用同一种映射关系从数值得到字符码。所以，我们就不能利用 0~9 和“0”~“9”之间与 10~15 和“A”~“F”之间原有的映射关系。

因为数值 0~15 和字符“0”~“F”之间没有一致的映射关系存在，所以，我们应该在它们之间建立新的映射关系。

具体的做法是，建立一张表，表中依次存储字符“0”~“F”，我们可以通过数值 0~15 直接查找到对应的字符。

子程序如下。

；用 al 传送要显示的数据

```
showbyte:  jmp short show

           table db '0123456789ABCDEF'          ;字符表

show:      push bx
           push es

           mov ah,al
           shr ah,1
           shr ah,1
           shr ah,1
           shr ah,1                                ;右移 4 位，ah 中得到高 4 位的值
           and al,00001111b                        ;al 中为低 4 位的值

           mov bl,ah
           mov bh,0
           mov ah,table[bx] ；用高 4 位的值作为相对于 table 的偏移，取得对应的字符

           mov bx,0b800h
           mov es,bx
           mov es:[160*12+40*2],ah

           mov bl,al
           mov bh,0
           mov al,table[bx] ；用低 4 位的值作为相对于 table 的偏移，取得对应的字符

           mov es:[160*12+40*2+2],al

           pop es
           pop bx

           ret
```

可以看出,在子程序中,我们在数值 0~15 和字符“0”~“F”之间建立的映射关系为:以数值 N 为 table 表中的偏移,可以找到对应的字符。

利用表,在两个数据集合之间建立一种映射关系,使我们可以用查表的方法根据给出的数据得到其在另一集合中的对应数据。这样做的目的一般来说有以下 3 个。

- (1) 为了算法的清晰和简洁;
- (2) 为了加快运算速度;
- (3) 为了使程序易于扩充。

在上面的子程序中,我们更多的是为了算法的清晰和简洁,而采用了查表的方法。下面我们来看一下,为了加快运算速度而采用查表的方法的情况。

编写一个子程序,计算 $\sin(x)$, $x \in \{0^\circ, 30^\circ, 60^\circ, 90^\circ, 120^\circ, 150^\circ, 180^\circ\}$, 并在屏幕中间显示计算结果。比如 $\sin(30)$ 的结果显示为“0.5”。

我们可以利用麦克劳林公式来计算 $\sin(x)$ 。x 为角度,麦克劳林公式中需要代入弧度,则:

$$\sin(x) = \sin(y) \approx y - \frac{1}{3!} y^3 + \frac{1}{5!} y^5$$

$$y = x / 180 * 3.1415926$$

可以看出,计算 $\sin(x)$ 需要进行多次乘法和除法。乘除是非常费时的运算,它们的执行时间大约是加法、比较等指令的 5 倍。如何才能够不做乘除而计算 $\sin(x)$ 呢?我们看一下需要计算的 $\sin(x)$ 的结果:

```
sin(0)=0
sin(30)=0.5
sin(60)=0.866
sin(90)=1
sin(120)=0.866
sin(150)=0.5
sin(180)=0
```

我们可以看出,其实用不着计算,可以占用一些内存空间来换取运算的速度。将所要计算的 $\sin(x)$ 的结果都存储到一张表中;然后用角度值来查表,找到对应的 $\sin(x)$ 的值。

用 ax 向子程序传递角度,程序如下:

```
showsine: jmp short show
```

```
table    dw  ag0,ag30,ag60,ag90,ag120,ag150,ag180    ;字符串偏移地址表
ag0      db  '0',0                                   ;sin(0) 对应的字符串“0”
```

```

    ag30    db '0.5',0                ;sin(30)对应的字符串“0.5”
    ag60    db '0.866',0              ;sin(60)对应的字符串“0.866”
    ag90    db '1',0                  ;sin(90)对应的字符串“1”
    ag120   db '0.866',0              ;sin(120)对应的字符串“0.866”
    ag150   db '0.5',0                ;sin(150)对应的字符串“0.5”
    ag180   db '0',0                  ;sin(180)对应的字符串“0”

show:  push bx
       push es
       push si
       mov bx,0b800h
       mov es,bx

;以下用角度值/30 作为相对于 table 的偏移，取得对应的字符串的偏移地址，放在 bx 中
       mov ah,0
       mov bl,30
       div bl
       mov bl,al
       mov bh,0
       add bx,bx
       mov bx,table[bx]

;以下显示 sin(x) 对应的字符串
       mov si,160*12+40*2
shows: mov ah,cs:[bx]
       cmp ah,0
       je showret
       mov es:[si],ah
       inc bx
       add si,2
       jmp short shows
showret:pop si
       pop es
       pop bx
       ret

```

在上面的子程序中，我们在角度值 X 和表示 $\sin(x)$ 的字符串集合 `table` 之间建立的映射关系为：以角度值/30 为 `table` 表中的偏移，可以找到对应的字符串的首地址。

编程的时候要注意程序的容错性，即对于错误的输入要有处理能力。在上面的子程序中，我们还应该再加上对提供的角度值是否超范围的检测。如果提供的角度值不在合法的集合中，程序将定位不到正确的字符串，出现错误。对于角度值的检测，请读者自行完成。

上面的两个子程序中，我们将通过给出的数据进行计算或比较而得到结果的问题，转化为用给出的数据作为查表的依据，通过查表得到结果的问题。具体的查表方法，是用查表的依据数据，直接计算出所要查找的元素在表中的位置。像这种可以通过依据数据，直接计算出所要找的元素的位置的表，我们称其为直接定址表。

16.4 程序入口地址的直接定址表

我们可以在直接定址表中存储子程序的地址，从而方便地实现不同子程序的调用。我们看下面的问题。

实现一个子程序 `setscreen`，为显示输出提供如下功能。

- (1) 清屏；
- (2) 设置前景色；
- (3) 设置背景色；
- (4) 向上滚动一行。

入口参数说明如下。

- (1) 用 `ah` 寄存器传递功能号：0 表示清屏，1 表示设置前景色，2 表示设置背景色，3 表示向上滚动一行；
- (2) 对于 1、2 号功能，用 `al` 传送颜色值， $(al) \in \{0, 1, 2, 3, 4, 5, 6, 7\}$ 。

下面我们讨论一下各种功能如何实现。

- (1) 清屏：将显存中当前屏幕中的字符设为空格符；
- (2) 设置前景色：设置显存中当前屏幕中处于奇地址的属性字节的第 0、1、2 位；
- (3) 设置背景色：设置显存中当前屏幕中处于奇地址的属性字节的第 4、5、6 位；
- (4) 向上滚动一行：依次将第 $n+1$ 行的内容复制到第 n 行处；最后一行为空。

我们将这 4 个功能分别写为 4 个子程序，请读者根据编程思想，自行读懂下面的程序。

```
sub1:  push bx
       push cx
       push es
       mov bx, 0b800h
       mov es, bx
       mov bx, 0
       mov cx, 2000
sub1s: mov byte ptr es:[bx], ' '
       add bx, 2
       loop sub1s
       pop es
       pop cx
       pop bx
       ret
```

```

sub2:  push bx
       push cx
       push es

       mov bx,0b800h
       mov es,bx
       mov bx,1
       mov cx,2000
sub2s:  and byte ptr es:[bx],11111000b
       or  es:[bx],al
       add bx,2
       loop sub2s

       pop es
       pop cx
       pop bx
       ret

sub3:   push bx
       push cx
       push es
       mov cl,4
       shl al,cl
       mov bx,0b800h
       mov es,bx
       mov bx,1
       mov cx,2000
sub3s:  and byte ptr es:[bx],10001111b
       or  es:[bx],al
       add bx,2
       loop sub3s

       pop es
       pop cx
       pop bx
       ret

sub4:   push cx
       push si
       push di
       push es
       push ds

       mov si,0b800h
       mov es,si
       mov ds,si
       mov si,160           ;ds:si 指向第 n+1 行
       mov di,0             ;es:di 指向第 n 行
       cld
       mov cx,24             ;共复制 24 行

sub4s:  push cx
       mov cx,160
       rep movsb             ;复制

```

```

        pop cx
        loop sub4s

        mov cx,80
        mov si,0
sub4s1: mov byte ptr [160*24+si], ' '      ;最后一行清空
        add si,2
        loop sub4s1

        pop ds
        pop es
        pop di
        pop si
        pop cx
        ret

```

我们可以将这些功能子程序的入口地址存储在一个表中，它们在表中的位置和功能号相对应。对应关系为：功能号*2=对应的功能子程序在地址表中的偏移。程序如下：

```

setscreen: jmp short set

        table dw sub1,sub2,sub3,sub4

        set: push bx

            cmp ah,3                ;判断功能号是否大于3
            ja sret
            mov bl,ah
            mov bh,0
            add bx,bx              ;根据 ah 中的功能号计算对应子程序在 table 表中的偏移

            call word ptr table[bx] ;调用对应的功能子程序

sret:    pop bx
        ret

```

当然，我们也可以将子程序 `setscreen` 如下实现。

```

setscreen: cmp ah,0
            je do1
            cmp ah,1
            je do2
            cmp ah,2
            je do3
            cmp ah,3
            je do4
            jmp short sret

do1:      call sub1
            jmp short sret
do2:      call sub2

```

```
        jmp short sret
do3:    call sub3
        jmp short sret
do4:    call sub4

sret:   ret
```

显然，用通过比较功能号进行转移的方法，程序结构比较混乱，不利于功能的扩充。比如说，在 `setscreen` 中再加入一个功能，则需要修改程序的逻辑，加入新的比较、转移指令。

