第7章
更灵活的定位内存地址的方法

第7章 更灵活的定位内存地址的方法

- 7.1 and和or指令
- 7.2 关于ASCII码
- 7.3 以字符形式给出的数据
- 7.4 大小写转换的问题
- 7.5 [bx+idata]
- 7.6 用[bx+idata]的方式进行数组的处理
- 7.7 SI和DI
- 7.8 [bx+si]和[bx+di]
- 7.9 [bx+si+idata]和[bx+di+idata]
- 7.10 不同的寻址方式的灵活应用

引言

- 访存指令中使用 [0]、[bx]方法定位内存单元的地址。
- 还有哪些更灵活的定位内存地址的方法？

7.1 and和or指令

- 按位逻辑运算指令： and和or。
- （1）and（逻辑与）指令：按位进行与运算。

  如　　mov al, 01100011B
  　　and al, 00111011B

  执行后：al = 00100011B

7.2 关于ASCII码

- ASCII编码
  - 字符的编码系统

如：61H 表示 “a”，
62H表示 “b”。

7.3 以字符形式给出的数据

- 在汇编程序中，可用 ‘....’ 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。

- 例如程序7.1

7.3 以字符形式给出的数据

- 程序7.1

```
assume 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
```

用‘XXX’的方式定义一组字符，编译器将把它们转化为相对应的ASCII码

7.3 以字符形式给出的数据

- 程序7.1

```
assume 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
```

即“db 75H,6EH,49H,58H”,

即“db 66H,6FH,52H,4BH”

即“mov al,61H”,因”a”的ASCII码为61H

即“mov al,62H”,因”b”的ASCII码为62H。

7.4 大小写转换的问题

- 补全以下代码，将“BaSiC”中小写字母转为大写字母，将“iNfOrMaTiOn”中大写字母转为小写字母

```
assume cs:codesg,ds:datasg

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

codesg segment
    start: ；请在此添加代码实现以上要求的大小写转换
codesg ends

end start
```

7.4 大小写转换的问题

- 字母的大写字符和小写字符所对应的ASCII：

大写	二进制	小写	二进制
A	01000001	a	01100001
B	01000010	b	01100010
C	01000011	c	01100011
…	…	…	…

- 规律：小写字母ASCII码比大写字母ASCII码大20H 。
  - 若将 “a” 的ASCII码值减去20H，可得 “A”；
  - 若将 “A”的ASCII码值加上20H，可得 “a”。

7.4 大小写转换的问题

- 按照这样的方法，可将 datasg段中：
  - 字符串“BaSiC”中的小写字母变成大写；
    - 小写字母的ASCII码减20H
  - 字符串 “iNfOrMaTiOn”中的大写字母变成小写。
    - 大写字母的ASCII码加20H

7.4 大小写转换的问题

- 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ASCII码，则：sub al,20H
       mov [bx],al
       inc bx
       loop s
        :
codesg ends
end start
```

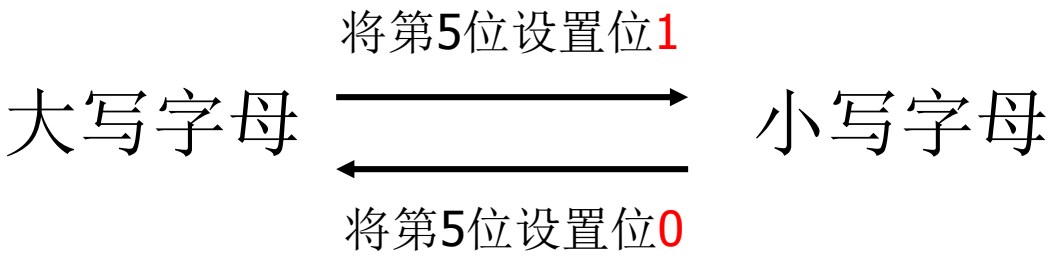
要判断字母的大小写，
但是判断指令没有学过。

7.4 大小写转换的问题

- 字母的大写字符和小写字符所对应的ASCII：

大写	二进制	小写	二进制
A	01000001	a	01100001
B	01000010	b	01100010
C	01000011	c	01100011
…	…	…	…

规律：



7.4 大小写转换的问题

- 如何将一个数据中的某一位 置0 或置1？

or和and指令。

- 完整的程序代码

7.4 大小写转换的问题

- 完整程序代码

