第七章 内存地址定位

and 和 or 指令

and

逻辑与指令，按位进行与运算。

```
mov ax, 01100011b
and ax, 00111011b
执行后ax 00100011b
```

or

逻辑或指令，按位进行或运算

```
mov ax, 01100011b
and ax, 00111011b
执行后ax 01111011b
```

通过字符形式给出数据

```
assume cs:code, ds:data

data segment
        db 'unIX' ; 四个字节 db 75h, 6eh, 49h, 58h
        db 'foRK' ; 四个字节 db 66h, 6fh, 52h, 4bh
data ends

code segment
start:
        mov al, 'a' ; mov al, 61h
```

```
        mov bl, 'b' ; mov bl, 62h

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

大小写转换问题

ASCII字符中大小写字母相差20H，也就是2^5，每一个大写字母的第六位都是0,所以将其变成1就成为了对应的小写字母。同理将第六位变成0,就变成了大写字母。

问题： 将给定的一个单词变成全部大写，第二个单词变成全部小写。

```
assume cs:code, ds:data

data segment
        db 'BaSiC'
        db 'iNfOrMaTion'
data ends

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

        mov bx, 0
        mov cx, 5
s:
        mov al, [bx]
        or al, 00100000b ; 将第六位变成1, 变成小写字母
        mov [bx], al
        inc bx
        loop s
```

```
21
22          mov bx, 5
23          mov cx, 11
24  s0:
25          mov al, [bx]
26          and al, 11011111b ; 将第六位变成0， 变成大写
    字母
27          mov [bx], al
28      inc bx
29      loop s0
30
31      mov ax, 4c00h
32      int 21h
33  code ends
34  end start
```

[bx+idata]

```
1  mov bx, 10
2  mov ax, [bx+200]
```

上面的语句其实相当与 `mov ax, ds:[10+200] => mov ax, ds:[210]`

`[bx+200]`也可以写成如下的几种形式

`[200+bx]`, `200[bx]`, `[bx].200`

通过[bx+idata]改写上面程序

```
1  assume cs:code, ds:data
2
3  data segment
4          db 'BaSiC'
5          db 'MinIx' ; 长度是一致的
6  data ends
```

```
7
8  code segment
9  start:
10         mov ax, data
11         mov ds, ax
12
13         mov bx, 0
14         mov cx, 5
15 s:
16         mov al, [bx] ; 或者写出 mov al, 0[bx]
17         or al, 00100000b ; 将第六位变成1, 变成小写字
   母
18         mov [bx], al
19         mov al, [bx+5] ; 或者写成 mov al, 5[bx]
20         and al, 11011111b ; 将第六位变成0,  变成大写
   字母
21         inc bx
22         loop s
23
24     mov ax, 4c00h
25     int 21h
26 code ends
27 end start
```

此时只需要一次循环就够了，而上面的`mov al, 0[bx]`和`mov al, 5[bx]`就非常向很多高级程序语言如C中的数组了。

- C语言， `a[i], b[i]`
- 汇编语言, `0[bx], 5[bx]`

这里的0和5就像是数组的名字，bx就像是数组的索引。

SI 和 DI

`si`和`di`是8086CPU中和`bx`功能相近的寄存器。

例：将字符串复制到他后面的数据区中。

```
assume cs:code, ds:data

data segment
        db 'Welcome to masm!'
        db '................'
data ends

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

        mov si, 0
        mov di, 16
        mov cx, 16
s:
        mov al, [si]
        mov [di], al
        inc si
        inc di
        loop s

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

上面是一个字节一个字节的复制，我们发现有16个字节，也就是8个字，其实通过字的形式复制效率更高一点，只需要八次循环。

```
assume cs:code, ds:data

data segment
        db 'Welcome to masm!'
        db '................'
data ends

```

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

        mov si, 0
        mov di, 16
        mov cx, 8 ; 修改循环次数
s:
        mov ax, [si] ; 处理字型数据
        mov [di], ax
        add si, 2
        add di, 2
        loop s

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

这题因为要处理的字符串长度是一致的，我们可以使用`[bx+idta]`来处理。（或者`[si+idata]`, `[di+idata]`）

```
assume cs:code, ds:data

data segment
        db 'Welcome to masm!'
        db '................'
data ends

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

        mov si, 0
        mov cx, 8 ; 修改循环次数
```

```
s:
        mov ax, 0[si] ; 处理字型数据
        mov 16[si], ax
        add si, 2
        loop s

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

这样处理起来更加简单。

其他寻址方式

[bx+si]和[bx+di]

意思非常容易理解，也可以写作是`[bx][si]`。

[bx+si+idata]和[bx+di+idata]

也可以写作如下的格式。

- `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`

寻址方式的应用

- `[idata]`使用一个常量表示地址，直接定位一个内存单元。
- `[bx]`使用一个变量表示地址，间接定位一个内存单元。
- `[bx+idata]`使用一个常量一个变量表示i地址，可在一个起始地址上使用变量间接定位一个内存单元。

- `[bx+si]`使用两个变量表示地址
- `[bx+si+idata]`使用两个变量一个常量表示地址。

例： 将数据段的每个单词的前4个字母大写

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

data segment
        db '1. display      '
        db '2. brows        '
        db '3. replace      '
        db '4. modify       '
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, 16h

        mov ax, data
        mov ds, ax

        mov bx, 3
        mov cx, 4
s: ; 外层循环
        push cx ; 使用栈来保存外层循环的cx
        mov cx, 4 ; 内层循环也需要使用到cx寄存器
s0: ;内层循环
        mov al, [bx]
        and al, 11011111b ; 第六位变0,变成大写字母
        mov [bx], al
        inc bx
```

```
        loop s0

        pop cx ; 将保持的外层循环的cx出栈
        add bx, 12
        loop s

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

当遇到多层循环时，我们需要使用栈来保存外层循环的cx。

一般来说，我们需要暂存数据时，都需要使用到栈。

两层循环的框架

```
        mov cx, xx ; 外层的循环次数
outer:
        push cx ; 暂存外层cx
        ...
        mov cx, xx ; 内层的循环次数
inner:
        ...
        loop inner

        ...
        pop cx ;取出外层cx
        loop outer
```