用根据功能号查找地址表的方法，程序的结构清晰，便于扩充。如果加入一个新的功能子程序，那么只需要在地址表中加入它的入口地址就可以了。

实验 16 编写包含多个功能子程序的中断例程

安装一个新的 `int 7ch` 中断例程，为显示输出提供如下功能子程序。

- (1) 清屏；
- (2) 设置前景色；
- (3) 设置背景色；
- (4) 向上滚动一行。

入口参数说明如下。

(1) 用 `ah` 寄存器传递功能号：0 表示清屏，1 表示设置前景色，2 表示设置背景色，3 表示向上滚动一行；

(2) 对于 1、2 号功能，用 `al` 传送颜色值， $(al) \in \{0,1,2,3,4,5,6,7\}$ 。

第 17 章 使用 BIOS 进行键盘输入和磁盘读写

大多数有用的程序都需要处理用户的输入，键盘输入是最基本的输入。程序和数据通常需要长期存储，磁盘是最常用的存储设备。BIOS 为这两种外设的 I/O 提供了最基本的中断例程，在本章中，我们对它们的应用和相关的问题进行讨论。

17.1 int 9 中断例程对键盘输入的处理

我们已经讲过，键盘输入将引发 9 号中断，BIOS 提供了 int 9 中断例程。CPU 在 9 号中断发生后，执行 int 9 中断例程，从 60h 端口读出扫描码，并将其转化为相应的 ASCII 码或状态信息，存储在内存的指定空间(键盘缓冲区或状态字节)中。

一般的键盘输入，在 CPU 执行完 int 9 中中断例程后，都放到了键盘缓冲区中。键盘缓冲区中有 16 个字单元，可以存储 15 个按键的扫描码和对应的 ASCII 码。

下面我们按照键盘缓冲区的逻辑结构，来看一下键盘输入的扫描码和对应的 ASCII 码是如何写入键盘缓冲区的。

注意：在我们的课程中，仅在逻辑结构的基础上，讨论 BIOS 键盘缓冲区的读写问题。其实键盘缓冲区是用环形队列结构管理的内存区，但我们不对队列和环形队列的实现进行讨论，因为那是另一门专业课《数据结构》的内容。

下面，我们通过下面几个键：

A, B, C, D, E, Shift A, A

的输入过程，简要地看一下 int 9 中断例程对键盘输入的处理方法。

- (1) 初始状态下, 没有键盘输入, 键盘缓冲区空, 此时没有任何元素。

[illegible]

(2) 按下 A 键, 引发键盘中断; CPU 执行 int 9 中断例程, 从 60h 端口读出 A 键的扫描码; 然后检测状态字节, 看看是否有 Shift、Ctrl 等切换键按下; 发现没有切换键按下, 则将 A 键的扫描码 1eh 和对应的 ASCII 码, 即字母 “a” 的 ASCII 码 61h, 写入键盘缓冲区。缓冲区的字单元中, 高位字节存储扫描码, 低位字节存储 ASCII 码。此时缓冲区中的内容如下。

[illegible]

(3) 按下 B 键，引发键盘中断；CPU 执行 int 9 中断例程，从 60h 端口读出 B 键的通码；然后检测状态字节，看看是否有切换键按下；发现没有切换键按下，将 B 键的扫描码 30h 和对应的 ASCII 码，即字母“b”的 ASCII 码 62h，写入键盘缓冲区。此时缓冲区中的内容如下。

1E61	3062														
------	------	--	--	--	--	--	--	--	--	--	--	--	--	--	--

(4) 按下 C、D、E 键后，缓冲区中的内容如下。

1E61	3062	2E63	2064	1265											
------	------	------	------	------	--	--	--	--	--	--	--	--	--	--	--

(5) 按下左 Shift 键，引发键盘中断；int 9 中断例程接收左 Shift 键的通码，设置 0040:17 处的状态字节的第 1 位为 1，表示左 Shift 键按下。

(6) 按下 A 键，引发键盘中断；CPU 执行 int 9 中断例程，从 60h 端口读出 A 键的通码；检测状态字节，看看是否有切换键按下；发现左 Shift 键被按下，则将 A 键的扫描码 1Eh 和 Shift_A 对应的 ASCII 码，即字母“A”的 ASCII 码 41h，写入键盘缓冲区。此时缓冲区中的内容如下。

1E61	3062	2E63	2064	1265	1E41										
------	------	------	------	------	------	--	--	--	--	--	--	--	--	--	--

(7) 松开左 Shift 键，引发键盘中断；int 9 中断例程接收左 Shift 键的通码，设置 0040:17 处的状态字节的第 1 位为 0，表示左 Shift 键松开。

(8) 按下 A 键，引发键盘中断；CPU 执行 int 9 中断例程，从 60h 端口读出 A 键的通码；然后检测状态字节，看看是否有切换键按下；发现没有切换键按下，则将 A 键的扫描码 1Eh 和 A 对应的 ASCII 码，即字母“a”的 ASCII 码 61h，写入键盘缓冲区。此时缓冲区中的内容如下。

1E61	3062	2E63	2064	1265	1E41	1E61									
------	------	------	------	------	------	------	--	--	--	--	--	--	--	--	--

17.2 使用 int 16h 中断例程读取键盘缓冲区

BIOS 提供了 int 16h 中断例程供程序员调用。int 16h 中断例程中包含的一个最重要的功能是从键盘缓冲区中读取一个键盘输入，该功能的编号为 0。下面的指令从键盘缓冲区中读取一个键盘输入，并且将其从缓冲区中删除：

```
mov ah,0
int 16h
```

结果：(ah)=扫描码，(al)=ASCII 码。

下面我们接着上一节中的键盘输入过程，看一下 int 16h 如何读取键盘缓冲区。

(1) 执行

```
mov ah,0
int 16h
```

后,缓冲区中的内容如下。

3062	2E63	2064	1265	1E41	1E61										
------	------	------	------	------	------	--	--	--	--	--	--	--	--	--	--

ah 中的内容为 1Eh, al 中的内容为 61h。

(2) 执行

```
mov ah,0
int 16h
```

后,缓冲区中的内容如下。

2E63	2064	1265	1E41	1E61											
------	------	------	------	------	--	--	--	--	--	--	--	--	--	--	--

ah 中的内容为 30h, al 中的内容为 62h。

(3) 执行

```
mov ah,0
int 16h
```

后,缓冲区中的内容如下。

2064	1265	1E41	1E61												
------	------	------	------	--	--	--	--	--	--	--	--	--	--	--	--

ah 中的内容为 2Eh, al 中的内容为 63h。

(4) 执行 4 次

```
mov ah,0
int 16h
```

后,缓冲区空。

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

ah 中的内容为 1Eh, al 中的内容为 61h。

(5) 执行

```
mov ah,0
int 16h
```

int 16h 中断例程检测键盘缓冲区,发现缓冲区空,则循环等待,直到缓冲区中有数据。

(6) 按下 A 键后, 缓冲区中的内容如下。

1E61															
------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

(7) 循环等待的 int 16h 中断例程检测到键盘缓冲区中有数据, 将其读出, 缓冲区又为空。

--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

ah 中的内容为 1Eh, al 中的内容为 61h。

从上面我们可以看出, int 16h 中断例程的 0 号功能, 进行如下的工作。

- (1) 检测键盘缓冲区中是否有数据;
- (2) 没有则继续做第 1 步;
- (3) 读取缓冲区第一个字单元中的键盘输入;
- (4) 将读取的扫描码送入 ah, ASCII 码送入 al;
- (5) 将已读取的键盘输入从缓冲区中删除。

可见, BIOS 的 int 9 中断例程和 int 16h 中断例程是一对相互配合的程序, int 9 中断例程向键盘缓冲区中写入, int 16h 中断例程从缓冲区中读出。它们写入和读出的时机不同, int 9 中断例程是在有键按下时向键盘缓冲区中写入数据; 而 int 16h 中断例程是在应用程序对其进行调用的时候, 将数据从键盘缓冲区中读出。

我们在编写一般的处理键盘输入的程序的时候, 可以调用 int 16h 从键盘缓冲区中读取键盘的输入。

编程, 接收用户的键盘输入, 输入“r”, 将屏幕上的字符设置为红色; 输入“g”, 将屏幕上的字符设置为绿色; 输入“b”, 将屏幕上的字符设置为蓝色。

程序如下, 画线处的程序比较技巧, 请读者自行分析。

```
assume cs:code

code segment
start: mov ah,0
      int 16h

      mov ah,1
      cmp al,'r'
      je red
      cmp al,'g'
      je green
      cmp al,'b'
      je blue
      jmp short sret

red:  shl ah,1
```

```
green: shl ah,1

blue:  mov bx,0b800h
      mov es,bx
      mov bx,1
      mov cx,2000
      s: and byte ptr es:[bx],11111000b
        or es:[bx],ah
        add bx,2
        loop s

sret:  mov ax,4c00h
      int 21h

code ends
end start
```

检测点 17.1

“在 int 16h 中断例程中，一定有设置 IF=1 的指令。”这种说法对吗？

17.3 字符串的输入

用户通过键盘输入的通常不仅仅是单个字符而是字符串。下面我们讨论字符串输入中的问题和简单的解决方法。

最基本的字符串输入程序，需要具备下面的功能。

- (1) 在输入的同时需要显示这个字符串；
- (2) 一般在输入回车符后，字符串输入结束；
- (3) 能够删除已经输入的字符。

对于这 3 个功能，我们可以想象在 DOS 中，输入命令行时的情况。

编写一个接收字符串输入的子程序，实现上面 3 个基本功能。因为在输入的过程中需要显示，子程序的参数如下：

(dh)、(dl)=字符串在屏幕上显示的行、列位置；
ds:si 指向字符串的存储空间，字符串以 0 为结尾符。

下面我们进行分析。

- (1) 字符的输入和删除。

每个新输入的字符都存储在前一个输入的字符之后，而删除是从最后面的字符进行的，我们看下面的过程。

空字符串:

