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间都是安全的，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间，则必须要在源程序中做出说明。我们通过在源程序中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，我们希望可以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   NV UP EI PL NZ NA PO NC
0B3D:0000 2301          AND     AX,[BX+DI]                          DS:0000=20CD
-u
0B3D:0000 2301          AND     AX,[BX+DI]
0B3D:0002 56            PUSH    SI
0B3D:0003 0489          ADD     AL,89
0B3D:0005 07            POP     ES
0B3D:0006 BC0AEF        MOV     SP,EF0A
0B3D:0009 0DED0F        OR      AX,0FED
0B3D:000C BA0C87        MOV     DX,870C
0B3D:000F 09BB0000      OR      [BP+DI+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 BC 0A-EF 0D ED 0F BA 0C 87 09   #.V.............
0B3D:0010  BB 00 00 B8 00 00 B9 08-00 2E 03 07 83 C3 02 E2   ................
0B3D:0020  F8 B8 00 4C CD 21 EA 99-0A C0 74 EA F6 C7 80 74   ...L.!....t....t
0B3D:0030  05 C6 06 15 98 01 E9 2D-01 3A C3 75 05 80 CF 80   .......-.:.u....
0B3D:0040  EB D4 3C 0D 75 03 E9 18-01 3A 06 B6 96 75 03 E9   ..<.u....:...u..
0B3D:0050  17 01 B2 3A 38 14 75 1D-80 3E A4 98 01 75 03 E8   ...:8.u..>...u..
0B3D:0060  EB E0 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 BB0000        MOV     BX,0000
0B3D:0013 B80000        MOV     AX,0000
0B3D:0016 B90800        MOV     CX,0008
0B3D:0019 2E            CS:
0B3D:001A 0307          ADD     AX,[BX]
0B3D:001C 83C302        ADD     BX,+02
0B3D:001F E2F8          LOOP    0019
0B3D:0021 B8004C        MOV     AX,4C00
0B3D:0024 CD21          INT     21
0B3D:0026 EA990AC074    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问题，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          ;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

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，这里进行一下复习。

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，编译器将把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:\masm>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   NV 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 masm!”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)。因为它们是

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)。

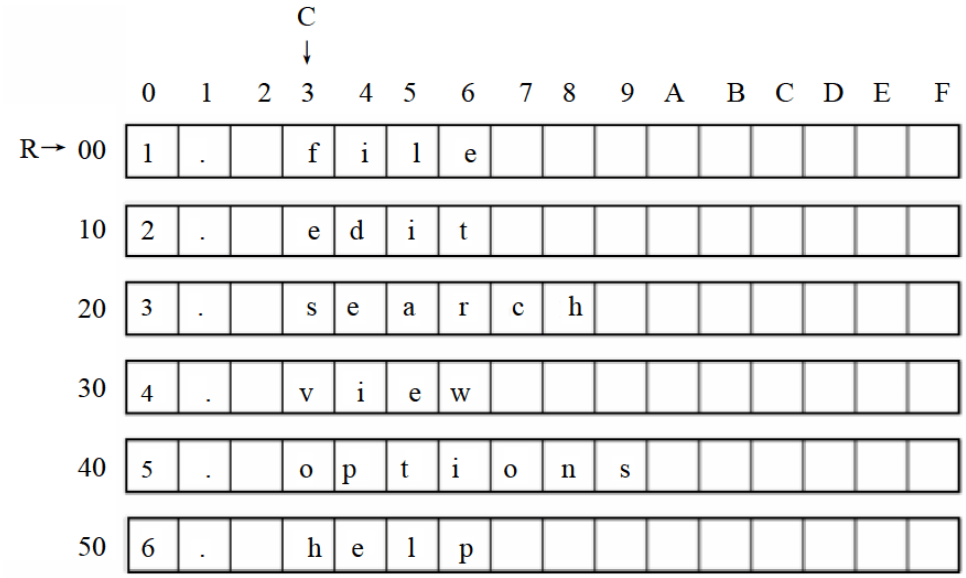

图 7.2　datasg 中数据的存储结构

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
```

问题 7.7

编程，将 datasg 段中每个单词改为大写字母。

