第 9 章　转移指令的原理

可以修改 IP，或同时修改 CS 和 IP 的指令统称为转移指令。概括地讲，转移指令就是可以控制 CPU 执行内存中某处代码的指令。

8086CPU 的转移行为有以下几类。

- 只修改 IP 时，称为段内转移，比如：jmp ax。
- 同时修改 CS 和 IP 时，称为段间转移，比如：jmp 1000:0。

由于转移指令对 IP 的修改范围不同，段内转移又分为：短转移和近转移。

- 短转移 IP 的修改范围为-128~127。
- 近转移 IP 的修改范围为-32768~32767。

8086CPU 的转移指令分为以下几类。

- 无条件转移指令(如：jmp)
- 条件转移指令
- 循环指令(如：loop)
- 过程
- 中断

这些转移指令转移的前提条件可能不同，但转移的基本原理是相同的。我们在这一章主要通过深入学习无条件转移指令 jmp 来理解 CPU 执行转移指令的基本原理。

9.1　操作符 offset

操作符 offset 在汇编语言中是由编译器处理的符号，它的功能是取得标号的偏移地址。比如下面的程序：

```
assume cs:codesg
codesg segment

  start:mov ax,offset start        ;相当于 mov ax,0
      s: mov ax,offset s           ;相当于 mov ax,3

codesg ends
end start
```

在上面的程序中，offset 操作符取得了标号 start 和 s 的偏移地址 0 和 3，所以指令：

mov ax,offset start 相当于指令 mov ax,0，因为 start 是代码段中的标号，它所标记的指

令是代码段中的第一条指令，偏移地址为 0；

mov ax,offset s 相当于指令 mov ax,3，因为 s 是代码段中的标号，它所标记的指令是代码段中的第二条指令，第一条指令长度为 3 个字节，则 s 的偏移地址为 3。

问题 9.1

有如下程序段，添写两条指令，使该程序在运行中将 s 处的一条指令复制到 s0 处。

```
assume cs:codesg
codesg segment
    s:   mov ax,bx            ;mov ax,bx 的机器码占两个字节
         mov si, offset s
         mov di, offset s0
         ____________
         ____________
   s0:   nop                  ;nop 的机器码占一个字节
         nop
codesg ends
end s
```

思考后看分析。

分析：

(1) s 和 s0 处的指令所在的内存单元的地址是多少？cs:offset s 和 cs:offset s0。

(2) 将 s 处的指令复制到 s0 处，就是将 cs:offset s 处的数据复制到 cs:offset s0 处。

(3) 段地址已知在 cs 中，偏移地址 offset s 和 offset s0 已经送入 si 和 di 中。

(4) 要复制的数据有多长？mov ax,bx 指令的长度为两个字节，即 1 个字。

程序如下。

```
assume cs:codesg
codesg segment
    s:   mov ax,bx            ;mov ax,bx 的机器码占两个字节
         mov si, offset s
         mov di, offset s0
         mov ax,cs:[si]
         mov cs:[di],ax
   s0:   nop                  ;nop 的机器码占一个字节
         nop
codesg ends

end s
```

9.2 jmp 指令

jmp 为无条件转移指令，可以只修改 IP，也可以同时修改 CS 和 IP。

jmp 指令要给出两种信息：

(1) 转移的目的地址
(2) 转移的距离(段间转移、段内短转移、段内近转移)

不同的给出目的地址的方法，和不同的转移位置，对应有不同格式的 jmp 指令。下面的几节内容中，我们以给出目的地址的不同方法为主线，讲解 jmp 指令的主要应用格式和 CPU 执行转移指令的基本原理。

9.3　依据位移进行转移的 jmp 指令

jmp short 标号(转到标号处执行指令)

这种格式的 jmp 指令实现的是段内短转移，它对 IP 的修改范围为-128~127，也就是说，它向前转移时可以最多越过 128 个字节，向后转移可以最多越过 127 个字节。jmp 指令中的“short”符号，说明指令进行的是短转移。jmp 指令中的“标号”是代码段中的标号，指明了指令要转移的目的地，转移指令结束后，CS:IP 应该指向标号处的指令。

比如：

程序 9.1

```
assume cs:codesg

codesg segment

  start:mov ax,0
        jmp short s
        add ax,1
      s: inc ax

codesg ends

end start
```

上面的程序执行后，ax 中的值为 1，因为执行 jmp short s 后，越过了 add ax,1，IP 指向了标号 s 处的 inc ax。也就是说，程序只进行了一次 ax 加 1 操作。

汇编指令 jmp short s 对应的机器指令应该是什么样的呢？我们先看一下别的汇编指令和其相对应的机器指令。

```
汇编指令                  机器指令
mov ax,0123h              B8 23 01
mov ax,ds:[0123h]         A1 23 01
push ds:[0123h]           FF 36 23 01
```

可以看到，在一般的汇编指令中，汇编指令中的 idata(立即数)，不论它是表示一个数

据还是内存单元的偏移地址，都会在对应的机器指令中出现，因为 CPU 执行的是机器指令，它必须要处理这些数据或地址。

现在我们在 Debug 中将程序 9.1 翻译成为机器码，看到的结果如图 9.1 所示。

```
-u
0BBD:0000 B80000        MOV     AX,0000
0BBD:0003 EB03          JMP     0008
0BBD:0005 050100        ADD     AX,0001
0BBD:0008 40            INC     AX
```

图 9.1　程序 9.1 的机器码

对照汇编源程序，我们可以看到，Debug 将 jmp short s 中的 s 表示为 inc ax 指令的偏移地址 8，并将 jmp short s 表示为 jmp 0008，表示转移到 cs:0008 处。这一切似乎合理，可是当我们查看 jmp short s 或是 jmp 0008 所对应的机器码，却发现了一些问题。

jmp 0008(Debug 中的表示)或 jmp short s(汇编语言中的表示)所对应的机器码为 **EB 03**，注意，这个机器码中竟不包含转移的目的地址，这意味着，CPU 在执行 EB 03 的时，并不知道转移的目的地址。那么，CPU 根据什么进行转移呢？它知道转移到哪里呢？

令人奇怪的是，汇编指令 jmp short s 中，明明是带有转移的目的地址(由标号 s 表示)的，可翻译成机器指令后，怎么目的地址就没了呢？没有了目的地址，CPU 如何知道转移到哪里呢？

我们把程序 9.1 改写一下，变成下面这样：

程序 9.2

```
assume cs:codesg

codesg segment

  start:mov ax,0
        mov bx,0
        jmp short s
        add ax,1
      s: inc ax

codesg ends

end start
```

我们在 Debug 中将程序 9.2 翻译成为机器码，看到的结果如图 9.2 所示。

```
-u
0BBD:0000 B80000        MOV     AX,0000
0BBD:0003 BB0000        MOV     BX,0000
0BBD:0006 EB03          JMP     000B
0BBD:0008 050100        ADD     AX,0001
0BBD:000B 40            INC     AX
```

图 9.2　程序 9.2 的机器码

比较一下程序 9.1 和 9.2 用 Debug 查看的结果，注意，两个程序中的 jmp 指令都要使 IP 指向 inc ax 指令，但是程序 9.1 的 inc ax 指令的偏移地址为 8，而程序 9.2 的 inc ax 指令的偏移地址为 000BH。我们再来看两个程序中的 jmp 指令所对应的机器码，都是 **EB 03**。这说明 **CPU 在执行 jmp 指令的时候并不需要转移的目的地址**。两个程序中的 jmp 指令的转移目的地址并不一样，一个是 cs:0008，另一个是 cs:000B，如果机器指令中包含了转移的目的地址的话，那么它们对应的机器码应该是不同的。可是它们对应的机器码都是 EB 03，这说明在机器指令中并不包含转移的目的地址。如果机器指令中不包含目的地址的话，那么也就是说，CPU 不需要这个目的地址就可以实现对 IP 的修改。

CPU 不是神仙，它只能处理你提供给它的东西，jmp 指令的机器码中不包含转移的目的地址，那么，CPU 如何知道将 IP 改为多少呢？所以，在 jmp 指令的机器码中，一定包含了某种信息，使得 CPU 可以将它当做修改 IP 的依据。这种信息是什么呢？我们一步步地分析。

我们先简单回忆一下 CPU 执行指令的过程(如果你需要更多的回忆，可以复习一下 2.10 节的内容)。

(1) 从 CS:IP 指向内存单元读取指令，读取的指令进入指令缓冲器；
(2) (IP)=(IP)+所读取指令的长度，从而指向下一条指令；
(3) 执行指令。转到 1，重复这个过程。

按照这个步骤，我们参照图 9.2 看一下，程序 9.2 中 jmp short s 指令的读取和执行过程：

(1) (CS)=0BBDH，(IP)=0006H，CS:IP 指向 EB 03(jmp short s 的机器码)；
(2) 读取指令码 EB 03 进入指令缓冲器；
(3) (IP)=(IP)+所读取指令的长度=(IP)+2=0008H，CS:IP 指向 add ax,1；
(4) CPU 执行指令缓冲器中的指令 EB 03；
(5) 指令 EB 03 执行后，(IP)=000BH，CS:IP 指向 inc ax。

从上面的过程中我们看到，CPU 将指令 EB 03 读入后，IP 指向了下一条指令，即 CS:0008 处的 add ax,1，接着执行 EB 03。如果 EB 03 没有对 IP 进行修改的话，那么，接下来 CPU 将执行 add ax,1，可是，CPU 执行的 EB 03 却是一条修改 IP 的转移指令，执行后(IP)=000BH，CS:IP 指向 inc ax，CS:0008 处的 add ax,1 没有被执行。

CPU 在执行 EB 03 的时候是根据什么修改的 IP，使其指向目标指令呢？就是根据指令码中的 03。注意，要转移的目的地址是 CS:000B，而 CPU 执行 EB 03 时，当前的(IP)=0008H，如果将当前的 IP 值加 3，使(IP)=000BH，CS:IP 就可指向目标指令。在转移指令 EB 03 中并没有告诉 CPU 要转移的目的地址，却告诉了 CPU 要转移的位移，即将当前的 IP 向后移动 3 个字节。因为程序 1、2 中的 jmp 指令转移的位移相同，都是向后 3 个字节，所以它们的机器码都是 EB 03。

原来如此，在“jmp short 标号”指令所对应的机器码中，并不包含转移的目的地址，而包含的是转移的位移。这个位移，是编译器根据汇编指令中的“标号”计算出来的，具体的计算方法如图 9.3 所示。

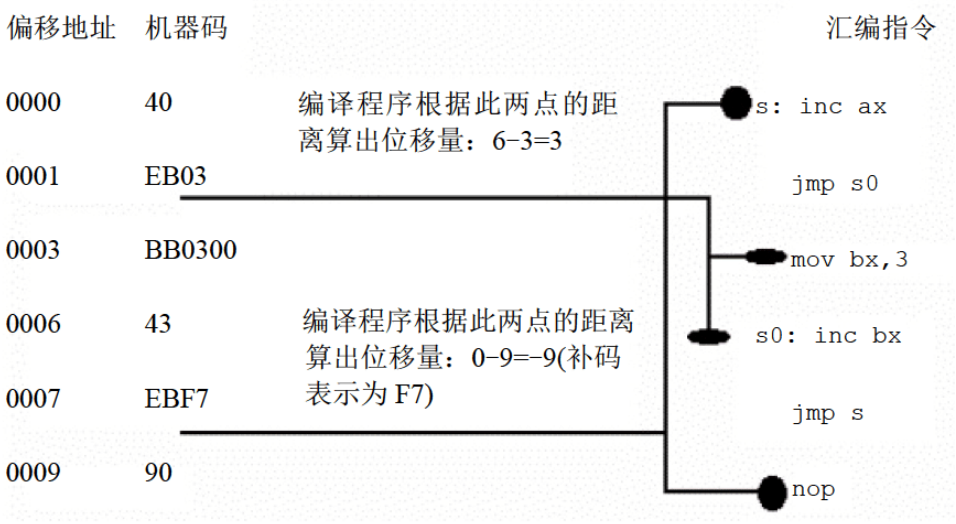


图 9.3 转移位移的计算方法

实际上，“jmp short 标号”的功能为：(IP)=(IP)+8 位位移。

(1) 8 位位移=标号处的地址-jmp 指令后的第一个字节的地址；

(2) short 指明此处的位移为 8 位位移；

(3) 8 位位移的范围为-128~127，用补码表示(如果你对补码还不了解，请阅读附注 2)；

(4) 8 位位移由编译程序在编译时算出。

还有一种和“jmp short 标号”功能相近的指令格式，jmp near ptr 标号，它实现的是段内近转移。

“jmp near ptr 标号”的功能为：(IP)=(IP)+16 位位移。

(1) 16 位位移=标号处的地址-jmp 指令后的第一个字节的地址；

(2) near ptr 指明此处的位移为 16 位位移，进行的是段内近转移；

(3) 16 位位移的范围为-32768～32767，用补码表示；

(4) 16 位位移由编译程序在编译时算出。

9.4 转移的目的地址在指令中的 jmp 指令

前面讲的 jmp 指令，其对应的机器指令中并没有转移的目的地址，而是相对于当前 IP 的转移位移。

“jmp far ptr 标号”实现的是段间转移，又称为远转移。功能如下：

(CS)=标号所在段的段地址；(IP)=标号在段中的偏移地址。

far ptr 指明了指令用标号的段地址和偏移地址修改 CS 和 IP。

看下面的程序：

程序 9.3

```
assume cs:codesg

codesg segment

  start:mov ax,0
        mov bx,0
        jmp far ptr s
        db 256 dup (0)
      s: add ax,1
        inc ax

codesg ends

end start
```

在 Debug 中将程序 9.3 翻译成为机器码，看到的结果如图 9.4 所示。

```
-u
0BBD:0000 B80000        MOV     AX,0000
0BBD:0003 BB0000        MOV     BX,0000
0BBD:0006 EA0B01BD0B    JMP     0BBD:010B
0BBD:000B 0000          ADD     [BX+SI],AL
0BBD:000D 0000          ADD     [BX+SI],AL
0BBD:000F 0000          ADD     [BX+SI],AL
0BBD:0011 0000          ADD     [BX+SI],AL
```

图 9.4　程序 9.3 的机器码

如图 9.4 中所示，源程序中的 db 256 dup (0)，被 Debug 解释为相应的若干条汇编指令。这不是关键，关键是，我们要注意一下 jmp far ptr s 所对应的机器码：EA 0B 01 BD 0B，其中包含转移的目的地址。“0B 01 BD 0B”是目的地址在指令中的存储顺序，高地址的“BD 0B”是转移的段地址：0BBDH，低地址的“0B 01”是偏移地址：010BH。

对于“jmp X 标号”格式的指令的深入分析请参看附注 3。

9.5　转移地址在寄存器中的 jmp 指令

指令格式：jmp 16 位 reg
功能：(IP)=(16 位 reg)

这种指令我们在前面的内容(参见 2.11 节)中已经讲过，这里就不再详述。

9.6 转移地址在内存中的 jmp 指令

转移地址在内存中的 jmp 指令有两种格式：

(1) jmp word ptr 内存单元地址(段内转移)

功能：从内存单元地址处开始存放着一个字，是转移的目的偏移地址。

内存单元地址可用寻址方式的任一格式给出。

比如，下面的指令：

```
mov ax,0123H
mov ds:[0],ax
jmp word ptr ds:[0]
```

执行后，(IP)=0123H。

又比如，下面的指令：

```
mov ax,0123H
mov [bx],ax
jmp word ptr [bx]
```

执行后，(IP)=0123H

(2) jmp dword ptr 内存单元地址(段间转移)

功能：从内存单元地址处开始存放着两个字，高地址处的字是转移的目的段地址，低地址处是转移的目的偏移地址。

(CS)=(内存单元地址+2)
(IP)=(内存单元地址)

内存单元地址可用寻址方式的任一格式给出。

比如，下面的指令：

```
mov ax,0123H
mov ds:[0],ax
mov word ptr ds:[2],0
jmp dword ptr ds:[0]
```

执行后，(CS)=0，(IP)=0123H，CS:IP 指向 0000:0123

又比如，下面的指令：

```
mov ax,0123H
mov [bx],ax
```

```
mov word ptr [bx+2],0
jmp dword ptr [bx]
```

执行后，(CS)=0，(IP)=0123H，CS:IP 指向 0000:0123

检测点 9.1

(1) 程序如下。

```
assume cs:code

data segment
   ?
data ends

code segment
  start:mov ax,data
        mov ds,ax
        mov bx,0
        jmp word ptr [bx+1]

code ends
end start
```

若要使程序中的 jmp 指令执行后，CS:IP 指向程序的第一条指令，在 data 段中应该定义哪些数据？

(2) 程序如下。

```
assume cs:code

data segment
  dd 12345678H
data ends

code segment

  start:mov ax,data
        mov ds,ax
        mov bx,0
        mov [bx],_______
        mov [bx+2],_______
        jmp dword ptr ds:[0]

code ends

end start
```

补全程序，使 jmp 指令执行后，CS:IP 指向程序的第一条指令。

(3) 用 Debug 查看内存，结果如下：

```
2000:1000 BE 00 06 00 00 00 ……
```

则此时，CPU 执行指令：

```
mov ax,2000H
mov es,ax
jmp dword ptr es:[1000H]
```

后，(CS)=？，(IP)=？

9.7 jcxz 指令

jcxz 指令为有条件转移指令，所有的有条件转移指令都是短转移，在对应的机器码中包含转移的位移，而不是目的地址。对 IP 的修改范围都为：-128~127。

指令格式：jcxz 标号(如果(cx)=0，转移到标号处执行。

操作：当(cx)=0 时，(IP)=(IP)+8 位位移；

8 位位移=标号处的地址-jcxz 指令后的第一个字节的地址；

8 位位移的范围为-128~127，用补码表示；

8 位位移由编译程序在编译时算出。

当(cx)≠0 时，什么也不做(程序向下执行)。

我们从 jcxz 的功能中可以看出，“jcxz 标号”的功能相当于：

```
if((cx)==0)jmp short 标号;
```

(这种用 C 语言和汇编语言进行的综合描述，或许能使你对有条件转移指令理解得更加清楚。)

检测点 9.2

补全程序，利用 jcxz 指令，实现在内存 2000H 段中查找第一个值为 0 的字节，找到后，将它的偏移地址存储在 dx 中。

```
assume cs:code
code segment
  start:mov ax,2000H
        mov ds,ax
        mov bx,0
      s:________
        ________
        ________
        ________
        jmp short s
```

```
      ok:mov dx,bx
         mov ax,4c00h
         int 21h
code ends
end start
```

9.8　loop 指令

loop 指令为循环指令，所有的循环指令都是**短转移**，在对应的机器码中包含转移的位移，而不是目的地址。对 IP 的修改范围都为：-128~127。

指令格式：loop 标号((cx)=(cx)-1，如果(cx)≠0，转移到标号处执行。

操作：

(1)　(cx)=(cx)-1；

(2)　如果(cx)≠0，(IP)=(IP)+8 位位移。

8 位位移=标号处的地址-loop 指令后的第一个字节的地址；

8 位位移的范围为-128~127，用补码表示；

8 位位移由编译程序在编译时算出。

如果(cx)=0，什么也不做(程序向下执行)。

我们从 loop 的功能中可以看出，“loop 标号”的功能相当于：

```
(cx)--;
if((cx)≠0)jmp short 标号;
```

检测点 9.3

补全程序，利用 loop 指令，实现在内存 2000H 段中查找第一个值为 0 的字节，找到后，将它的偏移地址存储在 dx 中。

```
assume cs:code
code segment
  start:mov ax,2000H
        mov ds,ax
        mov bx,0
      s: mov cl,[bx]
        mov ch,0
        ____________
        inc bx
        loop s
      ok:dec bx        ;dec 指令的功能和 inc 相反，dec bx 进行的操作为：(bx)=(bx)-1
        mov dx,bx
        mov ax,4c00h
```

```
        int 21h
code ends
end start
```

9.9 根据位移进行转移的意义

前面我们讲到：

jmp short 标号
jmp near ptr 标号
jcxz 标号
loop 标号

等几种汇编指令，它们对 IP 的修改是根据转移目的地址和转移起始地址之间的位移来进行的。在它们对应的机器码中不包含转移的目的地址，而包含的是到目的地址的位移。

这种设计，方便了程序段在内存中的浮动装配。

例如：

```
    汇编指令              机器代码
    mov cx,6              B9 06 00
    mov ax,10h            B8 10 00
s:  add ax,ax             01 C0
    loop s                E2 FC
```

这段程序装在内存中的不同位置都可正确执行，因为 loop s 在执行时只涉及 s 的位移(-4，前移 4 个字节，补码表示为 FCH)，而不是 s 的地址。如果 loop s 的机器码中包含的是 s 的地址，则就对程序段在内存中的偏移地址有了严格的限制，因为机器码中包含的是 s 的地址，如果 s 处的指令不在目的地址处，程序的执行就会出错。而 loop s 的机器码中包含的是转移的位移，就不存在这个问题了，因为，无论 s 处的指令的实际地址是多少，loop 指令的转移位移是不变的。

9.10 编译器对转移位移超界的检测

注意，根据位移进行转移的指令，它们的转移范围受到转移位移的限制，如果在源程序中出现了转移范围超界的问题，在编译的时候，编译器将报错。

比如，下面的程序将引起编译错误：

```
assume cs:code

code segment
  start:jmp short s
        db 128 dup (0)
```

```
     s: mov ax,0ffffh
code ends
end start
```

jmp short s 的转移范围是-128~127，IP 最多向后移动 127 个字节。

注意，我们在第 2 章中讲到的形如“jmp 2000:0100”的转移指令，是在 Debug 中使用的汇编指令，汇编编译器并不认识。如果在源程序中使用，编译时也会报错。

实验 8　分析一个奇怪的程序

分析下面的程序，在运行前思考：这个程序可以正确返回吗？

运行后再思考：为什么是这种结果？

通过这个程序加深对相关内容的理解。

```
assume cs:codesg
codesg segment

        mov ax,4c00h
        int 21h

start:  mov ax,0
    s:  nop
        nop

        mov di,offset s
        mov si,offset s2
        mov ax,cs:[si]
        mov cs:[di],ax

   s0:  jmp short s

   s1:  mov ax,0
        int 21h
        mov ax,0

   s2:  jmp short s1
        nop

codesg ends
end start
```

实验 9　根据材料编程

这个编程任务必须在进行下面的课程之前独立完成，因为后面的课程中，需要通过这个实验而获得的编程经验。

编程：在屏幕中间分别显示绿色、绿底红色、白底蓝色的字符串 'welcome to masm!'。

编程所需的知识通过阅读、分析下面的材料获得。

80×25 彩色字符模式显示缓冲区(以下简称为显示缓冲区)的结构：

内存地址空间中，B8000H~BFFFFH 共 32KB 的空间，为 80×25 彩色字符模式的显示缓冲区。向这个地址空间写入数据，写入的内容将立即出现在显示器上。

在 80×25 彩色字符模式下，显示器可以显示 25 行，每行 80 个字符，每个字符可以有 256 种属性(背景色、前景色、闪烁、高亮等组合信息)。

这样，一个字符在显示缓冲区中就要占两个字节，分别存放字符的 ASCII 码和属性。80×25 模式下，一屏的内容在显示缓冲区中共占 4000 个字节。

显示缓冲区分为 8 页，每页 4KB(≈4000B)，显示器可以显示任意一页的内容。一般情况下，显示第 0 页的内容。也就是说通常情况下，B8000H～B8F9FH 中的 4000 个字节的内容将出现在显示器上。

在一页显示缓冲区中：

```
偏移 000~09F 对应显示器上的第 1 行(80 个字符占 160 个字节);
偏移 0A0~13F 对应显示器上的第 2 行;
偏移 140~1DF 对应显示器上的第 3 行;
```

依此类推，可知，偏移 F00~F9F 对应显示器上的第 25 行。

在一行中，一个字符占两个字节的存储空间(一个字)，低位字节存储字符的 ASCII 码，高位字节存储字符的属性。一行共有 80 个字符，占 160 个字节。

即在一行中：

```
00~01 单元对应显示器上的第 1 列;
02~03 单元对应显示器上的第 2 列;
04~05 单元对应显示器上的第 3 列;
```