输入 “a”: a

输入 “b”: ab

输入 “c”: abc

输入 “d”: abcd

删除一个字符: abc

删除一个字符: ab

删除一个字符: a

删除一个字符:

可以看出在字符串输入的过程中, 字符的输入和输出是按照栈的访问规则进行的, 即后进先出。这样, 我们就可以用栈的方式来管理字符串的存储空间, 也就是说, 字符串的存储空间实际上是一个字符栈。字符栈中的所有字符, 从栈底到栈顶, 组成一个字符串。

(2) 在输入回车符后, 字符串输入结束。

输入回车符后, 可以在字符串中加入 0, 表示字符串结束。

(3) 在输入的同时需要显示这个字符串。

每次有新的字符输入和删除一个字符的时候, 都应该重新显示字符串, 即从字符栈的栈底到栈顶, 显示所有的字符。

(4) 程序的处理过程。

现在我们可以简单地确定程序的处理过程如下。

- ① 调用 int 16h 读取键盘输入;
- ② 如果是字符, 进入字符栈, 显示字符栈中的所有字符; 继续执行①;
- ③ 如果是退格键, 从字符栈中弹出一个字符, 显示字符栈中的所有字符; 继续执行①;
- ④ 如果是 Enter 键, 向字符栈中压入 0, 返回。

从程序的处理过程中可以看出, 字符栈的入栈、出栈和显示栈中的内容, 是需要有多处使用的功能, 我们应该将它们写为子程序。

子程序: 字符栈的入栈、出栈和显示。

参数说明: (ah)=功能号, 0 表示入栈, 1 表示出栈, 2 表示显示;

ds:si 指向字符栈空间;

对于 0 号功能: (al)=入栈字符;

对于 1 号功能: (al)=返回的字符;

对于 2 号功能: (dh)、(dl)=字符串在屏幕上显示的行、列位置。

```

charstack: jmp short charstart

table      dw charpush,charpop,charshow
top        dw 0                                ;栈顶

charstart: push bx
           push dx
           push di
           push es

           cmp ah,2
           ja sret
           mov bl,ah
           mov bh,0
           add bx,bx
           jmp word ptr table[bx]

charpush:  mov bx,top
           mov [si][bx],al
           inc top
           jmp sret

charpop:   cmp top,0
           je sret
           dec top
           mov bx,top
           mov al,[si][bx]
           jmp sret

charshow:  mov bx,0b800h
           mov es,bx
           mov al,160
           mov ah,0
           mul dh
           mov di,ax
           add dl,dl
           mov dh,0
           add di,dx

           mov bx,0

charshows: cmp bx,top
           jne noempty
           mov byte ptr es:[di], ' '
           jmp sret
noempty:   mov al,[si][bx]
           mov es:[di],al
           mov byte ptr es:[di+2], ' '
           inc bx
           add di,2
           jmp charshows

```

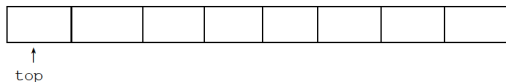
```

sret:      pop es
           pop di
           pop dx
           pop bx
           ret

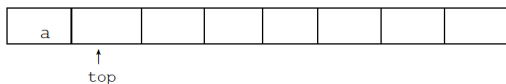
```

上面的子程序中，字符栈的访问规则如下所示。

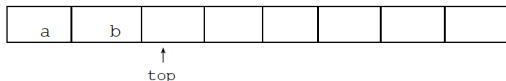
(1) 栈空



(2) “a”入栈



(3) “b”入栈



另外一个要注意的问题是，显示栈中字符的时候，要注意清除屏幕上上一次显示的内容。

我们现在写出完整的接收字符串输入的子程序，如下所示。

```

getstr:    push ax

getstrs:   mov ah,0
           int 16h
           cmp al,20h
           jb nochar           ;ASCII 码小于 20h, 说明不是字符
           mov ah,0
           call charstack      ;字符入栈
           mov ah,2
           call charstack      ;显示栈中的字符
           jmp getstrs

nochar:    cmp ah,0eh          ;退格键的扫描码
           je backspace
           cmp ah,1ch          ;Enter 键的扫描码
           je enter
           jmp getstrs

```

```
backspace: mov ah,1
            call charstack    ;字符出栈
            mov ah,2
            call charstack    ;显示栈中的字符
            jmp getstrs

enter:      mov al,0
            mov ah,0
            call charstack    ;0 入栈
            mov ah,2
            call charstack    ;显示栈中的字符
            pop ax
            ret
```

17.4 应用 int 13h 中断例程对磁盘进行读写

我们主要以 3.5 英寸软盘为例,进行讲解。

3.5 英寸软盘分为上下两面,每面有 80 个磁道,每个磁道又分为 18 个扇区,每个扇区的大小为 512 个字节。

则: 2 面*80 磁道*18 扇区*512 字节=1440KB $\approx$ 1.44MB

磁盘的实际访问由磁盘控制器进行,我们可以通过控制磁盘控制器来访问磁盘。只能以扇区为单位对磁盘进行读写。在读写扇区的时候,要给出面号、磁道号和扇区号。面号和磁道号从 0 开始,而扇区号从 1 开始。

如果我们通过直接控制磁盘控制器来访问磁盘,则需要涉及许多硬件细节。BIOS 提供了对扇区进行读写的中断例程,这些中断例程完成了许多复杂的和硬件相关的工作。我们可以通过调用 BIOS 中断例程来访问磁盘。

BIOS 提供的访问磁盘的中断例程为 int 13h。读取 0 面 0 道 1 扇区的内容到 0:200 的程序如下所示。

```
mov ax,0
mov es,ax
mov bx,200h

mov al,1
mov ch,0
mov cl,1
mov dl,0
mov dh,0
mov ah,2
int 13h
```

入口参数:

(ah)=int 13h 的功能号(2 表示读扇区)

(al)=读取的扇区数

(ch)=磁道号

(cl)=扇区号

(dh)=磁头号(对于软盘即面号, 因为一个面用一个磁头来读写)

(dl)=驱动器号 软驱从 0 开始, 0: 软驱 A, 1: 软驱 B;

硬盘从 80h 开始, 80h: 硬盘 C, 81h: 硬盘 D

es:bx 指向接收从扇区读入数据的内存区

返回参数:

操作成功: (ah)=0, (al)=读入的扇区数

操作失败: (ah)=出错代码

将 0:200 中的内容写入 0 面 0 道 1 扇区。

```
mov ax,0
mov es,ax
mov bx,200h
```

```
mov al,1
mov ch,0
mov cl,1
mov dl,0
mov dh,0
```

```
mov ah,3
int 13h
```

入口参数:

(ah)=int 13h 的功能号(3 表示写扇区)

(al)=写入的扇区数

(ch)=磁道号

(cl)=扇区号

(dh)=磁头号(面)

(dl)=驱动器号 软驱从 0 开始, 0: 软驱 A, 1: 软驱 B;

硬盘从 80h 开始, 80h: 硬盘 C, 81h: 硬盘 D

es:bx 指向将写入磁盘的数据

返回参数:

操作成功: (ah)=0, (al)=写入的扇区数

操作失败: (ah)=出错代码

注意,下面我们要使用 `int 13h` 中断例程对软盘进行读写。直接向磁盘扇区写入数据是很危险的,很可能覆盖掉重要的数据。如果向软盘的 0 面 0 道 1 扇区中写入了数据,要使软盘在现有的操作系统下可以使用,必须要重新格式化。在编写相关的程序之前,必须要找一张空闲的软盘。在使用 `int 13h` 中断例程时一定要注意驱动器号是否正确,千万不要随便对硬盘中的扇区进行写入。

编程:将当前屏幕的内容保存在磁盘上。

分析:1 屏的内容占 4000 个字节,需要 8 个扇区,用 0 面 0 道的 1~8 扇区存储显存中的内容。程序如下。

```
assume cs:code
code segment
start: mov ax,0b800h
      mov es,ax
      mov bx,0

      mov al,8
      mov ch,0
      mov cl,1
      mov dl,0
      mov dh,0
      mov ah,3
      int 13h

      mov ax,4c00h
      int 21h
code ends
end start
```

实验 17 编写包含多个功能子程序的中断例程

我们可以看出,用面号、磁道号、扇区号来访问磁盘不太方便。可以考虑对位于不同的磁道、面上的所有扇区进行统一编号。编号从 0 开始,一直到 2879,我们称这个编号为逻辑扇区编号。

编号的方法如下所示。

物理扇区号	逻辑扇区号
0 面 0 道 1 扇区	0
0 面 0 道 2 扇区	1
0 面 0 道 3 扇区	2
0 面 0 道 4 扇区	3
⋮	

0 面 0 道 18 扇区	17
0 面 1 道 1 扇区	18
0 面 1 道 2 扇区	19
0 面 1 道 3 扇区	20
0 面 1 道 4 扇区	21
⋮	
0 面 1 道 18 扇区	35
0 面 2 道 1 扇区	36
0 面 2 道 2 扇区	37
0 面 2 道 3 扇区	38
0 面 2 道 4 扇区	39
⋮	
0 面 79 道 18 扇区	1439
1 面 0 道 1 扇区	1440
1 面 0 道 2 扇区	1441
1 面 0 道 3 扇区	1442
1 面 0 道 4 扇区	1443

