

名,在命令行的结尾再加上分号,按 Enter 键后,连接程序就对当前路径下的 1.obj 进行处理,在当前路径下生成可执行文件 1.exe,并在过程中自动忽略中间文件的生成。

4.7 1.exe 的执行

现在,终于将我们的第一个汇编程序加工成了一个可在操作系统下执行的程序文件,我们现在执行一下,图 4.17 展示了 1.exe 的执行情况。



图 4.17 执行 1.exe

奇怪吗?程序运行后,竟然没有任何结果,就和没有运行一样。那么,程序到底运行了吗?

程序当然是运行了,只是从屏幕上不可能看到任何运行结果,因为,我们的程序根本没有向显示器输出任何信息。程序只是做了一些将数据送入寄存器和加法的操作,而这些事情,我们不可能从显示屏上看出来。程序执行完成后,返回,屏幕上再次出现操作系统的提示符(图 4.17 中第 2 行)。

当然,我们不能总是写这样的看不到任何结果的程序,随着课程的进行,我们将会向显示器上输出信息,不过那将是几章以后的事情了,请耐心等待。

4.8 谁将可执行文件中的程序装载进入内存并使它运行?

我们在前面讲过,在 DOS 中,可执行文件中的程序 P1 若要运行,必须有一个正在运行的程序 P2,将 P1 从可执行文件中加载入内存,将 CPU 的控制权交给它,P1 才能得以运行;当 P1 运行完毕后,应该将 CPU 的控制权交还给使它得以运行的程序 P2。

按照上面的原理,再来看一下 4.7 节中 1.exe 的执行过程(思考相关的问题)。

(1) 在提示符“c:\masm”后面输入可执行文件的名字“1”,按 Enter 键。这时,请思考问题 4.1。

(2) 1.exe 中的程序运行。

(3) 运行结束,返回,再次显示提示符“c:\masm”。请思考问题 4.2。

问题 4.1

此时,有一个正在运行的程序将 1.exe 中的程序加载入内存,这个正在运行的程序是什么?它将程序加载入内存后,如何使程序得以运行?

4.9 程序执行过程的跟踪

可以用 Debug 来跟踪一个程序的运行过程，这通常是必须要做的工作。我们写的程序在逻辑上不一定总是正确，对于简单的错误，仔细检查一下源程序就可以发现；而对于隐藏较深的错误，就必须对程序的执行过程进行跟踪分析才容易发现。

下面以在前面的内容中生成的可执行文件 1.exe 为例，讲解如何用 Debug 对程序的执行过程进行跟踪。

现在我们知道，在 DOS 中运行一个程序的时候，是由 command 将程序从可执行文件中加载入内存，并使其得以执行。但是，这样我们不能逐条指令地看到程序的执行过程，因为 command 的程序加载，设置 CS:IP 指向程序的入口的操作是连续完成的，而当 CS:IP 一指向程序的入口，command 就放弃了 CPU 的控制权，CPU 立即开始运行程序，直至程序结束。

为了观察程序的运行过程，可以使用 Debug。Debug 可以将程序加载入内存，设置 CS:IP 指向程序的入口，但 Debug 并不放弃对 CPU 的控制，这样，我们就可以使用 Debug 的相关命令来单步执行程序，查看每一条指令的执行结果。

具体方法如图 4.18 所示。

```
C:\>debug 1.exe
```

图 4.18 用 Debug 加载程序

在提示符后输入“debug 1.exe”，按 Enter 键，Debug 将程序从 1.exe 中加载入内存，进行相关的初始化后设置 CS:IP 指向程序的入口。

接下来可以用 R 命令看一下各个寄存器的设置情况，如图 4.19 所示。

```
C:\>debug 1.exe
-r
AX=0000 BX=0000 CX=000F DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=129E ES=129E SS=12AE CS=12AE IP=0000  NU UP EI PL NZ NA PO NC
12AE:0000 B82301    MOV     AX,0123
```

图 4.19 程序加载后各个寄存器的内容

可以看到，Debug 将程序从可执行文件加载入内存后，cx 中存放的是程序的长度。1.exe 中程序的机器码共有 15 个字节。则 1.exe 加载后，cx 中的内容为 000FH。

现在程序已从 1.exe 中装入内存，接下来查看一下它的内容，可是我们查看哪里的内容呢？程序被装入内存的什么地方？我们如何得知？

这里，需要讲解一下在 DOS 系统中.EXE 文件中的程序的加载过程。图 4.20 针对我

们的问题，简要地展示了这个过程。

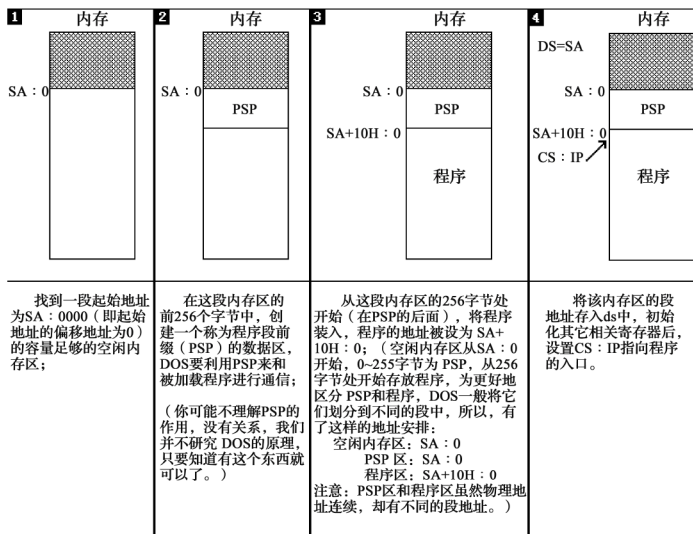


图 4.20 EXE 文件中程序的加载过程

注意,有一步称为重定位的工作在图 4.20 中没有讲解,因为这个问题和操作系统的关系较大,我们不作讨论。

那么,我们的程序被装入内存的什么地方?我们如何得知?从图 4.20 中我们知道以下的信息。

(1) 程序加载后,ds 中存放着程序所在内存区的段地址,这个内存区的偏移地址为 0,则程序所在的内存区的地址为 ds:0;

(2) 这个内存区的前 256 个字节中存放的是 PSP, DOS 用来和程序进行通信。从 256 字节处向后的空间存放的是程序。

所以,从 ds 中可以得到 PSP 的段地址 SA, PSP 的偏移地址为 0,则物理地址为 $SA \times 16 + 0$ 。

因为 PSP 占 256(100H)字节,所以程序的物理地址是:

$$SA \times 16 + 0 + 256 = SA \times 16 + 16 \times 16 + 0 = (SA + 16) \times 16 + 0$$

可用段地址和偏移地址表示为: SA+10H:0。

现在，我们看一下图 4.19 中 DS 的值，DS=129E，则 PSP 的地址为 129E:0，程序的地址为 12AE:0(即 129E+10:0)。

图 4.19 中，CS=12AE，IP=0000，CS:IP 指向程序的第一条指令。注意，源程序中的指令是 `mov ax,0123H`，在 Debug 中记为 `mov ax,0123`，这是因为 Debug 默认所有数据都用十六进制表示。

可以用 U 命令看一下其他指令，如图 4.21 所示。

```
C:\nasm>debug 1.exe
-r
AX=0000 BX=0000 CX=000F DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=129E ES=129E SS=12AE CS=12AE IP=0000  NU UP EI PL NZ NA PO NC
12AE:0000 B82301      MOV     AX,0123
12AE:0000 B82301      MOV     AX,0123
12AE:0003 B85604      MOV     BX,0456
12AE:0006 03C3      ADD     AX,BX
12AE:0008 03C8      ADD     AX,AX
12AE:000A B8004C      MOV     AX,4C00
12AE:000D CD21      INT     21
12AE:000F 83E201      AND     DX,+01
12AE:0012 B885E2      MOV     DX,E285
12AE:0015 2E      CS:
12AE:0016 A385E2      MOV     [E285],AX
12AE:0019 E9A5FC      JMP     FCC1
12AE:001C 803EE70400    CMP     BYTE PTR [04E7],00
```

图 4.21 1.exe 中程序的全部内容

可以看到，从 12AE:0000~12AE:000E 都是程序的机器码。

现在，我们可以开始跟踪了，用 T 命令单步执行程序中的每一条指令，并观察每条指令的执行结果，到了 `int 21`，我们要用 P 命令执行，如图 4.22 所示。

```
AX=00F2 BX=0456 CX=000F DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=129E ES=129E SS=12AE CS=12AE IP=000A  NU UP EI PL NZ AC PO NC
12AE:000A B8004C      MOV     AX,4C00
-t
AX=4C00 BX=0456 CX=000F DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=129E ES=129E SS=12AE CS=12AE IP=000D  NU UP EI PL NZ AC PO NC
12AE:000D CD21      INT     21
-p
Program terminated normally
-
```

图 4.22 程序返回

图 4.22 中，`int 21` 执行后，显示出“Program terminated normally”，返回到 Debug 中。表示程序正常结束。注意，要使用 P 命令执行 `int 21`。这里不必考虑是因为什么，只要记住这一点就可以了。

需要注意的是，在 DOS 中运行程序时，是 `command` 将程序加载入内存，所以程序运行结束后返回到 `command` 中，而在这里是 Debug 将程序加载入内存，所以程序运行结束后要返回到 Debug 中。

使用 Q 命令退出 Debug，将返回到 `command` 中，因为 Debug 是由 `command` 加载运行的。在 DOS 中用“`debug 1.exe`”运行 Debug 对 1.exe 进行跟踪时，程序加载的顺序是：`command` 加载 Debug，Debug 加载 1.exe。返回的顺序是：从 1.exe 中的程序返回到 Debug，从 Debug 返回到 `command`。

实验3 编程、编译、连接、跟踪

- (1) 将下面的程序保存为 t1.asm 文件，将其生成可执行文件 t1.exe。

```
assume cs:codesg
```

```
codesg segment
```

```
    mov ax,2000H
    mov ss,ax
    mov sp,0
    add sp,10
    pop ax
    pop bx
    push ax
    push bx
    pop ax
    pop bx
```

```
    mov ax,4c00H
    int 21H
```

```
codesg ends
```

```
end
```

- (2) 用 Debug 跟踪 t1.exe 的执行过程，写出每一步执行后，相关寄存器中的内容和栈顶的内容。

- (3) PSP 的头两个字节是 CD 20，用 Debug 加载 t1.exe，查看 PSP 的内容。

注意，一定要做完这个实验才能进行下面的课程。

第 5 章 [BX]和 loop 指令

1. [bx]和内存单元的描述

[bx]是什么呢？和[0]有些类似，[0]表示内存单元，它的偏移地址是 0。比如在下面的指令中(在 Debug 中使用)：

```
mov ax, [0]
```

将一个内存单元的内容送入 ax，这个内存单元的长度为 2 字节(字单元)，存放一个字，偏移地址为 0，段地址在 ds 中。

```
mov al, [0]
```

将一个内存单元的内容送入 al，这个内存单元的长度为 1 字节(字节单元)，存放一个字节，偏移地址为 0，段地址在 ds 中。

要完整地描述一个内存单元，需要两种信息：①内存单元的地址；②内存单元的长度(类型)。

用[0]表示一个内存单元时，0 表示单元的偏移地址，段地址默认在 ds 中，单元的长度(类型)可以由具体指令中的其他操作对象(比如说寄存器)指出。

[bx]同样也表示一个内存单元，它的偏移地址在 bx 中，比如下面的指令：

```
mov ax, [bx]
```

将一个内存单元的内容送入 ax，这个内存单元的长度为 2 字节(字单元)，存放一个字，偏移地址在 bx 中，段地址在 ds 中。

```
mov al, [bx]
```

将一个内存单元的内容送入 al，这个内存单元的长度为 1 字节(字节单元)，存放一个字节，偏移地址在 bx 中，段地址在 ds 中。

2. loop

英文单词“loop”有循环的含义，显然这个指令和循环有关。

我们在这一章，讲解[bx]和 loop 指令的应用、意义和相关的内容。

3. 我们定义的描述性的符号：“()”

为了描述上的简洁，在以后的课程中，我们将使用一个描述性的符号“()”来表示一个寄存器或一个内存单元中的内容。比如：

(ax)表示 ax 中的内容、(al)表示 al 中的内容；

(20000H)表示内存 20000H 单元的内容()中的内存单元的地址为物理地址);

((ds)*16+(bx))表示:

ds 中的内容为 ADRI, bx 中的内容为 ADRI, 内存 ADRI*16+ADRI 单元的内容。

也可以理解为: ds 中的 ADRI 作为段地址, bx 中的 ADRI 作为偏移地址, 内存 ADRI:ADRI 单元的内容。

注意,“()”中的元素可以有 3 种类型: ①寄存器名; ②段寄存器名; ③内存单元的
物理地址(一个 20 位数据)。比如:

(ax)、(ds)、(al)、(cx)、(20000H)、((ds)*16+(bx))等是正确的用法;

(2000:0)、((ds):1000H)等是不正确的用法。

我们看一下(X)的应用, 比如,

- (1) ax 中的内容为 0010H, 可以这样来描述: (ax)=0010H;
- (2) 2000:1000 处的内容为 0010H, 可以这样来描述: (21000H)=0010H;
- (3) 对于 mov ax,[2]的功能, 可以这样来描述: (ax)=((ds)*16+2);
- (4) 对于 mov [2],ax 的功能, 可以这样来描述: ((ds)*16+2)=(ax);
- (5) 对于 add ax,2 的功能, 可以这样来描述: (ax)=(ax)+2;
- (6) 对于 add ax,bx 的功能, 可以这样来描述: (ax)=(ax)+(bx);
- (7) 对于 push ax 的功能, 可以这样来描述:
(sp)=(sp)-2
((ss)*16+(sp))=(ax)
- (8) 对于 pop ax 的功能, 可以这样来描述:
(ax)=((ss)*16+(sp))
(sp)=(sp)+2

“(X)”所表示的数据有两种类型: ①字节; ②字。是哪种类型由寄存器名或具体的
运算决定, 比如:

(al)、(bl)、(cl)等得到的数据为字节型; (ds)、(ax)、(bx)等得到的数据为字型。

(al)=(20000H), 则(20000H)得到的数据为字节型; (ax)=(20000H), 则(20000H)得到的
数据为字型。

4. 约定符号 idata 表示常量

我们在 Debug 中写过类似的指令: mov ax,[0], 表示将 ds:0 处的数据送入 ax 中。指
令中, 在 “[...]”里用一个常量 0 表示内存单元的偏移地址。以后, 我们用 idata 表示常
量。比如:

mov ax,[idata] 就代表 mov ax,[1]、mov ax,[2]、mov ax,[3]等。

mov bx,idata 就代表 mov bx,1、mov bx,2、mov bx,3 等。

`mov ds,idata` 就代表 `mov ds,1`、`mov ds,2` 等，它们都是非法指令。

5.1 [BX]

看一看下面指令的功能。

```
mov ax, [bx]
```

功能：bx 中存放的数据作为一个偏移地址 EA，段地址 SA 默认在 ds 中，将 SA:EA 处的数据送入 ax 中。即： $(ax) = ((ds) * 16 + (bx))$ 。

```
mov [bx], ax
```

功能：bx 中存放的数据作为一个偏移地址 EA，段地址 SA 默认在 ds 中，将 ax 中的数据送入内存 SA:EA 处。即： $((ds) * 16 + (bx)) = (ax)$ 。

问题 5.1

程序和内存中的情况如图 5.1 所示，写出程序执行后，21000H~21007H 单元中的内容。

```
mov ax, 2000H
mov ds, ax
mov bx, 1000H
mov ax, [bx]
inc bx
inc bx
mov [bx], ax
inc bx
inc bx
mov [bx], ax
inc bx
mov [bx], al
inc bx
mov [bx], al
```

内存中的情况

BE	21000H
00	21001H
	21002H
	21003H
	21004H
	21005H
	21006H
	21007H

图 5.1 问题 5.1 程序和内存情况

思考后看分析。

注意，`inc bx` 的含义是 bx 中的内容加 1，比如下面两条指令：

```
mov bx, 1
inc bx
```

执行后，bx=2。

分析:

(1) 先看一下程序的前 3 条指令:

```
mov ax,2000H
mov ds,ax
mov bx,1000H
```

这 3 条指令执行后, ds=2000H, bx=1000H。

(2) 接下来, 第 4 条指令:

```
mov ax,[bx]
```

指令执行前: ds=2000H, bx=1000H, 则 mov ax,[bx] 将把内存 2000:1000 处的字型数据送入 ax 中。该指令执行后, ax=00beH。

(3) 接下来, 第 5、6 条指令:

```
inc bx
inc bx
```

这两条指令执行前 bx=1000H, 执行后 bx=1002H。

(4) 接下来, 第 7 条指令:

```
mov [bx],ax
```

指令执行前: ds=2000H, bx=1002H, 则 mov [bx],ax 将把 ax 中的数据送入内存 2000:1002 处。指令执行后, 2000:1002 单元的内容为 BE, 2000:1003 单元的内容为 00。

(5) 接下来, 第 8、9 条指令:

```
inc bx
inc bx
```

这两条指令执行前 bx=1002H, 执行后 bx=1004H。

(6) 接下来, 第 10 条指令:

```
mov [bx],ax
```

指令执行前: ds=2000H, bx=1004H, 则 mov [bx],ax 将把 ax 中的数据送入内存 2000:1004 处。指令执行后, 2000:1004 单元的内容为 BE, 2000:1005 单元的内容为 00。

(7) 接下来, 第 11 条指令:

```
inc bx
```

这条指令执行前 bx=1004H, 执行后 bx=1005H。

(8) 接下来, 第 12 条指令:

```
mov [bx], al
```

指令执行前: ds=2000H, bx=1005H, 则 mov [bx],al 将把 al 中的数据送入内存 2000:1005 处。指令执行后, 2000:1005 单元的内容为 BE。

(9) 接下来, 第 13 条指令:

```
inc bx
```

这条指令执行前 bx=1005H, 执行后 bx=1006H。

(10) 接下来, 第 14 条指令:

```
mov [bx], al
```

指令执行前: ds=2000H, bx=1006H, 则 mov [bx],al 将把 al 中的数据送入内存 2000:1006 处。指令执行后, 2000:1006 单元的内容为 BE。

BE	21000H
00	21001H
BE	21002H
00	21003H
BE	21004H
BE	21005H
BE	21006H
	21007H

图 5.2 内存中的情况

程序执行后, 内存中的情况如图 5.2 所示。

5.2 Loop 指令

loop 指令的格式是: loop 标号, CPU 执行 loop 指令的时候, 要进行两步操作, ①(cx)=(cx)-1; ②判断 cx 中的值, 不为零则转至标号处执行程序, 如果为零则向下执行。