依此类推，可知，9E～9F 单元对应显示器上的第 80 列。

例：在显示器的 0 行 0 列显示黑底绿色的字符串'ABCDEF'
('A'的 ASCII 码值为 41H，02H 表示黑底绿色)

显示缓冲区里的内容为：

```
           00  01  02  03  04  05  06  07  08  09  0A  0B  ... 0E  0F
B800:0000  41  02  42  02  43  02  44  02  45  02  46  02  ... ..  ..
  ⋮
  ⋮
B800:00A0  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ... ..  ..
```

可以看出，在显示缓冲区中，偶地址存放字符，奇地址存放字符的颜色属性。

一个在屏幕上显示的字符，具有前景(字符色)和背景(底色)两种颜色，字符还可以以高亮度和闪烁的方式显示。前景色、背景色、闪烁、高亮等信息被记录在属性字节中。

属性字节的格式：

	7	6	5	4	3	2	1	0
含义	BL	R	G	B	I	R	G	B
	闪烁	背景			高亮	前景		

R：红色
G：绿色
B：蓝色

可以按位设置属性字节，从而配出各种不同的前景色和背景色。

比如：

红底绿字，属性字节为：01000010B；
红底闪烁绿字，属性字节为：11000010B；
红底高亮绿字，属性字节为：01001010B；
黑底白字，属性字节为：00000111B；
白底蓝字，属性字节为：01110001B。

例：在显示器的 0 行 0 列显示红底高亮闪烁绿色的字符串'ABCDEF'
(红底高亮闪烁绿色，属性字节为：11001010B，CAH)

显示缓冲区里的内容为：

```
          00  01  02  03  04  05  06  07  08  09  0A  0B  ...  9E  9F
B800:0000 41  CA  42  CA  43  CA  44  CA  45  CA  46  CA  ...  ..  ..
  ⋮
  ⋮
B800:00A0 ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ..  ...  ..  ..
```

注意，闪烁的效果必须在全屏 DOS 方式下才能看到。

第 10 章　CALL 和 RET 指令

call 和 ret 指令都是转移指令，它们都修改 IP，或同时修改 CS 和 IP。它们经常被共同用来实现子程序的设计。这一章，我们讲解 call 和 ret 指令的原理。

10.1　ret 和 retf

ret 指令用栈中的数据，修改 IP 的内容，从而实现近转移；
retf 指令用栈中的数据，修改 CS 和 IP 的内容，从而实现远转移。

CPU 执行 ret 指令时，进行下面两步操作：

(1)　(IP)=((ss)*16+(sp))
(2)　(sp)=(sp)+2

CPU 执行 retf 指令时，进行下面 4 步操作：

(1)　(IP)=((ss)*16+(sp))
(2)　(sp)=(sp)+2
(3)　(CS)=((ss)*16+(sp))
(4)　(sp)=(sp)+2

可以看出，如果我们用汇编语法来解释 ret 和 retf 指令，则：

CPU 执行 ret 指令时，相当于进行：

```
pop IP
```

CPU 执行 retf 指令时，相当于进行：

```
pop IP
pop CS
```

例：

下面的程序中，ret 指令执行后，(IP)=0，CS:IP 指向代码段的第一条指令。

```
assume cs:code

stack segment
  db 16 dup (0)
stack ends

code segment
        mov ax,4c00h
```

```
        int 21h

start:  mov ax,stack
        mov ss,ax
        mov sp,16
        mov ax,0
        push ax
        mov bx,0
        ret
code ends

end start
```

下面的程序中，retf 指令执行后，CS:IP 指向代码段的第一条指令。

```
assume cs:code

stack segment
  db 16 dup (0)
stack ends

code segment
        mov ax,4c00h
        int 21h
start:  mov ax,stack
        mov ss,ax
        mov sp,16
        mov ax,0
        push cs
        push ax
        mov bx,0
        retf
code ends

end start
```

检测点 10.1

补全程序，实现从内存 1000:0000 处开始执行指令。

```
assume cs:code

stack segment
  db 16 dup (0)
stack ends

code segment
start:  mov ax,stack
        mov ss,ax
        mov sp,16
        mov ax,________
```

```
        push ax
        mov ax,________
        push ax
        retf
code ends

end start
```

10.2 call 指令

CPU 执行 call 指令时，进行两步操作：

(1) 将当前的 IP 或 CS 和 IP 压入栈中；
(2) 转移。

call 指令不能实现短转移，除此之外，call 指令实现转移的方法和 jmp 指令的原理相同，下面的几个小节中，我们以给出转移目的地址的不同方法为主线，讲解 call 指令的主要应用格式。

10.3 依据位移进行转移的 call 指令

call 标号(将当前的 IP 压栈后，转到标号处执行指令)

CPU 执行此种格式的 call 指令时，进行如下的操作：

(1) (sp)=(sp)−2
((ss)*16+(sp))=(IP)
(2) (IP)=(IP)+16 位位移。

16 位位移=标号处的地址−call 指令后的第一个字节的地址；
16 位位移的范围为−32768~32767，用补码表示；
16 位位移由编译程序在编译时算出。

从上面的描述中，可以看出，如果我们用汇编语法来解释此种格式的 call 指令，则：

CPU 执行“call 标号”时，相当于进行：

```
push IP
jmp near ptr 标号
```

检测点 10.2

下面的程序执行后，ax 中的数值为多少？

```
内存地址        机器码        汇编指令
1000:0          b8 00 00      mov ax,0
1000:3          e8 01 00      call s
1000:6          40            inc ax
1000:7          58          s:pop ax
```

10.4　转移的目的地址在指令中的 call 指令

前面讲的 call 指令，其对应的机器指令中并没有转移的目的地址，而是相对于当前 IP 的转移位移。

“call far ptr 标号”实现的是段间转移。

CPU 执行此种格式的 call 指令时，进行如下的操作。

(1) (sp)=(sp)−2
((ss)*16+(sp))=(CS)
(sp)=(sp)−2
((ss)*16+(sp))=(IP)

(2) (CS)=标号所在段的段地址
(IP)=标号在段中的偏移地址

从上面的描述中可以看出，如果我们用汇编语法来解释此种格式的 call 指令，则：

CPU 执行“call far ptr 标号”时，相当于进行：

```
push CS
push IP
jmp far ptr 标号
```

检测点 10.3

下面的程序执行后，ax 中的数值为多少？

```
内存地址        机器码                汇编指令
1000:0          b8 00 00              mov ax,0
1000:3          9A 09 00 00 10        call far ptr s
1000:8          40                    inc ax
1000:9          58                  s:pop ax
                                      add ax,ax
                                      pop bx
                                      add ax,bx
```

10.5 转移地址在寄存器中的 call 指令

指令格式：call 16 位 reg

功能：

(sp)=(sp)−2

((ss)*16+(sp))=(IP)

(IP)=(16 位 reg)

用汇编语法来解释此种格式的 call 指令，CPU 执行“call 16 位 reg”时，相当于进行：

```
push IP
jmp 16 位 reg
```

检测点 10.4

下面的程序执行后，ax 中的数值为多少？

内存地址	机器码	汇编指令
1000:0	b8 06 00	mov ax,6
1000:3	ff d0	call ax
1000:5	40	inc ax
1000:6		mov bp,sp
		add ax,[bp]

10.6 转移地址在内存中的 call 指令

转移地址在内存中的 call 指令有两种格式。

(1) call word ptr 内存单元地址

用汇编语法来解释此种格式的 call 指令，则：

CPU 执行“call word ptr 内存单元地址”时，相当于进行：

```
push IP
jmp word ptr 内存单元地址
```

比如，下面的指令：

```
mov sp,10h
mov ax,0123h
mov ds:[0],ax
```

```
call word ptr ds:[0]
```

执行后，(IP)=0123H，(sp)=0EH。

(2)　call dword ptr 内存单元地址

用汇编语法来解释此种格式的 call 指令，则：

CPU 执行“call dword ptr 内存单元地址”时，相当于进行：

```
push CS
push IP
jmp dword ptr 内存单元地址
```

比如，下面的指令：

```
mov sp,10h
mov ax,0123h
mov ds:[0],ax
mov word ptr ds:[2],0
call dword ptr ds:[0]
```

执行后，(CS)=0，(IP)=0123H，(sp)=0CH。

检测点 10.5

(1)　下面的程序执行后，ax 中的数值为多少？(注意：用 call 指令的原理来分析，不要在 Debug 中单步跟踪来验证你的结论。对于此程序，在 Debug 中单步跟踪的结果，不能代表 CPU 的实际执行结果。)

```
assume cs:code
stack segment
   dw 8 dup (0)
stack ends
code segment
  start:mov ax,stack
        mov ss,ax
        mov sp,16
        mov ds,ax
        mov ax,0
        call word ptr ds:[0EH]
        inc ax
        inc ax
        inc ax
        mov ax,4c00h
        int 21h
code ends
end start
```

(2) 下面的程序执行后，ax 和 bx 中的数值为多少？

```
assume cs:code
data segment
   dw 8 dup (0)
data ends
code segment
  start:mov ax,data
        mov ss,ax
        mov sp,16
        mov word ptr ss:[0],offset s
        mov ss:[2],cs
        call dword ptr ss:[0]
        nop
      s: mov ax,offset s
        sub ax,ss:[0cH]
        mov bx,cs
        sub bx,ss:[0eH]
        mov ax, 4c00h
        int 21h
code ends
end start
```

10.7 call 和 ret 的配合使用

前面，我们已经分别学习了 ret 和 call 指令的原理。现在来看一下，如何将它们配合使用来实现子程序的机制。

问题 10.1

下面程序返回前，bx 中的值是多少？

```
assume cs:code
code segment
 start: mov ax,1
        mov cx,3
        call s
        mov bx,ax          ;(bx)=?
        mov ax,4c00h
        int 21h
      s: add ax,ax
        loop s
        ret
code ends
end start
```

思考后看分析。

分析：

我们来看一下 CPU 执行这个程序的主要过程。

(1) CPU 将 call s 指令的机器码读入，IP 指向了 call s 后的指令 mov bx,ax，然后 CPU 执行 call s 指令，将当前的 IP 值(指令 mov bx,ax 的偏移地址)压栈，并将 IP 的值改变为标号 s 处的偏移地址；

(2) CPU 从标号 s 处开始执行指令，loop 循环完毕后，(ax)=8；

(3) CPU 将 ret 指令的机器码读入，IP 指向了 ret 指令后的内存单元，然后 CPU 执行 ret 指令，从栈中弹出一个值(即 call s 先前压入的 mov bx,ax 指令的偏移地址)送入 IP 中。则 CS:IP 指向指令 mov bx,ax；

(4) CPU 从 mov bx,ax 开始执行指令，直至完成。

程序返回前，(bx)=8。可以看出，从标号 s 到 ret 的程序段的作用是计算 2 的 N 次方，计算前，N 的值由 cx 提供。

我们再来看下面的程序：

```
源程序                          内存中的情况(假设程序从内存 1000:0 处装入)
assume cs:code

stack segment
  db 8 dup (0)                  1000:0000   00 00 00 00 00 00 00 00
  db 8 dup (0)                  1000:0008   00 00 00 00 00 00 00 00
stack ends

code segment
  start:mov ax,stack            1001:0000   B8 00 10
        mov ss,ax               1001:0003   8E D0
        mov sp,16               1001:0005   BC 10 00
        mov ax,1000             1001:0008   B8 E8 03
        call s                  1001:000B   E8 05 00
        mov ax,4c00h            1001:000E   B8 00 4C
        int 21h                 1001:0011   CD 21
      s:add ax,ax               1001:0013   03 C0
        ret                     1001:0015   C3
code ends

end start
```

看一下程序的主要执行过程。

(1) 前 3 条指令执行后，栈的情况如下：

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

(2) call 指令读入后，(IP)=000EH，CPU 指令缓冲器中的代码为：E8 05 00；

CPU 执行 E8 05 00，首先，栈中的情况变为：

```
1000:0000   00 00 00 00 00 00 00 00 00 00 00 00 00 00 0E 00
                                                      ↑
                                                    ss:sp
```

然后，(IP)=(IP)+0005=0013H。

(3) CPU 从 cs:0013H 处(即标号 s 处)开始执行。

(4) ret 指令读入后：

(IP)=0016H，CPU 指令缓冲器中的代码为：C3

CPU 执行 C3，相当于进行 pop IP，执行后，栈中的情况为：

```
1000:0000   00 00 00 00 00 00 00 00 00 00 00 00 00 00 0E 00
                                                         ↑
                                                       ss:sp
(IP)=000EH
```

(5) CPU 回到 cs:000EH 处(即 call 指令后面的指令处)继续执行。

从上面的讨论中我们发现，可以写一个具有一定功能的程序段，我们称其为子程序，在需要的时候，用 call 指令转去执行。可是执行完子程序后，如何让 CPU 接着 call 指令向下执行？call 指令转去执行子程序之前，call 指令后面的指令的地址将存储在栈中，所以可在子程序的后面使用 ret 指令，用栈中的数据设置 IP 的值，从而转到 call 指令后面的代码处继续执行。

这样，我们可以利用 call 和 ret 来实现子程序的机制。子程序的框架如下。

```
标号：
    指令
    ret
```

具有子程序的源程序的框架如下。

```
assume cs:code
code segment
  main:
        ⋮
        call  sub1              ;调用子程序 sub1
        ⋮
        ⋮
        mov ax,4c00h
        int 21h

  sub1:                         ;子程序 sub1 开始
        ⋮
        call sub2               ;调用子程序 sub2
        ⋮
        ⋮
```

```
        ret                         ;子程序返回

  sub2:                             ;子程序 sub2 开始
        ⋮
        ⋮
        ret                         ;子程序返回
code ends
end main
```

现在，可以从子程序的角度，回过头来再看一下本节中的两个程序。

10.8　mul 指令

因下面要用到，这里介绍一下 mul 指令，mul 是乘法指令，使用 mul 做乘法的时候，注意以下两点。

(1)　两个相乘的数：两个相乘的数，要么都是 8 位，要么都是 16 位。如果是 8 位，一个默认放在 AL 中，另一个放在 8 位 reg 或内存字节单元中；如果是 16 位，一个默认在 AX 中，另一个放在 16 位 reg 或内存字单元中。

(2)　结果：如果是 8 位乘法，结果默认放在 AX 中；如果是 16 位乘法，结果高位默认在 DX 中存放，低位在 AX 中放。

格式如下：

```
mul reg
mul 内存单元
```

内存单元可以用不同的寻址方式给出，比如：

```
mul byte ptr ds:[0]
```

含义：(ax)=(al)*((ds)*16+0)；

```
mul word ptr [bx+si+8]
```

含义：(ax)=(ax)*((ds)*16+(bx)+(si)+8)结果的低 16 位。
　　(dx)=(ax)*((ds)*16+(bx)+(si)+8)结果的高 16 位。

例：

(1)　计算 100*10。

100 和 10 小于 255，可以做 8 位乘法，程序如下。

```
mov al,100
mov bl,10
mul bl
```

结果：(ax)=1000(03E8H)

(2) 计算 100*10000

100 小于 255，可 10000 大于 255，所以必须做 16 位乘法，程序如下。

```
mov ax,100
mov bx,10000
mul bx
```

结果：(ax)=4240H，(dx)=000FH　　(F4240H=1000000)

10.9 模块化程序设计

从上面我们看到，call 与 ret 指令共同支持了汇编语言编程中的模块化设计。在实际编程中，程序的模块化是必不可少的。因为现实的问题比较复杂，对现实问题进行分析时，把它转化成为相互联系、不同层次的子问题，是必须的解决方法。而 call 与 ret 指令对这种分析方法提供了程序实现上的支持。利用 call 和 ret 指令，我们可以用简捷的方法，实现多个相互联系、功能独立的子程序来解决一个复杂的问题。

下面的内容中，我们来看一下子程序设计中的相关问题和解决方法。

10.10 参数和结果传递的问题

子程序一般都要根据提供的参数处理一定的事务，处理后，将结果(返回值)提供给调用者。其实，我们讨论参数和返回值传递的问题，实际上就是在探讨，应该如何存储子程序需要的参数和产生的返回值。

比如，设计一个子程序，可以根据提供的 N，来计算 N 的 3 次方。

这里面就有两个问题：

(1) 将参数 N 存储在什么地方？
(2) 计算得到的数值，存储在什么地方？

很显然，可以用寄存器来存储，可以将参数放到 bx 中；因为子程序中要计算 N*N*N，可以使用多个 mul 指令，为了方便，可将结果放到 dx 和 ax 中。子程序如下。

```
;说明：计算 N 的 3 次方
;参数：(bx)=N
;结果：(dx:ax)=N^3

cube:mov ax,bx
     mul bx
     mul bx
     ret
```

注意，我们在编程的时候要注意形成良好的风格，对于程序应有详细的注释。子程序

的注释信息应该包含对子程序的功能、参数和结果的说明。因为今天写的子程序，以后可能还会用到；自己写的子程序，也很可能要给别人使用，所以一定要有全面的说明。

用寄存器来存储参数和结果是最常使用的方法。对于存放参数的寄存器和存放结果的寄存器，调用者和子程序的读写操作恰恰相反：调用者将参数送入参数寄存器，从结果寄存器中取到返回值；子程序从参数寄存器中取到参数，将返回值送入结果寄存器。

编程，计算 data 段中第一组数据的 3 次方，结果保存在后面一组 dword 单元中。

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

我们可以用到已经写好的子程序，程序如下：

```
code segment

  start:mov ax,data
        mov ds,ax
        mov si,0                ;ds:si 指向第一组 word 单元
        mov di,16               ;ds:di 指向第二组 dword 单元

        mov cx,8
     s: mov bx,[si]
        call cube
        mov [di],ax
        mov [di].2,dx
        add si,2                ;ds:si 指向下一个 word 单元
        add di,4                ;ds:di 指向下一个 dword 单元
        loop s

        mov ax,4c00h
        int 21h

cube:   mov ax,bx
        mul bx
        mul bx
        ret

code ends
end start
```

10.11　批量数据的传递

前面的例程中，子程序 cube 只有一个参数，放在 bx 中。如果有两个参数，那么可以用两个寄存器来放，可是如果需要传递的数据有 3 个、4 个或更多直至 N 个，该怎样存放

呢？寄存器的数量终究有限，我们不可能简单地用寄存器来存放多个需要传递的数据。对于返回值，也有同样的问题。

在这种时候，我们将批量数据放到内存中，然后将它们所在内存空间的首地址放在寄存器中，传递给需要的子程序。对于具有批量数据的返回结果，也可用同样的方法。

下面看一个例子，设计一个子程序，功能：将一个全是字母的字符串转化为大写。

这个子程序需要知道两件事，字符串的内容和字符串的长度。因为字符串中的字母可能很多，所以不便将整个字符串中的所有字母都直接传递给子程序。但是，可以将字符串在内存中的首地址放在寄存器中传递给子程序。因为子程序中要用到循环，我们可以用 loop 指令，而循环的次数恰恰就是字符串的长度。出于方便的考虑，可以将字符串的长度放到 cx 中。

```
capital: and byte ptr [si],11011111b    ;将 ds:si 所指单元中的字母转化为大写
         inc si                         ;ds:si 指向下一个单元
         loop capital
         ret
```

编程，将 data 段中的字符串转化为大写。

```
assume cs:code

data segment
  db 'conversation'
data ends

code segment
  start:mov ax,data
        mov ds,ax
        mov si,0            ;ds:si 指向字符串(批量数据)所在空间的首地址
        mov cx,12           ;cx 存放字符串的长度
        call capital
        mov ax,4c00h
        int 21h

capital:and byte ptr [si],11011111b
        inc si
        loop capital
        ret
code ends
end start
```

注意，除了用寄存器传递参数外，还有一种通用的方法是用栈来传递参数。关于这种技术请参看附注 4。

10.12　寄存器冲突的问题

设计一个子程序，功能：将一个全是字母，以 0 结尾的字符串，转化为大写。

程序要处理的字符串以 0 作为结尾符，这个字符串可以如下定义：

```
db  'conversation',0
```

应用这个子程序，字符串的内容后面一定要有一个 0，标记字符串的结束。子程序可以依次读取每个字符进行检测，如果不是 0，就进行大写的转化；如果是 0，就结束处理。由于可通过检测 0 而知道是否已经处理完整个字符串，所以子程序可以不需要字符串的长度作为参数。可以用 jcxz 来检测 0。

```
;说明：将一个全是字母，以 0 结尾的字符串，转化为大写
;参数：ds:si 指向字符串的首地址
;结果：没有返回值

capital:mov cl,[si]
        mov ch,0
        jcxz ok                          ;如果(cx)=0，结束；如果不是 0，处理
        and byte ptr [si],11011111b      ;将 ds:si 所指单元中的字母转化为大写
        inc si                           ;ds:si 指向下一个单元
        jmp short capital
     ok:ret
```

来看一下这个子程序的应用。

(1)　将 data 段中字符串转化为大写。

```
assume cs:code
data segment
  db  'conversation',0
data ends
```

代码段中的相关程序段如下。

```
mov ax,data
mov ds,ax
mov si,0
call capital
```

(2)　将 data 段中的字符串全部转化为大写。

```
assume cs:code
data segment
  db  'word',0
  db  'unix',0
  db  'wind',0
  db  'good',0
data ends
```

可以看到，所有字符串的长度都是 5(算上结尾符 0)，使用循环，重复调用子程序 capital，完成对 4 个字符串的处理。完整的程序如下。

```
code segment

 start: mov ax,data
        mov ds,ax
        mov bx,0

        mov cx,4
      s: mov si,bx
        call capital
        add bx,5
        loop s

        mov ax,4c00h
        int 21h

capital:mov cl,[si]
        mov ch,0
        jcxz ok
        and byte ptr [si],11011111b
        inc si
        jmp short capital
      ok:ret

code ends

end start
```

问题 10.2

这个程序在思想上完全正确，但在细节上却有些错误，把错误找出来。

思考后看分析。

分析：

问题在于 cx 的使用，主程序要使用 cx 记录循环次数，可是子程序中也使用了 cx，在执行子程序的时候，cx 中保存的循环计数值被改变，使得主程序的循环出错。

从上面的问题中，实际上引出了一个一般化的问题：子程序中使用的寄存器，很可能在主程序中也要使用，造成了寄存器使用上的冲突。

那么如何来避免这种冲突呢？粗略地看，可以有以下两个方案。

(1) 在编写调用子程序的程序时，注意看看子程序中有没有用到会产生冲突的寄存器，如果有，调用者使用别的寄存器；

(2) 在编写子程序的时候，不要使用会产生冲突的寄存器。

我们来分析一下上面两个方案的可行性：

(1) 这将给调用子程序的程序的编写造成很大的麻烦，因为必须要小心检查所调用的子程序中是否有将产生冲突的寄存器。比如说，在上面的例子中，我们在编写主程序的循环的时候就得检查子程序中是否用到了 bx 和 cx，因为如果子程序中用到了这两个寄存器就会出现问题。如果采用这种方案来解决冲突的话，那么在主程序的循环中，就不能使用 cx 寄存器，因为子程序中已经用到。

(2) 这个方案是不可能实现的，因为编写子程序的时候无法知道将来的调用情况。

可见，我们上面所设想的两个方案都不可行。我们希望：

(1) 编写调用子程序的程序的时候不必关心子程序到底使用了哪些寄存器；
(2) 编写子程序的时候不必关心调用者使用了哪些寄存器；
(3) 不会发生寄存器冲突。

解决这个问题的简捷方法是，在子程序的开始将子程序中所有用到的寄存器中的内容都保存起来，在子程序返回前再恢复。可以用栈来保存寄存器中的内容。

以后，我们编写子程序的标准框架如下：

```
子程序开始：子程序中使用的寄存器入栈

            子程序内容

            子程序中使用的寄存器出栈

            返回(ret、retf)
```

我们改进一下子程序 capital 的设计：

```
capital:    push cx
            push si

 change:    mov cl,[si]
            mov ch,0
            jcxz ok
            and byte ptr [si],11011111b
            inc si
            jmp short change

     ok:    pop si
            pop cx
            ret
```