可以看出，逻辑扇区号和物理扇区号的关系如下：

逻辑扇区号 = (面号 * 80 + 磁道号) * 18 + 扇区号 - 1

那么如何根据逻辑扇区号算出物理编号呢？可以用下面的算法。

int()：描述性运算符，取商

rem()：描述性运算符，取余数

逻辑扇区号 = (面号 * 80 + 磁道号) * 18 + 扇区号 - 1

面号 = $\text{int}(\text{逻辑扇区号} / 1440)$

磁道号 = $\text{int}(\text{rem}(\text{逻辑扇区号} / 1440) / 18)$

扇区号 = $\text{rem}(\text{rem}(\text{逻辑扇区号} / 1440) / 18) + 1$

安装一个新的 int 7ch 中断例程，实现通过逻辑扇区号对软盘进行读写。

参数说明：

- (1) 用 ah 寄存器传递功能号：0 表示读，1 表示写；
- (2) 用 dx 寄存器传递要读写的扇区的逻辑扇区号；
- (3) 用 es:bx 指向存储读出数据或写入数据的内存区。

提示，用逻辑扇区号计算出面号、磁道号、扇区号后，调用 int 13h 中断例程进行实际的读写。

课程设计 2

阅读下面的材料:

开机后, CPU 自动进入到 FFFF:0 单元处执行, 此处有一条跳转指令。CPU 执行该指令后, 转去执行 BIOS 中的硬件系统检测和初始化程序。

初始化程序将建立 BIOS 所支持的中断向量, 即将 BIOS 提供的中断例程的入口地址登记在中断向量表中。

硬件系统检测和初始化完成后, 调用 int 19h 进行操作系统的引导。

如果设为从软盘启动操作系统, 则 int 19h 将主要完成以下工作。

- (1) 控制 0 号软驱, 读取软盘 0 道 0 面 1 扇区的内容到 0:7c00;
- (2) 将 CS:IP 指向 0:7c00。

软盘的 0 道 0 面 1 扇区中装有操作系统引导程序。int 19h 将其装到 0:7c00 处后, 设置 CPU 从 0:7c00 开始执行此处的引导程序, 操作系统被激活, 控制计算机。

如果在 0 号软驱中没有软盘, 或发生软盘 I/O 错误, 则 int 19h 将主要完成以下工作。

- (1) 读取硬盘 C 的 0 道 0 面 1 扇区的内容到 0:7c00;
- (2) 将 CS:IP 指向 0:7c00。

这次课程设计的任务是编写一个可以自行启动计算机, 不需要在现有操作系统环境中运行的程序。

该程序的功能如下。

- (1) 列出功能选项, 让用户通过键盘进行选择, 界面如下。

1) reset pc	;重新启动计算机
2) start system	;引导现有的操作系统
3) clock	;进入时钟程序
4) set clock	;设置时间

- (2) 用户输入“1”后重新启动计算机(提示: 考虑 ffff:0 单元)。
- (3) 用户输入“2”后引导现有的操作系统(提示: 考虑硬盘 C 的 0 道 0 面 1 扇区)。
- (4) 用户输入“3”后, 执行动态显示当前日期、时间的程序。

显示格式如下: 年/月/日 时:分:秒

进入此项功能后, 一直动态显示当前的时间, 在屏幕上将出现时间按秒变化的效果(提示: 循环读取 CMOS)。

当按下 F1 键后, 改变显示颜色; 按下 Esc 键后, 返回到主选单(提示: 利用键盘中断)。

(5) 用户输入“4”后可更改当前的日期、时间, 更改后返回到主选单(提示: 输入字符串)。

下面给出几点建议:

- (1) 在 DOS 下编写安装程序, 在安装程序中包含任务程序;
- (2) 运行安装程序, 将任务程序写到软盘上;
- (3) 若要任务程序可以在开机后自行执行, 要将它写到软盘的 0 道 0 面 1 扇区上。如果程序长度大于 512 个字节, 则需要用多个扇区存放, 这种情况下, 处于软盘 0 道 0 面 1 扇区中的程序就必须负责将其他扇区中的内容读入内存。

这个程序较为复杂, 它用到了我们所学到的所有技术, 需要进行仔细地分析和耐心地调试。这个程序对于我们的整个学习过程是具有总结性的, 希望读者能够尽力完成。

综合研究

对于已经按本书的要求完成了前面所有学习内容的学习者，如果有兴趣用汇编语言对一些相关问题进行一下深入的研究，可以学习本部分内容。

在这部分内容中，本书将启示我们如何进行独立研究和深度思考。同时使我们：

- (1) 认识到汇编语言对于深入理解其他领域知识的重要性。
- (2) 对前面所学习的汇编语言知识进行融会。
- (3) 对用研究的方法进行学习进行体验。

下面看一下我们要研究的问题。

- (1) 人们用 C 语言编程时都要用到变量。

比如，程序：打印从“a”到“h”的 8 个字符。

```
main()
{
    int n,ch;

    ch='a';

    for(n=0;n<8;n++)
    {
        printf("%c\n",ch+n);
    }
}
```

(2) C 语言规定，用户写的 C 语言程序都要从 **main** 函数开始运行，因此 **main** 函数又称为**主函数**。

- (3) **printf** 函数可以接收的参数数量不定，我们对此司空见惯，比如：

```
main()
{
    printf("hello world!");

    printf("%d + %d = %d",1,2,1+2);
}
```

注意，上面提到了几个关键词：都要用、规定、司空见惯，在看下面的内容的时候再仔细阅读上面的文字，找到这几个关键词。

思考下面几个问题：

- (1) 人们用 C 语言编程时都要用变量，我们就非用不可吗？

- (2) C 语言规定用户写的程序从 `main` 函数开始，我们就非要用 `main` 函数吗？
- (3) `printf` 函数可以接收不定数量的参数司空见惯，我们就不怀疑了吗？

我们把问题再精简一下，使其变得更本质：

- (1) 都在用，我们就非得用吗？
- (2) 规定了，我们就只知道遵守吗？
- (3) 司空见惯，我们就不怀疑了吗？

在许多领域内，我们被这些所谓都在用的，规定了的，司空见惯的，蒙蔽了多久呢？

如果我们被这些蒙蔽，那么，真正蒙蔽我们的是这些，还是我们自己？

现在我们提出要研究的 3 个问题：

- (1) 用 C 语言编程可以不用变量吗？
- (2) 用 C 语言编程可以不用 `main` 函数吗？
- (3) 我们能写一个 `printf` 函数吗？

注意：

(1) 我们使用 TC 2.0 编译器来进行研究，因为这是国内大多数学习者都会使用的 C 语言编译器。

(2) 我们的研究所用的基础知识大都是在前面汇编语言的课程中学习过的。只有极少数知识是我们前面的课程中没有讲解的，但有前面汇编语言的基础，这些知识学习者都可以通过自己学习和研究掌握。

(3) 这部分内容主要是启发学习者进行独立研究和深度思考，一定要注意这一点，相应地调整自己的学习思想。

研究试验 1 搭建一个精简的 C 语言开发环境

我们要对 C 语言进行深入的研究，就必须从准备一个清晰的 C 语言开发环境开始。

我们看一下 TC 2.0 的安装目录，有很多的文件和子目录，子目录下面还有很多程序和文件。这些程序和文件是我们现在都需要的吗？这些程序和文件会对我们研究的问题造成影响吗？问题是：这么多程序和文件混合在一起，如果其中有些程序和文件对我们研究的问题有影响，那么，我们容易判断出影响来自哪些程序和文件吗？

为了研究的过程清晰明了，我们的原则是：

- (1) 我们只运行解决当前问题所要用的，我们已知的程序；
- (2) 所有我们已知的程序在解决我们的问题的运行过程中，需要用到的程序和文件，也都是我们已知的。

这样,我们就可以清晰地知道,哪些程序和文件是用于解决哪些问题的。

这个原则决定了,我们在研究实践中,需要一步步地把我们已知的程序和文件与其他的程序和文件分离开来。

按照上面的原则,请完成以下试验。

(1) 在 d 盘建立一个目录(在 Windows 中称为文件夹)tc2.0。在 DOS 环境中,方法如下:

```
d:\md tc2.0
```

然后将 tc2.0 的所有文件都拷贝在 d:\tc2.0 目录下。

(2) 在 c 盘建立一个目录 minic。在 DOS 环境中,方法如下:

```
c:\md minic
```

这个目录用来存放我们已知的解决问题要用的程序和文件。

注意,一般的产品软件系统,都可以通过设置搜索路径的方式让系统提供的程序可以在相关文件不在相同目录的情况下,也可以找到相关的文件。这个做法可能会导致类似以下的情况:

我们在把一个程序拷贝到一个空的目录后,这个目录下只有这一个程序,然后我们运行它,它可以正确运行,我们就认为这个程序在运行过程中不需要别的文件。但是很可能它在运行过程中使用了别的文件,它不是在当前目录下,而是通过系统设置的搜索路径找到的相关文件。

如上情况的出现会影响我们对一个程序运行过程中使用哪些文件的掌握,而对一些问题产生错误的判断。

我们可以用两种方法解决这个问题:

- ① 不让设置的默认路径指向真的包含相关文件的目录;
- ② 把我们所要研究的系统的所有文件都拷贝到一个不可能是系统设置的搜索路径的目录中。

我们上面用的是第二种方法,将 tc2.0 的所有文件,都拷贝到 d:\tc2.0 目录下,因为这个目录基本上不可能被 tc2.0 设置成为相关文件的搜索路径。这样我们从这个目录拷贝到其他目录(比如 c:\minic)的程序,在运行过程中如果需要使用 tc2.0 中的相关文件,就会出现文件找不到的错误,我们根据提示信息,就可以知道找不到的是哪个文件,也就可能分析出这个文件是干什么用的。

(3) 把我们(国内大多数学习者)都已知的 tc.exe(集成开发环境)拷贝到 c:\minic 下:

```
c:\minic\copy d:\tc2.0\tc.exe
```

(4) 运行 tc.exe:

```
c:\minic\tc
```

用 tc 环境中的菜单项“Options”中的“Directories”选项,对 tc 的工作路径进行设置,将所有路径都清空,即都设置为当前路径。然后用“Save options”选项保存设置。

(5) 在 tc.exe 环境中编辑程序 simple.c, 保存到 c:\minic 下。

程序 simple.c:

```
main()
{
    printf("hello world!\n");
}
```

(6) 用 tc 环境中菜单项“Compile”中的“Compile to OBJ”, 对程序 simple.c 进行编译。看看显示出的提示信息,编译成功了吗?用菜单项“File”中“Quit”(或按 Alt-X)退出 tc 环境,在 c:\minic 下查找 simple.obj。

(7) 用如下的方法运行 tc.exe:

```
c:\minic\tc simple
```

因为 simple.obj 文件已经生成,所以我们用 tc 环境中菜单项“Compile”中的“Link EXE file”,将 simple.obj 连接为 simple.exe。

进行连接后,Message 窗口显示出提示信息:“Unable to open input file 'C0S.OBJ”。

看看显示出的提示信息,连接成功了吗?在 c:\minic 下查找 simple.exe,能找到吗?