```
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
```

每次从内存读入一个字符，进行大小写转换，再写回内存。

7.5 [bx+idata]

- 更为灵活的访存方式：

  [bx+idata]

  表示一个内存单元，其偏移地址为(bx)+idata。

- 如　　mov ax,[bx+200]
  - 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；偏移地址为bx中的数值加上200，段地址在ds中。
  - 数学化描述：(ax)=((ds)*16+(bx)+200)

7.5 [bx+idata]

- 指令mov ax,[bx+200]也可以写成如下格式（常用）：
  - mov ax,[200+bx]
  - mov ax,200[bx]
  - mov ax,[bx].200

7.5 [bx+idata]

- **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。

7.5 [bx+idata]

- **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。

7.5 [bx+idata]

- 问题7.1分析

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]
```

段地址在ds中，(ds)=2000H；
偏移地址在bx 中，(bx)=1000H；
指令执行后(ax)=00BEH。

7.5 [bx+idata]

- 问题7.1分析

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]
```

段地址在ds中，(ds)=2000H；
偏移地址=(bx)+1=1001H；
指令执行后(cx)=0600H。

7.5 [bx+idata]

- 问题7.1分析

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]
```

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

- [bx+idata]这种访存方式便于使用更高级的结构来看待所要处理的数据。

7.6 用[bx+idata]的方式进行数组的处理

- 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: ;请在此添加代码……
codesg ends
end start
```

7.6 用[bx+idata]的方式进行数组的处理

- 若用[bx]的方式定位字符串中的字符。

  代码段中的程序代码：

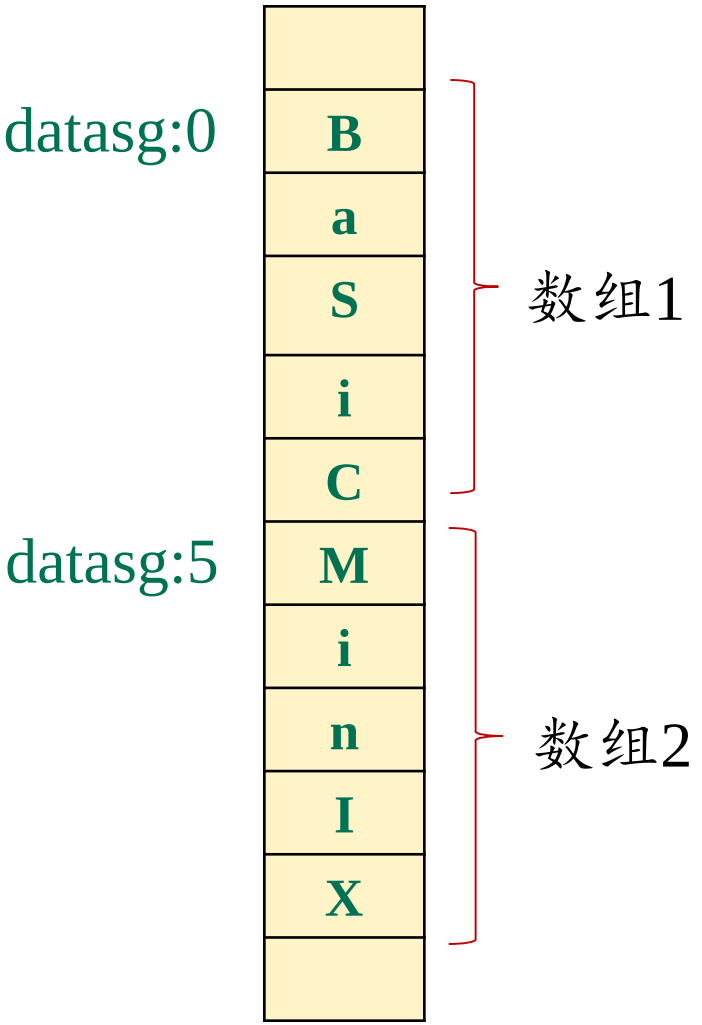
```
    mov ax,datasg
    mov ds,ax

    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
```

7.6 用[bx+idata]的方式进行数组的处理

- 采用[bx+idata]的方式可更简化地完成上面的程序。



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。

可用[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。

7.6 用[bx+idata]的方式进行数组的处理

- 改进的程序（以下2个版本等效）：

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

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

```
    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
```

7.6 用[bx+idata]的方式进行数组的处理

汇编语言实现

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

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

C语言实现

```
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);
}
```

比较C程序和上面的汇编程序的相似之处，可知[bx+idata]访存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**不可以**分成两个8 位寄存器来使用。

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

mov si,0
mov ax,[si]

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