要注意寄存器入栈和出栈的顺序。

实验 10　编写子程序

在这次实验中，我们将要编写 3 个子程序，通过它们来认识几个常见的问题和掌握解决这些问题的方法。同前面的所有实验一样，这个实验是必须独立完成的，在后面的课程中，将要用到这个实验中编写的 3 个子程序。

1. 显示字符串

问题

显示字符串是现实工作中经常要用到的功能，应该编写一个通用的子程序来实现这个功能。我们应该提供灵活的调用接口，使调用者可以决定显示的位置(行、列)、内容和颜色。

子程序描述

名称：show_str
功能：在指定的位置，用指定的颜色，显示一个用 0 结束的字符串。
参数：(dh)=行号(取值范围 0~24)，(dl)=列号(取值范围 0~79)，
　　(cl)=颜色，ds:si指向字符串的首地址
返回：无
应用举例：在屏幕的 8 行 3 列，用绿色显示 data 段中的字符串。

```
assume cs:code
data segment
  db 'Welcome to masm!',0
data ends

code segment
start:  mov dh,8
        mov dl,3
        mov cl,2
        mov ax,data
        mov ds,ax
        mov si,0
        call show_str

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

提示

(1)　子程序的入口参数是屏幕上的行号和列号，注意在子程序内部要将它们转化为显

存中的地址，首先要分析一下屏幕上的行列位置和显存地址的对应关系；

(2) 注意保存子程序中用到的相关寄存器；

(3) 这个子程序的内部处理和显存的结构密切相关，但是向外提供了与显存结构无关的接口。通过调用这个子程序，进行字符串的显示时可以不必了解显存的结构，为编程提供了方便。在实验中，注意体会这种设计思想。

2. 解决除法溢出的问题

问题

前面讲过，div 指令可以做除法。当进行 8 位除法的时候，用 al 存储结果的商，ah 存储结果的余数；进行 16 位除法的时候，用 ax 存储结果的商，dx 存储结果的余数。可是，现在有一个问题，如果结果的商大于 al 或 ax 所能存储的最大值，那么将如何？

比如，下面的程序段：

```
mov bh,1
mov ax,1000
div bh
```

进行的是8位除法，结果的商为1000，而1000在al中放不下。

又比如，下面的程序段：

```
mov ax,1000H
mov dx,1
mov bx,1
div bx
```

进行的是 16 位除法，结果的商为 11000H，而 11000H 在 ax 中存放不下。

我们在用 div 指令做除法的时候，很可能发生上面的情况：结果的商过大，超出了寄存器所能存储的范围。当 CPU 执行 div 等除法指令的时候，如果发生这样的情况，将引发 CPU 的一个内部错误，这个错误被称为：**除法溢出**。我们可以通过特殊的程序来处理这个错误，但在这里我们不讨论这个错误的处理，这是后面的课程中要涉及的内容。下面我们仅仅来看一下除法溢出发生时的一些现象，如图 10.1 所示。

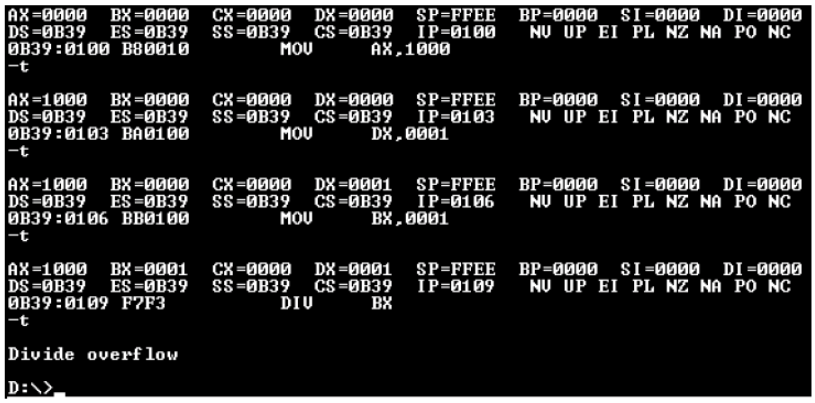


图 10.1　除法溢出时发生的现象

图中展示了在 Windows 2000 中使用 Debug 执行相关程序段的结果，div 指令引发了 CPU 的除法溢出，系统对其进行了相关的处理。

好了，我们已经清楚了问题的所在：用 div 指令做除法的时候可能产生除法溢出。由于有这样的问题，在进行除法运算的时候要注意除数和被除数的值，比如 1000000/10 就不能用 div 指令来计算。那么怎么办呢？我们用下面的子程序 divdw 解决。

子程序描述

名称：divdw

功能：进行不会产生溢出的除法运算，被除数为 dword 型，除数为 word 型，结果为 dword 型。

参数：(ax)=dword 型数据的低 16 位

(dx)=dword型数据的高16位

(cx)=除数

返回：(dx)=结果的高 16 位，(ax)=结果的低 16 位

(cx)=余数

应用举例：计算 1000000/10(F4240H/0AH)

```
mov ax,4240H
mov dx,000FH
mov cx,0AH
call divdw
```

结果：(dx)=0001H，(ax)=86A0H，(cx)=0

提示

给出一个公式：

X：被除数，范围：[0,FFFFFFFF]

N：除数，范围：[0,FFFF]

H：X 高 16 位，范围：[0,FFFF]

L：X 低 16 位，范围：[0,FFFF]

int()：描述性运算符，取商，比如，int(38/10)=3

rem()：描述性运算符，取余数，比如，rem(38/10)=8

公式：X/N = int(H/N)*65536 +[rem(H/N)*65536+L]/N

这个公式将可能产生溢出的除法运算：X/N，转变为多个不会产生溢出的除法运算。公式中，等号右边的所有除法运算都可以用 div 指令来做，肯定不会导致除法溢出。

(关于这个公式的推导，有兴趣的读者请参看附注 5。)

3. 数值显示

问题

编程，将 data 段中的数据以十进制的形式显示出来。

```
data segment
  dw 123,12666,1,8,3,38
data ends
```

这些数据在内存中都是二进制信息，标记了数值的大小。要把它们显示到屏幕上，成为我们能够读懂的信息，需要进行信息的转化。比如，数值 12666，在机器中存储为二进制信息：0011000101111010B(317AH)，计算机可以理解它。而要在显示器上读到可以理解的数值 12666，我们看到的应该是一串字符：“12666”。由于显卡遵循的是 ASCII 编码，为了让我们能在显示器上看到这串字符，它在机器中应以 ASCII 码的形式存储为：31H、32H、36H、36H、36H(字符“0”~“9”对应的 ASCII 码为 30H~39H)。

通过上面的分析可以看到，在概念世界中，有一个抽象的数据 12666，它表示了一个数值的大小。在现实世界中它可以有多种表示形式，可以在电子机器中以高低电平(二进制)的形式存储，也可以在纸上、黑板上、屏幕上以人类的语言“12666”来书写。现在，我们面临的问题就是，要将同一抽象的数据，从一种表示形式转化为另一种表示形式。

可见，要将数据用十进制形式显示到屏幕上，要进行两步工作：

(1) 将用二进制信息存储的数据转变为十进制形式的字符串；
(2) 显示十进制形式的字符串。

第二步我们在本次实验的第一个子程序中已经实现，在这里只要调用一下 show_str 即可。我们来讨论第一步，因为将二进制信息转变为十进制形式的字符串也是经常要用到的功能，我们应该为它编写一个通用的子程序。

子程序描述

名称：dtoc
功能：将 word 型数据转变为表示十进制数的字符串，字符串以 0 为结尾符。
参数：(ax)=word 型数据
　　ds:si指向字符串的首地址
返回：无
应用举例：编程，将数据 12666 以十进制的形式在屏幕的 8 行 3 列，用绿色显示出来。在显示时我们调用本次实验中的第一个子程序 show_str。

```
assume cs:code

data segment
  db 10 dup (0)
data ends
```

```
code segment
  start:mov ax,12666
        mov bx,data
        mov ds,bx
        mov si,0
        call dtoc

        mov dh,8
        mov dl,3
        mov cl,2
        call show_str
        ⋮
code ends
end start
```

提示

下面我们对这个问题进行一下简单的分析。

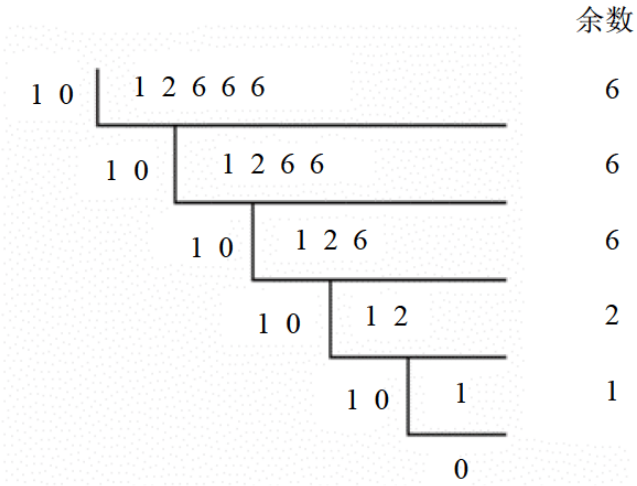
(1) 要得到字符串“12666”，就是要得到一列表示该字符串的 ASCII 码：31H、32H、36H、36H、36H。

十进制数码字符对应的 ASCII 码 = 十进制数码值 + 30H。

要得到表示十进制数的字符串，先求十进制数每位的值。

例：对于 12666，先求得每位的值：1、2、6、6、6。再将这些数分别加上 30H，便得到了表示 12666 的 ASCII 码串：31H、32H、36H、36H、36H。

(2) 那么，怎样得到每位的值呢？采用下面的方法：



可见，用 10 除 12666，共除 5 次，记下每次的余数，就得到了每位的值。

(3) 综合以上分析，可得出处理过程如下。

用 12666 除以 10，循环 5 次，记下每次的余数；将每次的余数分别加 30H，便得到了表示十进制数的 ASCII 码串。如下：

	余数	+ 30H	ASCII 码串	字符串
10 | 12666	6		36H	‘6’
10 | 1266	6		36H	‘6’
10 | 126	6		36H	‘6’
10 | 12	2		32H	‘2’
10 | 1	1		31H	‘1’
0				

(4)　对(3)的质疑。

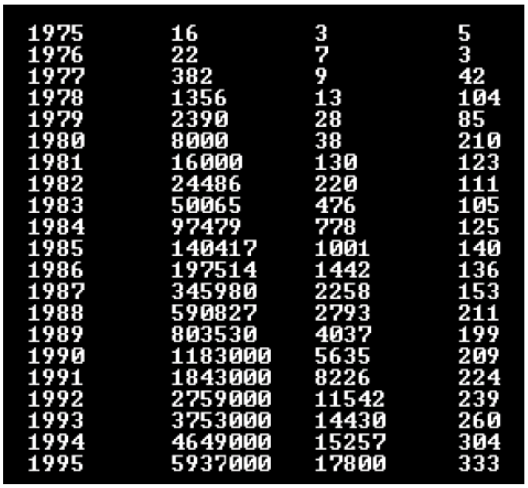
在已知数据是 12666 的情况下，知道进行 5 次循环。可在实际问题中，数据的值是多少程序员并不知道，也就是说，程序员不能事先确定循环次数。

那么，如何确定数据各位的值已经全部求出了呢？我们可以看出，只要是除到商为 0，各位的值就已经全部求出。可以使用 jcxz 指令来实现相关的功能。

课程设计 1

在整个课程中，我们一共有两个课程设计，编写两个比较综合的程序，这是第一个。

任务：将实验 7 中的 Power idea 公司的数据按照图 10.2 所示的格式在屏幕上显示出来。



1975	16	3	5
1976	22	7	3
1977	382	9	42
1978	1356	13	104
1979	2390	28	85
1980	8000	38	210
1981	16000	130	123
1982	24486	220	111
1983	50065	476	105
1984	97479	778	125
1985	140417	1001	140
1986	197514	1442	136
1987	345980	2258	153
1988	590827	2793	211
1989	803530	4037	199
1990	1183000	5635	209
1991	1843000	8226	224
1992	2759000	11542	239
1993	3753000	14430	260
1994	4649000	15257	304
1995	5937000	17800	333

图 10.2　Power idea 公司的数据

在这个程序中，要用到我们前面学到的几乎所有的知识，注意选择适当的寻址方式和

相关子程序的设计和应用。

另外，要注意，因为程序要显示的数据有些已经大于 65535，应该编写一个新的数据到字符串转化的子程序，完成 dword 型数据到字符串的转化，说明如下。

名称：dtoc
功能：将 dword 型数转变为表示十进制数的字符串，字符串以 0 为结尾符。
参数：(ax)=dword 型数据的低 16 位
(dx)=dword型数据的高16位
ds:si指向字符串的首地址
返回：无

在这个子程序中要注意除法溢出的问题，可以用我们在实验 10 中设计的子程序 divdw 来解决。

第 11 章　标志寄存器

CPU 内部的寄存器中，有一种特殊的寄存器(对于不同的处理机，个数和结构都可能不同)具有以下 3 种作用。

(1) 用来存储相关指令的某些执行结果；
(2) 用来为 CPU 执行相关指令提供行为依据；
(3) 用来控制 CPU 的相关工作方式。

这种特殊的寄存器在 8086CPU 中，被称为标志寄存器。8086CPU 的标志寄存器有 16 位，其中存储的信息通常被称为程序状态字(PSW)。我们已经使用过 8086CPU 的 ax、bx、cx、dx、si、di、bp、sp、IP、cs、ss、ds、es 等 13 个寄存器了，本章中的标志寄存器(以下简称为 flag)是我们要学习的最后一个寄存器。

flag 和其他寄存器不一样，其他寄存器是用来存放数据的，都是整个寄存器具有一个含义。而 flag 寄存器是按位起作用的，也就是说，它的每一位都有专门的含义，记录特定的信息。

8086CPU 的 flag 寄存器的结构如图 11.1 所示。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

图 11.1　flag 寄存器各位示意图

flag 的 1、3、5、12、13、14、15 位在 8086CPU 中没有使用，不具有任何含义。而 0、2、4、6、7、8、9、10、11 位都具有特殊的含义。

在这一章中，我们学习标志寄存器中的 CF、PF、ZF、SF、OF、DF 标志位，以及一些与其相关的典型指令。

11.1　ZF 标志

flag 的第 6 位是 ZF，零标志位。它记录相关指令执行后，其结果是否为 0。如果结果为 0，那么 zf=1；如果结果不为 0，那么 zf=0。

比如，指令：

```
mov ax,1
sub ax,1
```

执行后，结果为 0，则 zf=1。

```
mov ax,2
sub ax,1
```

执行后，结果不为 0，则 zf=0。

对于 zf 的值，我们可以这样来看，zf 标记相关指令的计算结果是否为 0，如果为 0，则 zf 要记录下“是 0”这样的肯定信息。在计算机中 1 表示逻辑真，表示肯定，所以当结果为 0 的时候 zf=1，表示“结果是 0”。如果结果不为 0，则 zf 要记录下“不是 0”这样的否定信息。在计算机中 0 表示逻辑假，表示否定，所以当结果不为 0 的时候 zf=0，表示“结果不是 0”。

比如，指令：

```
mov ax,1
and ax,0
```

执行后，结果为 0，则 zf=1，表示“结果是 0”。

```
mov ax,1
or ax,0
```

执行后，结果不为 0，则 zf=0，表示“结果非 0”。

注意，在 8086CPU 的指令集中，有的指令的执行是影响标志寄存器的，比如，add、sub、mul、div、inc、or、and 等，它们大都是运算指令(进行逻辑或算术运算)；有的指令的执行对标志寄存器没有影响，比如，mov、push、pop 等，它们大都是传送指令。在使用一条指令的时候，要注意这条指令的全部功能，其中包括，执行结果对标志寄存器的哪些标志位造成影响。

11.2 PF 标志

flag 的第 2 位是 PF，奇偶标志位。它记录相关指令执行后，其结果的所有 bit 位中 1 的个数是否为偶数。如果 1 的个数为偶数，pf=1，如果为奇数，那么 pf=0。

比如，指令：

```
mov al,1
add al,10
```

执行后，结果为 00001011B，其中有 3(奇数)个 1，则 pf=0；

```
mov al,1
or al,2
```

执行后，结果为 00000011B，其中有 2(偶数)个 1，则 pf=1；

```
sub al,al
```

执行后，结果为 00000000B，其中有 0(偶数)个 1，则 pf=1。

11.3　SF 标志

flag 的第 7 位是 SF，符号标志位。它记录相关指令执行后，其结果是否为负。如果结果为负，sf=1；如果非负，sf=0。

计算机中通常用补码来表示有符号数据。计算机中的一个数据可以看作是有符号数，也可以看成是无符号数。比如：

00000001B，可以看作为无符号数 1，或有符号数+1；
10000001B，可以看作为无符号数 129，也可以看作有符号数-127。

这也就是说，对于同一个二进制数据，计算机可以将它当作无符号数据来运算，也可以当作有符号数据来运算。比如：

```
mov al,10000001B
add al,1
```

结果：(al)=10000010B。

可以将 add 指令进行的运算当作无符号数的运算，那么 add 指令相当于计算 129+1，结果为 130(10000010B)；也可以将 add 指令进行的运算当作有符号数的运算，那么 add 指令相当于计算-127+1，结果为-126(10000010B)。

不管我们如何看待，CPU 在执行 add 等指令的时候，就已经包含了两种含义，也将得到用同一种信息来记录的两种结果。关键在于我们的程序需要哪一种结果。

SF 标志，就是 CPU 对有符号数运算结果的一种记录，它记录数据的正负。在我们将数据当作有符号数来运算的时候，可以通过它来得知结果的正负。如果我们将数据当作无符号数来运算，SF 的值则没有意义，虽然相关的指令影响了它的值。

这也就是说，CPU 在执行 add 等指令时，是必然要影响到 SF 标志位的值的。至于我们需不需要这种影响，那就看我们如何看待指令所进行的运算了。

比如：

```
mov al,10000001B
add al,1
```

执行后，结果为 10000010B，sf=1，表示：如果指令进行的是有符号数运算，那么结果为负；

```
mov al,10000001B
add al,01111111B
```

执行后，结果为 0，sf=0，表示：如果指令进行的是有符号数运算，那么结果为非负。

某些指令将影响标志寄存器中的多个标记位，这些被影响的标记位比较全面地记录了指令的执行结果，为相关的处理提供了所需的依据。比如指令 sub al,al 执行后，ZF、PF、SF 等标志位都要受到影响，它们分别为：1、1、0。

检测点 11.1

写出下面每条指令执行后，ZF、PF、SF 等标志位的值。

```
sub al,al       ZF=_____PF=_____SF=_____
mov al,1        ZF=_____PF=_____SF=_____
push ax         ZF=_____PF=_____SF=_____
pop bx          ZF=_____PF=_____SF=_____
add al,bl       ZF=_____PF=_____SF=_____
add al,10       ZF=_____PF=_____SF=_____
mul al          ZF=_____PF=_____SF=_____
```

11.4 CF 标志

flag 的第 0 位是 CF，进位标志位。一般情况下，在进行无符号数运算的时候，它记录了运算结果的最高有效位向更高位的进位值，或从更高位的借位值。

对于位数为 N 的无符号数来说，其对应的二进制信息的最高位，即第 N-1 位，就是它的最高有效位，而假想存在的第 N 位，就是相对于最高有效位的更高位，如图 11.2 所示。

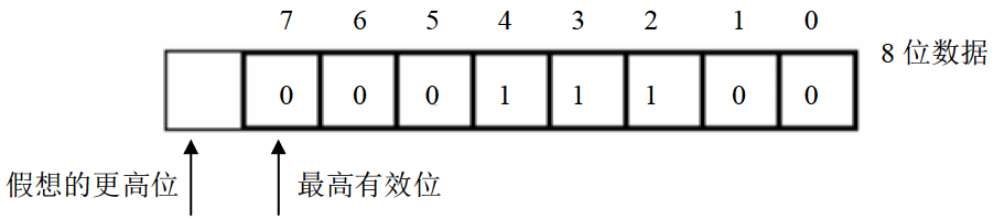


图 11.2 更高位

我们知道，当两个数据相加的时候，有可能产生从最高有效位向更高位的进位。比如，两个 8 位数据：98H+98H，将产生进位。由于这个进位值在 8 位数中无法保存，我们在前面的课程中，就只是简单地说这个进位值丢失了。其实 CPU 在运算的时候，并不丢弃这个进位值，而是记录在一个特殊的寄存器的某一位上。8086CPU 就用 flag 的 CF 位来记录这个进位值。比如，下面的指令：

```
mov al,98H
add al,al     ;执行后：(al)=30H，CF=1，CF 记录了从最高有效位向更高位的进位值
```

```
add al,al     ;执行后：(al)=60H，CF=0，CF 记录了从最高有效位向更高位的进位值
```

而当两个数据做减法的时候，有可能向更高位借位。比如，两个 8 位数据：97H-98H，将产生借位，借位后，相当于计算 197H-98H。而 flag 的 CF 位也可以用来记录这个借位值。比如，下面的指令：

```
mov al,97H
sub al,98H      ;执行后：(al)=FFH，CF=1，CF 记录了向更高位的借位值
sub al,al       ;执行后：(al)=0，CF=0，CF 记录了向更高位的借位值
```

11.5　OF 标志

我们先来谈谈溢出的问题。在进行有符号数运算的时候，如结果超过了机器所能表示的范围称为溢出。

那么，什么是机器所能表示的范围呢？

比如说，指令运算的结果用 8 位寄存器或内存单元来存放，比如，add al,3，那么对于 8 位的有符号数据，机器所能表示的范围就是-128~127。同理，对于 16 位有符号数据，机器所能表示的范围是-32768~32767。

如果运算结果超出了机器所能表达的范围，将产生溢出。

注意，这里所讲的溢出，只是对有符号数运算而言。下面我们看两个溢出的例子。

```
mov al,98
add al,99
```

执行后将产生溢出。因为 add al,99 进行的有符号数运算是：

(al)=(al)+99=98+99=197。

而结果 197 超出了机器所能表示的 8 位有符号数的范围：-128~127。

```
mov al,0F0H    ;F0H，为有符号数-16 的补码
add al,088H    ;88H，为有符号数-120 的补码
```

执行后，将产生溢出。因为 add al,088H 进行的有符号数运算是：

(al)=(al)+(-120)=(-16)+(-120)=-136

而结果-136 超出了机器所能表示的 8 位有符号数的范围：-128~127。

如果在进行有符号数运算时发生溢出，那么运算的结果将不正确。就上面的两个例子来说：

```
mov al,98
add al,99
```

add 指令运算的结果是(al)=0C5H，因为进行的是有符号数运算，所以 al 中存储的是有符号数，而 C5H 是有符号数-59 的补码。如果我们用 add 指令进行的是有符号数运算，则 98+99=-59 这样的结果让人无法接受。造成这种情况的原因，就是实际的结果 197，作为一个有符号数，在 8 位寄存器 al 中存放不下。

同样，对于：

```
mov al,0F0H    ;F0H，为有符号数-16 的补码
add al,088H    ;88H，为有符号数-120 的补码
```

add 指令运算的结果是(al)=78H，因为进行的是有符号数运算，所以 al 中存储的是有符号数，而 78H 表示有符号数 120。如果我们用 add 指令进行的是有符号数运算，则-16-120=120 这样的结果显然不正确。造成这种情况的原因，就是实际的结果-136，作为一个有符号数，在 8 位寄存器 al 中存放不下。

由于在进行有符号数运算时，可能发生溢出而造成结果的错误。则 CPU 需要对指令执行后是否产生溢出进行记录。

flag 的第 11 位是 OF，溢出标志位。一般情况下，OF 记录了有符号数运算的结果是否发生了溢出。如果发生溢出，OF=1；如果没有，OF=0。

一定要注意 CF 和 OF 的区别：CF 是对无符号数运算有意义的标志位，而 OF 是对有符号数运算有意义的标志位。比如：