从上面的描述中, 可以看到, cx 中的值影响着 loop 指令的执行结果。通常(注意, 我们说的是通常)我们用 loop 指令来实现循环功能, cx 中存放循环次数。

这里讲解 loop 指令的功能, 关于 loop 指令如何实现转至标号处的细节, 将在后面的课程中讲解。下面我们通过一个程序来看一下 loop 指令的具体应用。

任务 1: 编程计算 2^2 , 结果存在 ax 中。

分析: 设(ax)=2, 可计算(ax)=(ax)*2, 最后(ax)中为 2^2 的值。 $N*2$ 可用 $N+N$ 实现, 程序如下。

```
assume cs:code
code segment
    mov ax, 2
    add ax, ax

    mov ax, 4c00h
```

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

任务 2: 编程计算 2^3 。

分析: $2^3=2*2*2$, 若设(ax)=2, 可计算(ax)=(ax)*2*2, 最后(ax)中为 2^3 的值。 $N*2$ 可用 $N+N$ 实现, 程序如下。

```
assume cs:code
code segment
    mov ax,2
    add ax,ax
    add ax,ax

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

任务 3: 编程计算 2^{12} 。

分析: $2^{12}=2*2*2*2*2*2*2*2*2*2*2*2$, 若设(ax)=2, 可计算(ax)=(ax)*2*2*2*2*2*2*2*2*2*2*2*2, 最后(ax)中为 2^{12} 的值。 $N*2$ 可用 $N+N$ 实现, 程序如下。

```
assume cs:code
code segment
    mov ax,2
    ;做 11 次 add ax,ax
    mov ax,4c00h
int 21h
code ends
end
```

可见, 按照我们的算法, 计算 2^{12} 需要 11 条重复的指令 `add ax,ax`。我们显然不希望这样来写程序, 这里, 可用 `loop` 来简化我们的程序。

程序 5.1

```
assume cs:code
code segment
    mov ax,2

    mov cx,11
s:  add ax,ax
    loop s
```

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

下面分析一下程序 5.1。

(1) 标号

在汇编语言中，标号代表一个地址，程序 5.1 中有一个标号 s。它实际上标识了一个地址，这个地址处有一条指令：add ax,ax。

(2) loop s

CPU 执行 loop s 的时候，要进行两步操作：

- ① (cx)=(cx)-1;
- ② 判断 cx 中的值，不为 0 则转至标号 s 所标识的地址处执行(这里的指令是 add ax,ax)，如果为 0 则执行下一条指令(下一条指令是 mov ax,4c00h)。

(3) 以下 3 条指令

```
mov cx,11
s: add ax,ax
loop s
```

执行 loop s 时，首先要将(cx)减 1，然后若(cx)不为 0，则向前转至 s 处执行 add ax,ax。所以，可以利用 cx 来控制 add ax,ax 的执行次数。

下面我们详细分析一下这段程序的执行过程，从中体会如何用 cx 和 loop s 相配合实现循环功能。

- (1) 执行 mov cx,11，设置(cx)=11;
- (2) 执行 add ax,ax(第 1 次);
- (3) 执行 loop s 将(cx)减 1，(cx)=10，(cx)不为 0，所以转至 s 处;
- (4) 执行 add ax,ax(第 2 次);
- (5) 执行 loop s 将(cx)减 1，(cx)=9，(cx)不为 0，所以转至 s 处;
- (6) 执行 add ax,ax(第 3 次);
- (7) 执行 loop s 将(cx)减 1，(cx)=8，(cx)不为 0，所以转至 s 处;
- (8) 执行 add ax,ax(第 4 次);
- (9) 执行 loop s 将(cx)减 1，(cx)=7，(cx)不为 0，所以转至 s 处;
- (10) 执行 add ax,ax(第 5 次);
- (11) 执行 loop s 将(cx)减 1，(cx)=6，(cx)不为 0，所以转至 s 处;
- (12) 执行 add ax,ax(第 6 次);
- (13) 执行 loop s 将(cx)减 1，(cx)=5，(cx)不为 0，所以转至 s 处;

- (14) 执行 `add ax,ax`(第 7 次);
- (15) 执行 `loop s` 将(cx) 减 1, (cx)=4, (cx) 不为 0, 所以转至 s 处;
- (16) 执行 `add ax,ax`(第 8 次);
- (17) 执行 `loop s` 将(cx) 减 1, (cx)=3, (cx) 不为 0, 所以转至 s 处;
- (18) 执行 `add ax,ax`(第 9 次);
- (19) 执行 `loop s` 将(cx) 减 1, (cx)=2, (cx) 不为 0, 所以转至 s 处;
- (20) 执行 `add ax,ax`(第 10 次);
- (21) 执行 `loop s` 将(cx) 减 1, (cx)=1, (cx) 不为 0, 所以转至 s 处;
- (22) 执行 `add ax,ax`(第 11 次);
- (23) 执行 `loop s` 将(cx) 减 1, (cx)=0, (cx) 为 0, 所以向下执行。(结束循环)

从上面的过程中, 我们可以总结出用 `cx` 和 `loop` 指令相配合实现循环功能的 3 个要点:

- (1) 在 `cx` 中存放循环次数;
- (2) `loop` 指令中的标号所标识地址要在前面;
- (3) 要循环执行的程序段, 要写在标号和 `loop` 指令的中间。

用 `cx` 和 `loop` 指令相配合实现循环功能的程序框架如下。

```
mov cx, 循环次数
s:
    循环执行的程序段
loop s
```

问题 5.2

编程, 用加法计算 123×236 , 结果存在 `ax` 中。思考后看分析。

分析:

可用循环完成, 将 123 加 236 次。可先设 `(ax)=0`, 然后循环做 236 次 `(ax)=(ax)+123`。

程序如下。

程序 5.2

```
assume cs:code
code segment

    mov ax, 0
    mov cx, 236
s: add ax, 123
    loop s
```

```
mov ax,4c00h
int 21h

code ends
end
```

问题 5.3

改进程序 5.2，提高 123*236 的计算速度。思考后看分析。

分析：

程序 5.2 做了 236 次加法，我们可以将 236 加 123 次。可先设(ax)=0，然后循环做 123 次(ax)=(ax)+236，这样可以用 123 次加法实现相同的功能。

5.3 在 Debug 中跟踪用 loop 指令实现的循环程序

考虑这样一个问题，计算 ffff:0006 单元中的数乘以 3，结果存储在 dx 中。

我们分析一下。

(1) 运算后的结果是否会超出 dx 所能存储的范围？

ffff:0006 单元中的数是一个字节型的数据，范围在 0~255 之间，则用它和 3 相乘结果不会大于 65535，可以在 dx 中存放下。

(2) 用循环累加来实现乘法，用哪个寄存器进行累加？

将 ffff:0006 单元中的数赋值给 ax，用 dx 进行累加。先设(dx)=0，然后做 3 次 (dx)=(dx)+(ax)。

(3) ffff:6 单元是一个字节单元，ax 是一个 16 位寄存器，数据的长度不一样，如何赋值？

注意，我们说的是“赋值”，就是说，让 ax 中的数据的值(数据的大小)和 ffff:0006 单元中的数据的值(数据的大小)相等。8 位数据 01H 和 16 位数据 0001H 的数据长度不一样，但它们的值是相等的。

那么我们如何赋值？设 ffff:0006 单元中的数据是 XXH，若要 ax 中的值和 ffff:0006 单元中的相等，ax 中的数据应为 00XXH。所以，若实现 ffff:0006 单元向 ax 赋值，应该令 (ah)=0，(al)=(ffff:6)。

想清楚以上的 3 个问题之后，编写程序如下。

程序 5.3

```

assume cs:code
code segment
    mov ax,0ffffh
    mov ds,ax
    mov bx,6                ;以上,设置ds:bx指向ffff:6

    mov al,[bx]
    mov ah,0                ;以上,设置(al)=((ds*16)+(bx)),(ah)=0

    mov dx,0                ;累加寄存器清0

    mov cx,3                ;循环3次
s: add dx,ax                ;以上累加计算(ax)*3
    loop s

    mov ax,4c00h
    int 21h                ;程序返回

code ends
end

```

注意程序中的第一条指令 `mov ax,0ffffh`。我们知道,大于 9FFFh 的十六进制数据 A000H、A001H...C000H、c001H...FFFEH、FFFFH 等,在书写的时候都是以字母开头的。而在汇编源程序中,数据不能以字母开头,所以要在前面加 0。比如,9138h 在汇编源程序中可以直接写为“9138h”,而 A000h 在汇编源程序中要写为“0A000h”。

下面我们对程序的执行过程进行跟踪。首先,将它编辑为源程序文件,文件名定为 p3.asm; 对其进行编译连接后生成 p3.exe; 然后再用 Debug 对 p3.exe 中的程序进行跟踪。

用 Debug 加载 p3.exe 后,用 r 命令查看寄存器中的内容,如图 5.3 所示。

```

C:\nasnm>debug p3.exe
-r
AX=0000 BX=0000 CX=001B DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=0B2D ES=0B2D SS=0B3D CS=0B3D IP=0000  NU UP EI PL NZ NA PO NC
0B3D:0000 B8FFFF      MOV     AX,FFFF

```

图 5.3 用 r 命令查看寄存器

图 5.3 中(ds)=0B2DH, 所以, 程序在 0B3D:0 处(如果读者还不清楚这是为什么, 可以复习 4.9 节的内容)。我们看一下, (cs)=0B3DH, (IP)=0, CS:IP 正指向程序的第一条指令。再用 u 命令看一下被 Debug 加载入内存的程序, 如图 5.4 所示。

可以看到, 从 0B3D:0000~0B3D:001A 是我们的程序, 0B3D:0014 处是源程序中的指令 `loop s`, 只是此处 `loop s` 中的标号 `s` 已经变为一个地址 0012h。如果在执行“loop 0012”时, cx 减 1 后不为 0, “loop 0012”就把 IP 设置为 0012h, 从而使 CS:IP 指向 0B3D:0012 处的 `add dx,ax`, 实现转跳。

```

-u 0B3D:0
0B3D:0000 B8FFFF      MOV     AX,FFFF
0B3D:0003 8ED8        MOV     DS,AX
0B3D:0005 B80600      MOV     BX,0006
0B3D:0008 8A07        MOV     AL,[BX]
0B3D:000A B400        MOV     AH,00
0B3D:000C BA0000      MOV     DX,0000
0B3D:000F B90000      MOV     CX,0003
0B3D:0012 03D0        ADD     DX,AX
0B3D:0014 E2FC        LOOP    0012
0B3D:0016 B8004C      MOV     AX,4C00
0B3D:0019 CD21        INT     21
0B3D:001B E83E0D      CALL    0D5C
0B3D:001E 83C404      ADD     SP,+04

```

图 5.4 用 u 命令查看被 Debug 加载入内存的程序

我们开始跟踪，如图 5.5 所示。

```

-r
AX=FFFF BX=0000 CX=0010 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B20 SS=0B3D CS=0B3D IP=0000  NO UP EI PL NZ NA PO NC
0B3D:0000 B8FFFF      MOV     AX,FFFF
-t
AX=FFFF BX=0000 CX=0010 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B20 SS=0B3D CS=0B3D IP=0003  NO UP EI PL NZ NA PO NC
0B3D:0003 8ED8        MOV     DS,AX
-t
AX=FFFF BX=0000 CX=0010 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B20 SS=0B3D CS=0B3D IP=0005  NO UP EI PL NZ NA PO NC
0B3D:0005 B80600      MOV     BX,0006
-t
AX=FFFF BX=0006 CX=0010 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B20 SS=0B3D CS=0B3D IP=0008  NO UP EI PL NZ NA PO NC
0B3D:0008 8A07        MOV     AL,[BX]
DS:0006=32
-r

```

图 5.5 ds:bx 指向 ffff:6 单元

图 5.5 中，前 3 条指令执行后，(ds)=ffffh，(bx)=6，ds:bx 指向 ffff:6 单元。Debug 显示出当前要执行的指令“mov al,[bx]”，因为是读取内存的指令，所以 Debug 将要访问的内存单元中的内容也显示出来，可以看到屏幕最右边显示的“ds:0006=32”，由此，我们可以方便地知道目标单元(ffff:6)中的内容是 32h。

继续执行，如图 5.6 所示。

```

AX=FFFF BX=0006 CX=0010 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B20 SS=0B3D CS=0B3D IP=0008  NO UP EI PL NZ NA PO NC
0B3D:0008 8A07        MOV     AL,[BX]
DS:0006=32
-t
AX=FF32 BX=0006 CX=0010 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B20 SS=0B3D CS=0B3D IP=000A  NO UP EI PL NZ NA PO NC
0B3D:000A B400        MOV     AH,00
-t
AX=0032 BX=0006 CX=0010 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B20 SS=0B3D CS=0B3D IP=000C  NO UP EI PL NZ NA PO NC
0B3D:000C BA0000      MOV     DX,0000
-r

```

图 5.6 从 ffff:6 单元向 ax 赋值

图 5.6 中，这两条指令执行后，(ax)=0032h，完成了从 ffff:6 单元向 ax 的赋值。

继续，如图 5.7 所示。

```

AX=0032 BX=0006 CX=0010 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=000C  NO UP EI PL NZ NA PO NC
0B3D:000C 8A0000    MOV     DX,0000
-t
AX=0032 BX=0006 CX=0010 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=000F  NO UP EI PL NZ NA PO NC
0B3D:000F B90000    MOV     CX,0003
-t
AX=0032 BX=0006 CX=0003 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0012  NO UP EI PL NZ NA PO NC
0B3D:0012 0300     ADD     DX,AX

```

图 5.7 初始化累加寄存器和循环计数器

图 5.7 中, 这两条指令执行后, (dx)=0, 完成对累加寄存器的初始化; (cx)=3, 完成对循环计数寄存器的初始化。

下面, 将开始循环程序段的执行。我们继续, 如图 5.8 所示。

```

AX=0032 BX=0006 CX=0003 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0012  NO UP EI PL NZ NA PO NC
0B3D:0012 03D0     ADD     DX,AX
-t
AX=0032 BX=0006 CX=0003 DX=0032 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0014  NO UP EI PL NZ NA PO NC
0B3D:0014 E2FC     LOOP    0012
-t
AX=0032 BX=0006 CX=0002 DX=0032 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0012  NO UP EI PL NZ NA PO NC
0B3D:0012 03D0     ADD     DX,AX

```

图 5.8 第一次循环

图 5.8 中, CPU 执行 0B3D:0012 处的指令“add dx,ax”后, (IP)=0014h, CS:IP 指向 0B3D:0014 处的指令“loop 0012”。CPU 执行“loop 0012”, 第一步先将(cx)减 1, (cx)=2; 第二步因(cx)不等于 0, 将 IP 设为 0012h。指令“loop 0012”执行后, (IP)=0012h, CS:IP 再次指向 0B3D:0012 处的指令“add dx,ax”, 这条指令将再次得到执行。注意, “loop 0012”执行后(cx)=2, 也就是说, “loop 0012”还可以进行两次循环。

接着, 将重复执行“add dx,ax”和“loop 0012”, 直到(cx)=0 为止, 如图 5.9 所示。

```

AX=0032 BX=0006 CX=0002 DX=0032 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0012  NO UP EI PL NZ NA PO NC
0B3D:0012 03D0     ADD     DX,AX
-t
AX=0032 BX=0006 CX=0002 DX=0064 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0014  NO UP EI PL NZ NA PO NC
0B3D:0014 E2FC     LOOP    0012
-t
AX=0032 BX=0006 CX=0001 DX=0064 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0012  NO UP EI PL NZ NA PO NC
0B3D:0012 03D0     ADD     DX,AX
-t
AX=0032 BX=0006 CX=0001 DX=0096 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0014  NO UP EI PL NZ NA PO NC
0B3D:0014 E2FC     LOOP    0012
-t
AX=0032 BX=0006 CX=0000 DX=0096 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0016  NO UP EI PL NZ NA PO NC
0B3D:0016 B8004C    MOV     AX,4C00

```

图 5.9 得到计算结果

注意图 5.9 中，最后一次执行“loop 0012”的结果。执行前(cx)=1，CPU 执行“loop 0012”，第一步，(cx)=(cx)-1，(cx)=0；第二步，因为(cx)=0，所以 loop 指令不转跳，(IP)=0016h，CPU 向下执行 0B3D:0016 处的指令“mov ax,4c00”。

在完成最后一次“add dx,ax”后，(dx)=96h，此时 dx 中为累加计算(ax)*3 的最后结果。

我们继续，将程序执行完，如图 5.10 所示。

```

AX=0032  BX=0006  CX=0000  DX=0096  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0016  NO UP EI PL NZ NA PE NC
0B3D:0016  B8004C      mov     ax,4c00
-t
AX=4C00  BX=0006  CX=0000  DX=0096  SP=0000  BP=0000  SI=0000  DI=0000
DS=FFFF  ES=0B2D  SS=0B3D  CS=0B3D  IP=0019  NO UP EI PL NZ NA PE NC
0B3D:0019  CD21      int     21
-p
Program terminated normally

```

图 5.10 程序返回

图 5.10 中，执行完最后两条指令后，程序返回到 Debug 中。注意“int 21”要用 p 命令执行。

上面，我们通过对一个循环程序的跟踪，更进一步地讲解了 loop 指令实现循环的原理。下面，我们将程序 5.3 改一下，计算 ffff:0006 单元中的数乘以 123，结果存储在 dx 中。

这很容易完成，只要将循环的次数改为 123 就可以了。程序如下。

程序5.4

```

assume cs:code
code segment
    mov ax,0ffffh
    mov ds,ax
    mov bx,6                ;以上，设置 ds:bx 指向 ffff:6

    mov al,[bx]
    mov ah,0                ;以上，设置 (al)=((ds*16)+(bx))，(ah)=0

    mov dx,0                ;累加寄存器清 0

    mov cx,123              ;循环 123 次
s:  add dx,ax
    loop s                  ;以上累加计算 (ax)*123

    mov ax,4c00h            ;程序返回
    int 21h

```

```
code ends
end
```

我们用 Debug 对这个程序的循环程序段进行跟踪, 现在有这样一个问题: 前面的 7 条指令, 即标号 s 前的指令, 已经确定在逻辑上完全正确, 我们不想再一步步地跟踪了, 只想跟踪循环的过程。所以希望可以一次执行完标号 s 前的指令。可以用一个新的(对我们来说是新的, 因为以前没用过)Debug 命令 g 来达到目的。

下面来实际操作一下, 我们用程序 5.4 生成最终的可执行文件“c:\masm\p4.exe”, 用 Debug 加载 p4.exe, 然后看一下程序在内存中的情况, 如图 5.11 所示。

图 5.11 中, 循环程序段从 CS:0012 开始, CS:0012 前面的指令, 我们不想再一步步地跟踪, 希望能够一次执行完, 然后从 CS:0012 处开始跟踪。可以这样来使用 g 命令, “g 0012”, 它表示执行程序到当前代码段(段地址在 CS 中)的 0012h 处。也就是说“g 0012”将使 Debug 从当前的 CS:IP 指向的指令执行, 一直到(IP)=0012h 为止。具体的情况如图 5.12 所示。