```
mov bx,0
mov ax,[bx+123]

mov si,0
mov ax,[si+123]

mov di,0
mov ax,[di+123]
```

7.7 SI和DI

- **问题7.2**

  用寄存器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。

7.7 SI和DI

- 问题7.2分析

用寄存器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
```

源地址：datasg:0

目的地址：datasg:16

程序设计时，可用一个循环来完成复制，循环中：
用ds:si 指向要复制的源字符串，
用ds:di 指向复制的目的空间。

7.7 SI和DI

- 代码段：

```
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：程序中，使用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。

7.7 SI和DI

- **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。

7.7 SI和DI

- 问题7.3分析：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
```

用[bx（si或di）+idata]的方式，来使程序变得简洁

7.8 [bx+si]和[bx+di]

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

  - 以[bx+si]为例进行讲解其含义。

7.8 [bx+si]和[bx+di]

- 访存方式

  [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）。

- 例如　　mov ax,[bx+si]
  - 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字节（字单元），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))
  - 等效格式（常用）：　　mov ax,[bx][si]

7.8 [bx+si]和[bx+di]

- **问题7.4**

用Debug查看内存如下：2000:1000 BE 00 06 00 00 00 ……

```
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]
```

写出左侧程序执行后，ax、bx、cx中的内容。

思考后看分析

7.8 [bx+si]和[bx+di]

- 问题7.4分析

用Debug查看内存如下：2000:1000 BE 00 06 00 00 00 ……

```
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]
```

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

7.8 [bx+si]和[bx+di]

- 问题7.4分析

用Debug查看内存如下：2000:1000 BE 00 06 00 00 00 ……

```
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]
```

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

7.8 [bx+si]和[bx+di]

- 问题7.4分析

用Debug查看内存如下：2000:1000 BE 00 06 00 00 00 ……

```
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]
```

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]含义相似，以[bx+si+idata]为例进行讲解。

7.9 [bx+si+idata]和[bx+di+idata]

- 访存格式

  [bx+si+idata]

  表示一个内存单元，其偏移地址为(bx)+(si)+idata，即bx中数值加上si中数值再加上idata。

- 例如 mov ax,[bx+si+idata]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字节（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)

7.9 [bx+si+idata]和[bx+di+idata]

- 指令mov ax,[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

7.9 [bx+si+idata]和[bx+di+idata]

- **问题7.5**

用Debug查看内存如下：2000:1000 BE 00 06 00 6A 22 ……

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

写出左边的程序执行后，ax、bx、cx中的内容。

思考后看分析。

7.9 [bx+si+idata]和[bx+di+idata]

- 问题7.5分析

用Debug查看内存如下：2000:1000 BE 00 06 00 6A 22 ……

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

段地址在ds中，(ds)=2000H；
偏移地址=(bx)+(si)+2=1002H；
指令执行后(ax)=0006H。

7.9 [bx+si+idata]和[bx+di+idata]

- 问题7.5分析

用Debug查看内存如下：2000:1000 BE 00 06 00 6A 22 ……

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

段地址在ds中，(ds)=2000H；
偏移地址=(bx)+(si)+2=1003H；
指令执行后(cx)=6A00H。

7.9 [bx+si+idata]和[bx+di+idata]