```
mov al,98
add al,99
```

add 指令执行后：CF=0，OF=1。前面我们讲过，CPU 在执行 add 等指令的时候，就包含了两种含义：无符号数运算和有符号数运算。对于无符号数运算，CPU 用 CF 位来记录是否产生了进位；对于有符号数运算，CPU 用 OF 位来记录是否产生了溢出，当然，还要用 SF 位来记录结果的符号。对于无符号数运算，98+99 没有进位，CF=0；对于有符号数运算，98+99 发生溢出，OF=1。

```
mov al,0F0H
add al,88H
```

add 指令执行后：CF=1，OF=1。对于无符号数运算，0F0H+88H 有进位，CF=1；对于有符号数运算，0F0H+88H 发生溢出，OF=1。

```
mov al,0F0H
add al,78H
```

add 指令执行后：CF=1，OF=0。对于无符号运算，0F0H+78H 有进位，CF=1；对于有符号数运算，0F0H+78H 不发生溢出，OF=0。

我们可以看出，CF 和 OF 所表示的进位和溢出，是分别对无符号数和有符号数运算而言的，它们之间没有任何关系。

检测点 11.2

写出下面每条指令执行后，ZF、PF、SF、CF、OF 等标志位的值。

```
                CF      OF      SF      ZF      PF
sub al,al
mov al,10H
add al,90H
mov al,80H
add al,80H
mov al,0FCH
add al,05H
mov al,7DH
add al,0BH
```

11.6　adc 指令

adc 是带进位加法指令，它利用了 CF 位上记录的进位值。

指令格式：adc 操作对象 1，操作对象 2
功能：操作对象 1 = 操作对象 1 + 操作对象 2 + CF
比如指令 adc ax,bx 实现的功能是：(ax)=(ax)+(bx)+ CF

例：

```
mov ax,2
mov bx,1
sub bx,ax
adc ax,1
```

执行后，(ax)=4。adc 执行时，相当于计算：(ax)+1+CF=2+1+1=4。

```
mov ax,1
add ax,ax
adc ax,3
```

执行后，(ax)=5。adc 执行时，相当于计算：(ax)+3+CF=2+3+0=5。

```
mov al,98H
add al,al
adc al,3
```

执行后，(al)=34H。adc 执行时，相当于计算：(al)+3+CF=30H+3+1=34H。

可以看出，adc 指令比 add 指令多加了一个 CF 位的值。

为什么要加上 CF 的值呢？CPU 为什么要提供这样一条指令呢？

先来看一下 CF 的值的含义。在执行 adc 指令的时候加上的 CF 的值的含义，是由 adc 指令前面的指令决定的，也就是说，关键在于所加上的 CF 值是被什么指令设置的。显然，如果 CF 的值是被 sub 指令设置的，那么它的含义就是借位值；如果是被 add 指令设置的，那么它的含义就是进位值。我们来看一下两个数据：0198H 和 0183H 如何相加的：

```
    01  98
+   01  83
      1
-----------
    03  1B
```

可以看出，加法可以分两步来进行：①低位相加；②高位相加再加上低位相加产生的进位值。

下面的指令和 add ax, bx 具有相同的结果：

```
add al,bl
adc ah,bh
```

看来 CPU 提供 adc 指令的目的，就是来进行加法的第二步运算的。adc 指令和 add 指令相配合就可以对更大的数据进行加法运算。我们来看一个例子：

编程，计算 1EF000H+201000H，结果放在 ax(高 16 位)和 bx(低 16 位)中。

因为两个数据的位数都大于 16，用 add 指令无法进行计算。我们将计算分两步进行，先将低 16 位相加，然后将高 16 位和进位值相加。程序如下。

```
mov ax,001EH
mov bx,0F000H
add bx,1000H
adc ax,0020H
```

adc 指令执行后，也可能产生进位值，所以也会对 CF 位进行设置。由于有这样的功能，我们就可以对任意大的数据进行加法运算。看一个例子：

编程，计算 1EF0001000H+2010001EF0H，结果放在 ax(最高 16 位)，bx(次高 16 位)，cx(低 16 位)中。

计算分 3 步进行：

(1) 先将低 16 位相加，完成后，CF 中记录本次相加的进位值；

(2) 再将次高 16 位和 CF(来自低 16 位的进位值)相加，完成后，CF 中记录本次相加的进位值；

(3) 最后高 16 位和 CF(来自次高 16 位的进位值)相加，完成后，CF 中记录本次相加的进位值。

程序如下。

```
mov ax,001EH
mov bx,0F000H
mov cx,1000H
add cx,1EF0H
adc bx,1000H
adc ax,0020H
```

下面编写一个子程序，对两个 128 位数据进行相加。

名称：add128

功能：两个 128 位数据进行相加。

参数：ds:si 指向存储第一个数的内存空间，因数据为 128 位，所以需要 8 个字单元，由低地址单元到高地址单元依次存放 128 位数据由低到高的各个字。运算结果存储在第一个数的存储空间中。

ds:di 指向存储第二个数的内存空间。

程序如下。

```
add128: push ax
        push cx
        push si
        push di

        sub ax,ax           ;将 CF 设置为 0

        mov cx,8
      s: mov ax,[si]
        adc ax,[di]
        mov [si],ax
        inc si
        inc si
        inc di
        inc di
        loop s

        pop di
        pop si
        pop cx
        pop ax
        ret
```

inc 和 loop 指令不影响 CF 位，思考一下，上面的程序中，能不能将 4 个 inc 指令，用

```
add si,2
add di,2
```

来取代？

11.7 sbb 指令

sbb 是带借位减法指令，它利用了 CF 位上记录的借位值。

指令格式：sbb 操作对象 1，操作对象 2
功能：操作对象 1=操作对象 1-操作对象 2-CF
比如指令 sbb ax,bx 实现的功能是：(ax)=(ax)-(bx)-CF

sbb 指令执行后，将对 CF 进行设置。利用 sbb 指令可以对任意大的数据进行减法运算。比如，计算 003E1000H-00202000H，结果放在 ax,bx 中，程序如下：

```
mov bx,1000H
mov ax,003EH
sub bx,2000H
sbb ax,0020H
```

sbb 和 adc 是基于同样的思想设计的两条指令，在应用思路上和 adc 类似。在这里，我们就不再进行过多的讨论。通过学习这两条指令，我们可以进一步领会一下标志寄存器 CF 位的作用和意义。

11.8 cmp 指令

cmp 是比较指令，cmp 的功能相当于减法指令，只是不保存结果。cmp 指令执行后，将对标志寄存器产生影响。其他相关指令通过识别这些被影响的标志寄存器位来得知比较结果。

cmp 指令格式：cmp 操作对象 1，操作对象 2
功能：计算操作对象 1-操作对象 2 但并不保存结果，仅仅根据计算结果对标志寄存器进行设置。
比如，指令 cmp ax,ax，做(ax)-(ax)的运算，结果为 0，但并不在 ax 中保存，仅影响 flag 的相关各位。指令执行后：zf=1，pf=1，sf=0，cf=0，of=0。

下面的指令：

```
mov ax,8
mov bx,3
cmp ax,bx
```

执行后：(ax)=8，zf=0，pf=1，sf=0，cf=0，of=0。

其实，我们通过 cmp 指令执行后，相关标志位的值就可以看出比较的结果。

```
cmp ax,bx
```

如果(ax)＝(bx) 则(ax)−(bx)＝0，所以：zf=1；
如果(ax)≠(bx) 则(ax)−(bx)≠ 0，所以：zf=0；
如果(ax)＜(bx) 则(ax)−(bx)将产生借位，所以：cf=1；
如果(ax)≥(bx) 则(ax)−(bx)不必借位，所以：cf=0；
如果(ax)＞(bx) 则(ax)−(bx)既不必借位，结果又不为 0，所以：cf=0 并且 zf=0；
如果(ax)≤(bx) 则(ax)−(bx)既可能借位，结果可能为 0，所以：cf=1 或 zf=1。

现在我们可以看出比较指令的设计思路，即：通过做减法运算，影响标志寄存器，标志寄存器的相关位记录了比较的结果。反过来看上面的例子。

指令 cmp ax,bx 的逻辑含义是比较 ax 和 bx 中的值，如果执行后：

zf=1，说明(ax)＝(bx)
zf=0，说明(ax)≠(bx)
cf=1，说明(ax)＜(bx)
cf=0，说明(ax)≥(bx)
cf=0 并且 zf=0，说明(ax)＞(bx)
cf=1 或 zf=1，说明(ax)≤(bx)

同 add、sub 指令一样，CPU 在执行 cmp 指令的时候，也包含两种含义：进行无符号数运算和进行有符号数运算。所以利用 cmp 指令可以对无符号数进行比较，也可以对有符号数进行比较。上面所讲的是用 cmp 进行无符号数比较时，相关标志位对比较结果的记录。下面我们再来看一下如果用 cmp 来进行有符号数比较时，CPU 用哪些标志位对比较结果进行记录。我们以 cmp ah,bh 为例进行说明。

```
cmp ah,bh
```

如果(ah)＝(bh) 则(ah)−(bh)＝0，所以：zf=1；
如果(ah)≠(bh) 则(ah)−(bh)≠0，所以：zf=0；

所以，根据 cmp 指令执行后 zf 的值，就可以知道两个数据是否相等。

我们继续看，如果(ah)＜(bh)则可能发生什么情况呢？

对于有符号数运算，在(ah)＜(bh)情况下，(ah)−(bh)显然可能引起 sf=1，即结果为负。

比如：

(ah)=1，(bh)=2；则(ah)−(bh)=0FFH，0FFH 为−1 的补码，因为结果为负，所以 sf=1。
(ah)=0FEH，(bh)=0FFH；则(ah)−(bh)=−2−(−1)=0FFH，因为结果为负，所以 sf=1。

通过上面的例子，我们是不是可以得到这样的结论：“cmp 操作对象 1,操作对象 2”指令执行后，sf=1，就说明操作对象 1＜操作对象 2？

当然不是。

我们再看两个例子。

(ah)=22H，(bh)=0A0H；则(ah)-(bh)=34-(-96)=82H，82H 是-126 的补码
所以 sf=1

这里虽然 sf=1，但是并不能说明(ah)＜(bh)因为显然 34>-96。

两个有符号数 A 和 B 相减，得到的是负数，那么可以肯定 A<B，这个思路没有错误，关键在于我们根据什么来断定得到的是一个负数。CPU 将 cmp 指令得到的结果记录在 flag 的相关标志位中。我们可以根据指令执行后，相关标志位的值来判断比较的结果。单纯地考查 sf 的值不可能知道结果的正负。因为 sf 记录的只是可以在计算机中存放的相应位数的结果的正负。比如 add ah,al 执行后，sf 记录的是 ah 中的 8 位二进制信息所表示的数据的正负。cmp ah,bh 执行后，sf 记录的是(ah)-(bh)所得到的 8 位结果数据的正负，虽然这个结果没有在我们能够使用的寄存器或内存单元中保存，但是在指令执行的过程中，它暂存在 CPU 内部的暂存器中。

所得到的相应结果的正负，并不能说明，运算所应该得到的结果的正负。这是因为在运算的过程中可能发生溢出。如果有这样的情况发生，那么，sf 的值就不能说明任何问题。比如：

```
mov ah,22H
mov bh,0A0H
sub ah,bh
```

结果 sf=1，运算实际得到的结果是(ah)=82H，但是在逻辑上，运算所应该得到的结果是：34-(-96)=130。就是因为 130 这个结果作为一个有符号数超出了-128~127 这个范围，在 ah 中不能表示，而 ah 中的结果被 CPU 当作有符号数解释为-126。而 sf 被用来记录这个实际结果的正负，所以 sf=1。但 sf=1 不能说明在逻辑上，运算所得的正确结果的正负。

又比如：

```
mov ah,08AH
mov bh,070h
cmp ah,bh
```

结果 sf=0，运算(ah)-(bh)实际得到的结果是 1AH，但是在逻辑上，运算所应该得到的结果是：(-118)-112=-230。sf 记录实际结果的正负，所以 sf=0。但 sf=0 不能说明在逻辑上，运算所得的正确结果。

但是逻辑上的结果的正负，才是 cmp 指令所求的真正结果，因为我们就是要靠它得到两个操作对象的比较信息。所以 cmp 指令所作的比较结果，不是仅仅靠 sf 就能记录的，因为它只能记录实际结果的正负。

我们考虑一下，两种结果之间的关系，实际结果的正负，和逻辑上真正结果的正负，它们之间有多大的距离呢？从上面的分析中，我们知道，实际结果的正负，之所以不能说

明逻辑上真正结果的正负，关键的原因在于发生了溢出。如果没有溢出发生的话，那么，实际结果的正负和逻辑上真正结果的正负就一致了。

所以，我们应该在考查 sf(得知实际结果的正负)的同时考查 of(得知有没有溢出)，就可以得知逻辑上真正结果的正负，同时就可以知道比较的结果。

下面，我们以 cmp ah,bh 为例，总结一下 CPU 执行 cmp 指令后，sf 和 of 的值是如何来说明比较的结果的。

(1) 如果 sf=1，而 of=0

of=0，说明没有溢出，逻辑上真正结果的正负=实际结果的正负；

因 sf=1，实际结果为负，所以逻辑上真正的结果为负，所以(ah)＜(bh)。

(2) 如果 sf=1，而 of=1：

of=1，说明有溢出，逻辑上真正结果的正负≠实际结果的正负；

因 sf=1，实际结果为负。

实际结果为负，而又有溢出，这说明是由于溢出导致了实际结果为负，简单分析一下，就可以看出，**如果因为溢出导致了实际结果为负，那么逻辑上真正的结果必然为正。**

这样，sf=1，of=1，说明了(ah)＞(bh)。

(3) 如果 sf=0，而 of=1

of=1，说明有溢出，逻辑上真正结果的正负≠实际结果的正负；

因 sf=0，实际结果非负。而 of=1 说明有溢出，则结果非 0，所以，实际结果为正。

实际结果为正，而又有溢出，这说明是由于溢出导致了实际结果非负，简单分析一下，就可以看出，**如果因为溢出导致了实际结果为正，那么逻辑上真正的结果必然为负。**

这样，sf=0，of=1，说明了(ah)＜(bh)。

(4) 如果 sf=0，而 of=0

of=0，说明没有溢出，逻辑上真正结果的正负=实际结果的正负；

因 sf=0，实际结果非负，所以逻辑上真正的结果非负，所以(ah)≥(bh)。

上面，我们深入讨论了 cmp 指令在进行有符号数和无符号数比较时，对 flag 相关标志位的影响，和 CPU 如何通过相关的标志位来表示比较的结果。在学习中，要注意领会 8086CPU 这种工作机制的设计思想。实际上，这种设计思想对于各种处理机来说是普遍的。

下面的内容中我们将学习一些根据 cmp 指令的比较结果(即 cmp 指令执行后，相关标志位的值)进行工作的指令。

11.9　检测比较结果的条件转移指令

“转移”指的是它能够修改 IP，而“条件”指的是它可以根据某种条件，决定是否修改 IP。

比如，jcxz 就是一个条件转移指令，它可以检测 cx 中的数值，如果(cx)=0，就修改 IP，否则什么也不做。所有条件转移指令的转移位移都是[−128,127]。

除了 jcxz 之外，CPU 还提供了其他条件转移指令，大多数条件转移指令都检测标志寄存器的相关标志位，根据检测的结果来决定是否修改 IP。它们检测的是哪些标志位呢？就是被 cmp 指令影响的那些，表示比较结果的标志位。这些条件转移指令通常都和 cmp 相配合使用，就好像 call 和 ret 指令通常相配合使用一样。

因为 cmp 指令可以同时进行两种比较，无符号数比较和有符号数比较，所以根据 cmp 指令的比较结果进行转移的指令也分为两种，即根据无符号数的比较结果进行转移的条件转移指令(它们检测 zf、cf 的值)和根据有符号数的比较结果进行转移的条件转移指令(它们检测 sf、of 和 zf 的值)。

下面是常用的根据无符号数的比较结果进行转移的条件转移指令。

指令	含义	检测的相关标志位
je	等于则转移	zf=1
jne	不等于则转移	zf=0
jb	低于则转移	cf=1
jnb	不低于则转移	cf=0
ja	高于则转移	cf=0 且 zf=0
jna	不高于则转移	cf=1 或 zf=1

这些指令比较常用，它们都很好记忆，它们的第一个字母都是 j，表示 jump；后面的字母表示意义如下。

e：表示 equal
ne：表示 not equal
b：表示 below
nb：表示 not below
a：表示 above
na：表示 not above

注意观察一下它们所检测的标志位，都是 cmp 指令进行无符号数比较的时候，记录比较结果的标志位。比如 je，检测 zf 位，当 zf=1 的时候进行转移，如果在 je 前面使用了 cmp 指令，那么 je 对 zf 的检测，实际上就是间接地检测 cmp 的比较结果是否为两数相等。下面看一个例子。

编程实现如下功能：

如果(ah)=(bh)则(ah)=(ah)+(ah)，否则(ah)=(ah)+(bh)。

```
cmp ah,bh
je s
add ah,bh
jmp short ok
```

```
  s:add ah,ah
ok: …
```

上面的程序执行时，如果(ah)=(bh)，则 cmp ah,bh 使 zf=1，而 je 检测 zf 是否为 1，如果为 1，将转移到标号 s 处执行指令 add ah,ah。这也可以说，cmp 比较 ah、bh 后所得到的相等的结果使得 je 指令进行转移。从而很好地体现了 je 指令的逻辑含义，相等则转移。

虽然 je 的逻辑含义是“相等则转移”，但它进行的操作是 zf=1 时则转移。“相等则转移”这种逻辑含义，是通过和 cmp 指令配合使用来体现的，因为是 cmp 指令为“zf=1”赋予了“两数相等”的含义。

至于究竟在 je 之前使不使用 cmp 指令，在于我们的安排。je 检测的是 zf 位置，不管 je 前面是什么指令，只要 CPU 执行 je 指令时，zf=1，那么就会发生转移，比如：

```
    mov ax,0
    add ax,0
    je s
    inc ax
 s: inc ax
```

执行后，(ax)=1。add ax,0 使得 zf=1，所以 je 指令将进行转移。可在这个时候发生的转移的确不带有“相等则转移”的含义。因为此处的 je 指令检测到的 zf=1，不是由 cmp 等比较指令设置的，而是由 add 指令设置的，并不具有“两数相等”的含义。但无论“zf=1”的含义如何，是什么指令设置的，只要是 zf=1，就可以使得 je 指令发生转移。

CPU 提供了 cmp 指令，也提供了 je 等条件转移指令，如果将它们配合使用，可以实现根据比较结果进行转移的功能。但这只是“如果”，只是一种合理的建议，和事实上常用的方法。但究竟是否配合使用它们，完全是你自己的事情。这就好像 call 和 ret 指令的关系一样。

对于 jne、jb、jnb、ja、jna 等指令和 cmp 指令配合使用的思想和 je 相同，可以自己分析一下。

虽然我们分别讨论了 cmp 指令和与其比较结果相关的有条件转移指令，但是它们经常在一起配合使用。所以我们在联合应用它们的时候，不必再考虑 cmp 指令对相关标志位的影响和 je 等指令对相关标志位的检测。因为相关的标志位，只是为 cmp 和 je 等指令传递比较结果。我们可以直接考虑 cmp 和 je 等指令配合使用时，表现出来的逻辑含义。它们在联合使用的时候表现出来的功能有些像高级语言中的 IF 语句。

我们来看下面的一组程序。

data 段中的 8 个字节如下：

```
data segment
  db 8,11,8,1,8,5,63,38
data ends
```

(1) 编程，统计 data 段中数值为 8 的字节的个数，用 ax 保存统计结果。

编程思路：初始设置(ax)=0，然后用循环依次比较每个字节的值，找到一个和 8 相等的数就将 ax 的值加 1。程序如下。

```
      mov ax,data
      mov ds,ax
      mov bx,0                ;ds:bx 指向第一个字节
      mov ax,0                ;初始化累加器
      mov cx,8
  s:  cmp byte ptr [bx],8     ;和 8 进行比较
      jne  next               ;如果不相等转到 next，继续循环
      inc ax                  ;如果相等就将计数值加 1
next:inc bx
      loop s                  ;程序执行后：(ax)=3
```

这个程序也可以写成这样：

```
        mov ax,data
        mov ds,ax
        mov bx,0                ;ds:bx 指向第一个字节
        mov ax,0                ;初始化累加器
        mov cx,8
  s:    cmp byte ptr [bx],8     ;和 8 进行比较
        je  ok                  ;如果相等转到 ok
        jmp short next          ;如果不相等就转 next，继续循环
  ok:   inc ax                  ;如果相等就将计数值加 1
next:   inc bx
        loop s
```

比起第一个程序，它直接地遵循了“等于 8 则计数值加 1”的原则，用 je 指令检测等于 8 的情况，但是没有第一个程序精简。第一个程序用 jne 检测不等于 8 的情况，从而间接地检测等于 8 的情况。要注意在使用 cmp 和条件转移指令时的这种编程思想。

(2) 编程，统计 data 段中数值大于 8 的字节的个数，用 ax 保存统计结果。

编程思路：初始设置(ax)=0，然后用循环依次比较每个字节的值，找到一个大于 8 的就将 ax 的值加 1。程序如下。

```
        mov ax,data
        mov ds,ax
        mov ax,0                ;初始化累加器
        mov bx,0                ;ds:bx 指向第一个字节
```

```
        mov cx,8
s:      cmp byte ptr [bx],8     ;和 8 进行比较
        jna  next               ;如果不大于 8 转到 next，继续循环
        inc ax                  ;如果大于 8 就将计数值加 1
next:   inc bx
        loop s
```

程序执行后：(ax)=3

(3) 编程，统计 data 段中数值小于 8 的字节的个数，用 ax 保存统计结果。

编程思路：初始设置(ax)=0，然后用循环依次比较每个字节的值，找到一个小于 8 的就将 ax 的值加 1。程序如下。

```
        mov ax,data
        mov ds,ax
        mov ax,0                ;初始化累加器
        mov bx,0                ;ds:bx 指向第一个字节
        mov cx,8
s:      cmp byte ptr [bx],8     ;和 8 进行比较
        jnb  next               ;如果不小于 8 转到 next，继续循环
        inc ax                  ;如果小于 8 就将计数值加 1
next:   inc bx
        loop s
```

程序执行后：(ax)=2

上面讲解了根据无符号数的比较结果进行转移的条件转移指令。根据有符号数的比较结果进行转移的条件转移指令的工作原理和无符号的相同，只是检测了不同的标志位。我们在这里主要探讨的是 cmp、标志寄存器的相关位、条件转移指令三者配合应用的原理，这个原理具有普遍性，而不是逐条讲解条件转移指令。对这些指令感兴趣的读者可以查看相关的指令手册。

检测点 11.3

(1) 补全下面的程序，统计 F000:0 处 32 个字节中，大小在[32,128]的数据的个数。

```
mov ax,0f000h
mov ds,ax

mov bx,0
mov dx,0
mov cx,32
```

```
s:   mov al,[bx]
     cmp al,32
     ____________
    cmp al,128
     ____________
     inc dx
s0:  inc bx
     loop s
```

(2) 补全下面的程序，统计 F000:0 处 32 个字节中，大小在(32,128)的数据的个数。

```
     mov ax,0f000h
     mov ds,ax
     mov bx,0
     mov dx,0
     mov cx,32
s:   mov al,[bx]
     cmp al,32
     ____________
     cmp al,128
     ____________
     inc dx
s0:  inc bx
     loop s
```

11.10 DF 标志和串传送指令

flag 的第 10 位是 DF，方向标志位。在串处理指令中，控制每次操作后 si、di 的增减。

df=0 每次操作后 si、di 递增；
df=1 每次操作后 si、di 递减。

我们来看下面的一个串传送指令。

格式：movsb
功能：执行 movsb 指令相当于进行下面几步操作。

(1) ((es)*16+(di))=((ds)*16+(si))
(2) 如果 df=0 则： (si)=(si)+1
(di)=(di)+1

如果 df=1 则：　(si)=(si)−1
　　(di)=(di)−1

用汇编语法描述 movsb 的功能如下。