```
C:\masm>debug p4.exe
-r
AX=0000 BX=0000 CX=001B DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=0B2D ES=0B2D SS=0B3D CS=0B3D IP=0000
0B3D:0000 B0FFFF      MOV     AX,FFFF
-u 0B3D:0
0B3D:0000 B0FFFF      MOV     AX,FFFF
0B3D:0003 8ED8          MOV     DS,AX
0B3D:0005 B06000          MOV     BX,0006
0B3D:0008 8A07          MOV     AL,CX1
0B3D:000A D400          MOV     AH,00
0B3D:000C B00000          MOV     DX,0000
0B3D:000F B97B00          MOV     CX,007B
0B3D:0012 03D0          ADD     DX,AX
0B3D:0014 E2FC          LOOP   0012
0B3D:0016 B0004C          MOV     AX,4C00
0B3D:0019 CD21          INT     21
0B3D:001B EB3E0D          CALL   0D5C
0B3D:001E 83C404          ADD     SP,+04
-
```

图 5.11 程序在内存中的情况

```
-g 0012
AX=0032 BX=0006 CX=007B DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0012
0B3D:0012 03D0          ADD     DX,AX
```

图 5.12 CS:0012 前的程序段被执行

图 5.12 中, Debug 执行“g 0012”后, CS:0012 前的程序段被执行, 从各个相关的寄存器中的值, 我们可以看出执行的结果。

下面我们对循环的过程进行跟踪, 如图 5.13 所示。

图 5.13 中, 我们跟踪了两次循环的过程。其实, 通过这两次循环过程, 已经可以确定循环程序段在逻辑上是正确的。我们不想再继续一步步地观察循环的过程了, 怎样让程序向下执行呢? 继续像从前那样使用 t 命令? 显然这是不可行的, 因为还要进行 $121 \times (cx) = 79h$ 次循环, 如果像前两次那样使用 t 命令, 我们得使用 $121 \times 2 = 242$ 次 t 命令才

能从循环中出来。

```

AX=0032 BX=0006 CX=007B DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0012 NU UP EI PL NZ NA PO NC
0B3D:0012 03D0 ADD DX,AX
-t
AX=0032 BX=0006 CX=007B DX=0032 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0014 NU UP EI PL NZ NA PO NC
0B3D:0014 E2FC LOOP 0012
-t
AX=0032 BX=0006 CX=007B DX=0032 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0012 NU UP EI PL NZ NA PO NC
0B3D:0012 03D0 ADD DX,AX
-t
AX=0032 BX=0006 CX=007A DX=0064 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0014 NU UP EI PL NZ NA PO NC
0B3D:0014 E2FC LOOP 0012
-t
AX=0032 BX=0006 CX=0079 DX=0064 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0012 NU UP EI PL NZ NA PO NC
0B3D:0012 03D0 ADD DX,AX

```

图 5.13 两次循环的过程

这里的问题是，我们希望将循环一次执行完。可以使用 p 命令来达到目的。再次遇到 loop 指令时，使用 p 命令来执行，Debug 就会自动重复执行循环中的指令，直到(cx)=0 为止。具体情况如图 5.14 所示。

```

AX=0032 BX=0006 CX=0079 DX=0064 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0012 NU UP EI PL NZ NA PO NC
0B3D:0012 03D0 ADD DX,AX
-t
AX=0032 BX=0006 CX=0079 DX=0096 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0014 NU UP EI PL NZ NA PE NC
0B3D:0014 E2FC LOOP 0012
-p
AX=0032 BX=0006 CX=0000 DX=1806 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0016 NU UP EI PL NZ NA PE NC
0B3D:0016 B8004C MOV AX,4C00

```

图 5.14 用 p 命令执行 loop 指令

图 5.14 中，在遇到“loop 0012”时，用 p 命令执行，Debug 自动重复执行“loop 0012”和“add dx,ax”两条指令，直到(cx)=0。最后一次执行“loop 0012”后，(cx)=0，(IP)=0016h，当前指令为 CS:0016 处的“mov ax,4c00”。

当然，也可以用 g 命令来达到目的，可以用“g 0016”直接执行到 CS:0016 处。具体情况如图 5.15 所示。

```

AX=0032 BX=0006 CX=0079 DX=0064 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0012 NU UP EI PL NZ NA PO NC
0B3D:0012 03D0 ADD DX,AX
-g 0016
AX=0032 BX=0006 CX=0000 DX=1806 SP=0000 BP=0000 SI=0000 DI=0000
DS=FFFF ES=0B2D SS=0B3D CS=0B3D IP=0016 NU UP EI PL NZ NA PE NC
0B3D:0016 B8004C MOV AX,4C00

```

图 5.15 用 g 命令执行

5.4 Debug 和汇编编译器 masm 对指令的不同处理

本节知识点为下面课程的顺利进行提供一点预备知识。

我们在 Debug 中写过类似的指令：

```
mov ax,[0]
```

表示将 ds:0 处的数据送入 ax 中。

但是在汇编源程序中，指令“mov ax,[0]”被编译器当作指令“mov ax,0”处理。

下面通过具体的例子来看一下 Debug 和汇编编译器 masm 对形如“mov ax,[0]”这类指令的不同处理。

任务：将内存 2000:0、2000:1、2000:2、2000:3 单元中的数据送入 al、bl、cl、dl 中。

(1) 在 Debug 中编程实现：

```
mov ax,2000
mov ds,ax
mov al,[0]
mov bl,[1]
mov cl,[2]
mov dl,[3]
```

(2) 汇编源程序实现：

```
assume cs:code
code segment

    mov ax,2000h
    mov ds,ax
    mov al,[0]
    mov bl,[1]
    mov cl,[2]
    mov dl,[3]

    mov ax,4c00h
    int 21h

code ends
end
```

我们看一下两种实现的实际实施情况：

(1) Debug 中的情况如图 5.16 所示。

```

C:\masm>debug
~a
00E8:0100 mov ax,2000
00E8:0103 mov ds,ax
00E8:0105 mov al,[0]
00E8:0108 mov bl,[1]
00E8:010C mov cl,[2]
00E8:0110 mov dl,[3]
00E8:0114
~u
00E8:0100 B80020      MOV     AX,2000
00E8:0103 8ED0       MOV     DS,AX
00E8:0105 A00000     MOV     AL,[0000]
00E8:0108 8A1E0100   MOV     BL,[0001]
00E8:010C 8A0E0200   MOV     CL,[0002]
00E8:0110 8A160300   MOV     DL,[0003]

```

图 5.16 Debug 对 “mov al,[0]” 等指令的解释

(2) 将汇编源程序存储为 compare.asm, 用 masm、link 生成 compare.exe, 用 Debug 加载 compare.exe, 如图 5.17 所示。

```

C:\masm>debug compare.exe
~r
AX=0000 BX=0000 CX=0012 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=002D ES=002D SS=0B3D CS=0B3D IP=0000  NU UP EI PL NZ NA PO NC
00B3D:0000 B80020      MOV     AX,2000
~u 003d:0000
00B3D:0000 B80020      MOV     AX,2000
00B3D:0003 8ED0       MOV     DS,AX
00B3D:0005 B000       MOV     AL,00
00B3D:0007 E301       MOV     BL,01
00B3D:0009 B102       MOV     CL,02
00B3D:000B E203       MOV     DL,03
00B3D:000D B8004C     MOV     AX,4C00
00B3D:0010 CD21       INT     21

```

图 5.17 masm 对 “mov al,[0]” 等指令的解释

从图 5.16、图 5.17 中我们可以明显地看出, Debug 和编译器 masm 对形如 “mov ax, [0]” 这类指令在解释上的不同。我们在 Debug 中和源程序中写入同样形式的指令: “mov al,[0]”、“mov bl,[1]”、“mov cl,[2]”、“mov dl,[3]”, 但 Debug 和编译器对这些指令中的 “[idata]” 却有不同解释。Debug 将它解释为 “[idata]” 是一个内存单元, “[idata]” 是内存单元的偏移地址; 而编译器将 “[idata]” 解释为 “idata”。

那么我们如何在源程序中实现将内存 2000:0、2000:1、2000:2、2000:3 单元中的数据送入 al、bl、cl、dl 中呢?

目前的方法是, 可将偏移地址送入 bx 寄存器中, 用 [bx] 的方式来访问内存单元。比如我们可以这样访问 2000:0 单元:

```

mov ax,2000h
mov ds,ax           ;段地址 2000h 送入 ds
mov bx,0            ;偏移地址 0 送入 bx
mov al,[bx]         ;ds:bx 单元中的数据送入 al

```

这样做是可以, 可是比较麻烦, 我们要用 bx 来间接地给出内存单元的偏移地址。我们还是希望能够像在 Debug 中那样, 在 “[]” 中直接给出内存单元的偏移地址。这样做, 在汇编源程序中也是可以的, 只不过, 要在 “[]” 的前面显式地给出段地址所在的段寄存器。比如我们可以这样访问 2000:0 单元:

```
mov ax,2000h
mov ds,ax
mov al,ds:[0]
```

比较一下汇编源程序中以下指令的含义。

- “mov al,[0]”，含义：(al)=0，将常量0送入al中(与mov al,0含义相同)；
- “mov al,ds:[0]”，含义：(al)=((ds)*16+0)，将内存单元中的数据送入al中；
- “mov al,[bx]”，含义：(al)=((ds)*16+(bx))，将内存单元中的数据送入al中；
- “mov al,ds:[bx]”，含义：与“mov al,[bx]”相同。

从上面的比较中可以看出：

(1) 在汇编源程序中，如果用指令访问一个内存单元，则在指令中必须用“[...]”来表示内存单元，如果在“[]”里用一个常量idata直接给出内存单元的偏移地址，就要在“[]”的前面显式地给出段地址所在的段寄存器。比如

```
mov al,ds:[0]
```

如果没有在“[]”的前面显式地给出段地址所在的段寄存器，比如

```
mov al,[0]
```

那么，编译器masm将把指令中的“[idata]”解释为“idata”。

(2) 如果在“[]”里用寄存器，比如bx，间接给出内存单元的偏移地址，则段地址默认在ds中。当然，也可以显式地给出段地址所在的段寄存器。

5.5 loop和[bx]的联合应用

考虑这样一个问题，计算ffff:0~ffff:b单元中的数据的和，结果存储在dx中。

我们还是先分析一下。

(1) 运算后的结果是否会超出dx所能存储的范围？

ffff:0~ffff:b内存单元中的数据是字节型数据，范围在0~255之间，12个这样的数据相加，结果不会大于65535，可以在dx中存放下。

(2) 我们能否将ffff:0~ffff:b中的数据直接累加到dx中？

当然不行，因为ffff:0~ffff:b中的数据是8位的，不能直接加到16位寄存器dx中。

(3) 我们能否将ffff:0~ffff:b中的数据累加到dl中，并设置(dh)=0，从而实现累加到dx中？

这也不行，因为 `di` 是 8 位寄存器，能容纳的数据的范围在 0~255 之间，`ffff:0~ffff:b` 中的数据也都是 8 位，如果仅向 `di` 中累加 12 个 8 位数据，很有可能造成进位丢失。

(4) 我们到底怎样将 `ffff:0~ffff:b` 中的 8 位数据，累加到 16 位寄存器 `dx` 中？

从上面的分析中，可以看到，这里面有两个问题：类型的匹配和结果的不超界。具体地说，就是在做加法的时候，我们有两种方法：

- ① `(dx)=(dx)+内存中的 8 位数据`；
- ② `(di)=(di)+内存中的 8 位数据`。

第一种方法中的问题是两个运算对象的类型不匹配，第二种方法中的问题是结果有可能超界。

怎样解决这两个看似矛盾的问题？目前的方法(在后面的课程中我们还有别的方法)就是得用一个 16 位寄存器来做中介。将内存单元中的 8 位数据赋值到一个 16 位寄存器 `ax` 中，再将 `ax` 中的数据加到 `dx` 上，从而使两个运算对象的类型匹配并且结果不会超界。

想清楚以上的问题之后，编写程序如下。

程序5.5

```
assume cs:code

code segment

    mov ax,0ffffh
    mov ds,ax           ;设置(ds)=ffffh

    mov dx,0            ;初始化累加寄存器，(dx)=0

    mov al,ds:[0]
    mov ah,0            ;(ax)=(ds)*16+0=(ffff0h)
    add dx,ax           ;向dx中加上ffff:0单元的数据

    mov al,ds:[1]
    mov ah,0            ;(ax)=(ds)*16+1=(ffff1h)
    add dx,ax           ;向dx中加上ffff:1单元的数据

    mov al,ds:[2]
    mov ah,0            ;(ax)=(ds)*16+2=(ffff2h)
    add dx,ax           ;向dx中加上ffff:2单元的数据

    mov al,ds:[3]
    mov ah,0            ;(ax)=(ds)*16+3=(ffff3h)
    add dx,ax           ;向dx中加上ffff:3单元的数据

    mov al,ds:[4]
    mov ah,0            ;(ax)=(ds)*16+4=(ffff4h)
    add dx,ax           ;向dx中加上ffff:4单元的数据

    mov al,ds:[5]
```

```

mov ah,0                ; (ax)=(ds)*16+5)=(ffff5h)
add dx,ax                ; 向 dx 中加上 ffff:5 单元的数值

mov al,ds:[6]
mov ah,0                ; (ax)=(ds)*16+6)=(ffff6h)
add dx,ax                ; 向 dx 中加上 ffff:6 单元的数值

mov al,ds:[7]
mov ah,0                ; (ax)=(ds)*16+7)=(ffff7h)
add dx,ax                ; 向 dx 中加上 ffff:7 单元的数值

mov al,ds:[8]
mov ah,0                ; (ax)=(ds)*16+8)=(ffff8h)
add dx,ax                ; 向 dx 中加上 ffff:8 单元的数值

mov al,ds:[9]
mov ah,0                ; (ax)=(ds)*16+9)=(ffff9h)
add dx,ax                ; 向 dx 中加上 ffff:9 单元的数值

mov al,ds:[0ah]
mov ah,0                ; (ax)=(ds)*16+0ah)=(ffffah)
add dx,ax                ; 向 dx 中加上 ffff:a 单元的数值

mov al,ds:[0bh]
mov ah,0                ; (ax)=(ds)*16+0bh)=(ffffbh)
add dx,ax                ; 向 dx 中加上 ffff:b 单元的数值

mov ax,4c00h            ; 程序返回
int 21h

code ends

end

```

上面的程序很简单,不用解释,你一看就懂。不过,在看懂了之后,你是否觉得这个程序编得有些问题?它似乎没有必要写这么长。这是累加 `ffff:0~ffff:b` 中的 12 个数据,如果要累加 `0000:0~0000:7fff` 中的 32K 个数据,按照这个程序的思路,将要写将近 10 万行程序(写一个简单的操作系统也就这个长度了)。

问题 5.4

应用 `loop` 指令,改进程序 5.5,使它的指令行数让人能够接受。

思考后看分析。

分析:

可以看出,在程序中,有 12 个相似的程序段,我们将它们一般化地描述为:

```

mov al,ds:[X]            ; ds:X 指向 ffff:X 单元
mov ah,0                 ; (ax)=(ds)*16+(X)=(ffffXh)
add dx,ax                 ; 向 dx 中加上 ffff:X 单元的数值

```

我们可以看到，12 个相似的程序段中，只有 `mov al,ds:[X]` 指令中的内存单元的偏移地址是不同的，其他都一样。而这些不同的偏移地址是在 $0 \leq X \leq \text{bH}$ 的范围内递增变化的。

我们可以用数学语言来描述这个累加的运算：
$$\text{sum} = \sum_{X=0}^{\text{bH}} (\text{ffffh} * 10\text{h} + X)。$$

从程序实现上，我们将循环做。

```
(al) = ((ds) * 16 + X)
(ah) = 0
(dx) = (dx) + (ax)
```

一共循环 12 次，在循环开始前 $(\text{ds}) = \text{ffffh}$ ， $X = 0$ ， ds:X 指向第一个内存单元。每次循环后， X 递增， ds:X 指向下一个内存单元。

完整的算法描述如下。

初始化：

```
(ds) = ffffh
X = 0
(dx) = 0
```

循环 12 次：

```
(al) = ((ds) * 16 + X)
(ah) = 0
(dx) = (dx) + (ax)
X = X + 1
```

可见，表示内存单元偏移地址的 X 应该是一个变量，因为在循环的过程中，偏移地址必须能够递增。这样，在指令中，我们就不能用常量来表示偏移地址。我们可以将偏移地址放到 `bx` 中，用 `[bx]` 的方式访问内存单元。在循环开始前设 $(\text{bx}) = 0$ ，每次循环，将 `bx` 中的内容加 1 即可。

最后一个是，如何实现循环 12 次？我们的 `loop` 指令该发挥作用了。

更详细的算法描述如下。

初始化：

```
(ds) = ffffh
(bx) = 0
(dx) = 0
(cx) = 12
```

循环 12 次:

```
s: (al)=(ds)*16+(bx)
    (ah)=0
    (dx)=(dx)+(ax)
    (bx)=(bx)+1
    loop s
```

最后, 我们写出程序。

程序 5.6