(8) 可以看出,tc 进行连接要使用相关文件,但是找不到,所以出错。

为解决这个问题,需要从 d:\tc2.0 目录和子目录下查找到相关文件,将其拷贝到 c:\minic 下。

当然,也可以用 tc 环境中的 Options 菜单项下的相关功能设置相关文件(如.obj 文件)所在的目录的方法,解决找不到.obj 文件和.lib 文件的问题,但是,为了让学习者能够对此时需要哪些文件,以及这些文件在什么目录下,如何找到这些文件等问题有清晰的感性认识,这里我们不用这样的方法。

如果在 TC 2.0 安装目录下和各个子目录下都找不到所需的.obj 文件和.lib 文件,则需要重新安装一套完整的 TC 2.0。

想办法把所有 tc.exe 对程序 simple.obj 进行连接生成.exe 文件必须用到的相关文件都找到,拷贝到 c:\minic。注意,找的是必须用到的。

(9) 用 c:\minic\tc.exe 对 simple.c 进行编译,连接,生成 simple.exe。

研究试验 2 使用寄存器

我们用什么,不用什么,都要看我们要解决什么问题。搞清楚问题,就知道了我们的需要,然后我们就不会拘泥于一种方法,因为可能有很多方法都可以解决我们要解决的问题。

我们为什么必须用变量?因为我们在编程时必须存储数据。那么如果可以用别的方法存储数据,我们就可以不必因此目的而使用变量。

用什么方法来存储数据呢?在学习汇编语言时,我们如何存储数据?我们把数据存储在寄存器和内存空间中。

那么,在 C 语言中如何使用寄存器和内存空间呢?

在本次研究试验中,我们研究一下使用寄存器的问题。

在汇编语言中,要使用寄存器,必须要给出寄存器名,在 C 语言中也是如此。

tc2.0 提供的编译器支持如下寄存器名:

“_AX”, “_BX”, “_CX”, “_DX”, “_SI”, “_DI”, “_SP”,
“_BP”, “_CS”, “_DS”, “_SS”, “_ES”, “_AL”, “_AH”,
“_BL”, “_BH”, “_CL”, “_CH”, “_DL”, “_DH”。

从这些寄存器名称,可以看出它们对应的是哪个寄存器。

用 c:\minic 目录下的 tc.exe 完成以下试验。

(1) 编一个程序 url.c。

```
main ()
{
    _AX=1;

    _BX=1;

    _CX=2;

    _AX=_BX+_CX;

    _AH=_BL+_CL;

    _AL=_BH+_CH;
}
```

把这个程序保存在 c:\minic 下，然后，编译，连接，生成 url.exe。

(2) 用 Debug 加载 url.exe，用 u 命令查看 url.c 编译后的机器码和汇编代码。

思考：main 函数的代码在什么段中？用 Debug 怎样找到 url.exe 中 main 函数的代码？

(3) 用下面的方法打印出 url.exe 被加载运行时，main 函数在代码段中的偏移地址：

```
main()
{
    printf("%x\n",main);
}
```

“%x”指的是按照十六进制格式打印。

思考：为什么这个程序能够打印出 main 函数在代码段中的偏移地址？

(4) 用 Debug 加载 url.exe，根据上面打印出的 main 函数的偏移地址，用 u 命令察看 main 函数的汇编代码。仔细找到 url.c 中每条 C 语句对应的汇编代码。

注意：在这里，对于 main 函数汇编代码开始处的“push bp mov bp,sp”和结尾处的“pop bp”，这里只理解到：这是 C 编译器安排的为函数中可能使用到 bp 寄存器而设置的，就可以了。

(5) 通过 main 函数后面有 ret 指令，我们可以设想：C 语言将函数实现为汇编语言中的子程序。研究下面程序的汇编代码，验证我们的设想。

程序 ur2.c:

```
void f(void);

main()
{
    _AX=1;  _BX=1;  _CX=2;

    f();
}

void f(void)
{
    _AX=_BX+_CX;
}
```

研究试验 3 使用内存空间

寄存器只有十几个，但是内存空间可以很大。那么在 C 语言里如何使用内存空间呢？其实，寄存器也好，内存空间也好，都是存储空间，对于存储空间来说，要使用它们

一般都需要给出两个信息：①指明是存储空间所在、是哪个的信息；②指明存储空间有多大的类型信息。

对于寄存器来说，就需要给出寄存器的名称，寄存器的名称中也包含了它们的类型信息。

对于内存空间来说，就需要给出地址(准确地说，是内存空间首地址)和空间存储数据的类型。

我们知道，在 C 语言里，用指针型数据来表示内存空间的地址和空间存储数据的类型。比如要向偏移地址为 2000h、存储一个字节的内存空间写入一个字符'a'，我们用如下的方法：

```
*(char *)0x2000='a';
```

第一个“*”表示要访问的是一个内存空间；

“0x2000”是一个数值(0x 表示十六进制)，“(char *)”里面的“*”指明了这个数值表示一个内存空间的地址，“char”指明了这个地址是存储 char 型数据的内存空间的地址。

当然也可以用给出段地址和偏移地址的方法访问内存空间，比如我们要向地址为 2000:0、存储一个字节的内存空间写入字符'a'，如下：

```
*(char far *)0x20000000='a';
```

“far”指明内存空间的地址是段地址和偏移地址，“0x20000000”中的“0x2000”给出了段地址，“0000”给出了偏移地址。

不过这样直接用地址访问内存空间的方式是不安全的，因为，如果这些空间并不是分配给我们的程序使用的空间，这样做就可能改变了别的程序的代码或数据，引起错误。

我们可以按照上面的例子，举一反三，对以前学过的 C 语言相关知识进行深入的理解。

用 c:\minic 目录下的 tc.exe，完成下面的试验。

(1) 编一个程序 um1.c:

```
main()
{
    *(char *)0x2000='a';
    *(int *)0x2000=0xf;
    *(char far *)0x20001000='a';

    _AX=0x2000;
    *(char *)_AX='b';

    _BX=0x1000;
```

```
* (char *) (_BX+_BX)='a';

* (char far *) (0x20001000+_BX)=* (char *) _AX;
}
```

把 uml.c 保存在 c:\minic 下，编译，连接生成 uml.exe。然后用 Debug 加载 uml.exe，对 main 函数的汇编代码进行分析，找到每条 C 语句对应的汇编代码；对 main 函数进行单步跟踪，察看相关内存单元的内容。

(2) 编一个程序，用一条 C 语句实现在屏幕的中间显示一个绿色的字符“a”。

(3) 分析下面程序中所有函数的汇编代码，思考相关的问题。

```
int a1,a2,a3;

void f(void);

main()
{
    int b1,b2,b3;

    a1=0xa1;a2=0xa2;a3=0xa3;

    b1=0xb1;b2=0xb2;b3=0xb3;
}

void f(void)
{
    int c1,c2,c3;

    a1=0x0fa1;a2=0x0fa2;a3=0x0fa3;

    c1=0xc1;c2=0xc2;c3=0xc3;
}
```

问题：C 语言将全局变量存放在哪里？将局部变量存放在哪里？每个函数开头的“push bp mov bp sp”有何含义？

(4) 分析下面程序的汇编代码，思考相关的问题。

```
int f(void);

int a,b,ab;

main()
{
    int c;

    c=f();
}
```

```
int f(void)
{
    ab=a+b;

    return ab;
}
```

问题：C 语言将函数的返回值存放在哪里？

(5) 下面的程序向安全的内存空间写入从“a”到“h”的 8 个字符，理解程序的含义，深入理解相关的知识。(注意：请自己学习、研究 malloc 函数的用法)

```
#define Buffer ((char *)*(int far *)0x200)
main()
{
    Buffer=(char *)malloc(20);
    Buffer[10]=0;
    while(Buffer[10]!=8)
    {
        Buffer[Buffer[10]]='a'+Buffer[10];
        Buffer[10]++;
    }
    free(Buffer);
}
```

研究试验 4 不用 main 函数编程

在本研究试验中，我们看看如何不用 main 函数，编写可以正确运行的程序。我们用一个简单的程序来进行研究。

程序 f.c:

```
f()
{
    *(char far *) (0xb8000000+160*10+80)='a';

    *(char far *) (0xb8000000+160*10+81)=2;
}
```

下面，我们研究如何用 tc.exe 对 f.c 进行编译，连接，生成可正确运行的 f.exe。

我们用 c:\minic 下的 tc.exe 完成以下试验。

(1) 把程序 f.c 保存在 c:\minic 下，对其进行编译，连接。思考相关的问题。

问题:

- ① 编译和连接哪个环节会出问题?
- ② 显示出的错误信息是什么?
- ③ 这个错误信息可能与哪个文件相关?