```
assume  cs:codesg,ds:datasg

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

codesg segment
start:
codesg ends

end start
```

分析：

datasg 中的数据的存储结构如图 7.3 所示。

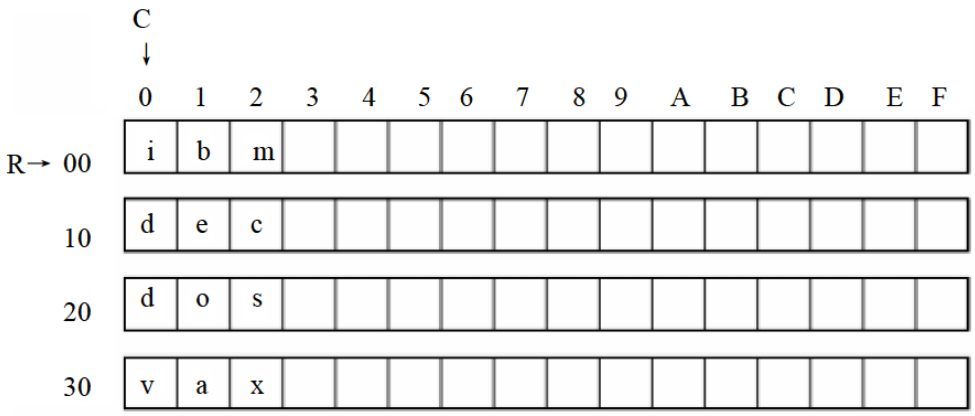

图 7.3　datasg 中数据的存储结构

在 datasg 中定义了 4 个字符串，每个长度为 16 个字节(注意，为了使我们在 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，我们需要修改每一个单词，即二维数组的每一行的前 3 列。

我们需要进行 4×3 次的二重循环，用变量 R 定位行，变量 C 定位列。外层循环按行来进行，内层按列来进行。首先用 R 定位第 1 行，然后循环修改 R 行的前 3 列；然后再用 R 定位到下一行，再次循环修改 R 行的前 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，你必须要记住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。

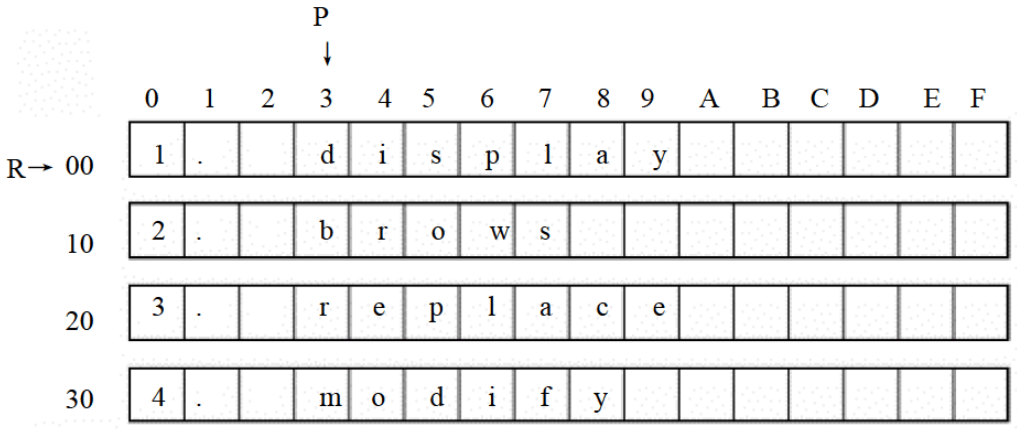

图 7.4　datasg 中数据的存储结构

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≤C≤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≤C≤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，现在我们进行一下总结。

(1)　在 8086CPU 中，只有这 4 个寄存器可以用在“[…]”中来进行内存单元的寻址。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,bl
mov al,ds:[0]
mov ds:[0],al
inc al
add al,100
```

(2) 在没有寄存器名存在的情况下，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。

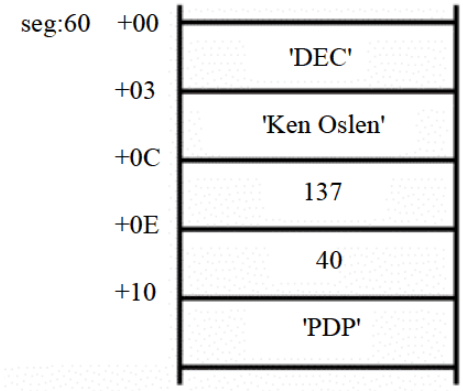

图 8.1　数据存放示意

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：(al)=(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。

```
db 3 dup ('abc','ABC')
```

定义了 18 个字节，它们是'abcABCabcABCabcABC'，相当于 db'abcABCabcABCabcABC'。

可见，dup 的使用格式如下。

```
db 重复的次数 dup (重复的字节型数据)
dw 重复的次数 dup (重复的字型数据)
dd 重复的次数 dup (重复的双字型数据)
```

dup 是一个十分有用的操作符，比如要定义一个容量为 200 个字节的栈段，如果不用 dup，则必须：

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

当然，你可以用 dd，使程序变得简短一些，但是如果要求定义一个容量为 1000 字节或 10000 字节的呢？如果没有 dup，定义部分的程序就变得太长了，有了 dup 就可以轻松解决。如下：

```
stack segment
  db 200 dup (0)
stack ends
```

实验 7　寻址方式在结构化数据访问中的应用

Power idea 公司从 1975 年成立一直到 1995 年的基本情况如下。

年份	收入(千美元)	雇员(人)	人均收入(千美元)
1975	16	3	?
1976	22	7	?
1977	382	9	?
1978	1356	13	?
1979	2390	28	?
1980	8000	38	?
⋮			
1995	5937000	17800	?

下面的程序中，已经定义好了这些数据：

```
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。