```
assume cs:code
code segment

    mov ax,0ffffh
    mov ds,ax
    mov bx,0           ;初始化 ds:bx 指向 ffff:0

    mov dx,0           ;初始化累加寄存器 dx, (dx)=0

    mov cx,12          ;初始化循环计数寄存器 cx, (cx)=12

s:  mov al,[bx]
    mov ah,0
    add dx,ax           ;间接向 dx 中加上 ((ds)*16+(bx)) 单元的数值
    inc bx              ;ds:bx 指向下一个单元
    loop s

    mov ax,4c00h
    int 21h

code ends
end
```

在实际编程中, 经常会遇到, 用同一种方法处理地址连续的内存单元中的数据的问题。我们需要用循环来解决这类问题, 同时我们必须能够在每次循环的时候按照同一种方法来改变要访问的内存单元的地址。这时, 就不能用常量来给出内存单元的地址(比如, [0]、[1]、[2]中, 0、1、2 是常量), 而应用变量。“mov al,[bx]”中的 bx 就可以看作一个代表内存单元地址的变量, 我们可以不写新的指令, 仅通过改变 bx 中的数值, 改变指令访问的内存单元。

5.6 段 前 缀

指令“mov ax,[bx]”中, 内存单元的偏移地址由 bx 给出, 而段地址默认在 ds 中。我们可以在访问内存单元的指令中显式地给出内存单元的段地址所在的段寄存器。比如:

(1) `mov ax,ds:[bx]`

将一个内存单元的内容送入 `ax`，这个内存单元的长度为 2 字节(字单元)，存放一个字，偏移地址在 `bx` 中，段地址在 `ds` 中。

(2) `mov ax,cs:[bx]`

将一个内存单元的内容送入 `ax`，这个内存单元的长度为 2 字节(字单元)，存放一个字，偏移地址在 `bx` 中，段地址在 `cs` 中。

(3) `mov ax,ss:[bx]`

将一个内存单元的内容送入 `ax`，这个内存单元的长度为 2 字节(字单元)，存放一个字，偏移地址在 `bx` 中，段地址在 `ss` 中。

(4) `mov ax,es:[bx]`

将一个内存单元的内容送入 `ax`，这个内存单元的长度为 2 字节(字单元)，存放一个字，偏移地址在 `bx` 中，段地址在 `es` 中。

(5) `mov ax,ss:[0]`

将一个内存单元的内容送入 `ax`，这个内存单元的长度为 2 字节(字单元)，存放一个字，偏移地址为 0，段地址在 `ss` 中。

(6) `mov ax,cs:[0]`

将一个内存单元的内容送入 `ax`，这个内存单元的长度为 2 字节(字单元)，存放一个字，偏移地址为 0，段地址在 `cs` 中。

这些出现在访问内存单元的指令中，用于显式地指明内存单元的段地址的“`ds:`”“`cs:`”“`ss:`”“`es:`”，在汇编语言中称为**段前缀**。

5.7 一段安全的空间

在 8086 模式中，随意向一段内存空间写入内容是很危险的，因为这段空间中可能存放着重要的系统数据或代码。比如下面的指令：

```
mov ax,1000h
mov ds,ax
mov al,0
mov ds:[0],al
```

我们以前在 Debug 中，为了讲解上的方便，写过类似的指令。但这种做法是不合理的，因为之前我们并没有论证过 1000:0 中是否存放着重要的系统数据或代码。如果 1000:0 中存放着重要的系统数据或代码，“`mov ds:[0],al`”将其改写，将引发错误。

比如下面的程序。

程序 5.7

```
assume cs:code
code segment

    mov ax,0
    mov ds,ax
    mov ds:[26h],ax

    mov ax,4c00h
    int 21h

code ends
end
```

将源程序编辑为 p7.asm, 编译、连接后生成 p7.exe, 用 Debug 加载, 跟踪它的运行, 如图 5.18 所示。

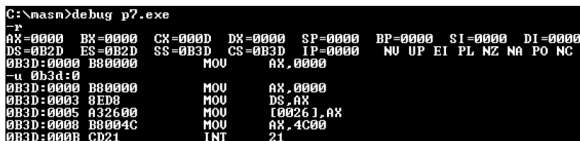


图 5.18 用 Debug 加载程序 5.7

图 5.18 中, 我们可以看到, 源程序中的“mov ds:[26h],ax”被 masm 翻译为机器码“a3 26 00”, 而 Debug 将这个机器码解释为“mov [0026],ax”。可见, 汇编源程序中的汇编指令“mov ds:[26h],ax”和 Debug 中的汇编指令“mov [0026],ax”同义。

我们看一下“mov [0026],ax”的执行结果, 如图 5.19 所示。

图 5.19 中, 是在 windows 2000 的 DOS 方式中, 在 Debug 里执行“mov [0026],ax”的结果。如果在实模式(即纯 DOS 方式)下执行程序 p7.exe, 将会引起死机。产生这种结果的原因是 0:0026 处存放着重要的系统数据, 而“mov [0026],ax”将其改写。

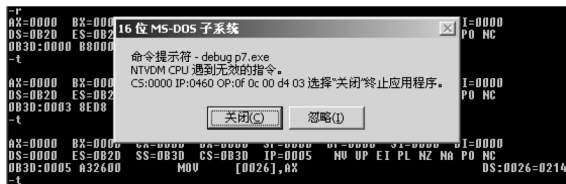


图 5.19 改写 0:0026 处存放的重要系统数据

可见，在不能确定一段内存空间中是否存放着重要的数据或代码的时候，不能随意向其中写入内容。

不要忘记，我们是在操作系统的环境中工作，操作系统管理所有的资源，也包括内存。如果我们需要向内存空间写入数据的话，要使用操作系统给我们分配的空间，而不应直接用地址任意指定内存单元，向里面写入。下一章我们会对“使用操作系统给我们分配的空间”有所认识。

但是，同样不能忘记，我们正在学习的是汇编语言，要通过它来获得底层的编程体验，理解计算机底层的基本工作机理。所以我们尽量直接对硬件编程，而不去理会操作系统。

我们似乎面临一种选择，是在操作系统中安全、规矩地编程，还是自由、直接地用汇编语言去操作真实的硬件，了解那些早已被层层系统软件掩盖的真相？在大部分的情况下，我们选择后者，除非我们就是在学习操作系统本身的内容。

注意，我们在纯 DOS 方式(实模式)下，可以不理睬 DOS，直接用汇编语言去操作真实的硬件，因为运行在 CPU 实模式下的 DOS，没有能力对硬件系统进行全面、严格的管理。但在 Windows 2000、Unix 这些运行于 CPU 保护模式下的操作系统中，不理睬操作系统，用汇编语言去操作真实的硬件，是根本不可能的。硬件已被这些操作系统利用 CPU 保护模式所提供的功能全面而严格地管理了。

在后面的课程中，我们需要直接向内存中写入内容，可我们又不希望发生图 5.19 中的那种情况。所以要找到一段安全的空间供我们使用。在一般的 PC 机中，DOS 方式下，DOS 和其他合法的程序一般都不会使用 0:200~0:2ff(00200h~002ffh)的 256 个字节的空间。所以，我们使用这段空间是安全的。不过为了谨慎起见，在进入 DOS 后，我们可以先用 Debug 查看一下，如果 0:200~0:2ff 单元的内容都是 0 的话，则证明 DOS 和其他合法的程序没有使用这里。

为什么 DOS 和其他合法的程序一般都不会使用 0:200~0:2ff 这段空间？我们将在以后的课程中讨论这个问题。

好了，我们总结一下：

- (1) 我们需要直接向一段内存中写入内容；
- (2) 这段内存空间不应存放系统或其他程序的数据或代码，否则写入操作很可能引发错误；
- (3) DOS 方式下，一般情况，0:200~0:2ff 空间中没有系统或其他程序的数据或代码；
- (4) 以后，我们需要直接向一段内存中写入内容时，就使用 0:200~0:2ff 这段空间。

5.8 段前缀的使用

我们考虑一个问题, 将内存 `ffff:0~ffff:b` 单元中的数据复制到 `0:200~0:20b` 单元中。

分析一下。

- (1) `0:200~0:20b` 单元等同于 `0020:0~0020:b` 单元, 它们描述的是同一段内存空间。
- (2) 复制的过程应用循环实现, 简要描述如下。

初始化:

`X=0`

循环 12 次:

将 `ffff:X` 单元中的数据送入 `0020:X` (需要用一个寄存器中转)

`X=X+1`

(3) 在循环中, 源始单元 `ffff:X` 和目标单元 `0020:X` 的偏移地址 `X` 是变量。我们用 `bx` 来存放。

(4) 将 `0:200~0:20b` 用 `0020:0~0020:b` 描述, 就是为了使目标单元的偏移地址和源始单元的偏移地址从同一数值 `0` 开始。

程序如下。

程序5.8

```
assume cs:code

code segment

    mov bx,0           ; (bx)=0, 偏移地址从 0 开始
    mov cx,12          ; (cx)=12, 循环 12 次

s:   mov ax,0ffffh
    mov ds,ax           ; (ds)=0ffffh
    mov dl,[bx]         ; (dl)=( (ds)*16+(bx) ), 将 ffff:bx 中的数据送入 dl

    mov ax,0020h
    mov ds,ax           ; (ds)=0020h
    mov [bx],dl         ; ( (ds)*16+(bx) )=(dl), 将中 dl 的数据送入 0020:bx

    inc bx              ; (bx)=(bx)+1
    loop s

    mov ax,4c00h
    int 21h

code ends

end
```

因源始单元 ffff:X 和目标单元 0020:X 相距大于 64KB, 在不同的 64KB 段里, 程序 5.8 中, 每次循环要设置两次 ds。这样做是正确的, 但是效率不高。我们可以使用两个段寄存器分别存放源始单元 ffff:X 和目标单元 0020:X 的段地址, 这样就可以省略循环中需要重复做 12 次的设置 ds 的程序段。

改进的程序如下。

程序 5.9

```
assume cs:code

code segment

    mov ax,0ffffh
    mov ds,ax           ; (ds)=0ffffh

    mov ax,0020h
    mov es,ax           ; (es)=0020h

    mov bx,0             ; (bx)=0, 此时 ds:bx 指向 ffff:0, es:bx 指向 0020:0

    mov cx,12            ; (cx)=12, 循环 12 次

s:   mov dl,[bx]          ; (dl)=( (ds)*16+(bx) ), 将 ffff:bx 中的数据送入 dl
     mov es:[bx],dl      ; ( (es)*16+(bx) )=(dl), 将 dl 中的数据送入 0020:bx
     inc bx              ; (bx)=(bx)+1
     loop s

    mov ax,4c00h
    int 21h

code ends

end
```

程序 5.9 中, 使用 es 存放目标空间 0020:0~0020:b 的段地址, 用 ds 存放源始空间 ffff:0~ffff:b 的段地址。在访问内存单元的指令 “mov es:[bx].al” 中, 显式地用段前缀 “es:” 给出单元的段地址, 这样就不必在循环中重复设置 ds。

实验 4 [bx]和 loop 的使用

- (1) 编程, 向内存 0:200~0:23F 依次传送数据 0~63(3FH)。
- (2) 编程, 向内存 0:200~0:23F 依次传送数据 0~63(3FH), 程序中只能使用 9 条指令, 9 条指令中包括 “mov ax,4c00h” 和 “int 21h”。
- (3) 下面的程序的功能是将 “mov ax,4c00h” 之前的指令复制到内存 0:200 处, 补全程序。上机调试, 跟踪运行结果。

```
assume cs:code

code segment
```

```
mov ax, _____  
mov ds, ax  
mov ax, 0020h  
mov es, ax  
mov bx, 0  
mov cx, _____  
s: mov al, [bx]  
mov es: [bx], al  
inc bx  
loop s  
mov ax, 4c00h  
int 21h  
code ends  
end
```

提示:

- (1) 复制的是什么? 从哪里到哪里?
- (2) 复制的是什么? 有多少个字节? 你如何知道要复制的字节的数量?

注意, 一定要做完这个实验才能进行下面的课程。

第 6 章 包含多个段的程序

前面的程序中，只有一个代码段。现在有一个问题是，如果程序需要用其他空间来存放数据，使用哪里呢？第 5 章中，我们讲到要使用一段安全的空间。可哪里安全呢？第 5 章中，我们说 0:200~0:2FF 是相对安全的，可这段空间的容量只有 256 个字节，如果我们需要的空间超过 256 个字节该怎么办呢？

在操作系统的环境中，合法地通过操作系统取得的空間都是安全的，因为操作系统不会让一个程序所用的空间和其他程序以及系统自己的空间相冲突。在操作系统允许的情况下，程序可以取得任意容量的空间。

程序取得所需空间的方法有两种，一是在加载程序的时候为程序分配，再就是程序在执行的过程中向系统申请。在我们的课程中，不讨论第二种方法。

加载程序的时候为程序分配空间，我们在前面已经有所体验，比如我们的程序在加载的时候，取得了代码段中的代码的存储空间。

我们若要一个程序在被加载的时候取得所需的空間，则必须要在源程序中做出说明。我们通过定义段来进行内存空间的获取。

上面是从内存空间获取的角度上，谈定义段的问题。我们再从程序规划的角度来谈一下定义段的问题。大多数有用的程序，都要处理数据，使用栈空间，当然也都必须有指令，为了程序设计上的清晰和方便，我们一般也都定义不同的段来存放它们。

对于使用多个段的问题，我们先简单说到这里，下面我们将以这样的顺序来深入地讨论多个段的问题：

- (1) 在一个段中存放数据、代码、栈，我们先来体会一下不使用多个段时的情况；
- (2) 将数据、代码、栈放入不同的段中。

6.1 在代码段中使用数据

考虑这样一个问题，编程计算以下 8 个数据的和，结果存在 ax 寄存器中：

0123h、0456h、0789h、0abch、0defh、0fedh、0cbah、0987h

在前面的课程中，我们都是累加某些内存单元中的数据，并不关心数据本身。可现在要累加的就是已经给定了数值的数据。我们可以将它们一个一个地加到 ax 寄存器中，但是，我们希望能用循环的方法来进行累加，所以在累加前，要将这些数据存储在一组地址连续的内存单元中。如何将这些数据存储在一组地址连续的内存单元中呢？我们可以用指令一个一个地将它们送入地址连续的内存单元中，可是这样又有一个问题，到哪里去找

这段内存空间呢？

从规范的角度来讲，我们是不能自己随便决定哪段空间可以使用的，应该让系统来为我们分配。我们可以在程序中，定义我们希望处理的数据，这些数据就会被编译、连接程序作为程序的一部分写到可执行文件中。当可执行文件中的程序被加载入内存时，这些数据也同时被加载入内存中。与此同时，我们要处理的数据也就自然而然地获得了存储空间。

具体的做法看下面的程序。

程序6.1

```
assume cs:code

code segment

    dw 0123h,0456h,0789h,0abch,0defh,0fedh,0cbah,0987h

    mov bx,0
    mov ax,0

    mov cx,8
s:add ax,cs:[bx]
    add bx,2
    loop s

    mov ax,4c00h
    int 21h

code ends

end
```

解释一下，程序第一行中的“dw”的含义是定义字型数据。dw 即“define word”。在这里，使用 dw 定义了 8 个字型数据(数据之间以逗号分隔)，它们所占的内存空间的大小为 16 个字节。

程序中的指令就要对这 8 个数据进行累加，可这 8 个数据在哪里呢？由于它们在代码段中，程序在运行的时候 CS 中存放代码段的段地址，所以可以从 CS 中得到它们的段地址。它们的偏移地址是多少呢？因为用 dw 定义的数据处于代码段的最开始，所以偏移地址为 0，这 8 个数据就在代码段的偏移 0、2、4、6、8、A、C、E 处。程序运行时，它们的地址就是 CS:0、CS:2、CS:4、CS:6、CS:8、CS:A、CS:C、CS:E。

程序中，用 bx 存放加 2 递增的偏移地址，用循环来进行累加。在循环开始前，设置 (bx)=0，cs:bx 指向第一个数据所在的字单元。每次循环中(bx)=(bx)+2，cs:bx 指向下一个数据所在的字单元。

将程序 6.1 编译、连接为可执行文件 p61.exe，先不要运行，用 Debug 加载查看一下，情况如图 6.1 所示。

```

C:\masm>debug p61.exe
-r
AX=0000 BX=0000 CX=0026 DX=0000 SP=0000 BP=0000 SI=0000 DI=0000
DS=0B2D ES=0B2D SS=0B3D CS=0B3D IP=0000      NU UP EI PL NZ NA PO NC
0B3D:0000 2301          AND     AX,[BX+01]
-u
0B3D:0000 2301          AND     AX,[BX+01]
0B3D:0002 56          PUSH    SI
0B3D:0003 0489        ADD     AL,89
0B3D:0005 07          POP     ES
0B3D:0006 BCDAEF     MOV     SP,EFAA
0B3D:0009 0DE0DF     OR      AX,0FED
0B3D:000C BADC87     MOV     DX,870C
0B3D:000F 09B80000    OR      [BP+01+0000],DI

```

图 6.1 用 u 命令从 0B3D:0000 查看程序

图 6.1 中，通过“DS=0B2D”，可知道程序从 0B3D:0000 开始存放。用 u 命令从 0B3D:0000 查看程序，却看到了一些让人读不懂的指令。

为什么没有看到程序中的指令呢？实际上用 u 命令从 0B3D:0000 查看到的也是程序中的内容，只不过不是源程序中的汇编指令所对应的机器码，而是源程序中，在汇编指令前面，用 dw 定义的数据。实际上，在程序中，有一个代码段，在代码段中，前面的 16 个字节是用“dw”定义的数据，从第 16 个字节开始才是汇编指令所对应的机器码。

可以用 d 命令更清楚地查看一下程序中前 16 个字节的内容，如图 6.2 所示。

```

-d 0b3d:0
0B3D:0000 23 01 56 04 89 07 0C 0A-EF 0D ED 0F BA 0C 87 09 #.V.....
0B3D:0010 B8 00 00 B8 00 00 B9 08-00 2E 03 07 83 C3 02 E2 .....
0B3D:0020 F8 B8 00 4C 0C 21 EA 99-0A C0 74 EA F6 C7 80 74 ...L.l...t...t
0B3D:0030 05 C6 06 15 98 01 E9 2D-01 3A C3 75 05 80 CF 80 ...:..:..:..:
0B3D:0040 EB 04 3C 0D 75 03 E9 18-01 3A 06 06 96 75 03 E9 ...<.u...:..u..
0B3D:0050 17 01 B2 3A 38 14 75 10-80 3E A4 98 01 75 03 E8 ...8.u...>..u..
0B3D:0060 EB ED E8 5C 01 AC E8 58-01 89 3E E6 99 C6 06 E8 ...\.X.\>..>..
0B3D:0070 99 00 E9 B3 00 89 3E E6-99 C6 06 E8 99 00 80 3E ...>.....>

```

图 6.2 查看程序中前 16 个字节的内容

可以从 0B3D:0010 查看程序中要执行的机器指令，如图 6.3 所示。

```

-u 0b3d:0010
0B3D:0010 B80000     MOV     BX,0000
0B3D:0013 B80000     MOV     AX,0000
0B3D:0016 B90000     MOV     CX,0000
0B3D:0019 2E          CS:
0B3D:001A 0307        ADD     AX,[BX]
0B3D:001C 33C302     ADD     BX,+02
0B3D:001F E2F8        LOOP    0019
0B3D:0021 B8004C     MOV     AX,4C00
0B3D:0024 C021        INT     21
0B3D:0026 E8990AC074 JMP     74C0:0A99
0B3D:002B EAF6C78074 JMP     7480:C7F6

```

图 6.3 查看程序中的机器指令

从图 6.2 和 6.3 中，我们可以看到程序加载到内存中后，所占内存空间的前 16 个单元存放在源程序中用“dw”定义的数据，后面的单元存放源程序中汇编指令所对应的机器指令。

怎样执行程序中的指令呢？用 Debug 加载后，可以将 IP 设置为 10h，从而使 CS:IP 指向程序中的第一条指令。然后再用 t 命令、p 命令，或者是 g 命令执行。

可是这样一来,我们就必须用 **Debug** 来执行程序。程序 6.1 编译、连接成可执行文件后,在系统中直接运行可能会出现問題,因为程序的入口处不是我们所希望执行的指令。如何让这个程序在编译、连接后可以在系统中直接运行呢?我们可以在源程序中指明程序的入口所在,具体做法如下。

程序6.2