```
mov es:[di],byte ptr ds:[si]   ;8086 并不支持这样的指令，这里只是个描述

如果 df=0：
inc si
inc di

如果 df=1：
dec si
dec di
```

可以看出，movsb 的功能是将 ds:si 指向的内存单元中的字节送入 es:di 中，然后根据标志寄存器 df 位的值，将 si 和 di 递增或递减。

当然，也可以传送一个字，指令如下。

格式：movsw

movsw 的功能是将 ds:si 指向的内存字单元中的字送入 es:di 中，然后根据标志寄存器 df 位的值，将 si 和 di 递增 2 或递减 2。

用汇编语法描述 movsw 的功能如下。

```
mov es:[di],word ptr ds:[si]   ;8086 并不支持这样的指令，这里只是个描述

如果 df=0：
add si,2
add di,2

如果 df=1：
sub si,2
sub di,2
```

movsb 和 movsw 进行的是串传送操作中的一个步骤，一般来说，movsb 和 movsw 都和 rep 配合使用，格式如下：

```
rep movsb
```

用汇编语法来描述 rep movsb 的功能就是：

```
s:movsb
  loop s
```

可见，rep 的作用是根据 cx 的值，重复执行后面的串传送指令。由于每执行一次 movsb 指令 si 和 di 都会递增或递减指向后一个单元或前一个单元，则 rep movsb 就可以循环实现(cx)个字符的传送。

同理，也可以使用这样的指令：rep movsw。

相当于：

```
s:movsw
  loop s
```

由于 flag 的 df 位决定着串传送指令执行后，si 和 di 改变的方向，所以 CPU 应该提供相应的指令来对 df 位进行设置，从而使程序员能够决定传送的方向。

8086CPU 提供下面两条指令对 df 位进行设置。

cld 指令：将标志寄存器的 df 位置 0
std 指令：将标志寄存器的 df 位置 1

我们来看下面的两个程序。

(1) 编程，用串传送指令，将 data 段中的第一个字符串复制到它后面的空间中。

```
data segment
  db 'Welcome to masm!'
  db 16 dup (0)
data ends
```

我们分析一下，使用串传送指令进行数据的传送，需要给它提供一些必要的信息，它们是：

① 传送的原始位置：ds:si；
② 传送的目的位置：es:di；
③ 传送的长度：cx；
④ 传送的方向：df。

在这个问题中，这些信息如下。

① 传送的原始位置：data:0；
② 传送的目的位置：data:0010；
③ 传送的长度：16；
④ 传送的方向：因为正向传送(每次串传送指令执行后，si 和 di 递增)比较方便，所以设置 df=0。

好了，明确了这些信息之后，我们来编写程序：

```
mov ax,data
mov ds,ax
mov si,0            ;ds:si 指向 data:0
mov es,ax
mov di,16           ;es:di 指向 data:0010
mov cx,16           ;(cx)=16，rep 循环 16 次
cld                 ;设置 df=0，正向传送
rep movsb
```

(2) 编程，用串传送指令，将 F000H 段中的最后 16 个字符复制到 data 段中。

```
data segment
  db 16 dup (0)
data ends
```

我们还是先来看一下应该为串传送指令提供什么样的信息。

要传送的字符串位于 F000H 段的最后 16 个单元中，那么它的最后一个字符的位置：F000:FFFF，是显而易见的。可以将 ds:si 指向 F000H 段的最后一个单元，将 es:di 指向 data 段中的最后一个单元，然后逆向(即从高地址向低地址)传送 16 个字节即可。

① 传送的原始位置：F000:FFFF；
② 传送的目的位置：data:000F；
③ 传送的长度：16；
④ 传送的方向：因为逆向传送(每次串传送指令执行后，si 和 di 递减)比较方便，所以设置 df=1。

程序如下。

```
mov ax,0f000h
mov ds,ax
mov si,0ffffh          ;ds:si 指向 f000:ffff
mov ax,data
mov es,ax
mov di,15              ;es:di 指向 data:000F
mov cx,16              ;(cx)=16，rep 循环 16 次
std                    ;设置 df=1，逆向传送
rep movsb
```

11.11　pushf 和 popf

pushf 的功能是将标志寄存器的值压栈，而 popf 是从栈中弹出数据，送入标志寄存器中。

pushf 和 popf，为直接访问标志寄存器提供了一种方法。

检测点 11.4

下面的程序执行后：(ax)=?

```
mov ax,0
push ax
popf
mov ax,0fff0h
add ax,0010h
pushf
```

```
pop ax
and al,11000101B
and ah,00001000B
```

11.12 标志寄存器在 Debug 中的表示

在 Debug 中，标志寄存器是按照有意义的各个标志位单独表示的。在 Debug 中，我们可以看到下面的信息。

```
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=****  ES=****  SS=****  CS=****  IP=0100  NV UP EI PL NZ NA PO NC
                                             ↑  ↑     ↑  ↑     ↑  ↑
                                             OF DF    SF ZF    PF CF
```

下面列出 Debug 对我们已知的标志位的表示。

标志	值为 1 的标记	值为 0 的标记
of	OV	NV
sf	NG	PL
zf	ZR	NZ
pf	PE	PO
cf	CY	NC
df	DN	UP

实验 11 编写子程序

编写一个子程序，将包含任意字符，以 0 结尾的字符串中的小写字母转变成大写字母，描述如下。

名称：letterc
功能：将以 0 结尾的字符串中的小写字母转变成大写字母
参数：ds:si 指向字符串首地址

应用举例：

```
assume  cs:codesg

datasg segment
    db "Beginner's All-purpose Symbolic Instruction Code.",0
datasg ends
```

```
codesg segment

  begin: mov ax,datasg
         mov ds,ax
         mov si,0
         call letterc

         mov ax,4c00h
         int 21h

letterc:
            ⋮
codesg ends
end begin
```

注意需要进行转化的是字符串中的小写字母 a~z，而不是其他字符。

第 12 章　内　中　断

任何一个通用的 CPU，比如 8086，都具备一种能力，可以在执行完当前正在执行的指令之后，检测到从 CPU 外部发送过来的或内部产生的一种特殊信息，并且可以立即对所接收到的信息进行处理。这种特殊的信息，我们可以称其为：中断信息。中断的意思是指，CPU 不再接着(刚执行完的指令)向下执行，而是转去处理这个特殊信息。

注意，我们这里所说的中断信息，是为了便于理解而采用的一种逻辑上的说法。它是对几个具有先后顺序的硬件操作所产生的事件的统一描述。“中断信息”是要求 CPU 马上进行某种处理，并向所要进行的该种处理提供了必备的参数的通知信息。因为本书的内容不是微机原理与接口或组成原理，我们只能用一些便于理解的说法来描述一些比较复杂的机器工作原理，从而使学习者忽略一些和我们的学习重心无关的内容。但笔者又需要对这些问题有一个严谨的交代，所以，有了这些补充说明的文字。如果你不理解这些文字所讲的东西，就不必去理解了。

中断信息可以来自 CPU 的内部和外部，这一章中，我们主要讨论来自于 CPU 内部的中断信息。

12.1　内中断的产生

当 CPU 的内部有什么事情发生的时候，将产生需要马上处理的中断信息呢？对于 8086CPU，当 CPU 内部有下面的情况发生的时候，将产生相应的中断信息。

(1) 除法错误，比如，执行 div 指令产生的除法溢出；
(2) 单步执行；
(3) 执行 into 指令；
(4) 执行 int 指令。

我们现在不要去管这 4 种情况的具体含义，只要知道 CPU 内部有 4 种情况可以产生需要及时处理的中断信息即可。虽然我们现在并不很清楚，这 4 种情况到底是什么，但是有一点是很清楚的，即，它们是不同的信息。既然是不同的信息，就需要进行不同的处理。要进行不同的处理，CPU 首先要知道，所接收到的中断信息的来源。所以中断信息中必须包含识别来源的编码。8086CPU 用称为中断类型码的数据来标识中断信息的来源。中断类型码为一个字节型数据，可以表示 256 种中断信息的来源。以后，我们将产生中断信息的事件，即中断信息的来源，简称为中断源，上述的 4 种中断源，在 8086CPU 中的中断类型码如下。

(1) 除法错误：0
(2) 单步执行：1

(3)　执行 into 指令：4

(4)　执行 int 指令，该指令的格式为 int n，指令中的 n 为字节型立即数，是提供给 CPU 的中断类型码。

12.2　中断处理程序

CPU 收到中断信息后，需要对中断信息进行处理。而如何对中断信息进行处理，可以由我们编程决定。我们编写的，用来处理中断信息的程序被称为中断处理程序。一般来说，需要对不同的中断信息编写不同的处理程序。

CPU 在收到中断信息后，应该转去执行该中断信息的处理程序。我们知道，若要 8086CPU 执行某处的程序，就要将 CS:IP 指向它的入口(即程序第一条指令的地址)。可见首要的问题是，CPU 在收到中断信息后，如何根据中断信息确定其处理程序的入口。

CPU 的设计者必须在中断信息和其处理程序的入口地址之间建立某种联系，使得 CPU 根据中断信息可以找到要执行的处理程序。

我们知道，中断信息中包含有标识中断源的类型码。根据 CPU 的设计，中断类型码的作用就是用来定位中断处理程序。比如 CPU 根据中断类型码 4，就可以找到 4 号中断的处理程序。可随之而来的问题是，若要定位中断处理程序，需要知道它的段地址和偏移地址，而如何根据 8 位的中断类型码得到中断处理程序的段地址和偏移地址呢？

12.3　中断向量表

CPU 用 8 位的中断类型码通过中断向量表找到相应的中断处理程序的入口地址。那么什么是中断向量表呢？中断向量表就是中断向量的列表。那么什么又是中断向量呢？所谓中断向量，就是中断处理程序的入口地址。展开来讲，中断向量表，就是中断处理程序入口地址的列表。

中断向量表在内存中保存，其中存放着 256 个中断源所对应的中断处理程序的入口，如图 12.1 所示。

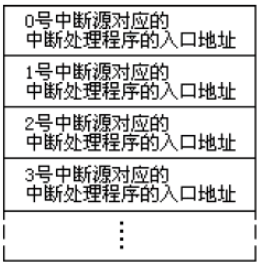


图 12.1　中断向量表

可以看到，CPU 只要知道了中断类型码，就可以将中断类型码作为中断向量表的表项号，定位相应的表项，从而得到中断处理程序的入口地址。

可见，CPU 用中断类型码，通过查找中断向量表，就可以得到中断处理程序的入口地址。在这个方案中，一个首要的问题是，CPU 如何找到中断向量表？现在，找到中断向量表成了通过中断类型码找到中断处理程序入口地址的先决条件。

中断向量表在内存中存放，对于 8086PC 机，中断向量表指定放在内存地址 0 处。从

内存 0000:0000 到 0000:03FF 的 1024 个单元中存放着中断向量表。能不能放在别处呢？不能，如果使用 8086CPU，中断向量表就必须放在 0000:0000~0000:03FF 单元中，这是规定，因为 8086CPU 就从这个地方读取中断向量表。

那么在中断向量表中，一个表项占多大的空间呢？一个表项存放一个中断向量，也就是一个中断处理程序的入口地址，对于 8086CPU，这个入口地址包括段地址和偏移地址，所以一个表项占两个字，高地址字存放段地址，低地址字存放偏移地址。

检测点 12.1

(1) 用 Debug 查看内存，情况如下：

```
0000:0000  68 10 A7 00 8B 01 70 00-16 00 9D 03 8B 01 70 00
```

则 3 号中断源对应的中断处理程序的入口地址为：_____。

(2) 存储 N 号中断源对应的中断处理程序入口的偏移地址的内存单元的地址为：_____。

存储 N 号中断源对应的中断处理程序入口的段地址的内存单元的地址为：_____。

12.4 中 断 过 程

从上面的讲解中，我们知道，可以用中断类型码，在中断向量表中找到中断处理程序的入口。找到这个入口地址的最终目的是用它设置 CS 和 IP，使 CPU 执行中断处理程序。用中断类型码找到中断向量，并用它设置 CS 和 IP，这个工作是由 CPU 的硬件自动完成的。CPU 硬件完成这个工作的过程被称为中断过程。

CPU 收到中断信息后，要对中断信息进行处理，首先将引发中断过程。硬件在完成中断过程后，CS:IP 将指向中断处理程序的入口，CPU 开始执行中断处理程序。

有一个问题需要考虑，CPU 在执行完中断处理程序后，应该返回原来的执行点继续执行下面的指令。所以在中断过程中，在设置 CS:IP 之前，还要将原来的 CS 和 IP 的值保存起来。在使用 call 指令调用子程序时有同样的问题，子程序执行后还要返回到原来的执行点继续执行，所以，call 指令先保存当前 CS 和 IP 的值，然后再设置 CS 和 IP。

下面是 8086CPU 在收到中断信息后，所引发的中断过程。

(1) (从中断信息中)取得中断类型码；

(2) 标志寄存器的值入栈(因为在中断过程中要改变标志寄存器的值，所以先将其保存在栈中)；

(3) 设置标志寄存器的第 8 位 TF 和第 9 位 IF 的值为 0(这一步的目的后面将介绍)；

(4) CS 的内容入栈；

(5) IP 的内容入栈；

(6) 从内存地址为中断类型码*4 和中断类型码*4+2 的两个字单元中读取中断处理程序的入口地址设置 IP 和 CS。

CPU 在收到中断信息之后，如果处理该中断信息，就完成一个由硬件自动执行的中断过程(程序员无法改变这个过程中所要做的工作)。中断过程的主要任务就是用中断类型码在中断向量表中找到中断处理程序的入口地址，设置 CS 和 IP。因为中断处理程序执行完成后，CPU 还要回过头来继续执行被中断的程序，所以要在设置 CS、IP 之前，先将它们的值保存起来。可以看到 CPU 将它们保存在栈中。我们注意到，在中断过程中还要做的一个工作就是设置标志寄存器的 TF、IF 位，对于这样做的目的，我们将在后面的内容和下一章中进行讨论。因为在执行完中断处理程序后，需要恢复在进入中断处理程序之前的 CPU 现场(某一时刻，CPU 中各个寄存器的值)。所以应该在修改标记寄存器之前，将它的值入栈保存。

我们更简洁地描述中断过程，如下：

(1) 取得中断类型码 N；
(2) pushf
(3) TF=0，IF=0
(4) push CS
(5) push IP
(6) (IP)=(N*4)，(CS)=(N*4+2)

在最后一步完成后，CPU 开始执行由程序员编写的中断处理程序。

12.5　中断处理程序和 iret 指令

由于 CPU 随时都可能检测到中断信息，也就是说，CPU 随时都可能执行中断处理程序，所以中断处理程序必须一直存储在内存某段空间之中。而中断处理程序的入口地址，即中断向量，必须存储在对应的中断向量表表项中。

中断处理程序的编写方法和子程序的比较相似，下面是常规的步骤：

(1) 保存用到的寄存器；
(2) 处理中断；
(3) 恢复用到的寄存器；
(4) 用 iret 指令返回。

iret 指令的功能用汇编语法描述为：

```
pop IP
pop CS
popf
```

iret 通常和硬件自动完成的中断过程配合使用。可以看到，在中断过程中，寄存器入栈的顺序是标志寄存器、CS、IP，而 iret 的出栈顺序是 IP、CS、标志寄存器，刚好和其相对应，实现了用执行中断处理程序前的 CPU 现场恢复标志寄存器和 CS、IP 的工作。iret 指令执行后，CPU 回到执行中断处理程序前的执行点继续执行程序。

12.6 除法错误中断的处理

下面的内容中，我们通过对 0 号中断，即除法错误中断的处理，来体会一下前面所讲的内容。

当 CPU 执行 div 等除法指令的时候，如果发生了除法溢出错误，将产生中断类型码为 0 的中断信息，CPU 将检测到这个信息，然后引发中断过程，转去执行 0 号中断所对应的中断处理程序。我们看一下下面程序的执行结果，如图 12.2 所示(不同的操作系统下显示可能不同)。

```
mov ax,1000h
mov bh,1
div bh
```

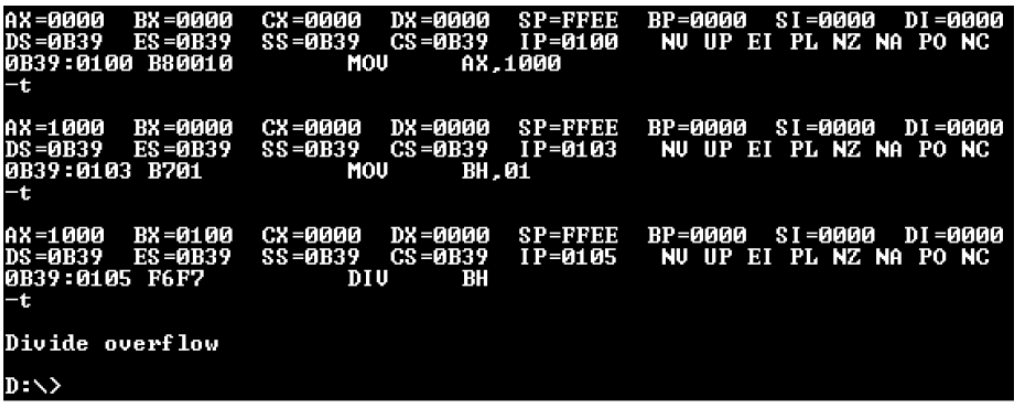


图 12.2 系统对 0 号中断的处理

可以看到，当 CPU 执行 div bh 时，发生了除法溢出错误，产生 0 号中断信息，从而引发中断过程，CPU 执行 0 号中断处理程序。我们从图中可以看出系统中的 0 号中断处理程序的功能：显示提示信息“Divide overflow”后，返回到操作系统中。

12.7 编程处理 0 号中断

现在我们考虑改变一下 0 号中断处理程序的功能，即重新编写一个 0 号中断处理程序，它的功能是在屏幕中间显示“overflow!”，然后返回到操作系统，如图 12.3 所示。

```
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B39  ES=0B39  SS=0B39  CS=0B39  IP=0100   NV UP EI PL NZ NA PO NC
0B39:0100 B80010        MOV    overflow!
-t

AX=1000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B39  ES=0B39  SS=0B39  CS=0B39  IP=0103   NV UP EI PL NZ NA PO NC
0B39:0103 B701          MOV      BH,01
-t

AX=1000  BX=0100  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0B39  ES=0B39  SS=0B39  CS=0B39  IP=0105   NV UP EI PL NZ NA PO NC
0B39:0105 F6F7          DIV      BH
-t

D:\>_
```

图 12.3　预编写程序对 0 号中断的处理

当 CPU 执行 div bh 后，发生了除法溢出错误，产生 0 号中断信息，引发中断过程，CPU 执行我们编写的 0 号中断处理程序。在屏幕中间显示提示信息“overflow!”后，返回到操作系统中。

编程：当发生除法溢出时，在屏幕中间显示“overflow!”，返回 DOS。

我们首先进行分析：

(1)　当发生除法溢出的时候，产生 0 号中断信息，从而引发中断过程。

此时，CPU 将进行以下工作。

①　取得中断类型码 0；
②　标志寄存器入栈，TF、IF 设置为 0；
③　CS、IP 入栈；
④　(IP)=(0*4)，(CS)=(0*4+2)。

(2)　可见，当中断 0 发生时，CPU 将转去执行中断处理程序。

只要按如下步骤编写中断处理程序，当中断 0 发生时，即可显示“overflow!”。

①　相关处理；
②　向显示缓冲区送字符串“overflow!”；
③　返回 DOS。

我们将这段程序称为：do0。

(3)　现在的问题是：do0 应存放在内存中。因为除法溢出随时可能发生，CPU 随时都可能将 CS:IP 指向 do0 的入口，执行程序。

那么 do0 应该放在哪里呢？

由于我们是在操作系统之上使用计算机，所有的硬件资源都在操作系统的管理之下，所以我们要想得到一块内存区存放 do0，应该向操作系统申请。

但在这里出于两个原因我们不想这样做：

① 过多地讨论申请内存将偏离问题的主线；

② 我们学习汇编的一个重要目的就是要获得对计算机底层的编程体验。所以，在可能的情况下，我们不去理会操作系统，而直接面向硬件资源。

问题变得简单而直接，我们只需找到一块别的程序不会用到的内存区，将 do0 传送到其中即可。

前面讲到，内存 0000:0000~0000:03FF，大小为 1KB 的空间是系统存放中断处理程序入口地址的中断向量表。8086 支持 256 个中断，但是，实际上，系统中要处理的中断事件远没有达到 256 个。所以在中断向量表中，有许多单元是空的。

中断向量表是 PC 系统中最重要的内存区，只用来存放中断处理程序的入口地址，DOS 系统和其他应用程序都不会随便使用这段空间。可以利用中断向量表中的空闲单元来存放我们的程序。一般情况下，从 0000:0200 至 0000:02FF 的 256 个字节的空间所对应的中断向量表项都是空的，操作系统和其他应用程序都不占用。我们在前面的课程中使用过这段空间(参见 5.7 节)。

根据以前的编程经验，我们可以估计出，do0 的长度不可能超过 256 个字节。

结论：我们可以将 do0 传送到内存 0000:0200 处。

(4) 将中断处理程序 do0 放到 0000:0200 后，若要使得除法溢出发生的时候，CPU 转去执行 do0，则必须将 do0 的入口地址，即 0000:0200 登记在中断向量表的对应表项中。因为除法溢出对应的中断类型码为 0，它的中断处理程序的入口地址应该从 0*4 地址单元开始存放，段地址存放在 0*4+2 字单元中，偏移地址存放在 0*4 字单元中。也就是说要将 do0 的段地址 0 存放在 0000:0002 字单元中，将偏移地址 200H 存放在 0000:0000 字单元中。

总结上面的分析，我们要做以下几件事情。

(1) 编写可以显示“overflow!”的中断处理程序：do0；
(2) 将 do0 送入内存 0000:0200 处；
(3) 将 do0 的入口地址 0000:0200 存储在中断向量表 0 号表项中。

程序的框架如下。

程序 12.1

```
assume cs:code

code segment

start:  do0 安装程序
        设置中断向量表
        mov ax,4c00h
        int 21h
```

```
do0:    显示字符串"overflow!"
        mov ax,4c00h
        int 21h

code ends

end start
```

可以看到，上面的程序分为两部分：

(1) 安装 do0，设置中断向量的程序；

(2) do0。

程序 12.1 执行时，do0 的代码是不执行的，它只是作为 do0 安装程序所要传送的数据。程序 12.1 执行时，首先执行 do0 安装程序，将 do0 的代码复制到内存 0:200 处，然后设置中断向量表，将 do0 的入口地址，即偏移地址 200H 和段地址 0，保存在 0 号表项中。这两部分工作完成后，程序就返回了。程序的目的就是在内存 0:200 处安装 do0 的代码，将 0 号中断处理程序的入口地址设置为 0:200。do0 的代码虽然在程序中，却不在程序执行的时候执行。它是在除法溢出发生的时候才得以执行的中断处理程序。

do0 部分代码的最后两条指令是依照我们的编程要求，用来返回 DOS 的。

现在，我们再反过来从 CPU 的角度看一下，什么是中断处理程序？我们来看一下 do0 是如何变成 0 号中断的中断处理程序的。

(1) 程序 12.1 在执行时，被加载到内存中，此时 do0 的代码在程序 12.1 所在的内存空间中，它只是存放在程序 12.1 的代码段中的一段要被传送到其他单元中的数据，我们不能说它是 0 号中断的中断处理程序；

(2) 程序 12.1 中安装 do0 的代码执行完后，do0 的代码被从程序 12.1 的代码段中复制到 0:200 处。此时，我们也不能说它是 0 号中断的中断处理程序，它只不过是存放在 0:200 处的一些数据；

(3) 程序 12.1 中设置中断向量表的代码执行完后，在 0 号表项中填入了 do0 的入口地址 0:200，此时 0:200 处的信息，即 do0 的代码，就变成了 0 号中断的中断处理程序。因为当除法溢出(即 0 号中断)发生时，CPU 将执行 0:200 处的代码。

回忆一下：

我们如何让一个内存单元成为栈顶？将它的地址放入 SS、SP 中；

我们如何让一个内存单元中的信息被 CPU 当作指令来执行？将它的地址放入 CS、IP 中；

那么，我们如何让一段程序成为 N 号中断的中断处理程序？将它的入口地址放入中断向量表的 N 号表项中。

下面的内容中，我们讨论每一部分程序的具体编写方法。

12.8 安　　装

可以使用 movsb 指令，将 do0 的代码送入 0:200 处。程序如下。