(2) 用学习汇编语言时使用的 `link.exe` 对 `tc.exe` 生成的 `f.obj` 文件进行连接, 生成 `f.exe`。用 `Debug` 加载 `f.exe`, 察看整个程序的汇编代码。思考相关的问题。

问题:

- ① `f.exe` 的程序代码总共有多少字节?
- ② `f.exe` 的程序能正确返回吗?
- ③ `f` 函数的偏移地址是多少?

(3) 写一个程序 `m.c`。

```
main()
{
    *(char far *) (0xb8000000+160*10+80)='a';

    *(char far *) (0xb8000000+160*10+81)=2;
}
```

用 `tc.exe` 对 `m.c` 进行编译, 连接, 生成 `m.exe`, 用 `Debug` 察看 `m.exe` 整个程序的汇编代码。思考相关的问题。

问题:

- ① `m.exe` 的程序代码总共有多少字节?
- ② `m.exe` 能正确返回吗?
- ③ `m.exe` 程序中的 `main` 函数和 `f.exe` 中的 `f` 函数的汇编代码有何不同?

(4) 用 `Debug` 对 `m.exe` 进行跟踪: ①找到对 `main` 函数进行调用的指令的地址; ②找到整个程序返回的指令。注意: 使用 `g` 命令和 `p` 命令。

(5) 思考如下几个问题:

- ① 对 `main` 函数调用的指令和程序返回的指令是哪里来的?
- ② 没有 `main` 函数时, 出现的错误信息里有和“`c0s`”相关的信息; 而前面在搭建开发环境时, 没有 `c0s.obj` 文件 `tc.exe` 就无法对程序进行连接。是不是 `tc.exe` 把 `c0s.obj` 和用户程序的 `.obj` 文件一起进行连接生成 `.exe` 文件?
- ③ 对用户程序的 `main` 函数进行调用的指令和程序返回的指令是否就来自 `c0s.obj` 文件?
- ④ 我们如何看到 `c0s.obj` 文件中的程序代码呢?
- ⑤ `c0s.obj` 文件里有我们设想的代码吗?

(6) 用 `link.exe` 对 `c:\minic` 目录下的 `c0s.obj` 进行连接, 生成 `c0s.exe`。

用 `Debug` 分别察看 `c0s.exe` 和 `m.exe` 的汇编代码。注意: 从头开始察看, 两个文件中的程序代码有何相同之处?

(7) 用 `Debug` 找到 `m.exe` 中调用 `main` 函数的 `call` 指令的偏移地址, 从这个偏移地址开始向后察看 10 条指令; 然后用 `Debug` 加载 `c0s.exe`, 从相同的偏移地址开始向后察看 10 条指令。对两处的指令进行对比。

(8) 从上我们可以看出, `tc.exe` 把 `c0s.obj` 和用户 `.obj` 文件一同进行连接, 生成 `.exe` 文件。按照这个方法生成的 `.exe` 文件中的程序的运行过程如下。

① `c0s.obj` 里的程序先运行, 进行相关的初始化, 比如, 申请资源、设置 `DS`、`SS` 等寄存器;

② `c0s.obj` 里的程序调用 `main` 函数, 从此用户程序开始运行;

③ 用户程序从 `main` 函数返回到 `c0s.obj` 的程序中;

④ `c0s.obj` 的程序接着运行, 进行相关的资源释放, 环境恢复等工作;

⑤ `c0s.obj` 的程序调用 `DOS` 的 `int 21h` 例程的 `4ch` 号功能, 程序返回。

看来, `C` 程序必须从 `main` 函数开始, 是 `C` 语言的规定, 这个规定不是在编译时保证的(`tc.exe` 对 `f.c` 的编译是可以通过的), 也不是连接的时候保证的(虽然, `tc.exe` 文件对 `f.obj` 文件不能连接成 `f.exe`, 但 `link.exe` 却可以), 而是用如下的机制保证的。

首先, `C` 开发系统提供了用户写的应用程序正确运行所必须的初始化和程序返回等相关程序, 这些程序存放在相关的 `.obj` 文件(比如, `c0s.obj`)中。

其次, 需要将这些文件 and 用户 `.obj` 文件一起进行连接, 才能生成可正确运行的 `.exe` 文件。

第三, 连接在用户 `.obj` 文件前面的由 `C` 语言开发系统提供的 `.obj` 文件里的程序要对 `main` 函数进行调用。

基于这种机制, 我们只要改写 `c0s.obj`, 让它调用其他函数, 编程时就可以不写 `main` 函数了。

下面, 我们用汇编语言编一个程序 `c0s.asm`, 然后把它编译为 `c0s.obj`, 替代 `c:\minic` 目录下的 `c0s.obj`。

程序 `c0s.asm`:

```
assume cs:code

data segment

    db 128 dup (0)
```

```
data ends

code segment

    start: mov ax,data
           mov ds,ax
           mov ss,ax
           mov sp,128

           call s

           mov ax,4c00h
           int 21h

s:

code ends

end start
```

用 `masm.exe` 对 `c0s.asm` 进行编译,生成 `c0s.obj`,把这个 `c0s.obj` 复制到 `c:\minic` 目录下覆盖由 `tc2.0` 提供的 `c0s.obj`。

(9) 在 `c:\minic` 目录下,用 `tc.exe` 将 `f.c` 重新进行编译,连接,生成 `f.exe`。这次能通过连接吗? `f.exe` 可以正确运行吗?用 `Debug` 察看 `f.exe` 的汇编代码。

(10)在新的 `c0s.obj` 的基础上,写一个新的 `f.c`,向安全的内存空间写入从“a”到“h”的8个字符。分析、理解 `f.c`。

程序 `f.c`:

```
#define Buffer ((char *)*(int far *)0x200)

f()
{
    Buffer=0;

    Buffer[10]=0;

    while(Buffer[10]!=8)
    {
        Buffer[Buffer[10]]='a'+Buffer[10];

        Buffer[10]++;
    }
}
```

注意,完成上面的相关试验后,把 `c:\minic` 目录下的 `c0s.obj` 文件恢复为 `tc2.0` 提供的 `c0s.obj` 文件。

研究试验 5 函数如何接收不定数量的参数

用 c:\minic 下的 tc.exe 完成下面的试验。

(1) 写一个程序 a.c:

```
void showchar(char a,int b);

main()
{
    showchar('a',2);
}

void showchar(char a,int b)
{
    *(char far *) (0xb8000000+160*10+80)=a;

    *(char far *) (0xb8000000+160*10+81)=b;
}
```

用 tc.exe 对 a.c 进行编译, 连接, 生成 a.exe。用 Debug 加载 a.exe, 对函数的汇编代码进行分析。解答这两个问题: main 函数是如何给 showchar 传递参数的? showchar 是如何接收参数的?

(2) 写一个程序 b.c:

```
void showchar(int,int,...);

main()
{
    showchar(8,2,'a','b','c','d','e','f','g','h');
}

void showchar(int n,int color,...)
{
    int a;

    for(a=0;a!=n;a++)
    {
        *(char far *) (0xb8000000+160*10+80+a)=*(int *) (_BP+8+a+a);

        *(char far *) (0xb8000000+160*10+81+a+a)=color;
    }
}
```

分析程序 b.c, 深入理解相关的知识。

思考: showchar 函数是如何知道要显示多少个字符的? printf 函数是如何知道有多少个参数的?

(3) 实现一个简单的 printf 函数, 只需要支持 “%c、%d” 即可。

附 注

附注 1 Intel 系列微处理器的 3 种工作模式

微机中常用的 Intel 系列微处理器的主要发展过程是：8080，8086/8088，80186，80286，80386，80486，Pentium，Pentium II，Pentium III，Pentium 4。

8086/8088 是一个重要的阶段，8086 和 8088 是略有区别的两个功能相同的 CPU。8088 被 IBM 用在了它所生产的第一台微机上，该微机的结构事实上成为以后微机的基本结构。

80386 是第二个重要的型号，随着微机应用及性能的发展，在微机上构造可靠的多任务操作系统的问题日益突出。人们希望(或许是一种潜在的希望，一旦被挖掘出来，便形成了一个最基本的需求)自己的 PC 机能够稳定地同时运行多个程序，同时处理多项工作；或将 PC 机用作主机服务器，运行 UNIX 那样的多用户系统。

8086/8088 不具备实现一个完善的多任务操作系统的功能。为此 Intel 开发了 80286，80286 具备了对多任务系统的支持。但对 8086/8088 的兼容却做得不好。这妨碍了用户对原 8086 机上的程序的使用。IBM 最早基于 80286 开发了多任务系统 OS/2，结果犯了一个战略错误。

随后 Intel 又开发了 80386 微处理器，这是一个划时代的产品。它可以在以下 3 个模式下工作。

- (1) 实模式：工作方式相当于一个 8086。
- (2) 保护模式：提供支持多任务环境的工作方式，建立保护机制(这与 VAX 等小型机类似)。
- (3) 虚拟 8086 模式：可从保护模式切换至其中的一种 8086 工作方式。这种方式的提供使用户可以方便地在保护模式下运行一个或多个原 8086 程序。

以后的各代微处理器都提供了上述 3 种工作模式。

你也许会说：“喂，先生，你说的太抽象了，这 3 种模式我如何感知？”