```
assume cs:code

code segment

    dw 0123h,0456h,0789h,0abch,0defh,0fedh,0cbah,0987h

    start:    mov bx,0
              mov ax,0

              mov cx,8
s:           add ax,cs:[bx]
              add bx,2
              loop s

              mov ax,4c00h
              int 21h

code ends

end start
```

注意在程序 6.2 中加入的新内容,在程序的第一条指令的前面加上了一个标号 **start**,而这个标号在伪指令 **end** 的后面出现。这里,我们要再次探讨 **end** 的作用。**end** 除了通知编译器程序结束外,还可以通知编译器程序的入口在什么地方。在程序 6.2 中我们用 **end** 指令指明了程序的入口在标号 **start** 处,也就是说,“**mov bx,0**”是程序的第一条指令。

在前面的课程中(参见 4.8 节),我们已经知道在单任务系统中,可执行文件中的程序执行过程如下。

- (1) 由其他的程序(**Debug**、**command** 或其他程序)将可执行文件中的程序加载入内存;
- (2) 设置 **CS:IP** 指向程序的第一条要执行的指令(即程序的入口),从而使程序得以运行;
- (3) 程序运行结束后,返回到加载者。

现在的问题是,根据什么设置 CPU 的 **CS:IP** 指向程序的第一条要执行的指令?也就是说,如何知道哪一条指令是程序的第一条要执行的指令?这一点,是由可执行文件中的描述信息指明的。我们知道可执行文件由描述信息和程序组成,程序来自于源程序中的汇编指令和定义的数据;描述信息则主要是编译、连接程序对源程序中相关伪指令进行处理所得到的信息。我们在程序 6.2 中,用伪指令 **end** 描述了程序的结束和程序的入口。在编译、连接后,由“**end start**”指明的程序入口,被转化为一个入口地址,存储在可执行文

件的描述信息中。在程序 6.2 生成的可执行文件中，这个入口地址的偏移地址部分为：10H。当程序被加载入内存之后，加载者从程序的可执行文件的描述信息中读到程序的入口地址，设置 CS:IP。这样 CPU 就从我们希望的地址处开始执行。

归根结底，我们若要 CPU 从何处开始执行程序，只要在源程序中用“end 标号”指明就可以了。

有了这种方法，就可以这样来安排程序的框架：

```
assume cs:code

code segment
    :
    :
    数据
    :
    :
start:
    :
    :
    代码
    :
    :
code ends

end start
```

6.2 在代码段中使用栈

完成下面的程序，利用栈，将程序中定义的数据逆序存放。

```
assume cs:codesg

codesg segment

    dw 0123h,0456h,0789h,0abch,0defh,0fedh,0cbah,0987h
    ?
codesg ends

end
```

程序的思路大致如下。

程序运行时，定义的数据存放在 cs:0~cs:F 单元中，共 8 个字单元。依次将这 8 个字单元中的数据入栈，然后再依次出栈到这 8 个字单元中，从而实现数据的逆序存放。

问题是，我们首先要有一段可当作栈的内存空间。如前所述，这段空间应该由系统来分配。可以在程序中通过定义数据来取得一段空间，然后将这段空间当作栈空间来用。程序如下。

程序6.3

```

assume cs:codesg

codesg segment

    dw 0123h,0456h,0789h,0abch,0defh,0fedh,0cbah,0987h

    dw 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
        ;用 dw 定义 16 个字型数据,在程序加载后,将取得 16 个字的
        ;内存空间,存放这 16 个数据。在后面的程序中将这段
        ;空间当作栈来使用

start:  mov ax,cs
        mov ss,ax
        mov sp,30h    ;将设置栈顶 ss:sp 指向 cs:30

        mov bx,0
        mov cx,8
s:      push cs:[bx]
        add bx,2
        loop s        ;以上将代码段 0~15 单元中的 8 个字型数据依次入栈

        mov bx,0
        mov cx,8
s0:     pop cs:[bx]
        add bx,2
        loop s0       ;以上依次出栈 8 个字型数据到代码段 0~15 单元中

        mov ax,4c00h
        int 21h

codesg ends

end start                ;指明程序的入口在 start 处

```

注意程序 6.3 中的指令:

```

mov ax,cs
mov ss,ax
mov sp,30h

```

我们要将 cs:10~cs:2F 的内存空间当作栈来用,初始状态下栈为空,所以 ss:sp 要指向栈底,则设置 ss:sp 指向 cs:30。如果对这点还有疑惑,建议回头认真复习一下第 3 章。

在代码段中定义了 16 个字型数据,它们的数值都是 0。这 16 个字型数据的值是多少,对程序来说没有意义。我们用 dw 定义 16 个数据,即在程序中写入了 16 个字型数据,而程序在加载后,将用 32 个字节的内存空间来存放它们。这段内存空间是我们所需要的,程序将它用作栈空间。可见,我们定义这些数据的最终目的是,通过它们取得一定容量的内存空间。所以我们在描述 dw 的作用时,可以说用它定义数据,也可以说用它开辟内存空间。比如对于:

```
dw 0123h,0456h,0789h,0abch,0defh,0fedh,0cbah,0987h
```

可以说，定义了8个字型数据，也可以说，开辟了8个字的内存空间，这段空间中每个字单元中的数据依次是：0123h、0456h、0789h、0abch、0defh、0fedh、0cbah、0987h。因为它们最终的效果是一样的。

检测点 6.1

(1) 下面的程序实现依次用内存 0:0~0:15 单元中的内容改写程序中的数据，完成程序：

```
assume cs:codesg

codesg segment

    dw 0123h,0456h,0789h,0abch,0defh,0fedh,0cbah,0987h

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

      mov cx,8
s:     mov ax,[bx]

      _____

      add bx,2
      loop s

      mov ax,4c00h
      int 21h

codesg ends

end start
```

(2) 下面的程序实现依次用内存 0:0~0:15 单元中的内容改写程序中的数据，数据的传送用栈来进行。栈空间设置在程序内。完成程序：

```
assume cs:codesg

codesg segment

    dw 0123h,0456h,0789h,0abch,0defh,0fedh,0cbah,0987h

    dw 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ;10 个字单元用作栈空间

start: mov ax,_____
      mov ss,ax
      mov sp,_____

      mov ax,0
      mov ds,ax
      mov bx,0
      mov cx,8
```

```
s: push [bx]

    _____
    add bx,2
    loop s

    mov ax,4c00h
    int 21h

codesg ends

end start
```

6.3 将数据、代码、栈放入不同的段

在前面的内容中，我们在程序中用到了数据和栈，将数据、栈和代码都放到了一个段里面。我们在编程的时候要注意何处是数据，何处是栈，何处是代码。这样做显然有两个问题：

- (1) 把它们放到一个段中使程序显得混乱；
- (2) 前面程序中处理的数据很少，用到的栈空间也小，加上没有多长的代码，放到一个段里面没有问题。但如果数据、栈和代码需要的空间超过 64KB，就不能放在一个段中（一个段的容量不能大于 64KB，是我们在学习中所用的 8086 模式的限制，并不是所有的处理器都这样）。

所以，应该考虑用多个段来存放数据、代码和栈。

怎样做呢？我们用和定义代码段一样的方法来定义多个段，然后在这些段里面定义需要的数据，或通过定义数据来取得栈空间。具体做法如下面的程序所示，这个程序实现了和程序 6.3 一样的功能，不同之处在于它将数据、栈和代码放到了不同的段中。

程序6.4

```
assume cs:code,ds:data,ss:stack

data segment

    dw 0123h,0456h,0789h,0abch,0defh,0fedh,0cbah,0987h

data ends

stack segment

    dw 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

stack ends

code segment

start: mov ax,stack
       mov ss,ax
```

```

mov sp,20h    ;设置栈顶 ss:sp 指向 stack:20

mov ax,data
mov ds,ax     ;ds 指向 data 段

mov bx,0      ;ds:bx 指向 data 段中的第一个单元

mov cx,8
s: push [bx]
add bx,2
loop s        ;以上将 data 段中的 0~15 单元中的 8 个字型数据依次入栈

mov bx,0

mov cx,8
s0: pop [bx]
add bx,2
loop s0       ;以上依次出栈 8 个字型数据到 data 段的 0~15 单元中

mov ax,4c00h
int 21h

code ends

end start

```

下面对程序 6.4 做出说明。

(1) 定义多个段的方法

这点，我们从程序中可明显地看出，定义一个段的方法和前面所讲的定义代码段的方法没有区别，只是对于不同的段，要有不同的段名。

(2) 对段地址的引用

现在，程序中有多个段了，如何访问段中的数据呢？当然要通过地址，而地址是分为两部分的，即段地址和偏移地址。如何指明要访问的数据的段地址呢？在程序中，段名就相当于一个标号，它代表了段地址。所以指令“mov ax,data”的含义就是将名称为“data”的段的段地址送入 ax。一个段中的数据的段地址可由段名代表，偏移地址就要看它在段中的位置了。程序中“data”段中的数据“0abch”的地址就是：data:6。要将它送入 bx 中，就要用如下的代码：

```

mov ax,data
mov ds,ax
mov bx,ds:[6]

```

我们不能用下面的指令：

```

mov ds,data
mov bx,ds:[6]

```

其中指令“mov ds,data”是错误的，因为 8086CPU 不允许将一个数值直接送入段寄

寄存器中。程序中对段名的引用,如指令“`mov ds,data`”中的“`data`”,将被编译器处理为一个表示段地址的数值。

(3) “代码段”、“数据段”、“栈段”完全是我们的安排

现在,我们以一个具体的程序来再次讨论一下所谓的“代码段”、“数据段”、“栈段”。在汇编源程序中,可以定义许多的段,比如在程序 6.4 中,定义了 3 个段,“`code`”、“`data`”和“`stack`”。我们可以分别安排它们存放代码、数据和栈。那么我们如何让 CPU 按照我们的这种安排来执行这个程序呢?下面来看看源程序中对这 3 个段所做的处理。

① 我们在源程序中为这 3 个段起了具有含义的名称,用来放数据的段我们将其命名为“`data`”,用来放代码的段我们将其命名为“`code`”,用作栈空间的段命名为“`stack`”。

这样命名了之后,CPU 是否就去执行“`code`”段中的内容,处理“`data`”段中的数据,将“`stack`”当做栈了呢?

当然不是,我们这样命名,仅仅是为了使程序便于阅读。这些名称同“`start`”、“`s`”、“`s0`”等标号一样,仅在源程序中存在,CPU 并不知道它们。

② 我们在源程序中用伪指令“`assume cs:code,ds:data,ss:stack`”将 `cs`、`ds` 和 `ss` 分别和 `code`、`data`、`stack` 段相连。这样做了之后,CPU 是否就会将 `cs` 指向 `code`,`ds` 指向 `data`,`ss` 指向 `stack`,从而按照我们的意图来处理这些段呢?

当然也不是,要知道 `assume` 是伪指令,是由编译器执行的,也是仅在源程序中存在的信息,CPU 并不知道它们。我们不必深究 `assume` 的作用,只要知道需要用它将你定义的具有一定用途的段和相关的寄存器联系起来就可以了。

③ 若要 CPU 按照我们的安排行事,就要用机器指令控制它,源程序中的汇编指令是 CPU 要执行的内容。CPU 如何知道去执行它们?我们在源程序的最后用“`end start`”说明了程序的入口,这个入口将被写入可执行文件的描述信息,可执行文件中的程序被加载入内存后,CPU 的 `CS:IP` 被设置指向这个入口,从而开始执行程序中的第一条指令。标号“`start`”在“`code`”段中,这样 CPU 就将 `code` 段中的内容当作指令来执行了。我们在 `code` 段中,使用指令:

```
mov ax,stack
mov ss,ax
mov sp,20h
```

设置 `ss` 指向 `stack`,设置 `ss:sp` 指向 `stack:20`,CPU 执行这些指令后,将把 `stack` 段当做栈空间来用。CPU 若要访问 `data` 段中的数据,则可用 `ds` 指向 `data` 段,用其他的寄存器(如 `bx`)来存放 `data` 段中数据的偏移地址。

总之,CPU 到底如何处理我们定义的段中的内容,是当作指令执行,当作数据访

问, 还是当作栈空间, 完全是靠程序中具体的汇编指令, 和汇编指令对 CS:IP、SS:SP、DS 等寄存器的设置来决定的。完全可以将程序 6.4 写成下面的样子, 实现同样的功能。

```

assume cs:b,ds:a,ss:c

a segment
    dw 0123h,0456h,0789h,0abch,0defh,0fedh,0cbah,0987h
a ends

c segment
    dw 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
c ends

b segment

d: mov ax,c
    mov ss,ax
    mov sp,20h          ;希望用 c 段当作栈空间, 设置 ss:sp 指向 c:20

    mov ax,a
    mov ds,ax          ;希望用 ds:bx 访问 a 段中的数据, ds 指向 a 段

    mov bx,0           ;ds:bx 指向 a 段中的第一个单元
    mov cx,8

s: push [bx]
    add bx,2
    loop s              ;以上将 a 段中的 0~15 单元中的 8 个字型数据依次入栈

    mov bx,0
    mov cx,8
s0: pop [bx]
    add bx,2
    loop s0             ;以上依次出栈 8 个字型数据到 a 段的 0~15 单元中

    mov ax,4c00h
    int 21h

b ends

end d                  ;d 处是要执行的第一条指令, 即程序的入口

```

实验 5 编写、调试具有多个段的程序

这一章的内容较少, 有些知识需要在实践中掌握。这个实验, 既是实验, 也是学习内容。必须完成这个实验, 才能继续向下学习。

(1) 将下面的程序编译、连接, 用 Debug 加载、跟踪, 然后回答问题。

```

assume cs:code,ds:data,ss:stack

data segment
    dw 0123h,0456h,0789h,0abch,0defh,0fedh,0cbah,0987h
data ends

```

```
stack segment
    dw 0,0,0,0,0,0,0,0
stack ends
```

```
code segment
```

```
start: mov ax,stack
       mov ss,ax
       mov sp,16

       mov ax,data
       mov ds,ax

       push ds:[0]
       push ds:[2]
       pop ds:[2]
       pop ds:[0]

       mov ax,4c00h
       int 21h
```

```
code ends
```

```
end start
```

- ① CPU 执行程序，程序返回前，**data** 段中的数据为多少？
- ② CPU 执行程序，程序返回前，cs=_____、ss=_____、ds=_____。
- ③ 设程序加载后，**code** 段的段地址为 X，则 **data** 段的段地址为_____，**stack** 段的段地址为_____。

(2) 将下面的程序编译、连接，用 Debug 加载、跟踪，然后回答问题。

```
assume cs:code,ds:data,ss:stack
```

```
data segment
    dw 0123H,0456H
data ends
```

```
stack segment
    dw 0,0
stack ends
```

```
code segment
```

```
start: mov ax,stack
       mov ss,ax
       mov sp,16

       mov ax,data
       mov ds,ax

       push ds:[0]
       push ds:[2]
```

```

        pop ds:[2]
        pop ds:[0]

        mov ax,4c00h
        int 21h

code ends

end start

```

- ① CPU 执行程序，程序返回前，**data** 段中的数据为多少？
- ② CPU 执行程序，程序返回前，**cs**=_____、**ss**=_____、**ds**=_____。
- ③ 设程序加载后，**code** 段的段地址为 **X**，则 **data** 段的段地址为_____，**stack** 段的段地址为_____。
- ④ 对于如下定义的段：

```

name segment
:
name ends

```

如果段中的数据占 **N** 个字节，则程序加载后，该段实际占有的空间为_____。

- (3) 将下面的程序编译、连接，用 **Debug** 加载、跟踪，然后回答问题。

```

assume cs:code,ds:data,ss:stack

code segment

start:  mov ax,stack
        mov ss,ax
        mov sp,16

        mov ax,data
        mov ds,ax

        push ds:[0]
        push ds:[2]
        pop ds:[2]
        pop ds:[0]

        mov ax,4c00h
        int 21h

code ends

data segment

dw 0123H,0456H

data ends

stack segment

dw 0,0

```

```
stack ends
```

```
end start
```

- ① CPU 执行程序, 程序返回前, **data** 段中的数据为多少?
- ② CPU 执行程序, 程序返回前, **cs**=____、**ss**=____、**ds**=_____。
- ③ 设程序加载后, **code** 段的段地址为 **X**, 则 **data** 段的段地址为____, **stack** 段的段地址为_____。

(4) 如果将(1)、(2)、(3)题中的最后一条伪指令“**end start**”改为“**end**”(也就是说, 不指明程序的入口), 则哪个程序仍然可以正确执行? 请说明原因。

(5) 程序如下, 编写 **code** 段中的代码, 将 **a** 段和 **b** 段中的数据依次相加, 将结果存到 **c** 段中。

```
assume cs:code
```

```
a segment
```

```
    db 1,2,3,4,5,6,7,8
```

```
a ends
```

```
b segment
```

```
    db 1,2,3,4,5,6,7,8
```

```
b ends
```

```
c segment
```

```
    db 0,0,0,0,0,0,0,0
```

```
c ends
```

```
code segment
```

```
start:
```

```
    ?
```

```
code ends
```

```
end start
```

(6) 程序如下, 编写 **code** 段中的代码, 用 **push** 指令将 **a** 段中的前 8 个字型数据, 逆序存储到 **b** 段中。

```
assume cs:code
```

```
a segment
```

```
    dw 1,2,3,4,5,6,7,8,9,0ah,0bh,0ch,0dh,0eh,0fh,0ffh
```

```
a ends
```

```
b segment
```

```
        dw 0,0,0,0,0,0,0,0
b ends

code segment
start:
    ?

code ends
end start
```

第 7 章 更灵活的定位内存地址的方法

前面，我们用[0]、[bx]的方法，在访问内存的指令中，定位内存单元的地址。本章我们主要通过具体的问题来讲解一些更灵活的定位内存地址的方法和相关的编程方法。我们的讲解将通过具体的问题来进行。

7.1 and 和 or 指令

首先，介绍两条指令 **and** 和 **or**，因为我们下面的例程中要用到它们。

(1) **and** 指令：逻辑与指令，按位进行与运算。

例如指令：

```
mov al,01100011B
and al,00111011B
```

执行后：al=00100011B

通过该指令可将操作对象的相应位设为 0，其他位不变。

例如：

将 al 的第 6 位设为 0 的指令是：and al,10111111B

将 al 的第 7 位设为 0 的指令是：and al,01111111B

将 al 的第 0 位设为 0 的指令是：and al,11111110B

(2) **or** 指令：逻辑或指令，按位进行或运算。

例如指令：