```
assume cs:code
code segment

  start:设置 es:di 指向目的地址
        设置 ds:si 指向源地址
        设置 cx 为传输长度
        设置传输方向为正
        rep movsb

        设置中断向量表

        mov ax,4c00h
        int 21h

    do0:显示字符串"overflow!"
        mov ax,4c00h
        int 21h

code ends
end start
```

我们来看一下，用 rep movsb 指令的时候要确定的信息。

(1) 传送的原始位置，段地址：code，偏移地址：offset do0；
(2) 传送的目的位置：0:200；
(3) 传送的长度：do0 部分代码的长度；
(4) 传送的方向：正向。

更明确的程序如下。

```
assume cs:code
code segment

start:  mov ax,cs
        mov ds,ax
        mov si,offset do0           ;设置 ds:si 指向源地址

        mov ax,0
        mov es,ax
        mov di,200h                 ;设置 es:di 指向目的地址

        mov cx,do0 部分代码的长度   ;设置 cx 为传输长度

        cld                         ;设置传输方向为正
```

```
        rep movsb

        设置中断向量表

        mov ax,4c00h
        int 21h

    do0: 显示字符串"overflow!"
        mov ax,4c00h
        int 21h

code ends
end start
```

问题是，我们如何知道 do0 代码的长度？最简单的方法是，计算一下 do0 中所有指令码的字节数。但是这样做太麻烦了，因为只要 do0 的内容发生了改变，我们都要重新计算它的长度。

可以利用编译器来计算 do0 的长度，具体做法如下。

```
assume cs:code
code segment
start:  mov ax,cs
        mov ds,ax
        mov si,offset do0                   ;设置 ds:si 指向源地址
        mov ax,0
        mov es,ax
        mov di,200h                         ;设置 es:di 指向目的地址

        mov cx,offset do0end-offset do0     ;设置 cx 为传输长度

        cld                                 ;设置传输方向为正
        rep movsb

        设置中断向量表

        mov ax,4c00h
        int 21h

   do0: 显示字符串"overflow!"
        mov ax,4c00h
        int 21h

do0end: nop

code ends
end start
```

“-”是编译器识别的运算符号，编译器可以用它来进行两个常数的减法。

比如，指令：mov ax,8-4，被编译器处理为指令：mov ax,4。

汇编编译器可以处理表达式。

比如，指令：mov ax,(5+3)*5/10，被编译器处理为指令：mov ax,4。

好了，知道了“-”的含义，对于用 offset do0end-offset do0，得到 do0 代码的长度的原理，这里就不再多说了，相信到了现在，读者已可以自己进行分析了。

下面我们编写 do0 程序。

12.9 do0

do0 程序的主要任务是显示字符串，程序如下。

```
do0:  设置 ds:si 指向字符串
      mov ax,0b800h
      mov es,ax
      mov di,12*160+36*2        ;设置 es:di 指向显存空间的中间位置

      mov cx,9                  ;设置 cx 为字符串长度
    s: mov al,[si]
      mov es:[di],al
      inc si
      add di,2
      loop s

      mov ax,4c00h
      int 21h

do0end:nop
```

程序写好了，可要显示的字符串放在哪里呢？我们看下面的程序。

程序 12.2

```
assume cs:code

data segment
 db "overflow!"
data ends

code segment

start:  mov ax,cs
        mov ds,ax
        mov si,offset do0                    ;设置 ds:si 指向源地址
        mov ax,0
        mov es,ax
        mov di,200h                          ;设置 es:di 指向目的地址
```

```
        mov cx,offset do0end-offset do0     ;设置 cx 为传输长度
        cld                                 ;设置传输方向为正
        rep movsb

        设置中断向量表
        mov ax,4c00h
        int 21h

do0:    mov ax,data
        mov ds,ax
        mov si,0                            ;设置 ds:si 指向字符串

        mov ax,0b800h
        mov es,ax
        mov di,12*160+36*2                  ;设置 es:di 指向显存空间的中间位置

        mov cx,9                            ;设置 cx 为字符串长度
    s:  mov al,[si]
        mov es:[di],al
        inc si
        add di,2
        loop s

        mov ax,4c00h
        int 21h

do0end:nop

code ends
end start
```

上面的程序，看似合理，可实际上却大错特错。注意，“overflow!”在程序 12.2 的 data 段中。程序 12.2 执行完成后返回，它所占用的内存空间被系统释放，而在其中存放的“overflow!”也将很可能被别的信息覆盖。而 do0 程序被放到了 0:200 处，随时都会因发生了除法溢出而被 CPU 执行，很难保证 do0 程序从原来程序 12.2 所处的空间中取得的是要显示的字符串“overflow!”。

因为 do0 程序随时可能被执行，而它要用到字符串“overflow!”，所以该字符串也应该存放在一段不会被覆盖的空间中。正确的程序如下。

程序 12.3

```
assume cs:code

code segment
start:  mov ax,cs
        mov ds,ax
        mov si,offset do0                   ;设置 ds:si 指向源地址
        mov ax,0
        mov es,ax
```

```
        mov di,200h                          ;设置 es:di 指向目的地址
        mov cx,offset do0end-offset do0      ;设置 cx 为传输长度
        cld                                  ;设置传输方向为正
        rep movsb
        设置中断向量表

        mov ax,4c00h
        int 21h

   do0: jmp short do0start
        db "overflow!"

do0start:   mov ax,cs
            mov ds,ax
            mov si,202h              ;设置 ds:si 指向字符串

            mov ax,0b800h
            mov es,ax
            mov di,12*160+36*2       ;设置 es:di 指向显存空间的中间位置

            mov cx,9                 ;设置 cx 为字符串长度
         s: mov al,[si]
            mov es:[di],al
            inc si
            add di,2
            loop s

            mov ax,4c00h
            int 21h

do0end:nop
code ends
end start
```

在程序 12.3 中，将“overflow!”放到 do0 程序中，程序 12.3 执行时，将标号 do0 到标号 do0end 之间的内容送到 0000:0200 处。

注意，因为在 do0 程序开始处的“overflow!”不是可以执行的代码，所以在“overflow!”之前加上一条 jmp 指令，转移到正式的 do0 程序。当除法溢出发生时，CPU 执行 0:200 处的 jmp 指令，跳过后面的字符串，转到正式的 do0 程序执行。

do0 程序执行过程中必须要找到“overflow!”，那么它在哪里呢？首先来看段地址，“overflow!”和 do0 的代码处于同一个段中，而除法溢出发生时，CS 中必然存放 do0 的段地址，也就是“overflow!”的段地址；再来看偏移地址，0:200 处的指令为 jmp short do0start，这条指令占两个字节，所以“overflow!”的偏移地址为 202h。

12.10　设置中断向量

下面，将 do0 的入口地址 0:200，写入中断向量表的 0 号表项中，使 do0 成为 0 号中断的中断处理程序。

0 号表项的地址为 0:0，其中 0:0 字单元存放偏移地址，0:2 字单元存放段地址。程序如下。

```
mov ax,0
mov es,ax
mov word ptr es:[0*4],200h
mov word ptr es:[0*4+2],0
```

12.11　单 步 中 断

基本上，CPU 在执行完一条指令之后，如果检测到标志寄存器的 TF 位为 1，则产生单步中断，引发中断过程。单步中断的中断类型码为 1，则它所引发的中断过程如下。

(1) 取得中断类型码 1；
(2) 标志寄存器入栈，TF、IF 设置为 0；
(3) CS、IP 入栈；
(4) (IP)=(1*4)，(CS)=(1*4+2)。

如上所述，如果 TF=1，则执行一条指令后，CPU 就要转去执行 1 号中断处理程序。CPU 为什么要提供这样的功能呢？

我们在使用 Debug 的 t 命令的时候，有没有想过这样的问题，Debug 如何能让 CPU 在执行一条指令后，就显示各个寄存器的状态？我们知道，CPU 在执行程序的时候是从 CS:IP 指向的某个地址开始，自动向下读取指令执行。也就是说，如果 CPU 不提供其他功能的话，就按这种方式工作，只要 CPU 一加电，它就从预设的地址开始一直执行下去，不可能有任何程序能控制它在执行完一条指令后停止，去做别的事情。可是，我们在 Debug 中看到的情况却是，Debug 可以控制 CPU 执行被加载程序中的一条指令，然后让它停下来，显示寄存器的状态。

Debug 有特殊的能力吗？我们只能说 Debug 利用了 CPU 提供的一种功能。只有 CPU 提供了在执行一条指令后就转去做其他事情的功能，Debug 或是其他的程序才能利用 CPU 提供的这种功能做出我们使用 T 命令时的效果。

好了，我们来简要地考虑一下 Debug 是如何利用 CPU 所提供的单步中断的功能的。首先，Debug 提供了单步中断的中断处理程序，功能为显示所有寄存器中的内容后等待输入命令。然后，在使用 t 命令执行指令时，Debug 将 TF 设置为 1，使得 CPU 工作于单步

中断方式下，则在 CPU 执行完这条指令后就引发单步中断，执行单步中断的中断处理程序，所有寄存器中的内容被显示在屏幕上，并且等待输入命令。

那么，接下来的问题是，当 TF=1 时，CPU 在执行完一条指令后将引发单步中断，转去执行中断处理程序。注意，中断处理程序也是由一条条指令组成的，如果在执行中断处理程序之前，TF=1，则 CPU 在执行完中断处理程序的第一条指令后，又要产生单步中断，则又要转去执行单步中断的中断处理程序，在执行完中断处理程序的第一条指令后，又要产生单步中断，则又要转去执行单步中断的中断处理程序……

看来，上面的过程将陷入一个永远不能结束的循环，CPU 永远执行单步中断处理程序的第一条指令。

CPU 当然不能让这种情况发生，解决的办法就是，在进入中断处理程序之前，设置 TF=0。从而避免 CPU 在执行中断处理程序的时候发生单步中断。这就是为什么在中断过程中有 TF=0 这个步骤，我们再来看一下中断过程。

(1) 取得中断类型码 N；
(2) 标志寄存器入栈，TF=0、IF=0；
(3) CS、IP 入栈；
(4) (IP)=(N*4)，(CS)=(N*4+2)。

最后，CPU 提供单步中断功能的原因就是，为单步跟踪程序的执行过程，提供了实现机制。

12.12 响应中断的特殊情况

一般情况下，CPU 在执行完当前指令后，如果检测到中断信息，就响应中断，引发中断过程。可是，在有些情况下，CPU 在执行完当前指令后，即便是发生中断，也不会响应。对于这些情况，我们不一一列举，只是用一种情况来进行说明。

在执行完向 ss 寄存器传送数据的指令后，即便是发生中断，CPU 也不会响应。这样做的主要原因是，ss:sp 联合指向栈顶，而对它们的设置应该连续完成。如果在执行完设置 ss 的指令后，CPU 响应中断，引发中断过程，要在栈中压入标志寄存器、CS 和 IP 的值。而 ss 改变，sp 并未改变，ss:sp 指向的不是正确的栈顶，将引起错误。所以 CPU 在执行完设置 ss 的指令后，不响应中断。这给连续设置 ss 和 sp 指向正确的栈顶提供了一个时机。即，我们应该利用这个特性，将设置 ss 和 sp 的指令连续存放，使得设置 sp 的指令紧接着设置 ss 的指令执行，而在此之间，CPU 不会引发中断过程。比如，我们要将栈顶设为 1000:0，应该：

```
mov ax,1000h
mov ss,ax
mov sp,0
```

而不应该：

```
mov ax,1000h
mov ss,ax
mov ax,0
mov sp,0
```

好了，现在我们回过来看一下，实验 2 中的“(3)下一条指令执行了吗？”现在你知道原因了吗？

Debug 利用单步中断来实现 T 命令的功能，也就是说，用 T 命令执行一条指令后，CPU 响应单步中断，执行 Debug 设置好的处理程序，才能在屏幕上显示寄存器的状态，并等待命令的输入。而在 mov ss,ax 指令执行后，CPU 根本就不响应任何中断，其中也包括单步中断，所以 Debug 设置好的用来显示寄存器状态和等待输入命令的中断处理程序根本没有得到执行，所以我们看不到预期的结果。CPU 接着向下执行后面的指令 mov sp,10h，然后响应单步中断，我们才看到正常的结果。

实验 12　编写 0 号中断的处理程序

编写 0 号中断的处理程序，使得在除法溢出发生时，在屏幕中间显示字符串“divide error!”，然后返回到 DOS。

要求：仔细跟踪调试，在理解整个过程之前，不要进行后面课程的学习。

第 13 章　int 指令

中断信息可以来自 CPU 的内部和外部，当 CPU 的内部有需要处理的事情发生的时候，将产生需要马上处理的中断信息，引发中断过程。在第 12 章中，我们讲解了中断过程和两种内中断的处理。

这一章中，我们讲解另一种重要的内中断，由 int 指令引发的中断。

13.1　int 指令

int 指令的格式为：int n，n 为中断类型码，它的功能是引发中断过程。

CPU 执行 int n 指令，相当于引发一个 n 号中断的中断过程，执行过程如下。

(1)　取中断类型码 n；
(2)　标志寄存器入栈，IF=0，TF=0；
(3)　CS、IP 入栈；
(4)　(IP)=(n*4)，(CS)=(n*4+2)。

从此处转去执行 n 号中断的中断处理程序。

可以在程序中使用 int 指令调用任何一个中断的中断处理程序。例如，下面的程序：

```
assume cs:code

code segment

  start:mov ax,0b800h
        mov es,ax
        mov byte ptr es:[12*160+40*2],'!'
        int 0

code ends

end start
```

这个程序在 Windows 2000 中的 DOS 方式下执行时，将在屏幕中间显示一个“!”，然后显示“Divide overflow”后返回到系统中。“!”是我们编程显示的，而“Divide overflow”是哪里来的呢？我们的程序中又没有做除法，不可能产生除法溢出。

程序是没有做除法，但是在结尾使用了 int 0 指令。CPU 执行 int 0 指令时，将引发中断过程，执行 0 号中断处理程序，而系统设置的 0 号中断处理程序的功能是显示“Divide overflow”，然后返回到系统。

可见，int 指令的最终功能和 call 指令相似，都是调用一段程序。

一般情况下，系统将一些具有一定功能的子程序，以中断处理程序的方式提供给应用程序调用。我们在编程的时候，可以用 int 指令调用这些子程序。当然，也可以自己编写一些中断处理程序供别人使用。以后，我们可以将中断处理程序简称为中断例程。

13.2　编写供应用程序调用的中断例程

前面，我们已经编写过中断 0 的中断例程了，现在我们讨论可以供应用程序调用的中断例程的编写方法。下面通过两个问题来讨论。

问题一：编写、安装中断 7ch 的中断例程。

功能：求一 word 型数据的平方。
参数：(ax)=要计算的数据。
返回值：dx、ax 中存放结果的高 16 位和低 16 位。
应用举例：求 2*3456^2。

```
assume cs:code

code segment

  start:mov ax,3456          ;(ax)=3456
        int 7ch              ;调用中断 7ch 的中断例程，计算 ax 中的数据的平方
        add ax,ax
        adc dx,dx            ;dx:ax 存放结果，将结果乘以 2
        mov ax,4c00h
        int 21h

code ends

end start
```

分析一下，我们要做以下 3 部分工作。

(1)　编写实现求平方功能的程序；
(2)　安装程序，将其安装在 0:200 处；
(3)　设置中断向量表，将程序的入口地址保存在 7ch 表项中，使其成为中断 7ch 的中断例程。

安装程序如下。

```
assume cs:code

code segment
start:  mov ax,cs
        mov ds,ax
```

```
        mov si,offset sqr                   ;设置 ds:si 指向源地址
        mov ax,0
        mov es,ax
        mov di,200h                         ;设置 es:di 指向目的地址
        mov cx,offset sqrend-offset sqr     ;设置 cx 为传输长度
        cld                                 ;设置传输方向为正
        rep movsb

        mov ax,0
        mov es,ax
        mov word ptr es:[7ch*4],200h
        mov word ptr es:[7ch*4+2],0

        mov ax,4c00h
        int 21h

  sqr:  mul ax
        iret

sqrend: nop

code ends
end start
```

注意，在中断例程 sqr 的最后，要使用 iret 指令。用汇编语法描述，iret 指令的功能为：

```
pop IP
pop CS
popf
```

CPU 执行 int 7ch 指令进入中断例程之前，标志寄存器、当前的 CS 和 IP 被压入栈中，在执行完中断例程后，应该用 iret 指令恢复 int 7ch 执行前的标志寄存器和 CS、IP 的值，从而接着执行应用程序。

int 指令和 iret 指令的配合使用与 call 指令和 ret 指令的配合使用具有相似的思路。

问题二：编写、安装中断 7ch 的中断例程。

功能：将一个全是字母，以 0 结尾的字符串，转化为大写。
参数：ds:si 指向字符串的首地址。
应用举例：将 data 段中的字符串转化为大写。

```
assume cs:code

data segment
    db  'conversation',0
data ends
```

```
code segment
start:  mov ax,data
        mov ds,ax
        mov si,0
        int 7ch

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

安装程序如下。

```
assume cs:code
code segment

   start:   mov ax,cs
            mov ds,ax
            mov si,offset capital
            mov ax,0
            mov es,ax
            mov di,200h
            mov cx,offset capitalend-offset capital
            cld
            rep movsb

            mov ax,0
            mov es,ax
            mov word ptr es:[7ch*4],200h
            mov word ptr es:[7ch*4+2],0
            mov ax,4c00h
            int 21h

capital:    push cx
            push si
  change:   mov cl,[si]
            mov ch,0
            jcxz ok
            and byte ptr [si],11011111b
            inc si
            jmp short change
        ok: pop si
            pop cx
            iret
capitalend: nop

code ends
end start
```

在中断例程 capital 中用到了寄存器 si 和 cx，编写中断例程和编写子程序的时候具有

同样的问题，就是要避免寄存器的冲突。应该注意例程中用到的寄存器的值的保存和恢复。

13.3 对 int、iret 和栈的深入理解

问题：用 7ch 中断例程完成 loop 指令的功能。

loop s 的执行需要两个信息，循环次数和到 s 的位移，所以，7ch 中断例程要完成 loop 指令的功能，也需要这两个信息作为参数。我们用 cx 存放循环次数，用 bx 存放位移。

应用举例：在屏幕中间显示 80 个“!”。

```
assume  cs:code

code segment

  start:mov ax,0b800h
        mov es,ax
        mov di,160*12

        mov bx,offset s-offset se       ;设置从标号 se 到标号 s 的转移位移
        mov cx,80
      s: mov byte ptr es:[di],'!'
        add di,2
        int 7ch                         ;如果(cx)≠0，转移到标号 s 处
      se:nop

        mov ax,4c00h
        int 21h

code ends

end start
```

在上面的程序中，用 int 7ch 调用 7ch 中断例程进行转移，用 bx 传递转移的位移。

分析：为了模拟 loop 指令，7ch 中断例程应具备下面的功能。

(1) dec cx；
(2) 如果(cx)≠0，转到标号 s 处执行，否则向下执行。

下面我们分析 7ch 中断例程如何实现到目的地址的转移。

(1) 转到标号 s 显然应设(CS)=标号 s 的段地址，(IP)=标号 s 的偏移地址。

(2) 那么，中断例程如何得到标号 s 的段地址和偏移地址呢？

int 7ch 引发中断过程后，进入 7ch 中断例程，在中断过程中，当前的标志寄存器、CS 和 IP 都要压栈，此时压入的 CS 和 IP 中的内容，分别是调用程序的段地址(可以认为是标号 s 的段地址)和 int 7ch 后一条指令的偏移地址(即标号 se 的偏移地址)。

可见，在中断例程中，可以从栈里取得标号 s 的段地址和标号 se 的偏移地址，而用标号 se 的偏移地址加上 bx 中存放的转移位移就可以得到标号 s 的偏移地址。

(3) 现在知道，可以从栈中直接和间接地得到标号 s 的段地址和偏移地址，那么如何用它们设置 CS:IP 呢？

可以利用 iret 指令，我们将栈中的 se 的偏移地址加上 bx 中的转移位移，则栈中的 se 的偏移地址就变为了 s 的偏移地址。我们再使用 iret 指令，用栈中的内容设置 CS、IP，从而实现转移到标号 s 处。

7ch 中断例程如下。

```
    lp:  push bp
         mov bp,sp
         dec cx
         jcxz lpret
         add [bp+2],bx
lpret:   pop bp
         iret
```

因为要访问栈，使用了 bp，在程序开始处将 bp 入栈保存，结束时出栈恢复。当要修改栈中 se 的偏移地址的时候，栈中的情况为：栈顶处是 bp 原来的数值，下面是 se 的偏移地址，再下面是 s 的段地址，再下面是标志寄存器的值。而此时，bp 中为栈顶的偏移地址，所以((ss)*16+(bp)+2)处为 se 的偏移地址，将它加上 bx 中的转移位移就变为 s 的偏移地址。最后用 iret 出栈返回，CS:IP 即从标号 s 处开始执行指令。

如果(cx)=0，则不需要修改栈中 se 的偏移地址，直接返回即可。CPU 从标号 se 处向下执行指令。

检测点 13.1

(1) 在上面的内容中，我们用 7ch 中断例程实现 loop 的功能，则上面的 7ch 中断例程所能进行的最大转移位移是多少？

(2) 用 7ch 中断例程完成 jmp near ptr s 指令的功能，用 bx 向中断例程传送转移位移。

应用举例：在屏幕的第 12 行，显示 data 段中以 0 结尾的字符串。

```
assume cs:code
data segment
    db  'conversation',0
data ends
code segment
start:  mov ax,data
```

```
        mov ds,ax
        mov si,0
        mov ax,0b800h
        mov es,ax
        mov di,12*160
    s:  cmp byte ptr [si],0
        je ok                          ;如果是 0 跳出循环
        mov al,[si]
        mov es:[di],al
        inc si
        add di,2
        mov bx,offset s-offset ok      ;设置从标号 ok 到标号 s 的转移位移
        int 7ch                        ;转移到标号 s 处
   ok:  mov ax,4c00h
        int 21h
code ends
end start
```

13.4 BIOS 和 DOS 所提供的中断例程

在系统板的 ROM 中存放着一套程序，称为 BIOS(基本输入输出系统)，BIOS 中主要包含以下几部分内容。

(1) 硬件系统的检测和初始化程序；
(2) 外部中断(第 15 章中进行讲解)和内部中断的中断例程；
(3) 用于对硬件设备进行 I/O 操作的中断例程；
(4) 其他和硬件系统相关的中断例程。

操作系统 DOS 也提供了中断例程，从操作系统的角度来看，DOS 的中断例程就是操作系统向程序员提供的编程资源。

BIOS 和 DOS 在所提供的中断例程中包含了许多子程序，这些子程序实现了程序员在编程的时候经常需要用到的功能。程序员在编程的时候，可以用 int 指令直接调用 BIOS 和 DOS 提供的中断例程，来完成某些工作。

和硬件设备相关的 DOS 中断例程中，一般都调用了 BIOS 的中断例程。

13.5 BIOS 和 DOS 中断例程的安装过程

前面的课程中，我们都是自己编写中断例程，将它们放到安装程序中，然后运行安装程序，将它们安装到指定的内存区中。此后，别的应用程序才可以调用。

而 BIOS 和 DOS 提供的中断例程是如何安装到内存中的呢？我们下面讲解它们的安装过程。

(1) 开机后，CPU 一加电，初始化(CS)=0FFFFH，(IP)=0，自动从 FFFF:0 单元开始执行程序。FFFF:0 处有一条转跳指令，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。

(2) 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。注意，对于 BIOS 所提供的中断例程，只需将入口地址登记在中断向量表中即可，因为它们是固化到 ROM 中的程序，一直在内存中存在。