- 问题7.5分析

用Debug查看内存如下：2000:1000 BE 00 06 00 6A 22 ……

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

段地址在ds中，(ds)=2000H；
偏移地址=(bx)+(di)+2=1004H；
指令执行后(ax)=226AH。

7.10 不同的寻址方式的灵活应用

- 比较定位内存地址的方法（即**寻址方式**）：

（1）[idata]用一个常量表示内存地址，可用于直接定位一个内存单元；

（2）[bx]用一个变量表示内存地址，可用于间接定位一个内存单元；

（3）[bx+idata] 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。

7.10 不同的寻址方式的灵活应用

- 灵活的方式来定位一个内存单元的地址，使我们可以从更加结构化的角度来看待所要处理的数据。

以下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。

7.10 不同的寻址方式的灵活应用

- **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。

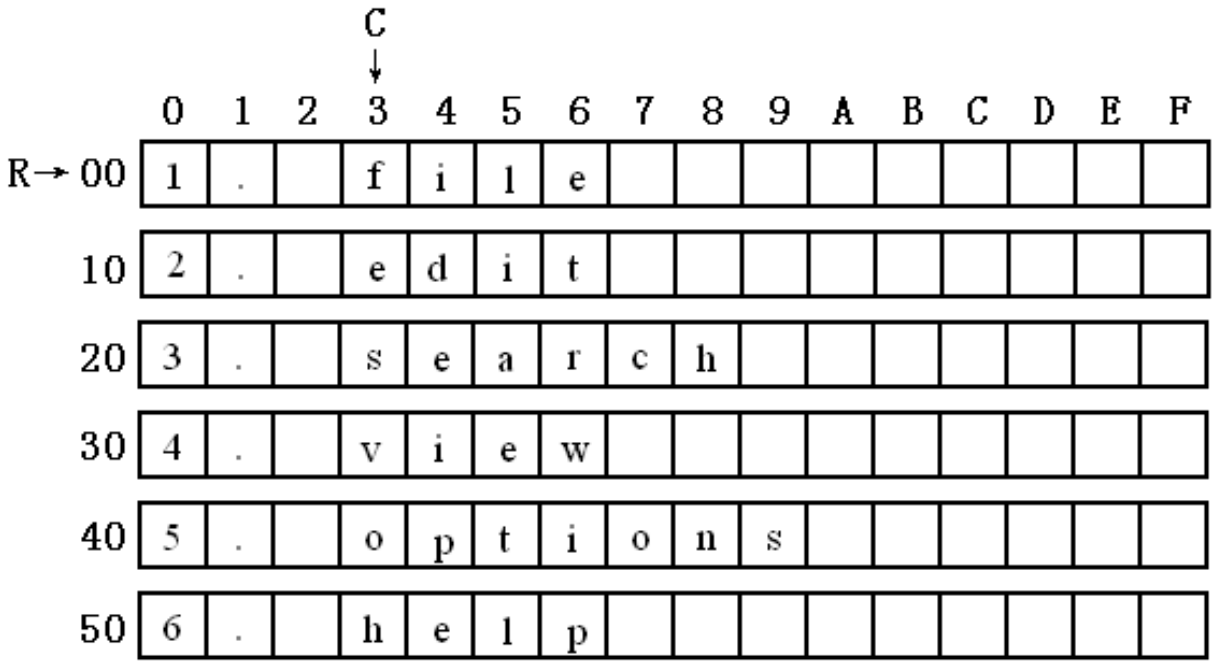
```
assume cs:codesg,ds:datasg
datasg segment
   db '1. file         '
   db '2. edit         '
   db '3. search       '
   db '4. view         '
   db '5. options      '
   db '6. help         '
datasg ends

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

7.10 不同的寻址方式的灵活应用

- 问题7.6分析

datasg中的数据的存储结构，如图：



这6个字符数组连续存放，可看成一个6行16列的二维数组。

按要求，修改每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）。

7.10 不同的寻址方式的灵活应用

- 问题7.6分析

  需要进行6次循环，用一个变量R定位行，用常量3定位列。处理的过程伪代码如下：

```
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
```

7.10 不同的寻址方式的灵活应用

- 问题7.6分析

```
    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
```

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，程序左图

7.10 不同的寻址方式的灵活应用

- 问题7.7

编程：将datasg段中每个单词改为大写字母。

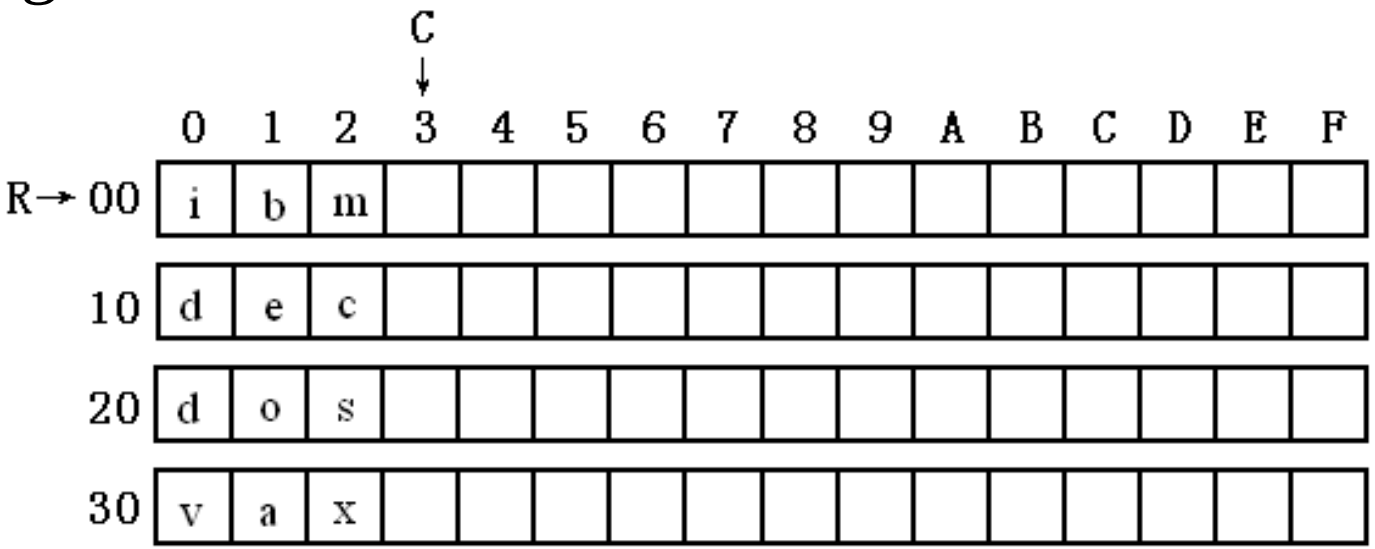
```
assume cs:codesg,ds:datasg
datasg segment
   db 'ibm          '
   db 'dec          '
   db 'dos          '
   db 'vax          '
datasg ends