```
mov al,01100011B
or al,00111011B
```

执行后：al=01111011B

通过该指令可将操作对象的相应位设为 1，其他位不变。

例如：

将 al 的第 6 位设为 1 的指令是：or al,01000000B

将 al 的第 7 位设为 1 的指令是：or al,10000000B

将 al 的第 0 位设为 1 的指令是：or al,00000001B

7.2 关于 ASCII 码

我们可能已经学习过 ASCII 码的知识了，这里进行一次复习。

计算机中，所有的信息都是二进制，而人能理解的信息是已经具有约定意义的字符。比如说，人在有一定上下文的情况下看到“123”，就可知道这是一个数值，它的大小为123；看到“BASIC”就知道这是在说 BASIC 这种编程语言；看到“desk”，就知道说的是桌子。而我们要把这些信息存储在计算机中，就要对其进行编码，将其转化为二进制信息进行存储。而计算机要将这些存储的信息再显示给我们看，就要再对其进行解码。只要编码和解码采用同样的规则，我们就可以将人能理解的信息存入到计算机，再从计算机中取出。

世界上有很多编码方案，有一种方案叫做 ASCII 编码，是在计算机系统中通常被采用的。简单地说，所谓编码方案，就是一套规则，它约定了用什么样的信息来表示现实对象。比如说，在 ASCII 编码方案中，用 61H 表示“a”，62H 表示“b”。一种规则需要人们遵守才有意义。

一个文本编辑过程中，就包含着按照 ASCII 编码规则进行的编码和解码。在文本编辑过程中，我们按一下键盘的 a 键，就会在屏幕上看到“a”。这是怎样一个过程呢？我们按下键盘的 a 键，这个按键的信息被送入计算机，计算机用 ASCII 码的规则对其进行编码，将其转化为 61H 存储在内存的指定空间中；文本编辑软件从内存中取出 61H，将其送到显卡上的显存中；工作在文本模式下的显卡，用 ASCII 码的规则解释显存中的内容，61H 被当作字符“a”，显卡驱动显示器，将字符“a”的图像画在屏幕上。我们可以看到，显卡在处理文本信息的时候，是按照 ASCII 码的规则进行的。这也就是说，如果我们要想在显示器上看到“a”，就要给显卡提供“a”的 ASCII 码，61H。如何提供？当然是写入显存中。

7.3 以字符形式给出的数据

我们可以在汇编程序中，用'.....'的方式指明数据是以字符的形式给出的，编译器将它们转化为相对应的 ASCII 码。如下面的程序。

程序 7.1

```
assume cs:code,ds:data
data segment
    db 'unIX'
    db 'foRK'
data ends
code segment
```

```

start: mov al,'a'
      mov bl,'b'
      mov ax,4c00h
      int 21h
code ends
end start

```

上面的源程序中:

“db 'unIX'” 相当于 “db 75H,6EH,49H,58H”, “u”、“n”、“I”、“X”的 ASCII 码分别为 75H、6EH、49H、58H;

“db 'foRK'” 相当于 “db 66H, 6FH, 52H, 4BH”, “f”、“o”、“R”、“K”的 ASCII 码分别为 66H、6FH、52H、4BH;

“mov al,'a'” 相当于 “mov al,61H”, “a”的 ASCII 码为 61H;

“mov bl,'b'” 相当于 “mov al,62H”, “b”的 ASCII 码为 62H。

将程序 7.1 编译为可执行文件后,用 Debug 加载查看 data 段中的内容,如图 7.1 所示。

```

C:\>debug p7_1.exe
-r
AX=0000  BX=0000  CX=0019  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=0B2D  ES=0B2D  SS=0B3D  CS=0B3E  IP=0000  NU UP EI PL NZ NA PO NC
0B3E:0000  B061      MOV     AL,61
-d 0B3D:0
0B3D:0000  75 6E 49 58 66 6F 52 4B-00 00 00 00 00 00 00 00 unIXfoRK...

```

图 7.1 查看 data 段中的内容

图 7.1 中,先用 r 命令分析一下 data 段的地址,因“ds=0B2D”,所以程序从 0B3DH 段开始,data 段又是程序中的第一个段,它就在程序的起始处,所以它的段地址为 0B3DH。

用 d 命令查看 data 段,Debug 以十六进制数码和 ASCII 码字符的形式显示出其中的内容,从中,可以看出 data 段中的每个数据所对应的 ASCII 字符。

7.4 大小写转换的问题

下面考虑这样一个问题,在 codesg 中填写代码,将 datasg 中的第一个字符串转化为大写,第二个字符串转化为小写。

```

assume cs:codesg,ds:datasg

datasg segment
db 'BaSiC'
db 'iNfOrMaTiOn'
datasg ends

```

```
codesg segment
start:
codesg ends
end start
```

首先分析一下，我们知道同一个字母的大写字符和小写字符对应的 ASCII 码是不同的，比如“A”的 ASCII 码是 41H，“a”的 ASCII 码是 61H。要改变一个字母的大小写，实际上就是要改变它所对应的 ASCII 码。我们可以将所有的字母的大写字符和小写字符所对应的 ASCII 码列出来，进行一下对比，从中找到规律。

大写	十六进制	二进制	小写	十六进制	二进制
A	41	01000001	a	61	01100001
B	42	01000010	b	62	01100010
C	43	01000011	c	63	01100011
D	44	01000100	d	64	01100100
E	45	01000101	e	65	01100101
F	46	01000110	f	66	01100110
:					

通过对比，我们可以看出来，小写字母的 ASCII 码值比大写字母的 ASCII 码值大 20H。这样，我们可以想到，如果将“a”的 ASCII 码值减去 20H，就可以得到“A”；如果将“A”的 ASCII 码值加上 20H 就可以得到“a”。按照这样的方法，可以将 `datasg` 段中的第一个字符串“BaSiC”中的小写字母变成大写，第二个字符串“iNfOrMaTiOn”中的大写字母变成小写。

要注意的是，对于字符串“BaSiC”，应只对其中的小写字母所对应的 ASCII 码进行减 20H 的处理，将其转为大写，而对其中的大写字母不进行改变；对于字符串“iNfOrMaTiOn”，我们应只对其中的大写字母所对应的 ASCII 码进行加 20H 的处理，将其转为小写，而对于其中的小写字母不进行改变。这里面就存在着一个前提，程序必须要能够判断一个字母是大写还是小写。以“BaSiC”讨论，程序的流程将是这样的：

```
assume cs:codesg,ds:datasg

datasg segment
db 'BaSiC'
db 'iNfOrMaTiOn'
datasg ends

codesg segment
start:mov ax,datasg
      mov ds,ax
      mov bx,0
```

```
        mov cx,5
s:mov al,[bx]
        如果(al)>61H,则为小写字母的ASCII码,则: sub al,20H
        mov [bx],al
        inc bx
        loop s
        :
codesg ends
end start
```

判断将用到一些我们目前还没有学习到的指令。现在面临的问题是,用已学的指令来解决这个问题,则不能对字母的大小写进行任何判断。

但是,现实的问题却要求程序必须要能区别对待大写字母和小写字母。那么怎么办呢?

如果一个问题的解决方案,使我们陷入一种矛盾之中。那么,很可能是我们考虑问题的出发点有了问题,或是说,我们起初运用的规律并不合适。

我们前面所运用的规律是,小写字母的 ASCII 码值,比大写字母的 ASCII 码值大 20H。考虑问题的出发点是:大写字母+20H=小写字母,小写字母-20H=大写字母。这使我们最终落入了这样一个矛盾之中:必须判断是大写字母还是小写字母,才能决定进行何种处理,而我们现在又没有可以使用的用于判断的指令。

我们应该重新观察,寻找新的规律。可以看出,就 ASCII 码的二进制形式来看,除第 5 位(位数从 0 开始计算)外,大写字母和小写字母的其他各位都一样。大写字母 ASCII 码的第 5 位为 0,小写字母的第 5 位为 1。这样,我们就有了新的方法,一个字母,不管它原来是大写还是小写,将它的第 5 位置 0,它就必将变为大写字母;将它的第 5 位置 1,它就必将变为小写字母。在这个方法中,我们不需要在处理前判断字母的大小写。比如:对于“BaSiC”中的“B”,按要求,它已经是大写字母了,不应进行改变,将它的第 5 位设为 0,它还是大写字母,因为它的第 5 位本来就是 0。

用什么方法将一个数据中的某一位置 0 还是置 1?当然是用我们刚刚学过的 or 和 and 指令。

完整的程序如下。

```
assume cs:codesg,ds:datasg

datasg segment
db 'BaSiC'
db 'iNfOrMaTiOn'
datasg ends
```

```

codesg segment
start: mov ax,datasg
      mov ds,ax          ;设置ds 指向 datasg 段

      mov bx,0           ;设置(bx)=0,ds:bx 指向 'BaSiC' 的第一个字母

      mov cx,5           ;设置循环次数 5,因为 'BaSiC' 有 5 个字母
s:mov al,[bx]           ;将 ASCII 码从 ds:bx 所指向的单元中取出
   and al,11011111B     ;将 al 中的 ASCII 码的第 5 位置为 0, 变为大写字母
   mov [bx],al          ;将转变后的 ASCII 码写回原单元
   inc bx               ;(bx) 加 1, ds:bx 指向下一个字母
   loop s

      mov bx,5           ;设置(bx)=5,ds:bx 指向 'iNfOrMaTiOn' 的第一个字母

      mov cx,11          ;设置循环次数 11, 因为 'iNfOrMaTiOn' 有 11 个字母
s0:mov al,[bx]
   or al,00100000B      ;将 al 中的 ASCII 码的第 5 位置为 1, 变为小写字母
   mov [bx],al
   inc bx
   loop s0

      mov ax,4c00h
      int 21h

codesg ends
end start

```

7.5 [bx+idata]

在前面, 我们用[bx]的方式来指明一个内存单元, 还可以用一种更为灵活的方式来指明内存单元: [bx+idata]表示一个内存单元, 它的偏移地址为(bx)+idata(bx 中的数值加上 idata)。

我们看一下指令 `mov ax,[bx+200]` 的含义:

将一个内存单元的内容送入 ax, 这个内存单元的长度为 2 个字节(字单元), 存放一个字, 偏移地址为 bx 中的数值加上 200, 段地址在 ds 中。

数学化的描述为: $(ax) = ((ds) * 16 + (bx) + 200)$

该指令也可以写成如下格式(常用):

```
mov ax, [200+bx]
```

```
mov ax,200[bx]
mov ax,[bx].200
```

问题 7.1

用 Debug 查看内存,结果如下:

2000:1000 BE 00 06 00 00 00

写出下面的程序执行后,ax、bx、cx 中的内容。

```
mov ax,2000H
mov ds,ax
mov bx,1000H
mov ax,[bx]
mov cx,[bx+1]
add cx,[bx+2]
```

思考后看分析。

分析:

```
mov ax,[bx]
```

访问的字单元的段地址在 ds 中, (ds)=2000H; 偏移地址在 bx 中, (bx)=1000H; 指令执行后(ax)=00BEH。

```
mov cx,[bx+1]
```

访问的字单元的段地址在 ds 中, (ds)=2000H; 偏移地址=(bx)+1=1001H; 指令执行后(cx)=0600H。

```
add cx,[bx+2]
```

访问的字单元的段地址在 ds 中, (ds)=2000H; 偏移地址=(bx)+2=1002H; 指令执行后(cx)=0606H。

7.6 用[bx+idata]的方式进行数组的处理

有了[bx+idata]这种表示内存单元的方式,我们就可以用更高级的结构来看待所要处理的数据。我们通过下面的问题来理解这一点。

在 codesg 中填写代码,将 datasg 中定义的第一个字符串转化为大写,第二个字符串转化为小写。

```
assume cs:codesg,ds:datasg

datasg segment
    db 'BaSiC'
    db 'MinIX'
datasg ends

codesg segment
    start:
codesg ends

end start
```

按照我们原来的方法，用[**bx**]的方式定位字符串中的字符。代码段中的程序如下。

```
mov ax,datasg
mov ds,ax
mov bx,0

mov cx,5
s: mov al,[bx]
    and al,11011111b
    mov [bx],al
    inc bx
    loop s

mov bx,5
mov cx,5
s0: mov al,[bx]
    or al,00100000b
    mov [bx],al
    inc bx
    loop s0
```

现在，我们有了[**bx+idata**]的方式，就可以用更简化的方法来完成上面的程序。观察 **datasg** 段中的两个字符串，一个的起始地址为 0，另一个的起始地址为 5。我们可以将这两个字符串看作两个数组，一个从 0 地址开始存放，另一个从 5 开始存放。那么我们可以用[**0+bx**]和[**5+bx**]的方式在同一个循环中定位这两个字符串中的字符。在这里，0 和 5 给定了两个字符串的起始偏移地址，**bx** 中给出了从起始偏移地址开始的相对地址。这两个字符串在内存中的起始地址是不一样的，但是，它们中的每一个字符，从起始地址开始的相对地址的变化是相同的。改进的程序如下。

```
mov ax,datasg
mov ds,ax
```

```
        mov bx,0

        mov cx,5
s:      mov al,[bx]           ;定位第一个字符串中的字符
        and al,11011111b
        mov [bx],al
        mov al,[5+bx]       ;定位第二个字符串中的字符
        or al,00100000b
        mov [5+bx],al
        inc bx
        loop s
```

程序也可以写成下面的样子:

```
        mov ax,datasg
        mov ds,ax
        mov bx,0

        mov cx,5
s:      mov al,0[bx]
        and al,11011111b
        mov 0[bx],al
        mov al,5[bx]
        or al,00100000b
        mov 5[bx],al
        inc bx
        loop s
```

如果用高级语言,比如 C 语言来描述上面的程序,大致是这样的:

```
char a[5]="BaSiC";
char b[5]="MinIX";

main()
{
    int i;
    i=0;
    do
    {
        a[i]=a[i]&0xDF;
        b[i]=b[i]|0x20;
        i++;
    }
    while(i<5);
}
```

如果你熟悉 C 语言的话，可以比较一下这个 C 程序和上面的汇编程序的相似之处。尤其注意它们定位字符串中字符的方式。

C 语言: `a[i], b[i]`

汇编语言: `0[bx], 5[bx]`

通过比较，我们可以发现，`[bx+idata]`的方式为高级语言实现数组提供了便利机制。

7.7 SI 和 DI

`si` 和 `di` 是 8086CPU 中和 `bx` 功能相近的寄存器，`si` 和 `di` 不能够分成两个 8 位寄存器来使用。下面的 3 组指令实现了相同的功能。

```
(1)  mov bx,0  
     mov ax,[bx]
```

```
(2)  mov si,0  
     mov ax,[si]
```

```
(3)  mov di,0  
     mov ax,[di]
```

下面的 3 组指令也实现了相同的功能。

```
(1)  mov bx,0  
     mov ax,[bx+123]
```

```
(2)  mov si,0  
     mov ax,[si+123]
```

```
(3)  mov di,0  
     mov ax,[di+123]
```

问题 7.2

用 `si` 和 `di` 实现将字符串 `'welcome to masm'` 复制到它后面的数据区中。

```
assume  cs:codesg,ds:datasg  
  
datasg segment  
    db 'welcome to masm!'  
    db '.....'  
datasg ends
```

思考后看分析。

分析：

我们编写的程序大都是进行数据的处理，而数据在内存中存放，所以我们在处理数据之前首先要搞清楚数据存储在什么地方，也就是说数据的内存地址。现在我们要对 `datasg` 段中的数据进行复制，先来看一下要复制的数据在什么地方，`datasg:0`，这是要进行复制的数据的地址。那么复制到哪里去呢？它后面的数据区。“welcome to masml!”从偏移地址 0 开始存放，长度为 16 个字节，所以，它后面的数据区的偏移地址为 16，就是字符串“.....”存放的空间。清楚了地址之后，我们就可以进行处理了。我们用 `ds:si` 指向要复制的源始字符串，用 `ds:di` 指向复制的目的空间，然后用一个循环来完成复制。代码段如下。

```
codesg segment

start: mov ax,datasg
      mov ds,ax
      mov si,0
      mov di,16

      mov cx,8

s:    mov ax,[si]
      mov [di],ax
      add si,2
      add di,2
      loop s

      mov ax,4c00h
      int 21h

codesg ends
end start
```

注意，在程序中，用 16 位寄存器进行内存单元之间的数据传送，一次复制 2 个字节，一共循环 8 次。

问题 7.3

用更少的代码，实现问题 7.2 中的程序。

思考后看分析。

分析：

我们可以利用 `[bx(si 或 di)+idata]` 的方式，来使程序变得简洁。程序如下。

```
codesg segment

start: mov ax,datasg
      mov ds,ax
      mov si,0
      mov cx,8
s:    mov ax,0[si]
      mov 16[si],ax
      add si,2
      loop s

      mov ax,4c00h
      int 21h

codesg ends
end start
```

7.8 [bx+si]和[bx+di]

在前面，我们用[bx(si 或 di)]和[bx(si 或 di)+idata]的方式来指明一个内存单元，我们还可以用更为灵活的方式：[bx+si]和[bx+di]。

[bx+si]和[bx+di]的含义相似，我们以[bx+si]为例进行讲解。

[bx+si]表示一个内存单元，它的偏移地址为(bx)+(si)(即 bx 中的数值加上 si 中的数值)。

指令 `mov ax,[bx+si]` 的含义如下：

将一个内存单元的内容送入 ax，这个内存单元的长度为 2 字节(字单元)，存放一个字，偏移地址为 bx 中的数值加上 si 中的数值，段地址在 ds 中。

数学化的描述为： $(ax) = ((ds) * 16 + (bx) + (si))$

该指令也可以写成如下格式(常用)：

```
mov ax, [bx][si]
```

问题 7.4

用 Debug 查看内存，结果如下：

```
2000:1000 BE 00 06 00 00 00 .....
```

写出下面的程序执行后，ax、bx、cx 中的内容。

```
mov ax,2000H
mov ds,ax
mov bx,1000H
mov si,0
mov ax,[bx+si]
inc si
mov cx,[bx+si]
inc si
mov di,si
add cx,[bx+di]
```

思考后看分析。

分析：

```
mov ax,[bx+si]
```

访问的字单元的段地址在 `ds` 中, $(ds)=2000H$; 偏移地址 $=(bx)+(si)=1000H$; 指令执行后 $(ax)=00BEH$ 。

```
mov cx,[bx+si]
```

访问的字单元的段地址在 `ds` 中, $(ds)=2000H$; 偏移地址 $=(bx)+(si)=1001H$; 指令执行后 $(cx)=0600H$ 。