(3) 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。从此将计算机交由操作系统控制。

(4) DOS 启动后，除完成其他工作外，还将它所提供的中断例程装入内存，并建立相应的中断向量。

检测点 13.2

判断下面说法的正误:

(1) 我们可以编程改变 FFFF:0 处的指令，使得 CPU 不去执行 BIOS 中的硬件系统检测和初始化程序。

(2) int 19h 中断例程，可以由 DOS 提供。

13.6　BIOS 中断例程应用

下面我们举几个例子，来看一下 BIOS 中断例程的应用。

int 10h 中断例程是 BIOS 提供的中断例程，其中包含了多个和屏幕输出相关的子程序。

一般来说，一个供程序员调用的中断例程中往往包括多个子程序，中断例程内部用传递进来的参数来决定执行哪一个子程序。BIOS 和 DOS 提供的中断例程，都用 ah 来传递内部子程序的编号。

下面看一下 int 10h 中断例程的设置光标位置功能。

```
mov ah,2        ;置光标
mov bh,0        ;第 0 页
mov dh,5        ;dh 中放行号
mov dl,12       ;dl 中放列号
int 10h
```

(ah)=2 表示调用第 10h 号中断例程的 2 号子程序，功能为设置光标位置，可以提供光标所在的行号(80*25 字符模式下：0~24)、列号(80*25 字符模式下：0~79)，和页号作为参数。

(bh)=0，(dh)=5，(dl)=12，设置光标到第 0 页，第 5 行，第 12 列。

bh 中页号的含义：内存地址空间中，B8000H~BFFFFH 共 32kB 的空间，为 80*25 彩色字符模式的显示缓冲区。一屏的内容在显示缓冲区中共占 4000 个字节。

显示缓冲区分为 8 页，每页 4KB(≈4000B)，显示器可以显示任意一页的内容。一般情况下，显示第 0 页的内容。也就是说，通常情况下，B8000H~B8F9FH 中的 4000 个字节的内容将出现在显示器上。

再看一下 int 10h 中断例程的在光标位置显示字符功能。

```
mov ah,9          ;在光标位置显示字符
mov al,'a'        ;字符
mov bl,7          ;颜色属性
mov bh,0          ;第 0 页
mov cx,3          ;字符重复个数
int 10h
```

(ah)=9 表示调用第 10h 号中断例程的 9 号子程序，功能为在光标位置显示字符，可以提供要显示的字符、颜色属性、页号、字符重复个数作为参数。

bl 中的颜色属性的格式如下。

	7	6	5	4	3	2	1	0
含义	BL	R	G	B	I	R	G	B
	闪烁	背景			高亮	前景		

可以看出，和显存中的属性字节的格式相同。

编程：在屏幕的 5 行 12 列显示 3 个红底高亮闪烁绿色的“a”。

```
assume cs:code
code segment

     mov ah,2            ;置光标
     mov bh,0            ;第 0 页
     mov dh,5            ;dh 中放行号
     mov dl,12           ;dl 中放列号
     int 10h

     mov ah,9            ;在光标位置显示字符
     mov al,'a'          ;字符
     mov bl,11001010b    ;颜色属性
     mov bh,0            ;第 0 页
     mov cx,3            ;字符重复个数
     int 10h

     mov ax,4c00h
     int 21h

code ends
end
```

注意，闪烁的效果必须在全屏 DOS 方式下才能看到。

13.7　DOS 中断例程应用

int 21h 中断例程是 DOS 提供的中断例程，其中包含了 DOS 提供给程序员在编程时调用的子程序。

我们前面一直使用的是 int 21h 中断例程的 4ch 号功能，即程序返回功能，如下：

```
mov ah,4ch        ;程序返回
mov al,0          ;返回值
int 21h
```

(ah)=4ch 表示调用第 21h 号中断例程的 4ch 号子程序，功能为程序返回，可以提供返回值作为参数。

我们前面使用这个功能的时候经常写做：

```
mov ax,4c00h
int 21h
```

我们看一下 int 21h 中断例程在光标位置显示字符串的功能：

```
ds:dx 指向字符串        ;要显示的字符串需用"$"作为结束符
mov ah,9                ;功能号 9，表示在光标位置显示字符串
int 21h
```

(ah)=9 表示调用第 21h 号中断例程的 9 号子程序，功能为在光标位置显示字符串，可以提供要显示字符串的地址作为参数。

编程：在屏幕的 5 行 12 列显示字符串“Welcome to masm!”。

```
assume cs:code

data segment
  db 'Welcome to masm', '$'
data ends

code segment
  start:mov ah,2        ;置光标
        mov bh,0        ;第 0 页
        mov dh,5        ;dh 中放行号
        mov dl,12       ;dl 中放列号
        int 10h

        mov ax,data
        mov ds,ax
        mov dx,0        ;ds:dx 指向字符串的首地址 data:0
        mov ah,9
```

```
        int 21h

        mov ax,4c00h
        int 21h

code ends
end start
```

上述程序在屏幕的 5 行 12 列显示字符串“Welcome to masm!”，直到遇见“$”(“$”本身并不显示，只起到边界的作用)。

如果字符串比较长，遇到行尾，程序会自动转到下一行开头处继续显示；如果到了最后一行，还能自动上卷一行。

DOS 为程序员提供了许多可以调用的子程序，都包含在 int 21h 中断例程中。我们这里只对原理进行了讲解，对于 DOS 提供的所有可调用子程序的情况，读者可以参考相关的书籍。

实验 13　编写、应用中断例程

(1) 编写并安装 int 7ch 中断例程，功能为显示一个用 0 结束的字符串，中断例程安装在 0:200 处。

参数：(dh)=行号，(dl)=列号，(cl)=颜色，ds:si 指向字符串首地址。

以上中断例程安装成功后，对下面的程序进行单步跟踪，尤其注意观察 int、iret 指令执行前后 CS、IP 和栈中的状态。

```
assume cs:code
data segment
    db "welcome to masm! ",0
data ends
code segment
start:  mov dh,10
        mov dl,10
        mov cl,2
        mov ax,data
        mov ds,ax
        mov si,0
        int 7ch
        mov ax,4c00h
        int 21h
code ends
end start
```

(2) 编写并安装 int 7ch 中断例程，功能为完成 loop 指令的功能。

参数：(cx)=循环次数，(bx)=位移。

以上中断例程安装成功后，对下面的程序进行单步跟踪，尤其注意观察 int、iret 指令执行前后 CS、IP 和栈中的状态。

在屏幕中间显示 80 个“!”。

```
assume  cs:code
code segment
start:  mov ax,0b800h
        mov es,ax
        mov di,160*12
        mov bx,offset s-offset se     ;设置从标号 se 到标号 s 的转移位移
        mov cx,80
   s:   mov byte ptr es:[di],'!'
        add di,2
        int 7ch                       ;如果(cx)≠0，转移到标号 s 处
   se:  nop
        mov ax,4c00h
        int 21h
code ends
end start
```

(3)　下面的程序，分别在屏幕的第 2、4、6、8 行显示 4 句英文诗，补全程序。

```
assume cs:code
code segment
  s1:   db 'Good,better,best,','$'
  s2:   db 'Never let it rest,','$'
  s3:   db 'Till good is better,','$'
  s4:   db 'And better,best.','$'
  s :   dw  offset s1,offset s2,offset s3,offset s4
  row:  db  2,4,6,8

  start:mov ax,cs
        mov ds,ax
        mov bx,offset s
        mov si,offset row
        mov cx,4
     ok:mov bh,0
        mov dh,________
        mov dl,0
        mov ah,2
        int 10h

        mov dx,________
        mov ah,9
        int 21h
        ____________
        ____________
        loop ok
```

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

完成后编译运行，体会其中的编程思想。

第 14 章　端　　口

我们前面讲过，各种存储器都和 CPU 的地址线、数据线、控制线相连。CPU 在操控它们的时候，把它们都当作内存来对待，把它们总地看做一个由若干存储单元组成的逻辑存储器，这个逻辑存储器我们称其为内存地址空间(可参见 1.15 节)。

在 PC 机系统中，和 CPU 通过总线相连的芯片除各种存储器外，还有以下 3 种芯片。

(1) 各种接口卡(比如，网卡、显卡)上的接口芯片，它们控制接口卡进行工作；
(2) 主板上的接口芯片，CPU 通过它们对部分外设进行访问；
(3) 其他芯片，用来存储相关的系统信息，或进行相关的输入输出处理。

在这些芯片中，都有一组可以由 CPU 读写的寄存器。这些寄存器，它们在物理上可能处于不同的芯片中，但是它们在以下两点上相同。

(1) 都和 CPU 的总线相连，当然这种连接是通过它们所在的芯片进行的；
(2) CPU 对它们进行读或写的时候都通过控制线向它们所在的芯片发出端口读写命令。

可见，从 CPU 的角度，将这些寄存器都当作端口，对它们进行统一编址，从而建立了一个统一的端口地址空间。每一个端口在地址空间中都有一个地址。

CPU 可以直接读写以下 3 个地方的数据。

(1) CPU 内部的寄存器；
(2) 内存单元；
(3) 端口。

这一章，我们讨论端口的读写。

14.1　端口的读写

在访问端口的时候，CPU 通过端口地址来定位端口。因为端口所在的芯片和 CPU 通过总线相连，所以，端口地址和内存地址一样，通过地址总线来传送。在 PC 系统中，CPU 最多可以定位 64KB 个不同的端口。则端口地址的范围为 0~65535。

对端口的读写不能用 mov、push、pop 等内存读写指令。端口的读写指令只有两条：in 和 out，分别用于从端口读取数据和往端口写入数据。

我们看一下 CPU 执行内存访问指令和端口访问指令时，总线上的信息：

(1) 访问内存：

```
mov ax,ds:[8]        ;假设执行前(ds)=0
```

执行时与总线相关的操作如下所示。

① CPU 通过地址线将地址信息 8 发出；

② CPU 通过控制线发出内存读命令，选中存储器芯片，并通知它，将要从中读取数据；

③ 存储器将 8 号单元中的数据通过数据线送入 CPU。

(2) 访问端口：

```
in al,60h            ;从 60h 号端口读入一个字节
```

执行时与总线相关的操作如下。

① CPU 通过地址线将地址信息 60h 发出；

② CPU 通过控制线发出端口读命令，选中端口所在的芯片，并通知它，将要从中读取数据；

③ 端口所在的芯片将 60h 端口中的数据通过数据线送入 CPU。

注意，在 in 和 out 指令中，只能使用 ax 或 al 来存放从端口中读入的数据或要发送到端口中的数据。访问 8 位端口时用 al，访问 16 位端口时用 ax。

对 0~255 以内的端口进行读写时：

```
in al,20h            ;从 20h 端口读入一个字节
out 20h,al           ;往 20h 端口写入一个字节
```

对 256~65535 的端口进行读写时，端口号放在 dx 中：

```
mov dx,3f8h          ;将端口号 3f8h 送入 dx
in al,dx             ;从 3f8h 端口读入一个字节
out dx,al            ;向 3f8h 端口写入一个字节
```

14.2 CMOS RAM 芯片

下面的内容中，我们通过对 CMOS RAM 的读写来体会一下对端口的访问。

PC 机中，有一个 CMOS RAM 芯片，一般简称为 CMOS。此芯片的特征如下。

(1) 包含一个实时钟和一个有 128 个存储单元的 RAM 存储器(早期的计算机为 64 个字节)。

(2) 该芯片靠电池供电。所以，关机后其内部的实时钟仍可正常工作，RAM 中的信息不丢失。

(3) 128 个字节的 RAM 中，内部实时钟占用 0~0dh 单元来保存时间信息，其余大部分单元用于保存系统配置信息，供系统启动时 BIOS 程序读取。BIOS 也提供了相关的程

序，使我们可以在开机的时候配置 CMOS RAM 中的系统信息。

(4) 该芯片内部有两个端口，端口地址为 70h 和 71h。CPU 通过这两个端口来读写 CMOS RAM。

(5) 70h 为地址端口，存放要访问的 CMOS RAM 单元的地址；71h 为数据端口，存放从选定的 CMOS RAM 单元中读取的数据，或要写入到其中的数据。可见，CPU 对 CMOS RAM 的读写分两步进行，比如，读 CMOS RAM 的 2 号单元：

① 将 2 送入端口 70h；
② 从端口 71h 读出 2 号单元的内容。

检测点 14.1

(1) 编程，读取 CMOS RAM 的 2 号单元的内容。
(2) 编程，向 CMOS RAM 的 2 号单元写入 0。

14.3　shl 和 shr 指令

shl 和 shr 是逻辑移位指令，后面的课程中我们要用到移位指令，这里进行一下讲解。

shl 是逻辑左移指令，它的功能为：

(1) 将一个寄存器或内存单元中的数据向左移位；
(2) 将最后移出的一位写入 CF 中；
(3) 最低位用 0 补充。

指令：

```
mov al,01001000b
shl al,1                ;将 al 中的数据左移一位
```

执行后(al)=10010000b，CF=0。

我们来看一下 shl al,1 的操作过程。

(1) 左移
　　原数据：　01001000
　　左移后：　01001000

(2) 将最后移出的一位写入 CF 中
　　原数据：　01001000
　　左移后：　1001000　　CF=0

(3) 最低位用 0 补充
　　原数据：　01001000
　　左移后：　10010000

如果接着上面，继续执行一条 shl al,1，则执行后：(al)=00100000b，CF=1。shl 指令的操作过程如下。

(1) 左移

原数据： 10010000

左移后： 10010000

(2) 将最后移出的一位写入 CF 中

原数据： 10010000

左移后： 0010000 CF=1

(3) 最低位用 0 补充

原数据： 10010000

左移后： 00100000

如果移动位数大于 1 时，必须将移动位数放在 cl 中。

比如，指令：

```
mov al,01010001b
mov cl,3
shl al,cl
```

执行后(al)=10001000b，因为最后移出的一位是 0，所以 CF=0。

可以看出，将 X 逻辑左移一位，相当于执行 X=X*2。

比如：

```
mov al,00000001b        ;执行后(al)=00000001b=1
shl al,1                ;执行后(al)=00000010b=2
shl al,1                ;执行后(al)=00000100b=4
shl al,1                ;执行后(al)=00001000b=8
mov cl,3
shl al,cl               ;执行后(al)=01000000b=64
```

shr 是逻辑右移指令，它和 shl 所进行的操作刚好相反。

(1) 将一个寄存器或内存单元中的数据向右移位；
(2) 将最后移出的一位写入 CF 中；
(3) 最高位用 0 补充。

指令：

```
mov al,10000001b
shr al,1                ;将 al 中的数据右移一位
```

执行后(al)=01000000b，CF=1。

如果接着上面，继续执行一条 shr al,1，则执行后：(al)=00100000b，CF=0。

如果移动位数大于 1 时，必须将移动位数放在 cl 中。

比如，指令：

```
mov al,01010001b
mov cl,3
shr al,cl
```

执行后(al)=00001010b，因为最后移出的一位是 0，所以 CF=0。

可以看出将 X 逻辑右移一位，相当于执行 X=X/2。

检测点 14.2

编程，用加法和移位指令计算(ax)=(ax)*10。

提示，(ax)*10=(ax)*2+(ax)*8。

14.4　CMOS RAM 中存储的时间信息

在 CMOS RAM 中，存放着当前的时间：年、月、日、时、分、秒。这 6 个信息的长度都为 1 个字节，存放单元为：

秒：0　　分：2　　时：4　　日：7　　月：8　　年：9

这些数据以 BCD 码的方式存放。

BCD 码是以 4 位二进制数表示十进制数码的编码方法，如下所示。

十进制数码：　0　1　2　3　4　5　6　7　8　9

对应的 BCD 码：0000 0001 0010 0011 0100 0101 0110 0111 1000 1001

比如，数值 26，用 BCD 码表示为：0010 0110。

可见，一个字节可表示两个 BCD 码。则 CMOS RAM 存储时间信息的单元中，存储了用两个 BCD 码表示的两位十进制数，高 4 位的 BCD 码表示十位，低 4 位的 BCD 码表示个位。比如，00010100b 表示 14。

编程，在屏幕中间显示当前的月份。

分析，这个程序主要做以下两部分工作。

(1)　从 CMOS RAM 的 8 号单元读出当前月份的 BCD 码。

要读取 CMOS RAM 的信息，首先要向地址端口 70h 写入要访问的单元的地址：

```
mov al,8
out 70h,al
```

然后从数据端口 71h 中取得指定单元中的数据：

```
in al,71h
```

(2) 将用 BCD 码表示的月份以十进制的形式显示到屏幕上。

我们可以看出，BCD 码值=十进制数码值，则 BCD 码值+30h=十进制数对应的 ASCII 码。

从 CMOS RAM 的 8 号单元读出的一个字节中，包含了用两个 BCD 码表示的两位十进制数，高 4 位的 BCD 码表示十位，低 4 位的 BCD 码表示个位。比如，00010100b 表示 14。

我们需要进行以下两步工作。

① 将从 CMOS RAM 的 8 号单元中读出的一个字节，分为两个表示 BCD 码值的数据。

```
mov ah,al            ;al 中为从 CMOS RAM 的 8 号单元中读出的数据
mov cl,4
shr ah,cl            ;ah 中为月份的十位数码值
and al,00001111b     ;al 中为月份的个位数码值
```

② 显示(ah)+30h 和(al)+30h 对应的 ASCII 码字符。

完整的程序如下。

```
assume cs:code
code segment
start:  mov al,8
        out 70h,al
        in al,71h

        mov ah,al
        mov cl,4
        shr ah,cl
        and al,00001111b

        add ah,30h
        add al,30h

        mov bx,0b800h
        mov es,bx
        mov byte ptr es:[160*12+40*2],ah        ;显示月份的十位数码
        mov byte ptr es:[160*12+40*2+2],al      ;接着显示月份的个位数码

        mov ax,4c00h
```

```
        int 21h

code ends
end start
```

实验 14　访问 CMOS RAM

编程，以“年/月/日 时:分:秒”的格式，显示当前的日期、时间。

注意：CMOS RAM 中存储着系统的配置信息，除了保存时间信息的单元外，不要向其他的单元中写入内容，否则将引起一些系统错误。

第 15 章　外　中　断

以前我们讨论的都是 CPU 对指令的执行。我们知道，CPU 在计算机系统中，除了能够执行指令，进行运算以外，还应该能够对外部设备进行控制，接收它们的输入，向它们进行输出。也就是说，CPU 除了有运算能力外，还要有 I/O(Input/Output，输入/输出)能力。比如，我们按下键盘上的一个键，CPU 最终要能够处理这个键。在使用文本编辑器时，按下 a 键后，我们可以看到屏幕上出现“a”，是 CPU 将从键盘上输入的键所对应的字符送到显示器上的。

要及时处理外设的输入，显然需要解决两个问题：①外设的输入随时可能发生，CPU 如何得知？②CPU 从何处得到外设的输入？

这一章中，我们以键盘输入为例，讨论这两个问题。

15.1　接口芯片和端口

第 14 章我们讲过，PC 系统的接口卡和主板上，装有各种接口芯片。这些外设接口芯片的内部有若干寄存器，CPU 将这些寄存器当作端口来访问。

外设的输入不直接送入内存和 CPU，而是送入相关的接口芯片的端口中；CPU 向外设的输出也不是直接送入外设，而是先送入端口中，再由相关的芯片送到外设。CPU 还可以向外设输出控制命令，而这些控制命令也是先送到相关芯片的端口中，然后再由相关的芯片根据命令对外设实施控制。

可见，CPU 通过端口和外部设备进行联系。

15.2　外中断信息

现在，我们知道了外设的输入被存放在端口中，可是外设的输入随时都有可能到达，CPU 如何及时地知道，并进行处理呢？更一般地讲，就是外设随时都可能发生需要 CPU 及时处理的事件，CPU 如何及时得知并进行处理？

CPU 提供中断机制来满足这种需要。前面讲过，当 CPU 的内部有需要处理的事情发生的时候，将产生中断信息，引发中断过程。这种中断信息来自 CPU 的内部。

还有一种中断信息，来自于 CPU 外部，当 CPU 外部有需要处理的事情发生的时候，比如说，外设的输入到达，相关芯片将向 CPU 发出相应的中断信息。CPU 在执行完当前指令后，可以检测到发送过来的中断信息，引发中断过程，处理外设的输入。

在 PC 系统中，外中断源一共有以下两类。

1. 可屏蔽中断

可屏蔽中断是 CPU 可以不响应的外中断。CPU 是否响应可屏蔽中断，要看标志寄存器的 IF 位的设置。当 CPU 检测到可屏蔽中断信息时，如果 IF=1，则 CPU 在执行完当前指令后响应中断，引发中断过程；如果 IF=0，则不响应可屏蔽中断。

我们回忆一下内中断所引发的中断过程：

(1) 取中断类型码 n；
(2) 标志寄存器入栈，IF=0，TF=0；
(3) CS、IP 入栈；
(4) (IP)=(n*4)，(CS)=(n*4+2)

由此转去执行中断处理程序。

可屏蔽中断所引发的中断过程，除在第 1 步的实现上有所不同外，基本上和内中断的中断过程相同。因为可屏蔽中断信息来自于 CPU 外部，中断类型码是通过数据总线送入 CPU 的；而内中断的中断类型码是在 CPU 内部产生的。

现在，我们可以解释中断过程中将 IF 置为 0 的原因了。将 IF 置 0 的原因就是，在进入中断处理程序后，禁止其他的可屏蔽中断。

当然，如果在中断处理程序中需要处理可屏蔽中断，可以用指令将 IF 置 1。8086CPU 提供的设置 IF 的指令如下：

sti，设置 IF=1；
cli，设置 IF=0。

2. 不可屏蔽中断

不可屏蔽中断是 CPU 必须响应的外中断。当 CPU 检测到不可屏蔽中断信息时，则在执行完当前指令后，立即响应，引发中断过程。

对于 8086CPU，不可屏蔽中断的中断类型码固定为 2，所以中断过程中，不需要取中断类型码。则不可屏蔽中断的中断过程为：

(1) 标志寄存器入栈，IF=0，TF=0；
(2) CS、IP 入栈；
(3) (IP)=(8)，(CS)=(0AH)。

几乎所有由外设引发的外中断，都是可屏蔽中断。当外设有需要处理的事件(比如说键盘输入)发生时，相关芯片向 CPU 发出可屏蔽中断信息。不可屏蔽中断是在系统中有必须处理的紧急情况发生时用来通知 CPU 的中断信息。在我们的课程中，主要讨论可屏蔽中断。

15.3 PC 机键盘的处理过程

下面我们看一下键盘输入的处理过程，并以此来体会一下 PC 机处理外设输入的基本方法。

1. 键盘输入

键盘上的每一个键相当于一个开关，键盘中有一个芯片对键盘上的每一个键的开关状态进行扫描。

按下一个键时，开关接通，该芯片就产生一个扫描码，扫描码说明了按下的键在键盘上的位置。扫描码被送入主板上的相关接口芯片的寄存器中，该寄存器的端口地址为 60h。

松开按下的键时，也产生一个扫描码，扫描码说明了松开的键在键盘上的位置。松开按键时产生的扫描码也被送入 60h 端口中。

一般将按下一个键时产生的扫描码称为通码，松开一个键产生的扫描码称为断码。扫描码长度为一个字节，通码的第 7 位为 0，断码的第 7 位为 1，即：

断码＝通码 ＋80h

比如，g 键的通码为 22h，断码为 a2h。

表 15.1 是键盘上部分键的扫描码，只列出通码。断码＝通码 ＋80h。

表 15.1 键盘上部分键的扫描码

键	扫描码	键	扫描码	键	扫描码	键	扫描码
Esc	01	U	16	H	23	B	30
1~9	02~0A	I	17	J	24	N	31
0	0B	O	18	K	25	M	32
-	0C	P	19	L	26	,	33
=	0D	[	1A	;	27	.	34
Backspace	0E	]	1B	,	28	/	35
Tab	0F	Enter	1C	、	29	Shift(右)	36
Q	10	Ctrl	1D	Shift(左)	2A	PrtSc	37
W	11	A	1E	\	2B	Alt	38
E	12	S	1F	Z	2C	Space	39
R	13	D	20	X	2D	CapsLock	3A
T	14	F	21	C	2E	F1~F10	3B~44
Y	15	G	22	V	2F	NumLock	45