其实 CPU 的这 3 种模式只要用过 PC 机的人都经历过。任何一台使用 Intel 系列 CPU 的 PC 机只要一开机，CPU 就工作在实模式下。如果你的机器装的是 DOS，那么在 DOS 加载后 CPU 仍以实模式工作。如果你的机器装的是 Windows，那么 Windows 加载后，将由 Windows 将 CPU 切换到保护模式下工作，因为 Windows 是多任务系统，它必须在保护模式下运行。如果你在 Windows 中运行一个 DOS 下的程序，那么 Windows 将 CPU 切换

到虚拟 8086 模式下运行该程序。或者是这样,你点击开始菜单在程序项中进入 MS-DOS 方式,这时,Windows 也将 CPU 切换到虚拟 8086 模式下运行。

可以从保护模式直接进入能运行原 8086 程序的虚拟 8086 模式是很有意义的,这为用户提供了一种机制,可以在现有的多任务系统中方便地运行原 8086 系统中的程序。这一点,在 Windows 中我们都可以体会到,你在 Windows 中想运行一个原 DOS 中的程序,只用鼠标点击一下它的图标即可。80286CPU 的缺陷在于,它只提供了实模式和保护模式,但没有提供虚拟 8086 模式。这使基于 80286 构造的多任务系统,不能方便地运行原 8086 系统中的程序。如果运行原 8086 系统中的程序,需要重新启动计算机,使 CPU 工作在实模式下才行。这意味着什么?意味着将给用户造成很大的不方便。假设你使用的是基于 80286 构造的 Windows 系统,就会发生这样的情况:你正在用 Word 写一篇论文,其中用到了一些从前的数据,你必须运行原 DOS 下的 DBASE 系统来看一下这些数据。这时你只能停下现有的工作,重新启动计算机,进入实模式工作。你看完了数据,继续写论文,可过了一会儿,你发现又有些数据需要参考,于是你又得停下现有的工作,重新启动计算机……

幸运的是,我们用的 Windows 是基于 80386 的,我们可以以这样轻松的方式工作,开两个窗口,一个是工作于保护模式的 Word,一个是工作于虚拟 8086 模式的 DBASE,我们可以方便地在两个窗口中切换,只要用鼠标点一下就行。

前面讲过,我们在 8086PC 机的基础上学习汇编语言。但现在知道,我们实际的编程环境是当前 CPU 的实模式。当然,有些程序也可以在虚拟 8086 模式下运行。

如果你仔细阅读了上面的内容,或已具备相关的知识,你会发现,从 80386 到当前的 CPU,提供 8086 实模式的目的是为了兼容。现今 CPU 的真正有效力的工作模式是支持多任务操作系统的保护模式。这也许会引发你的一个疑问:“为什么我们不在保护模式下学习汇编语言?”

类似的问题很多,我们都希望学习更新的东西,但学习的过程是客观的。任何合理的学习过程(尽可能排除走弯路、盲目探索、不成系统)都是一个循序渐进的过程。我们必须先通过一个易于全面把握的事物,来学习和探索一般的规律和方法。信息技术是一个发展非常快、日新月异的技术,新的东西不断出现,使人在学习的时候往往无所适从。在你的身边不断有这样的故事出现:COOL 先生用了 3 天(或更短)的时间就学会了某某语言,并开始用它编写软件。在这个故事的感召下,一个初学者也去尝试,但完全是另外一种结果。COOL 先生的快速学习只是露出水面的冰山一角,深藏水下的是他的较为系统的相关基础知识和相关的技术。在开始的时候学习保护模式下的编程,是不现实的,保护模式下所涉及的东西对初学者来说太复杂。你必须知道很多知识后,才能开始编写第一个小程序。相比之下 8086 就合适得多。

附注 2 补 码

以 8 位的数据为例，对于无符号数来说是从 00000000b~11111111b 到 0~255 一一对应的。那么我们如何对有符号数进行编码呢？即我们如何用 8 位数据表示有符号数呢？

既然表示的数有符号，则必须要能够区分正、负。

首先，我们可以考虑用 8 位数据的最高位来表示符号，1 表示负，0 表示正，而用其他位表示数值。如下：

```
00000000b: 0
00000001b: 1
00000010b: 2
01111111b: 127
10000000b: ?
10000001b: -1
10000010b: -2
11111111b: -127
```

可见，用上面的表示方法，8 位数据可以表示 -127~127 的 254 个有符号数。从这里我们看出一些问题，8 位数据可以表示 255 种不同的信息，也就是说应该可以表示 255 个有符号数，可用上面的方法，只能表示 254 个有符号数。注意，用上面的方法，00000000b 和 10000000b 都表示 0，一个是 0，一个是 -0，当然不可能有 -0。可以看出，这种表示有符号数的方法是有点问题的，它并不能正确地表示有符号数。

我们再考虑用反码来表示，这种思想是，我们先确定用 00000000b~01111111b 表示 0~127，然后再用它们按位取反后的数据表示负数。如下：

```
00000000b: 0          11111111b: ?
00000001b: 1          11111110b: -1
00000010b: 2          11111110b: -2
01111111b: 127       10000000b: -127
```

可以看出，用反码表示有符号数存在同样的问题，0 出现重码。

为了解决这种问题，采用一种称为补码的编码方法。这种思想是：先确定用 00000000b~01111111b 表示 0~127，然后再用它们按位取反加 1 后的数据表示负数。如下：

```
00000000b: 0          11111111b+1=00000000b: 0
00000001b: 1          11111110b+1=11111111b: -1
00000010b: 2          11111110b+1=11111110b: -2
01111111b: 127       10000000b+1=10000001b: -127
```

观察上面的数据，我们可以发现，在补码方案中：

(1) 最高位为 1, 表示负数;

(2) 正数的补码取反加 1 后, 为其对应的负数的补码; 负数的补码取反加 1 后, 为其绝对值。比如:

1 的补码为: 00000001b, 取反加 1 后为: 11111111b, 表示-1;

-1 的补码为: 11111111b, 取反加 1 后为: 00000001b, 其绝对值为 1。

我们从一个负数的补码不太容易看出它所表示的数据, 比如: 11010101b 表示的数据是多少?

但是我们利用补码的特性, 将 11010101b 取反加 1 后为: 00101011b。可知 11010101b 表示的负数的绝对值为: 2BH, 则 11010101b 表示的负数为-2BH。

那么-20 的补码是多少呢?

用补码的特性, -20 的绝对值是 20, 00010100b, 将其取反加 1 后为: 11101100b。可知-20H 的补码为: 11101100b。

那么 10000000b 表示多少呢?

10000000b 取反加 1 后为: 10000000b, 其大小为 128, 所以 10000000b 表示-128。

8 位补码所表示的数的范围: -128~127。

补码为有符号数的运算提供了方便, 运算后的结果依旧满足补码规则。

比如:

计算	补码表示
10	00001010b
<u>+ (-20)</u>	11101100b
-10	11110110b

附注 3 汇编编译器(masm.exe)对 jmp 的相关处理

1. 向前转移

```
s:  :  
    :  
    :  
    jmp s (jmp short s, jmp near ptr s, jmp far ptr s)
```

编译器中有一个地址计数器(AC), 编译器在编译程序过程中, 每读到一个字节 AC 就加 1。当编译器遇到一些伪操作的时候, 也会根据具体情况使 AC 增加, 如 db、dw 等。

在向前转移时, 编译器可以在读到标号 s 后记下 AC 的值 as, 在读到 jmp ... s 后记下

AC 的值 `aj`。编译器可以用 `as-aj` 算出位移量 `disp`。

此时，编译器作如下处理。

(1) 如果 `disp` $\in [-128, 127]$ ，则不管汇编指令格式是：

```
jmp s
jmp short s
jmp near ptr s
jmp far ptr s
```

中的哪一种，都将它转变为 `jmp short s` 所对应的机器码。

`jmp short s` 所对应的机器码格式为：EB `disp`(占两个字节)

编译，连接以下程序，用 Debug 进行反汇编查看。

```
assume cs:code
code segment
s:    jmp s
      jmp short s
      jmp near ptr s
      jmp far ptr s
code ends
end s
```

(2) 如果 `disp` $\in [-32768, 32767]$ ，则：

对于 `jmp short s` 将产生编译错误；

对于 `jmp s`、`jmp near ptr s` 将产生 `jmp near ptr s` 所对应的机器码，`jmp near ptr s` 所对应的机器码格式为：E9 `disp`(占 3 个字节)；

对于 `jmp far ptr s` 将产生相应的编码，`jmp far ptr s` 所对应的机器码格式为：EA 偏移地址 段地址(占 5 个字节)。

编译，连接以下程序。

```
assume cs:code
code segment
s:    db 100 dup (0b8h, 0, 0)
      jmp short s
      jmp s
      jmp near ptr s
      jmp far ptr s
code ends
end s
```

编译中将产生错误，错误是由 `jmp short s` 引起的，去掉 `jmp short s` 后再编译就可以通过。用 Debug 进行反汇编查看。

2. 向后转移

```

    jmp s (jmp short s, jmp near ptr s, jmp far ptr s)
    :
    :
s:  :
```

在这种情况下, 编译器先读到 `jmp ... s` 指令。由于它还没有读到标号 `s`, 所以编译器此时还不能确定标号 `s` 处的 AC 值。也就是说, 编译器不能确定位移量 `disp` 的大小。

此时, 编译器将 `jmp ... s` 指令都当作 `jmp short s` 来读取, 记下 `jmp ... s` 指令的位置和 AC 的值 `aj`, 并作如下处理。