```
add cx,[bx+di]
```

访问的字单元的段地址在 `ds` 中, $(ds)=2000H$; 偏移地址 $=(bx)+(di)=1002H$; 指令执行后 $(cx)=0606H$ 。

7.9 `[bx+si+idata]`和`[bx+di+idata]`

`[bx+si+idata]`和`[bx+di+idata]`的含义相似,我们以`[bx+si+idata]`为例进行讲解。

`[bx+si+idata]`表示一个内存单元,它的偏移地址为 $(bx)+(si)+idata$ (即 `bx` 中的数值加上 `si` 中的数值再加上 `idata`)。

指令 `mov ax,[bx+si+idata]`的含义如下:

将一个内存单元的内容送入 `ax`, 这个内存单元的长度为 2 字节(字单元), 存放一个字, 偏移地址为 `bx` 中的数值加上 `si` 中的数值再加上 `idata`, 段地址在 `ds` 中。

数学化的描述为: $(ax) = ((ds) * 16 + (bx) + (si) + idata)$

该指令也可以写成如下格式(常用):

```
mov ax,[bx+200+si]
mov ax,[200+bx+si]
mov ax,200[bx][si]
mov ax,[bx].200[si]
mov ax,[bx][si].200
```

问题 7.5

用 Debug 查看内存，结果如下：

2000:1000 BE 00 06 00 6A 22

写出下面的程序执行后，ax、bx、cx 中的内容。

```
mov ax,2000H
mov ds,ax
mov bx,1000H
mov si,0
mov ax,[bx+2+si]
inc si
mov cx,[bx+2+si]
inc si
mov di,si
mov bx,[bx+2+di]
```

思考后看分析。

分析：

```
mov ax,[bx+2+si]
```

访问的字单元的段地址在 ds 中，(ds)=2000H；偏移地址=(bx)+(si)+2=1002H；指令执行后(ax)=0006H。

```
mov cx,[bx+2+si]
```

访问的字单元的段地址在 ds 中，(ds)=2000H；偏移地址=(bx)+(si)+2=1003H；指令执行后(cx)=6A00H。

```
mov bx,[bx+2+di]
```

访问的字单元的段地址在 ds 中，(ds)=2000H；偏移地址=(bx)+(di)+2=1004H；指令执行后(bx)=226AH。

7.10 不同的寻址方式的灵活应用

如果我们比较一下前面用到的几种定位内存地址的方法(可称为寻址方式),就可以发现:

- (1) `[idata]`用一个常量来表示地址,可用于直接定位一个内存单元;
- (2) `[bx]`用一个变量来表示内存地址,可用于间接定位一个内存单元;
- (3) `[bx+idata]`用一个变量和常量表示地址,可在一个起始地址的基础上用变量间接定位一个内存单元;
- (4) `[bx+si]`用两个变量表示地址;
- (5) `[bx+si+idata]`用两个变量和一个常量表示地址。

可以看到,从`[idata]`一直到`[bx+si+idata]`,我们可以用更加灵活的方式来定位一个内存单元的地址。这使我们可以从更加结构化的角度来看待所要处理的数据。下面我们通过一个问题的系列来体会 CPU 提供多种寻址方式的用意,并学习一些相关的编程技巧。

问题 7.6

编程,将 `datasg` 段中每个单词的头一个字母改为大写字母。

```
assume cs:codesg,ds:datasg

datasg segment
    db '1. file      '
    db '2. edit      '
    db '3. search    '
    db '4. view      '
    db '5. options   '
    db '6. help      '
datasg ends

codesg segment
    start:
codesg ends

end start
```

分析:

`datasg` 中的数据的存储结构,如图 7.2 所示。

我们可以看到,在 `datasg` 中定义了 6 个字符串,每个长度为 16 个字节(注意,为了直观,每个字符串的后面都加上了空格符,以使它们的长度刚好为 16 个字节)。因为它们

连续存放的, 可以将这 6 个字符串看成一个 6 行 16 列的二维数组。按照要求, 需要修改每一个单词的第一个字母, 即二维数组的每一行的第 4 列(相对于行首的偏移地址为 3)。

			C ↓													
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
R→ 00	1	.		f	i	l	e									
10	2	.		e	d	i	t									
20	3	.		s	e	a	r	c	h							
30	4	.		v	i	e	w									
40	5	.		o	p	t	i	o	n	s						
50	6	.		h	e	l	p									

图 7.2 datasg 中数据的存储结构

我们需要进行 6 次循环, 用一个变量 R 定位行, 用常量 3 定位列。处理的过程如下。

R=第一行的地址

mov cx, 6

s: 改变 R 行, 3 列的字母为大写

R=下一行的地址

loop s

我们用 bx 作变量, 定位每行的起始地址, 用 3 定位要修改的列, 用[bx+idata]的方式来对目标单元进行寻址, 程序如下。

mov ax, datasg

mov ds, ax

mov bx, 0

mov cx, 6

s: mov al, [bx+3]

and al, 11011111b

mov [bx+3], al

add bx, 16

loop s


```
R=第一行的地址;

mov cx,4

s0: C=第一列的地址

mov cx,3

s:  改变 R 行, C 列的字母为大写

C=下一列的地址;

loop s

R=下一行的地址

loop s0
```

我们用 `bx` 来作变量, 定位每行的起始地址, 用 `si` 定位要修改的列, 用 `[bx+si]` 的方式来对目标单元进行寻址, 程序如下。

```
mov ax,datasg
mov ds,ax
mov bx,0

mov cx,4

s0: mov si,0
    mov cx,3

s:  mov al,[bx+si]
    and al,11011111b
    mov [bx+si],al
    inc si

    loop s

    add bx,16
    loop s0
```

问题 7.8

仔细阅读上面的程序, 看看有什么问题?

思考后看分析。

分析:

问题在于 `cx` 的使用, 我们进行二重循环, 却只用了一个循环计数器, 造成在进行内层循环的时候, 覆盖了外层循环的循环计数值。多用一个计数器又不可能, 因为 `loop` 指

令默认 `cx` 为循环计数器。怎么办呢?

我们应该在每次开始内层循环的时候,将外层循环的 `cx` 中的数值保存起来,在执行外层循环的 `loop` 指令前,再恢复外层循环的 `cx` 数值。可以用寄存器 `dx` 来临时保存 `cx` 中的数值,改进的程序如下。

```

mov ax,datasg
mov ds,ax
mov bx,0

mov cx,4

s0: mov dx,cx          ;将外层循环的 cx 值保存在 dx 中
    mov si,0
    mov cx,3          ;cx 设置为内层循环的次数

s:   mov al,[bx+si]
    and al,11011111b
    mov [bx+si],al
    inc si
    loop s

    add bx,16
    mov cx,dx          ;用 dx 中存放的外层循环的计数值恢复 cx
    loop s0            ;外层循环的 loop 指令将 cx 中的计数值减 1

```

上面的程序用 `dx` 来暂时存放 `cx` 中的值,如果在内层循环中,`dx` 寄存器也被使用,该怎么办?我们似乎可以使用别的寄存器,但是 CPU 中的寄存器数量毕竟是有限的,如 8086CPU 只有 14 个寄存器。在上面的程序中,`si`、`cx`、`ax`、`bx`,显然不能用来暂存 `cx` 中的值,因为这些寄存器在循环中也要使用;`cs`、`ip`、`ds` 也不能用,因为 `cs:ip` 时刻指向当前指令,`ds` 指向 `datasg` 段;可用的就只有: `dx`、`di`、`es`、`ss`、`sp`、`bp` 等 6 个寄存器了。可是如果循环中的程序比较复杂,这些寄存器也都被使用的话,那么该如何?

我们在这里讨论的问题是,程序中经常需要进行数据的暂存,怎样做将更为合理。这些数据可能是寄存器中的,也可能是内存中的。我们可以用寄存器暂存它们,但是这不是一个一般化的解决方案,因为寄存器的数量有限,每个程序中可使用的寄存器都不一样。我们希望寻找一个通用的方案,来解决这种在编程中经常会出现的问题。

显然,我们不能选择寄存器,那么可以使用的就是内存了。可以考虑将需要暂存的数据放到内存单元中,需要使用的时候,再从内存单元中恢复。这样我们就需要开辟一段内存空间。再次改进的程序如下。

```

assume cs:codesg,ds:datasg

datasg segment

```

```

db 'ibm'          '
db 'dec'          '
db 'dos'          '
db 'vax'          '

dw 0              ;定义一个字，用来暂存 cx
datasg ends

codesg segment

    start:mov ax,datasg
           mov ds,ax
           mov bx,0
           mov cx,4

    s0:  mov ds:[40H],cx      ;将外层循环的 cx 值保存在 datasg:40H 单元中
           mov si,0
           mov cx,3          ;cx 设置为内层循环的次数

    s:   mov al,[bx+si]
           and al,11011111b
           mov [bx+si],al
           inc si
           loop s

           add bx,16
           mov cx,ds:[40H]    ;用 datasg:40H 单元中的值恢复 cx
           loop s0           ;外层循环的 loop 指令将 cx 中的计数值减 1

           mov ax,4c00H
           int 21H

codesg ends
end start

```

上面的程序中，用内存单元来保存数据，可是上面的做法却有些麻烦，因为如果需要保存多个数据的时候，你必须要注意数据放到了哪个单元中，这样程序容易混乱。

我们使用内存来暂存数据，这一点是确定了的，但是值得推敲的是，我们用怎样的结构来保存这些数据，而使得我们的程序更加清晰。一般来说，在需要暂存数据的时候，我们都应该使用栈。回忆一下，栈空间在内存中，采用相关的指令，如 `push`、`pop` 等，可对其进行特殊的操作。下面，再次改进我们的程序。

```

assume cs:codesg,ds:datasg,ss:stacksg

datasg segment

```

```

    db 'ibm'          '
    db 'dec'          '
    db 'dos'          '
    db 'vax'          '
datasg ends

stacksg segment      ;定义一个段，用来做栈段，容量为16个字节
    dw 0,0,0,0,0,0,0,0
stacksg ends

codesg segment

    start:mov ax,stacksg
           mov ss,ax
           mov sp,16
           mov ax,datasg
           mov ds,ax
           mov bx,0

           mov cx,4

s0:  push cx          ;将外层循环的 cx 值压栈
           mov si,0
           mov cx,3    ;cx 设置为内层循环的次数

s:   mov al,[bx+si]
           and al,11011111b
           mov [bx+si],al
           inc si
           loop s

           add bx,16

           pop cx      ;从栈顶弹出原 cx 的值，恢复 cx
           loop s0     ;外层循环的 loop 指令将 cx 中的计数值减 1

           mov ax,4c00H
           int 21H

codesg ends
end start

```

问题 7.9

编程，将 `datasg` 段中每个单词的前 4 个字母改为大写字母。

```

assume cs:codesg,ss:stacksg,ds:datasg

stacksg segment
    dw 0,0,0,0,0,0,0,0
stacksg ends

datasg segment
    db '1. display      '
    db '2. brows       '
    db '3. replace      '
    db '4. modify       '
datasg ends

codesg segment
    start:
codesg ends

end start

```

分析:

datasg 中的数据 的存储结构, 如图 7.4 所示。

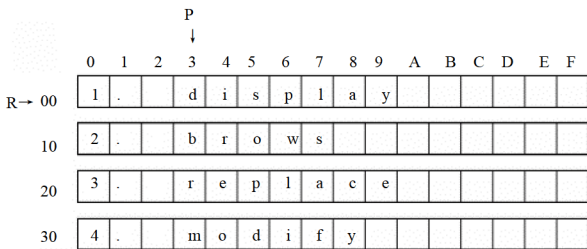


图 7.4 datasg 中数据的存储结构

在 datasg 中定义了 4 个字符串, 每个长度为 16 字节(注意, 为了使我们能在 Debug 中可以直观地查看, 每个字符串的后面都加上了空格符, 以使它们的长度刚好为 16 个字节)。因为它们是连续存放的, 我们可以将这 4 个字符串看成一个 4 行 16 列的二维数组, 按照要求, 我们需要修改每个单词的前 4 个字母, 即二维数组的每一行的 3~6 列。

我们需要进行 4×4 次的二重循环, 用变量 R 定位行, 常量 3 定位每行要修改的起始列, 变量 C 定位相对于起始列的要修改的列。外层循环按行来进行, 内层按列来进行。我们首先用 R 定位第 1 行, 循环修改 R 行的 $3+C(0 \leq C \leq 3)$ 列; 然后再用 R 定位到下一行, 再次循环修改 R 行的 $3+C(0 \leq C \leq 3)$ 列……, 如此重复直到所有的数据修改完毕。处

理的过程大致如下。

```
R=第一行的地址;

mov cx, 4

s0: C=第一个要修改的列相对于起始列的地址

mov cx, 4

s:  改变 R 行, 3+C 列的字母为大写

C=下一个要修改的列相对于起始列的地址

loop s

R=下一行的地址

loop s0
```

我们用 `bx` 来作变量, 定位每行的起始地址, 用 `si` 定位要修改的列, 用 `[bx+3+si]` 的方式来对目标单元进行寻址。

请在实验中自己完成这个程序。

这一章中, 我们主要讲解了更灵活的寻址方式的应用和一些编程方法, 主要内容有:

- 寻址方式 `[bx(或 si、di)+idata]`、`[bx+si(或 di)]`、`[bx+si(或 di)+idata]` 的意义和应用;
- 二重循环问题的处理;
- 栈的应用;
- 大小写转化的方法;
- `and`、`or` 指令。

下一章中, 我们将对寻址方式的问题进行更深入的探讨。之所以如此重视这个问题, 是因为寻址方式的适当应用, 使我们可以以更合理的结构来看待所要处理的数据。而为所要处理的看似杂乱的数据设计一种清晰的数据结构是程序设计的一个关键的问题。

实验 6 实践课程中的程序

(1) 将课程中所有讲解过的程序上机调试, 用 `Debug` 跟踪其执行过程, 并在过程中进一步理解所讲内容。

(2) 编程, 完成问题 7.9 中的程序。

第 8 章 数据处理的两个基本问题

本章对前面的所有内容是具有总结性的。我们知道，计算机是进行数据处理、运算的机器，那么有两个基本的问题就包含在其中：

- (1) 处理的数据在什么地方？
- (2) 要处理的数据有多长？

这两个问题，在机器指令中必须给以明确或隐含的说明，否则计算机就无法工作。本章中，我们就要针对 8086CPU 对这两个基本问题进行讨论。虽然讨论是在 8086CPU 的基础上进行的，但是这两个基本问题却是普遍的，对任何一个处理机都存在。

我们定义的描述性符号：**reg** 和 **sreg**。

为了描述上的简洁，在以后的课程中，我们将使用描述性的符号 **reg** 来表示一个寄存器，用 **sreg** 表示一个段寄存器。

reg 的集合包括：ax、bx、cx、dx、ah、al、bh、bl、ch、cl、dh、dl、sp、bp、si、di；

sreg 的集合包括：ds、ss、cs、es。

8.1 bx、si、di 和 bp

前 3 个寄存器我们已经用过了，现在进行一下总结。

(1) 在 8086CPU 中，只有这 4 个寄存器可以用在 “[...]” 中来内存单元的寻址。比如下面的指令都是正确的：

```
mov ax, [bx]
mov ax, [bx+si]
mov ax, [bx+di]
mov ax, [bp]
mov ax, [bp+si]
mov ax, [bp+di]
```

而下面的指令是错误的：

```
mov ax, [cx]
mov ax, [ax]
mov ax, [dx]
mov ax, [ds]
```

(2) 在 “[...]” 中，这 4 个寄存器可以单个出现，或只能以 4 种组合出现：bx 和 si、bx 和

di、bp 和 si、bp 和 di。比如下面的指令是正确的：

```
mov ax, [bx]
mov ax, [si]
mov ax, [di]
mov ax, [bp]
mov ax, [bx+si]
mov ax, [bx+di]
mov ax, [bp+si]
mov ax, [bp+di]
mov ax, [bx+si+idata]
mov ax, [bx+di+idata]
mov ax, [bp+si+idata]
mov ax, [bp+di+idata]
```

下面的指令是错误的：

```
mov ax, [bx+bp]
mov ax, [si+di]
```

(3) 只要在[...]中使用寄存器 bp，而指令中没有显性地给出段地址，段地址就默认在 ss 中。比如下面的指令。

mov ax, [bp]	含义: $(ax) = ((ss) * 16 + (bp))$
mov ax, [bp+idata]	含义: $(ax) = ((ss) * 16 + (bp) + idata)$
mov ax, [bp+si]	含义: $(ax) = ((ss) * 16 + (bp) + (si))$
mov ax, [bp+si+idata]	含义: $(ax) = ((ss) * 16 + (bp) + (si) + idata)$

8.2 机器指令处理的数据在什么地方

绝大部分机器指令都是进行数据处理的指令，处理大致可分为 3 类：读取、写入、运算。在机器指令这一层来讲，并不关心数据的值是多少，而关心指令执行前一刻，它将要处理的数据所在的位置。指令在执行前，所要处理的数据可以在 3 个地方：CPU 内部、内存、端口(端口将在后面的课程中进行讨论)，比如表 8.1 中所列的指令。

表 8.1 指令举例

机器码	汇编指令	指令执行前数据的位置
8E1E0000	mov bx,[0]	内存, ds:0 单元
89C3	mov bx,ax	CPU 内部, ax 寄存器
BB0100	mov bx,1	CPU 内部, 指令缓冲器

8.3 汇编语言中数据位置的表达

在汇编语言中如何表达数据的位置？汇编语言中用 3 个概念来表达数据的位置。

(1) 立即数(idata)

对于直接包含在机器指令中的数据(执行前在 CPU 的指令缓冲器中), 在汇编语言中称为: 立即数(idata), 在汇编指令中直接给出。

```
例: mov ax, 1
      add bx, 2000h
      or  bx, 00010000b
      mov al, 'a'
```

(2) 寄存器

指令要处理的数据在寄存器中, 在汇编指令中给出相应的寄存器名。

```
例: mov ax, bx
      mov ds, ax
      push bx
      mov ds:[0], bx
      push ds
      mov ss, ax
      mov sp, ax
```

(3) 段地址(SA)和偏移地址(EA)

指令要处理的数据在内存中, 在汇编指令中可用[X]的格式给出 EA, SA 在某个段寄存器中。

存放段地址的寄存器可以是默认的, 比如:

```
mov ax, [0]
mov ax, [di]
mov ax, [bx+8]
mov ax, [bx+si]
mov ax, [bx+si+8]
```

等指令, 段地址默认在 ds 中:

```
mov ax, [bp]
mov ax, [bp+8]
mov ax, [bp+si]
mov ax, [bp+si+8]
```

等指令, 段地址默认在 ss 中。

存放段地址的寄存器也可以是显性给出的, 比如以下的指令。