续表

键	扫描码	键	扫描码	键	扫描码	键	扫描码
ScrollLock	46	-	4A	End	4F	Del	53
Home	47	←	4B	↓	50		
↑	48	→	4D	PgDn	51		
PgUp	49	+	4E	Ins	52		

2. 引发 9 号中断

键盘的输入到达 60h 端口时，相关的芯片就会向 CPU 发出中断类型码为 9 的可屏蔽中断信息。CPU 检测到该中断信息后，如果 IF=1，则响应中断，引发中断过程，转去执行 int 9 中断例程。

3. 执行 int 9 中断例程

BIOS 提供了 int 9 中断例程，用来进行基本的键盘输入处理，主要的工作如下：

(1) 读出 60h 端口中的扫描码；

(2) 如果是字符键的扫描码，将该扫描码和它所对应的字符码(即 ASCII 码)送入内存中的 BIOS 键盘缓冲区；如果是控制键(比如 Ctrl)和切换键(比如 CapsLock)的扫描码，则将其转变为状态字节(用二进制位记录控制键和切换键状态的字节)写入内存中存储状态字节的单元；

(3) 对键盘系统进行相关的控制，比如说，向相关芯片发出应答信息。

BIOS 键盘缓冲区是系统启动后，BIOS 用于存放 int 9 中断例程所接收的键盘输入的内存区。该内存区可以存储 15 个键盘输入，因为 int 9 中断例程除了接收扫描码外，还要产生和扫描码对应的字符码，所以在 BIOS 键盘缓冲区中，一个键盘输入用一个字单元存放，高位字节存放扫描码，低位字节存放字符码。

0040:17 单元存储键盘状态字节，该字节记录了控制键和切换键的状态。键盘状态字节各位记录的信息如下。

0：右 Shift 状态，置 1 表示按下右 Shift 键；
1：左 Shift 状态，置 1 表示按下左 Shift 键；
2：Ctrl 状态，置 1 表示按下 Ctrl 键；
3：Alt 状态，置 1 表示按下 Alt 键；
4：ScrollLock 状态，置 1 表示 Scroll 指示灯亮；
5：NumLock 状态，置 1 表示小键盘输入的是数字；
6：CapsLock 状态，置 1 表示输入大写字母；
7：Insert 状态，置 1 表示处于删除态。

15.4 编写 int 9 中断例程

从上面的内容中，可以看出键盘输入的处理过程：①键盘产生扫描码；②扫描码送入 60h 端口；③引发 9 号中断；④CPU 执行 int 9 中断例程处理键盘输入。

上面的过程中，第 1、2、3 步都是由硬件系统完成的。我们能够改变的只有 int 9 中断处理程序。我们可以重新编写 int 9 中断例程，按照自己的意图来处理键盘的输入。但是，在课程中，我们不准备完整地编写一个键盘中断的处理程序，因为要涉及一些硬件细节，而这些内容脱离了我们的内容主线。

但是，我们却还要编写新的键盘中断处理程序，来进行一些特殊的工作，那么这些硬件细节如何处理呢？这点比较简单，因为 BIOS 提供的 int 9 中断例程已经对这些硬件细节进行了处理。我们只要在自己编写的中断例程中调用 BIOS 的 int 9 中断例程就可以了。

编程：在屏幕中间依次显示“a”~“z”，并可以让人看清。在显示的过程中，按下 Esc 键后，改变显示的颜色。

我们先来看一下如何依次显示“a”~“z”。

```
assume cs:code
code segment
start:  mov ax,0b800h
        mov es,ax
        mov ah,'a'
    s:  mov es:[160*12+40*2],ah
        inc ah
        cmp ah,'z'
        jna s
        mov ax,4c00h
        int 21h
code ends
end start
```

在上面的程序的执行过程中，我们无法看清屏幕上的显示。因为一个字母刚显示到屏幕上，CPU 执行几条指令后，就又变成了另一个字母，字母之间切换得太快，无法看清。

应该在每显示一个字母后，延时一段时间，让人看清后，再显示下一个字母。那么如何延时呢？我们让 CPU 执行一段时间的空循环。因为现在 CPU 的速度都非常快，所以循环的次数一定要大，用两个 16 位寄存器来存放 32 位的循环次数。如下：

```
        mov dx,10h
        mov ax,0
s:      sub ax,1
        sbb dx,0
        cmp ax,0
```

```
    jne s
    cmp dx,0
    jne s
```

上面的程序，循环 100000h 次。我们可以将循环延时的程序段写为一个子程序。

现在，我们的程序如下：

```
assume cs:code

stack segment
  db 128 dup (0)
stack ends

code segment

start:  mov ax,stack
        mov ss,ax
        mov sp,128

        mov ax,0b800h
        mov es,ax
        mov ah,'a'
    s:  mov es:[160*12+40*2],ah
        call delay
        inc ah
        cmp ah,'z'
        jna s

        mov ax,4c00h
        int 21h

delay:  push ax
        push dx
        mov dx,1000h    ;循环 10000000h 次，读者可以根据自己机器的速度调整循环次数
        mov ax,0
   s1:  sub ax,1
        sbb dx,0
        cmp ax,0
        jne s1
        cmp dx,0
        jne s1
        pop dx
        pop ax
        ret

code ends
end start
```

显示“a”~“z”，并可以让人看清，这个任务已经实现。那么如何实现，按下 Esc

键后，改变显示的颜色呢？

键盘输入到达 60h 端口后，就会引发 9 号中断，CPU 则转去执行 int 9 中断例程。我们可以编写 int 9 中断例程，功能如下。

(1) 从 60h 端口读出键盘的输入；
(2) 调用 BIOS 的 int 9 中断例程，处理其他硬件细节；
(3) 判断是否为 Esc 的扫描码，如果是，改变显示的颜色后返回；如果不是则直接返回。

下面对这些功能的实现一一进行分析。

1. 从端口 60h 读出键盘的输入

```
in al,60h
```

2. 调用 BIOS 的 int 9 中断例程

有一点要注意的是，我们写的中断处理程序要成为新的 int 9 中断例程，主程序必须要将中断向量表中的 int 9 中断例程的入口地址改为我们写的中断处理程序的入口地址。则在新的中断处理程序中调用原来的 int 9 中断例程时，中断向量表中的 int 9 中断例程的入口地址却不是原来的 int 9 中断例程的地址。所以不能使用 int 指令直接调用。

要能在我们写的新中断例程中调用原来的中断例程，就必须在将中断向量表中的中断例程的入口地址改为新地址之前，将原来的入口地址保存起来。这样，在需要调用的时候，我们才能找到原来的中断例程的入口。

对于我们现在的问题，假设将原来 int 9 中断例程的偏移地址和段地址保存在 ds:[0]和 ds:[2]单元中。那么我们在需要调用原来的 int 9 中断例程时，就可以在 ds:[0]、ds:[2]单元中找到它的入口地址。

那么，有了入口地址后，如何进行调用呢？

当然不能使用指令 int 9 来调用。我们可以用别的指令来对 int 指令进行一些模拟，从而实现对中断例程的调用。

我们来看，int 指令在执行的时候，CPU 进行下面的工作。

(1) 取中断类型码 n；
(2) 标志寄存器入栈；
(3) IF=0，TF=0；
(4) CS、IP 入栈；
(5) (IP)=(n*4)，(CS)=(n*4+2)。

取中断类型码是为了定位中断例程的入口地址，在我们的问题中，中断例程的入口地址已经知道。所以，我们用别的指令模拟 int 指令时，不需要做第(1)步。在假设要调用的

中断例程的入口地址在 ds:0 和 ds:2 单元中的前提下，我们将 int 过程用下面几步模拟。

(1) 标志寄存器入栈；
(2) IF=0，TF=0；
(3) CS、IP 入栈；
(4) (IP)=((ds)*16+0)，(CS)=((ds)*16+2)。

可以注意到第(3)、(4)步和 call dword ptr ds:[0]的功能一样，call dword ptr ds:[0]的功能也是：

(1) CS、IP 入栈；
(2) (IP)=((ds)*16+0)，(CS)=((ds)*16+2)。

如果还有疑问，复习 10.6 节的内容。

所以 int 过程的模拟过程变为：

(1) 标志寄存器入栈；
(2) IF=0，TF=0；
(3) call dword ptr ds:[0]。

对于(1)，可用 pushf 实现；

对于(2)，可用下面的指令实现：

```
pushf
pop ax
and ah,11111100b        ;IF 和 TF 为标志寄存器的第 9 位和第 8 位
push ax
popf
```

则模拟 int 指令的调用功能，调用入口地址在 ds:0、ds:2 中的中断例程的程序为：

```
pushf                   ;标志寄存器入栈

pushf
pop ax
and ah,11111100b
push ax
popf                    ;IF=0，TF=0

call dword ptr ds:[0]   ;CS、IP 入栈；(IP)=((ds)*16+0)，(CS)=((ds)*16+2)
```

3. 如果是 Esc 的扫描码，改变显示的颜色后返回

如何改变显示的颜色？

显示的位置是屏幕的中间，即第 12 行 40 列，显存中的偏移地址为：160*12+40*2。所以字符的 ASCII 码要送入段地址 b800h，偏移地址 160*12+40*2 处。而段地址 b800h，

偏移地址 160*12+40*2+1 处是字符的属性，只要改变此处的数据就可以改变在段地址 b800h，偏移地址 160*12+40*2 处显示的字符的颜色了。

该程序的最后一个问题是，要在程序返回前，将中断向量表中的 int 9 中断例程的入口地址恢复为原来的地址。否则程序返回后，别的程序将无法使用键盘。

经过分析，完整的程序如下。

```
assume cs:code

stack segment
 db 128 dup (0)
stack ends

data segment
 dw 0,0
data ends

code segment
start:  mov ax,stack
        mov ss,ax
        mov sp,128

        mov ax,data
        mov ds,ax

        mov ax,0
        mov es,ax

        push es:[9*4]
        pop ds:[0]
        push es:[9*4+2]
        pop ds:[2]      ;将原来的 int 9 中断例程的入口地址保存在 ds:0、ds:2 单元中

        mov word ptr es:[9*4],offset int9
        mov es:[9*4+2],cs       ;在中断向量表中设置新的 int 9 中断例程的入口地址

        mov ax,0b800h
        mov es,ax
        mov ah,'a'
   s:   mov es:[160*12+40*2],ah
        call delay
        inc ah
        cmp ah,'z'
        jna s

        mov ax,0
        mov es,ax

        push ds:[0]
```

```
        pop es:[9*4]
        push ds:[2]
        pop es:[9*4+2]      ;将中断向量表中 int 9 中断例程的入口恢复为原来的地址

        mov ax,4c00h
        int 21h

delay:  push ax
        push dx
        mov dx,1000h
        mov ax,0
  s1:   sub ax,1
        sbb dx,0
        cmp ax,0
        jne s1
        cmp dx,0
        jne s1
        pop dx
        pop ax
        ret

;------以下为新的 int 9 中断例程-----------------

int9:   push ax
        push bx
        push es

        in al,60h

        pushf
        pushf
        pop bx
        and bh,11111100b
        push bx
        popf
        call dword ptr ds:[0]   ;对 int 指令进行模拟，调用原来的 int 9 中断例程

        cmp al,1
        jne int9ret

        mov ax,0b800h
        mov es,ax
        inc byte ptr es:[160*12+40*2+1]         ;将属性值加 1，改变颜色

int9ret:pop es
        pop bx
        pop ax
        iret

code ends
end start
```

注意，本章中所有关于键盘的程序，因要直接访问真实的硬件，则必须在 DOS 实模式下运行。在 Windows 2000 的 DOS 方式下运行，会出现一些和硬件工作原理不符合的现象。

检测点 15.1

(1) 仔细分析一下上面的 int 9 中断例程，看看是否可以精简一下？

其实在我们的 int 9 中断例程中，模拟 int 指令调用原 int 9 中断例程的程序段是可以精简的，因为在进入中断例程后，IF 和 TF 都已经置 0，没有必要再进行设置了。对于程序段：

```
pushf
pushf
pop ax
and ah,11111100b
push ax
popf
call dword ptr ds:[0]
```

可以精简为：

_____

_____

两条指令。

(2) 仔细分析上面程序中的主程序，看看有什么潜在的问题？

在主程序中，如果在执行设置 int 9 中断例程的段地址和偏移地址的指令之间发生了键盘中断，则 CPU 将转去一个错误的地址执行，将发生错误。

找出这样的程序段，改写它们，排除潜在的问题。

提示，注意 sti 和 cli 指令的用法。

15.5 安装新的 int 9 中断例程

下面，我们安装一个新的 int 9 中断例程，使得原 int 9 中断例程的功能得到扩展。

任务：安装一个新的 int 9 中断例程。
功能：在 DOS 下，按 F1 键后改变当前屏幕的显示颜色，其他的键照常处理。

我们进行一下分析。

(1)　改变屏幕的显示颜色

改变从 B8000H 开始的 4000 个字节中的所有奇地址单元中的内容，当前屏幕的显示颜色即发生改变。程序如下：

```
    mov ax,0b800h
    mov es,ax
    mov bx,1
    mov cx,2000
s:  inc byte ptr es:[bx]
    add bx,2
    loop s
```

(2)　其他键照常处理

可以调用原 int 9 中断处理程序，来处理其他的键盘输入。

(3)　原 int 9 中断例程入口地址的保存

因为在编写的新 int 9 中断例程中要调用原 int 9 中断例程，所以，要保存原 int 9 中断例程的入口地址。保存在哪里？显然不能保存在安装程序中，因为安装程序返回后地址将丢失。我们将地址保存在 0:200 单元处。

(4)　新 int 9 中断例程的安装

这个问题在前面已经详细讨论过。我们可将新的 int 9 中断例程安装在 0:204 处。

完整的程序如下。

```
assume cs:code

stack segment
  db 128 dup (0)
stack ends

code segment
start:  mov ax,stack
        mov ss,ax
        mov sp,128

        push cs
        pop ds

        mov ax,0
        mov es,ax

        mov si,offset int9                  ;设置 ds:si 指向源地址
        mov di,204h                         ;设置 es:di 指向目的地址
        mov cx,offset int9end-offset int9   ;设置 cx 为传输长度
        cld                                 ;设置传输方向为正
```

```
        rep movsb

        push es:[9*4]
        pop es:[200h]
        push es:[9*4+2]
        pop es:[202h]

        cli
        mov word ptr es:[9*4],204h
        mov word ptr es:[9*4+2],0
        sti

        mov ax,4c00h
        int 21h

int9:   push ax
        push bx
        push cx
        push es

        in al,60h

        pushf
        call dword ptr cs:[200h]        ;当此中断例程执行时(CS)=0

        cmp al,3bh                      ;F1 的扫描码为 3bh
        jne int9ret

        mov ax,0b800h
        mov es,ax
        mov bx,1
        mov cx,2000
  s:    inc byte ptr es:[bx]
        add bx,2
        loop s

int9ret:pop es
        pop cx
        pop bx
        pop ax
        iret

int9end:nop

code ends
end start
```

这一章中，我们通过对键盘输入的处理，讲解了 CPU 对外设输入的通常处理方法。即：

(1) 外设的输入送入端口；

(2)　向 CPU 发出外中断(可屏蔽中断)信息；

(3)　CPU 检测到可屏蔽中断信息，如果 IF=1，CPU 在执行完当前指令后响应中断，执行相应的中断例程；

(4)　可在中断例程中实现对外设输入的处理。

端口和中断机制，是 CPU 进行 I/O 的基础。

实验 15　安装新的 int 9 中断例程

安装一个新的 int 9 中断例程，功能：在 DOS 下，按下“A”键后，除非不再松开，如果松开，就显示满屏幕的“A”，其他的键照常处理。

提示，按下一个键时产生的扫描码称为通码，松开一个键产生的扫描码称为断码。断码＝通码 ＋80h。

指令系统总结

我们对 8086CPU 的指令系统进行一下总结。读者若要详细了解 8086 指令系统中的各个指令的用法，可以查看有关的指令手册。

8086CPU 提供以下几大类指令。

1. 数据传送指令

比如，mov、push、pop、pushf、popf、xchg 等都是数据传送指令，这些指令实现寄存器和内存、寄存器和寄存器之间的单个数据传送。

2. 算术运算指令

比如，add、sub、adc、sbb、inc、dec、cmp、imul、idiv、aaa 等都是算术运算指令，这些指令实现寄存器和内存中的数据的算数运算。它们的执行结果影响标志寄存器的 sf、zf、of、cf、pf、af 位。

3. 逻辑指令

比如，and、or、not、xor、test、shl、shr、sal、sar、rol、ror、rcl、rcr 等都是逻辑指令。除了 not 指令外，它们的执行结果都影响标志寄存器的相关标志位。

4. 转移指令

可以修改 IP，或同时修改 CS 和 IP 的指令统称为转移指令。转移指令分为以下几类。

(1) 无条件转移指令，比如，jmp；

(2) 条件转移指令，比如，jcxz、je、jb、ja、jnb、jna 等；

(3) 循环指令，比如，loop；

(4) 过程，比如，call、ret、retf；

(5) 中断，比如，int、iret。

5. 处理机控制指令

这些指令对标志寄存器或其他处理机状态进行设置，比如，cld、std、cli、sti、nop、clc、cmc、stc、hlt、wait、esc、lock 等都是处理机控制指令。

6. 串处理指令

这些指令对内存中的批量数据进行处理，比如，movsb、movsw、cmps、scas、lods、stos 等。若要使用这些指令方便地进行批量数据的处理，则需要和 rep、repe、repne 等前缀指令配合使用。

第 16 章　直接定址表

这一章，我们讨论如何有效合理地组织数据，以及相关的编程技术。

16.1　描述了单元长度的标号

前面的课程中，我们一直在代码段中使用标号来标记指令、数据、段的起始地址。比如，下面的程序将 code 段中的 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

code segment

      a: db 1,2,3,4,5,6,7,8
      b: dw 0

start:mov si,offset a
      mov bx,offset b
      mov cx,8
   s: mov al,cs:[si]
      mov ah,0
      add cs:[bx],ax
      inc si
      loop s

      mov ax,4c00h
      int 21h

code ends
end start
```

程序中，code、a、b、start、s 都是标号。这些标号仅仅表示了内存单元的地址。

但是，我们还可以使用一种标号，这种标号不但表示内存单元的地址，还表示了内存单元的长度，即表示在此标号处的单元，是一个字节单元，还是字单元，还是双字单元。上面的程序还可以写成这样：

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

  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
```

在 code 段中使用的标号 a、b 后面没有“:”，它们是同时描述内存地址和单元长度的标号。标号 a，描述了地址 code:0，和从这个地址开始，以后的内存单元都是字节单元；而标号 b 描述了地址 code:8，和从这个地址开始，以后的内存单元都是字单元。

因为这种标号包含了对单元长度的描述，所以在指令中，它可以代表一个段中的内存单元。比如，对于程序中的“b dw 0”：

指令： mov ax,b
相当于：mov ax,cs:[8]

指令： mov b,2
相当于：mov word ptr cs:[8],2

指令： inc b
相当于：inc word ptr cs:[8]

在这些指令中，标号 b 代表了一个内存单元，地址为 code:8，长度为两个字节。

下面的指令会引起编译错误：

```
mov al,b
```

因为 b 代表的内存单元是字单元，而 al 是 8 位寄存器。

如果我们将程序中的指令“add b,ax”，写为“add b,al”，将出现同样的编译错误。

对于程序中的“a db 1,2,3,4,5,6,7,8”：

指令： mov al,a[si]
相当于：mov al,cs:0[si]

指令： mov al,a[3]
相当于：mov al,cs:0[3]

指令： mov al,a[bx+si+3]
相当于：mov al,cs:0[bx+si+3]

可见，使用这种包含单元长度的标号，可以使我们以简洁的形式访问内存中的数据。以后，我们将这种标号称为数据标号，它标记了存储数据的单元的地址和长度。它不同于

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∈{0°，30°，60°，90°，120°，150°，180°}，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：

```
showsin:  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)∈{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)∈{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。

(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 键的通码；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。

1E61															

(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 键的断码，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的时候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，是需要在多处使用的功能，我们应该将它们写为子程序。

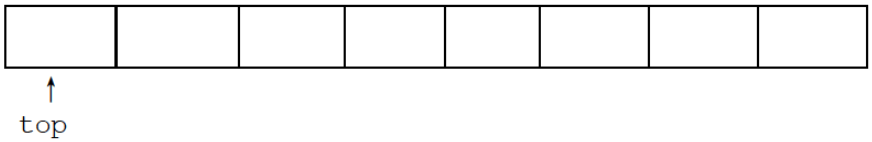
子程序：字符栈的入栈、出栈和显示。
参数说明：　(ah)=功能号，0 表示入栈，1 表示出栈，2 表示显示；
ds:si 指向字符栈空间；
对于 0 号功能：(al)=入栈字符；
对于 1 号功能：(al)=返回的字符；
对于 2 号功能：(dh)、(dl)=字符串在屏幕上显示的行、列位置。

```
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
```

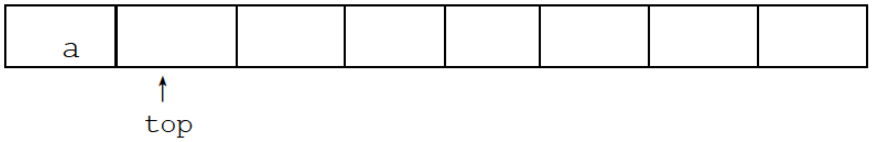
```
sret:       pop es
            pop di
            pop dx
            pop bx
            ret
```

上面的子程序中，字符栈的访问规则如下所示。

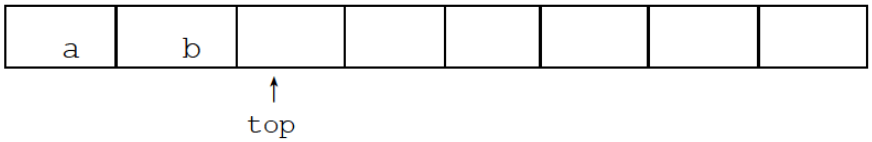
(1)　栈空



(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≈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=int(逻辑扇区号/1440)
磁道号=int(rem(逻辑扇区号/1440)/18)
扇区号= rem(rem(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 ur1.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 ur1.exe。

(2) 用 Debug 加载 ur1.exe，用 u 命令查看 ur1.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 ur1.exe 中 main 函数的代码？

(3) 用下面的方法打印出 ur1.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 ur1.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 ur1.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 um1.c 保存在 c:\minic 下，编译，连接生成 um1.exe。然后用 Debug 加载 um1.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，需要将这些文件和用户.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+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，PentiumⅡ，PentiumⅢ，Pentium4。

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                11111101b：-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                11111101b+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
+(-20)          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∈[-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∈[-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间，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)。

作完以上处理后，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∈[−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 disp∈ [−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 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 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≠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=int(X/65536)
L=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=int(X/65536)；L=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=int(H/n)*n+rem(H/n)

所以有：

(H/n)*65536+(L/n)
=[int(H/n)*n+rem(H/n)]/n*65536+(L/n)
=int(H/n)*65536+rem(H/n)*65536/n+L/n
=int(H/n)*65536+[rem(H/n)*65536+L]/n

现在将计算 X/n 转化为计算：

int(H/n)*65536+[rem(H/n)*65536+L]/n
H=int(X/65536)；L=rem(X/65536)

在这里要进行两次除法运算：

第一次：H/n
第二次：[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：

① L≤65535
② rem(H/n)≤n−1
由②有：
③ rem(H/n)*65536≤(n−1)*65536
由①，③有：
④ rem(H/n)*65536+L≤(n−1)*65536+65535
由④有：
⑤ [rem(H/n)*65536+L]/n≤[(n−1)*65536+65535]/n
由⑤有：
⑥ [rem(H/n)*65536+L]/n≤65536−(1/n)

所以 [rem(H/n)*65536+L]/n 不会产生除法溢出。

则：X/n=int(H/n)*65536+[rem(H/n)*65536+L]/n
H=int(X/65536)；L=rem(X/65536)