对于 `jmp short s`, 编译器生成 EB 和 1 个 `nop` 指令(相当于预留 1 个字节的空间, 存放 8 位 `disp`);

对于 `jmp s` 和 `jmp near ptr s`, 编译器生成 EB 和两个 `nop` 指令(相当于预留两个字节的空, 存放 16 位 `disp`);

对于 `jmp far ptr s`, 编译器生成 EB 和 4 个 `nop` 指令(相当于预留 4 个字节的空间, 存放段地址和偏移地址)。

做完以上处理后, 编译器继续工作, 当向后读到标号 `s` 时, 记下 AC 的值 `as`, 并计算出转移的位移量: $disp = as - aj$ 。

此时, 编译器作如下处理。

(1) 当 $disp \in [-128, 127]$ 时, 不管指令格式是:

```

jmp short s
jmp s
jmp near ptr s
jmp far ptr s
```

中的哪一种, 都在前面记下的 `jmp ... s` 指令位置处添上 `jmp short s` 对应的机器码(格式为: EB disp)。

注意, 此时, 对于 `jmp s` 和 `jmp near ptr s` 格式, 在机器码 EB disp 后还有 1 条 `nop` 指令; 对于 `jmp far ptr s` 格式, 在机器码 EB disp 后还有 3 条 `nop` 指令。

编译, 连接以下程序, 用 Debug 进行反汇编查看。

```

assume cs:code
code segment
begin: jmp short s
        jmp s
        jmp near ptr s
        jmp far ptr s
s:      mov ax, 0
code ends
end begin
```

(2) 当 $\text{disp} \in [-32768, 32767]$ 时, 则:

对于 `jmp short s`, 将产生编译错误;

对于 `jmp s`、`jmp near ptr s`, 在前面记下的 `jmp ... s` 指令位置处添上 `jmp near ptr s` 所对应的机器码(格式为: `E9 disp`);

对于 `jmp far ptr s`, 在前面记下的 `jmp ... s` 指令位置处添上相应的代码。

编译, 连接以下程序。

```
assume cs:code
code segment
begin: jmp short s
      jmp s
      jmp near ptr s
      jmp far ptr s
      db 100 dup (0b8h, 0, 0)
s:     mov ax, 2
code ends
end begin
```

在编译中将产生错误, 错误是由 `jmp short s` 引起的, 去掉 `jmp short s` 后再编译就可通过。用 `Debug` 进行反汇编查看。

附注 4 用栈传递参数

这种技术和高级语言编译器的工作原理密切相关。我们下面结合 C 语言的函数调用, 看一下用栈传递参数的思想。

用栈传递参数的原理十分简单, 就是由调用者将需要传递给子程序的参数压入栈中, 子程序从栈中取得参数。我们看下面的例子。

```
;说明: 计算 (a-b)^3, a、b 为字型数据
;参数: 进入子程序时, 栈顶存放 IP, 后面依次存放 a、b
;结果: (dx:ax) = (a-b)^3
```

```
difcube: push bp
         mov bp, sp
         mov ax, [bp+4]    ;将栈中 a 的值送入 ax 中
         sub ax, [bp+6]    ;减栈中 b 的值
         mov bp, ax
         mul bp
         mul bp
         pop bp
         ret 4
```

指令 `ret n` 的含义用汇编语法描述为:

```
pop ip
add sp, n
```

因为用栈传递参数,所以调用者在调用程序的时候要向栈中压入参数,子程序在返回的时候可以用 `ret n` 指令将栈顶指针修改为调用前的值。调用上面的子程序之前,需要压入两个参数,所以用 `ret 4` 返回。

我们看一下如何调用上面的程序,设 $a=3$ 、 $b=1$,下面的程序段计算 $(a-b)^3$:

```
mov ax,1
push ax
mov ax,3
push ax           ;注意参数压栈的顺序
call difcube
```

程序的执行过程中栈的变化如下。

(1) 假设栈的初始情况如下:

```
1000:0000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
                                                    ↑
                                                    ss:sp
```

(2) 执行以下指令:

```
mov ax,1
push ax
mov ax,3
push ax
```

栈的情况变为:

```

                                a      b
1000:0000  00 00 00 00 00 00 00 00 00 00 00 00 03 00 01 00
                                                    ↑
                                                    ss:sp
```

(3) 执行指令 `call difcube`, 栈的情况变为:

```

                                IP      a      b
1000:0000  00 00 00 00 00 00 00 00 00 XX XX 03 00 01 00
                                                    ↑
                                                    ss:sp
```

(4) 执行指令 `push bp`, 栈的情况变为:

```

                                bp      IP      a      b
1000:0000  00 00 00 00 00 00 00 XX XX XX XX 03 00 01 00
                                                    ↑
                                                    ss:sp
```

(5) 执行指令 `mov bp,sp`, `ss:bp` 指向 1000:8

(6) 执行以下指令:

```
mov ax,[bp+4]           ;将栈中 a 的值送入 ax 中
```

```

sub ax, [bp+6]      ;减栈中b的值
mov bp, ax
mul bp
mul bp

```

(7) 执行指令 `pop bp`, 栈的情况变为:

```

                                IP      a      b
1000:0000  00 00 00 00 00 00 00 00 XX XX XX XX 03 00 01 00
                                ↑
                                ss:sp

```

(8) 执行指令 `ret 4`, 栈的情况变为:

```

1000:0000  00 00 00 00 00 00 00 00 XX XX XX XX 03 00 01 00
                                                                ↑
                                                                ss:sp

```

下面, 我们通过一个 C 语言程序编译后的汇编语言程序, 看一下栈在参数传递中的应用。要注意的是, 在 C 语言中, 局部变量也在栈中存储。

C 程序

```

void add(int, int, int);

main()
{
    int a=1;
    int b=2;
    int c=0;
    add(a, b, c);
    c++;
}

void add(int a, int b, int c)
{
    c=a+b;
}

```

编译后的汇编程序

```

mov bp, sp
sub sp, 6
mov word ptr [bp-6], 0001    ;int a
mov word ptr [bp-4], 0002    ;int b
mov word ptr [bp-2], 0000    ;int c
push [bp-2]
push [bp-4]
push [bp-6]

```

```

call ADDR
add sp,6
inc word ptr [bp-2]

ADDR:  push bp
        mov bp,sp
        mov ax,[bp+4]
        add ax,[bp+6]
        mov [bp+8],ax
        mov sp,bp
        pop bp
        ret

```

附注 5 公式证明

问题：计算 X/n ($X < 65536 * 65536$, $n \neq 0$)

在计算过程中要保证不会出现除法溢出。

分析：

(1) 在计算过程中不会出现除法溢出，也就是说，在计算过程中除法运算的商要小于 65536。

设： $X/n = (H * 65536 + L)/n = (H/n) * 65536 + (L/n)$

$H = \text{int}(X/65536)$

$L = \text{rem}(X/65536)$

因为 $H < 65536$ ， $L < 65536$ 所以

将计算 X/n 转化为计算： $(H/n) * 65536 + (L/n)$ 可以消除溢出的可能性。

(2) 将计算 X/n 分解为计算：

$(H/n) * 65536 + (L/n)$ ； $H = \text{int}(X/65536)$ ； $L = \text{rem}(X/65536)$

DIV 指令只能得出余数和商，而我们只保留商。余数必然小于除数，一次正确的除法运算只能丢掉一个余数。

我们虽然在具体处理时进行了两次除法运算 H/n 和 L/n ；但这实质上是一次除法运算 X/n 问题的分解。也就是说，为保证最终结果的正确，两次除法运算只能丢掉一个余数。

在这个问题中， H/n 产生的余数是绝对不能丢的，因为丢掉了它(设为 r)就相当于丢掉了 $r * 65536$ (这是一个相当大的误差)。

那么如何处理 H/n 产生的余数呢？

我们知道： $H = \text{int}(H/n) * n + \text{rem}(H/n)$

所以有：

$$\begin{aligned} & (H/n)*65536+(L/n) \\ &= [\text{int}(H/n)*n+\text{rem}(H/n)]/n*65536+(L/n) \\ &= \text{int}(H/n)*65536+\text{rem}(H/n)*65536/n+L/n \\ &= \text{int}(H/n)*65536+[\text{rem}(H/n)*65536+L]/n \end{aligned}$$

现在将计算 X/n 转化为计算：

$$\begin{aligned} & \text{int}(H/n)*65536+[\text{rem}(H/n)*65536+L]/n \\ & H=\text{int}(X/65536); \quad L=\text{rem}(X/65536) \end{aligned}$$

在这里要进行两次除法运算：

第一次： H/n

第二次： $[\text{rem}(H/n)*65536+L]/n$

我们知道第一次不会产生除法溢出。

现证明第二次：

$$\textcircled{1} \quad L \leq 65535$$

$$\textcircled{2} \quad \text{rem}(H/n) \leq n-1$$

由②有：

$$\textcircled{3} \quad \text{rem}(H/n)*65536 \leq (n-1)*65536$$

由①，③有：

$$\textcircled{4} \quad \text{rem}(H/n)*65536+L \leq (n-1)*65536+65535$$

由④有：

$$\textcircled{5} \quad [\text{rem}(H/n)*65536+L]/n \leq [(n-1)*65536+65535]/n$$

由⑤有：

$$\textcircled{6} \quad [\text{rem}(H/n)*65536+L]/n \leq 65536-(1/n)$$

所以 $[\text{rem}(H/n)*65536+L]/n$ 不会产生除法溢出。

则： $X/n=\text{int}(H/n)*65536+[\text{rem}(H/n)*65536+L]/n$

$$H=\text{int}(X/65536); \quad L=\text{rem}(X/65536)$$