```
mov ax, ds:[bp]      含义: (ax) = ((ds) * 16 + (bp))
mov ax, es:[bx]      含义: (ax) = ((es) * 16 + (bx))
```

mov ax,ss:[bx+si] 含义: $(ax) = ((ss) * 16 + (bx) + (si))$
 mov ax,cs:[bx+si+8] 含义: $(ax) = ((cs) * 16 + (bx) + (si) + 8)$

8.4 寻址方式

当数据存放在内存中的时候,我们可以用多种方式来给定这个内存单元的偏移地址,这种定位内存单元的方法一般被称为寻址方式。

8086CPU 有多种寻址方式,我们在前面的课程中都已经用到了,这里进行一下总结,如表 8.2 所列。

表 8.2 寻址方式小结

寻址方式	含 义	名 称	常用格式举例
[idata]	EA=idata;SA=(ds)	直接寻址	[idata]
[bx]	EA=(bx);SA=(ds)	寄存器间接寻址	[bx]
[si]	EA=(si);SA=(ds)		
[di]	EA=(di);SA=(ds)		
[bp]	EA=(bp);SA=(ss)		
[bx+idata]	EA=(bx)+idata;SA=(ds)	寄存器相对寻址	用于结构体: [bx].idata 用于数组: idata[si],idata[di] 用于二维数组: [bx][idata]
[si+idata]	EA=(si)+idata;SA=(ds)		
[di+idata]	EA=(di)+idata;SA=(ds)		
[bp+idata]	EA=(bp)+idata;SA=(ss)		
[bx+si]	EA=(bx)+(si);SA=(ds)	基址变址寻址	用于二维数组: [bx][si]
[bx+di]	EA=(bx)+(di);SA=(ds)		
[bp+si]	EA=(bp)+(si);SA=(ss)		
[bp+di]	EA=(bp)+(di);SA=(ss)		
[bx+si+idata]	EA=(bx)+(si)+idata; SA=(ds)	相对基址变址寻址	用于表格(结构)中的数组项: [bx].idata[si] 用于二维数组: idata[bx][si]
[bx+di+idata]	EA=(bx)+(di)+idata; SA=(ds)		
[bp+si+idata]	EA=(bp)+(si)+idata; SA=(ss)		
[bp+di+idata]	EA=(bp)+(di)+idata; SA=(ss)		

8.5 指令要处理的数据有多长

8086CPU 的指令，可以处理两种尺寸的数据，byte 和 word。所以在机器指令中要指明，指令进行的是字操作还是字节操作。对于这个问题，汇编语言中用以下方法处理。

(1) 通过寄存器名指明要处理的数据的尺寸。

例如，下面的指令中，寄存器指明了指令进行的是字操作。

```
mov ax,1
mov bx,ds:[0]
mov ds,ax
mov ds:[0],ax
inc ax
add ax,1000
```

下面的指令中，寄存器指明了指令进行的是字节操作。

```
mov al,1
mov al,b1
mov al,ds:[0]
mov ds:[0],al
inc al
add al,100
```

(2) 在没有寄存器名存在的条件下，用操作符 X ptr 指明内存单元的长度，X 在汇编指令中可以为 word 或 byte。

例如，下面的指令中，用 word ptr 指明了指令访问的内存单元是一个字单元。

```
mov word ptr ds:[0],1
inc word ptr [bx]
inc word ptr ds:[0]
add word ptr [bx],2
```

下面的指令中，用 byte ptr 指明了指令访问的内存单元是一个字节单元。

```
mov byte ptr ds:[0],1
inc byte ptr [bx]
inc byte ptr ds:[0]
add byte ptr [bx],2
```

在没有寄存器参与的内存单元访问指令中，用 word ptr 或 byte ptr 显性地指明所要访问的内存单元的长度是很必要的。否则，CPU 无法得知所要访问的单元是字单元，还是字节单元。假设我们用 Debug 查看内存的结果如下：

```
2000: 1000 FF FF FF FF FF FF .....
```

那么指令：

```
mov ax,2000H
mov ds,ax
mov byte ptr [1000H],1
```

将使内存中的内容变为:

2000:1000 01 FF FF FF FF FF

而指令:

```
mov ax,2000H
mov ds,ax
mov word ptr [1000H],1
```

将使内存中的内容变为:

2000:1000 01 00 FF FF FF FF

这是因为 `mov byte ptr [1000H],1` 访问的是地址为 `ds:1000H` 的字节单元,修改的是 `ds:1000H` 单元的内容;而 `mov word ptr [1000H],1` 访问的是地址为 `ds:1000H` 的字单元,修改的是 `ds:1000H` 和 `ds:1001H` 两个单元的内容。

(3) 其他方法

有些指令默认了访问的是字单元还是字节单元,比如, `push [1000H]` 就不用指明访问的是字单元还是字节单元,因为 `push` 指令只进行字操作。

8.6 寻址方式的综合应用

下面我们通过一个问题来进一步讨论一下各种寻址方式的作用。

关于 DEC 公司的一条记录(1982 年)如下。

公司名称: DEC
总裁姓名: Ken Olsen
排 名: 137
收 入: 40(40 亿美元)
著名产品: PDP(小型机)

这些数据在内存中以图 8.1 所示的方式存放。

可以看到,这些数据被存放在 `seg` 段中从偏移地址 `60H` 起始的位置,从 `seg:60` 起始以 ASCII 字符的形式存储了 3 个字节的公司名称;从 `seg:60+3` 起始以 ASCII 字符的形式存储了 9 个字节的总裁姓名;从 `seg:60+0C` 起始存放了一个字型数据,总裁在富翁榜上的排名;从 `seg:60+0E` 起始存放了一个字型数据,公司的收入;从 `seg:60+10` 起始以 ASCII 字符的形式存储了 3 个字节的产品名称。

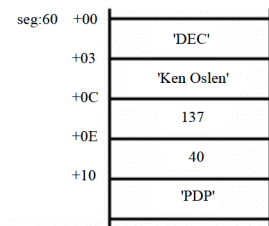


图 8.1 数据存放示意

以上是该公司 1982 年的情况，到了 1988 年 DEC 公司的信息有了如下变化。

- (1) Ken Olsen 在富翁榜上的排名已升至 38 位；
- (2) DEC 的收入增加了 70 亿美元；
- (3) 该公司的著名产品已变为 VAX 系列计算机。

我们提出的任务是，编程修改内存中的过时数据。

首先，我们应该分析一下要修改的数据。

要修改内容是：

- (1) (DEC 公司记录)的(排名字段)
- (2) (DEC 公司记录)的(收入字段)
- (3) (DEC 公司记录)的(产品字段)的(第一个字符)、(第二个字符)、(第三个字符)

从要修改的内容，我们就可以逐步地确定修改的方法。

- (1) 要访问的数据是 DEC 公司的记录，所以，首先要确定 DEC 公司记录的位置：

R=seg:60

确定了公司记录的位置后，下面就进一步确定要访问的内容在记录中的位置。

- (2) 确定排名字段在记录中的位置：0CH。
- (3) 修改 R+0CH 处的数据。
- (4) 确定收入字段在记录中的位置：0EH。
- (5) 修改 R+0EH 处的数据。
- (6) 确定产品字段在记录中的位置：10H。

要修改的产品字段是一个字符串(或一个数组)，需要访问字符串中的每一个字符。所以要进一步确定每一个字符在字符串中的位置。

- (7) 确定第一个字符在产品字段中的位置：P=0。
- (8) 修改 R+10H+P 处的数据；P=P+1。
- (9) 修改 R+10H+P 处的数据；P=P+1。

(10) 修改 R+10H+P 处的数据。

根据上面的分析, 程序如下。

```
mov ax, seg
mov ds, ax
mov bx, 60h ;确定记录地址, ds:bx

mov word ptr [bx+0ch], 38 ;排名字段改为 38
add word ptr [bx+0eh], 70 ;收入字段增加 70

mov si, 0 ;用 si 来定位产品字符串中的字符
mov byte ptr [bx+10h+si], 'V'
inc si
mov byte ptr [bx+10h+si], 'A'
inc si
mov byte ptr [bx+10h+si], 'X'
```

如果你熟悉 C 语言的话, 我们可以用 C 语言来描述这个程序, 大致应该是这样的:

```
struct company { /*定义一个公司记录的结构体*/
    char cn[3]; /*公司名称*/
    char hn[9]; /*总裁姓名*/
    int pm; /*排 名*/
    int sr; /*收 入*/
    char cp[3]; /*著名产品*/
};

struct company dec={"DEC", "Ken Olsen", 137, 40, "PDP"};
/*定义一个公司记录的变量, 内存中将存有一条公司的记录*/

main()
{
    int i;
    dec.pm=38;
    dec.sr=dec.sr+70;
    i=0;
    dec.cp[i]='V';
    i++;
    dec.cp[i]='A';
    i++;
    dec.cp[i]='X';
    return 0;
}
```

我们再按照 C 语言的风格, 用汇编语言写一下这个程序, 注意和 C 语言相关语句的比对:

```
mov ax, seg
mov ds, ax
mov bx, 60h ;记录首址送 BX
```

```

mov word ptr [bx].0ch,38      ;排名字段改为 38
                               ;C: dec.pm=38;
add word ptr [bx].0eh,70      ;收入字段增加 70
                               ;C: dec.sr=dec.sr+70;
                               ;产品字段改为字符串'VAX'
mov si,0                      ;C: i=0;
mov byte ptr [bx].10h[si],'V' ; dec.cp[i]='V';
inc si                        ; i++;
mov byte ptr [bx].10h[si],'A' ; dec.cp[i]='A';
inc si                        ; i++;
mov byte ptr [bx].10h[si],'X' ; dec.cp[i]='X';

```

我们可以看到, 8086CPU 提供的如[**bx+si+idata**]的寻址方式为结构化数据的处理提供了方便。使得我们可以在编程的时候, 从结构化的角度去看待所要处理的数据。从上面可以看到, 一个结构化的数据包含了多个数据项, 而数据项的类型又不相同, 有的是字型数据, 有的是字节型数据, 有的是数组(字符串)。一般来说, 我们可以用[**bx+idata+si**]的方式来访问结构体中的数据。用 **bx** 定位整个结构体, 用 **idata** 定位结构体中的某一个数据项, 用 **si** 定位数组项中的每个元素。为此, 汇编语言提供了更为贴切的书写方式, 如: [bx].idata、[bx].idata[si]。

在 C 语言程序中我们看到, 如: dec.cp[i], dec 是一个变量名, 指明了结构体变量的地址, cp 是一个名称, 指明了数据项 cp 的地址, 而 i 用来定位 cp 中的每一个字符。汇编语言中的做法是: bx.10h[si]。看一下, 是不是很相似?

8.7 div 指令

div 是除法指令, 使用 div 做除法的时候应注意以下问题。

- (1) 除数: 有 8 位和 16 位两种, 在一个 reg 或内存单元中。
- (2) 被除数: 默认放在 AX 或 DX 和 AX 中, 如果除数为 8 位, 被除数则为 16 位, 默认在 AX 中存放; 如果除数为 16 位, 被除数则为 32 位, 在 DX 和 AX 中存放, DX 存放高 16 位, AX 存放低 16 位。
- (3) 结果: 如果除数为 8 位, 则 AL 存储除法操作的商, AH 存储除法操作的余数; 如果除数为 16 位, 则 AX 存储除法操作的商, DX 存储除法操作的余数。

格式如下:

```

div reg
div 内存单元

```

现在, 我们可以用多种方法来表示一个内存单元了, 比如下面的例子:

```

div byte ptr ds:[0]
含义: (ax)=(ax)/((ds)*16+0) 的商
      (ah)=(ax)/((ds)*16+0) 的余数

```

```
div word ptr es:[0]
```

含义: $(ax) = [(dx) * 10000H + (ax)] / ((es) * 16 + 0)$ 的商
 $(dx) = [(dx) * 10000H + (ax)] / ((es) * 16 + 0)$ 的余数

```
div byte ptr [bx+si+8]
```

含义: $(al) = (ax) / ((ds) * 16 + (bx) + (si) + 8)$ 的商
 $(ah) = (ax) / ((ds) * 16 + (bx) + (si) + 8)$ 的余数

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

含义: $(ax) = [(dx) * 10000H + (ax)] / ((ds) * 16 + (bx) + (si) + 8)$ 的商
 $(dx) = [(dx) * 10000H + (ax)] / ((ds) * 16 + (bx) + (si) + 8)$ 的余数

编程, 利用除法指令计算 $100001/100$ 。

首先分析一下, 被除数 100001 大于 65535, 不能用 ax 寄存器存放, 所以只能用 dx 和 ax 两个寄存器联合存放 100001, 也就是说要进行 16 位的除法。除数 100 小于 255, 可以在一个 8 位寄存器中存放, 但是, 因为被除数是 32 位的, 除数应为 16 位, 所以要用一个 16 位寄存器来存放除数 100。

因为要分别为 dx 和 ax 赋 100001 的高 16 位值和低 16 位值, 所以应先将 100001 表示为 16 进制形式: 186A1H。程序如下:

```
mov dx,1
mov ax,86A1H      ; (dx)*10000H+(ax)=100001
mov bx,100
div bx
```

程序执行后, $(ax)=03E8H$ (即 1000), $(dx)=1$ (余数为 1)。读者可自行在 Debug 中实践。

编程, 利用除法指令计算 $1001/100$ 。

首先分析一下, 被除数 1001 可用 ax 寄存器存放, 除数 100 可用 8 位寄存器存放, 也就是说, 要进行 8 位的除法。程序如下。

```
mov ax,1001
mov bl,100
div bl
```

程序执行后, $(al)=0AH$ (即 10), $(ah)=1$ (余数为 1)。读者可自行在 Debug 中实践。

8.8 伪指令 dd

前面我们用 db 和 dw 定义字节型数据和字型数据。dd 是用来定义 dword(double word, 双字)型数据的。比如:

```
data segment
db 1
```

```
dw 1
dd 1
data ends
```

在 data 段中定义了 3 个数据：

第一个数据为 01H，在 data:0 处，占 1 个字节；

第二个数据为 0001H，在 data:1 处，占 1 个字；

第三个数据为 00000001H，在 data:3 处，占 2 个字。

问题 8.1

用 div 计算 data 段中第一个数据除以第二个数据后的结果，商存在第三个数据的存储单元中。

```
data segment
dd 100001
dw 100
dw 0
data ends
```

思考后看分析。

分析：

data 段中的第一个数据是被除数，为 dword(双字)型，32 位，所以在做除法之前，用 dx 和 ax 存储。应将 data:0 字单元中的低 16 位存储在 ax 中，data:2 字单元中的高 16 位存储在 dx 中。程序如下。

```
mov ax,data
mov ds,ax
mov ax,ds:[0]           ;ds:0 字单元中的低 16 位存储在 ax 中
mov dx,ds:[2]           ;ds:2 字单元中的高 16 位存储在 dx 中
div word ptr ds:[4]      ;用 dx:ax 中的 32 位数据除以 ds:4 字单元中的数据
mov ds:[6],ax           ;将商存储在 ds:6 字单元中
```

8.9 dup

dup 是一个操作符，在汇编语言中同 db、dw、dd 等一样，也是由编译器识别处理的符号。它是和 db、dw、dd 等数据定义伪指令配合使用的，用来进行数据的重复。比如：

```
db 3 dup (0)
```

定义了 3 个字节，它们的值都是 0，相当于 db 0,0,0。

```
db 3 dup (0,1,2)
```

定义了 9 个字节，它们是 0、1、2、0、1、2、0、1、2，相当于 db 0,1,2,0,1,2,0,1,2。


```

assume cs:codesg

data segment
    db '1975','1976','1977','1978','1979','1980','1981','1982','1983'
    db '1984','1985','1986','1987','1988','1989','1990','1991','1992'
    db '1993','1994','1995'
;以上是表示 21 年的 21 个字符串

    dd 16,22,382,1356,2390,8000,16000,24486,50065,97479,140417,197514
    dd 345980,590827,803530,1183000,1843000,2759000,3753000,4649000,5937000
;以上是表示 21 年公司总收入的 21 个 dword 型数据

    dw 3,7,9,13,28,38,130,220,476,778,1001,1442,2258,2793,4037,5635,8226
    dw 11542,14430,15257,17800
;以上是表示 21 年公司雇员人数的 21 个 word 型数据

data ends

table segment
    db 21 dup ('year summ ne ?? ')
table ends

```

编程，将 data 段中的数据按如下格式写入到 table 段中，并计算 21 年中的人均收入(取整)，结果也按照下面的格式保存在 table 段中。

	年份(4 字节)				空 格	收入(4 字节)				空 格	雇员数 (2 字节)		空 格	人均收入 (2 字节)		空 格
行内 地址 1 年 占 1 行, 每行的 起始地址	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
table:0	'1	9	7	5'		16					3			?		
table:10H	'1	9	7	6'		22					7			?		
table:20H	'1	9	7	7'		382					9			?		
table:30H	'1	9	7	8'		1356					13			?		
table:40H	'1	9	7	9'		2390					28			?		
table:50H	'1	9	8	0'		8000					38			?		
⋮																
table:140H	'1	9	9	5'		5937000					17800			?		

提示, 可将 **data** 段中的数据看成是多个数组, 而将 **table** 中的数据看成是一个结构型数据的数组, 每个结构型数据中包含多个数据项。可用 **bx** 定位每个结构型数据, 用 **idata** 定位数据项, 用 **si** 定位数组项中的每个元素, 对于 **table** 中的数据的访问可采用 **[bx].idata** 和 **[bx].idata[si]** 的寻址方式。

注意, 这个程序是到目前为止最复杂的程序, 它几乎用到了我们以前学过的所有知识和编程技巧。所以, 这个程序是对我们从前学习的最好的实践总结。请认真完成。

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 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	B80000	MOV	EB,0000
0BBD:0006	EB03	JMP	0008
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) $(IP) = (IP) + \text{所读取指令的长度}$, 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) $(IP) = (IP) + \text{所读取指令的长度} = (IP) + 2 = 0008H$, `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。

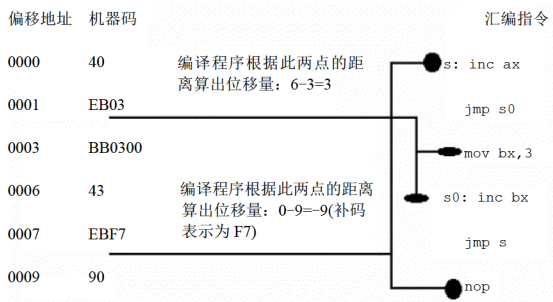


图 9.3 转移位移的计算方法

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, 请阅读附录 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	BB0000	MOV	AX,0000
	0BBD:0003	BB0000	MOV	EBX,0000
	0BBD:0006	EA0B01BD0B	JMP	0BBD:010B
	0BBD:000B	0000	ADD	[EBX*SI],AL
	0BBD:000D	0000	ADD	[EBX*SI],AL
	0BBD:000F	0000	ADD	[EBX*SI],AL
	0BBD:0011	0000	ADD	[EBX*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 个字节的內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 .. .. .. .. .. .. .. .. .. .. .. ..
```