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

7.10 不同的寻址方式的灵活应用

- 问题7.7分析

datasg中数据的存储结构，如图：



- 4个字符数组，每个长度为16字节。因它们是连续存放的，可将它们看成一个4行16列的二维数组。
- 要求，需修改每一个单词，即二维数组的每一行的前3列。

7.10 不同的寻址方式的灵活应用

- 问题7.7分析
  - 需要进行4x3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。
  - 程序设计思路

（1）用R 定位第1行，然后循环修改R行的前3列；

（2）然后再用R定位到下一行，再次循环修改R行的前3列……，

（3）如此重复直到所有的数据修改完毕

7.10 不同的寻址方式的灵活应用

- 问题7.7分析

```
      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
```

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，程序代码如左图。

问题7.8

程序什么问题？

思考后看分析。

7.10 不同的寻址方式的灵活应用

- 问题7.8分析

```
    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
```

问题在于cx的使用——进行二重循环，却只用了一个循环计数器cx，所以在进行内层的时候覆盖了外层循环的循环计数值。

怎么办呢？

思路：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。

7.10 不同的寻址方式的灵活应用

- 问题7.8分析

```
    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设置为内存循环的次数
 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
```

7.10 不同的寻址方式的灵活应用

- 程序中经常需要进行数据的暂存。
  - 用**寄存器**来暂存数据
    - 这不是一般化的解决方案，因寄存器的数量有限，且每个程序可能使用的寄存器都不一样。
    - 如上面的程序用dx来暂时存放cx中的值。若dx寄存器（甚至其他所有寄存器均）另有他用，该怎么办？
  - 用内存来暂存数据
    - 需要使用的时候，再从内存单元中恢复。
    - 这样就需要开辟一段内存空间。

7.10 不同的寻址方式的灵活应用

- 再次改进的程序(完整代码参见课本)

```
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
```

7.10 不同的寻址方式的灵活应用

- 以上程序用内存单元来保存数据，这种方法有些麻烦，因为如果需要保存多个数据的时候，程序员必须要记住数据各自放到了哪个内存单元中，这样程序容易混乱。

- 用怎样的结构来暂存数据，而使得程序更加清晰？
  - 一般来说，**使用内存的栈来暂存数据**。

7.10 不同的寻址方式的灵活应用

- 再次改进的程序（完整程序参见课本）

```
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
```

7.10 不同的寻址方式的灵活应用

- **问题7.9**

  编程，将datasg段中每个单词的前四个字母改为大写字母：

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

7.10 不同的寻址方式的灵活应用

- 问题7.9分析

datasg中的数据的存储结构，如图：

- 4个字符数组，每个长度为16字节。它们连续存放，可将它们看成一个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四个字母，即二维数组的每一行的3~6列。

7.10 不同的寻址方式的灵活应用

- 问题7.9分析
  - 我们需要进行4x4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。
  - 可以外层循环按行来进行，内层按列来进行。
  - 思路：

（1）首先用R定位第1行，循环修改R行的3+C（0≤C≤3）列；

（2）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……，

（3）如此重复直到所有的数据修改完毕。

7.10 不同的寻址方式的灵活应用

- 处理过程的伪代码大致如下：

```
    R=第一行的首地址;
    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
```

7.10 不同的寻址方式的灵活应用

- 可以用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。

7.10 不同的寻址方式的灵活应用

- 这一章中，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。

7.10 不同的寻址方式的灵活应用

- 下一章中，将对寻址方式的问题进行更深入地探讨。
- 之所以如此重视这个问题，是因为寻址方式的适当应用，使程序员能够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。