

汇编语言程序设计

lesson 3

上节回顾

- 1、CS存放指令的段地址，IP存放指令的偏移地址，CPU将CS:IP指向的内容当作指令执行。
- 2、8086CPU的工作过程：（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指向下一条指令；（3）执行指令。（转到步骤（1），重复这个过程。）
- 3、8086CPU提供转移指令修改CS、IP的内容：jmp。

上节回顾

- 4、字在内存中存储时，要用两个地址连续的内存单元来存放，字的低位字节存放在低地址单元中，高位字节存放在再高地址单元中。
- 5、用 **mov** 指令要访问内存单元，可以在**mov** 指令中只给出单元的偏移地址，此时，段地址默认在**DS**寄存器中。
- 6、在内存和寄存器之间传送字型数据时，高地址单元和高**8**位寄存器、低地址单元和低**8**位寄存器相对应。

上节回顾

- 7、8086CPU提供了栈操作机制，方案如下：
在SS，SP中存放栈顶的段地址和偏移地址；
提供入栈和出栈指令，他们根据SS:SP指示的地址，
按照栈的方式访问内存单元，任意时刻，SS:SP指向栈顶元素。
- 8、push指令的执行步骤：
 - 1) $SP=SP-2$;
 - 2) 向SS:SP指向的字单元中送入数据。
- 9、pop指令的执行步骤：
 - 1) 从SS:SP指向的字单元中读取数据;
 - 2) $SP=SP+2$ 。

Debug和汇编编译器Masm对指令的不同处理

- Debug和编译器对指令“mov ax,[0]”的不同处理
- 默认进制的区别

第5章 [bx]和loop指令

- 5.1 [bx]
- 5.2 Loop指令
- 5.3 loop和[bx]的联合应用

5.1 [bx]

- 我们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： $(ds * 16 + (bx)) = (ax)$ 。

5.1 [bx]

■ 问题5.1

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

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

```
1  mov ax,2000H
2  mov ds,ax
3  mov bx,1000H
4  mov ax,[bx]
5  inc bx
6  inc bx
7  mov [bx],ax
8  inc bx
9  inc bx
10 mov [bx],ax
11 inc bx
12 mov [bx],al
13 inc bx
14 mov [bx],al
```


5.1 [bx]

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

- 问题5.1:
 - 指令执行后:
2000:1006单元的内容为BE。

5.2 Loop指令

- 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。
- 通常我们用 loop 指令来实现循环功能，CX 中存放循环次数。

5.2 Loop指令

- 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 ，结果存放在ax中。

任务1分析

- 任务2：编程计算 2^3 。

任务2分析

- 任务3：编程计算 2^{12} 。

任务3分析

5.2 Loop指令

- 任务1：编程计算 2^2 ，结果存放在`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。

程序代码

5.2 Loop指令

- 任务1：编程计算 2^2 ，结果存放在ax中。

程序代码：

```
assume cs:code
```

```
code segment
```

```
mov ax,2
```

```
add ax,ax
```

```
mov ax,4c00h
```

```
int 21h
```

```
code ends
```

```
end
```

5.2 Loop指令

- 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。

程序代码

5.2 Loop指令

- 任务2：编程计算 2^3 。

程序代码：

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

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

5.2 Loop指令

- 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。

程序代码

5.2 Loop指令

- 任务3：编程计算 2^{12} 。

程序代码：

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

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

5.2 Loop指令

- 任务3：编程计算 2^{12} 。

程序代码：

```
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
```

5.2 Loop指令

编程计算 2^{12} 。

程序代码：

```
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
```

■ 程序分析：

(1) 标号

在汇编语言中，标号代表一个地址，此程序中有一个标号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**), 如果为零则执行下一条指令(下一条指令是**mov ax,4c00h**)。

5.2 Loop指令

- 程序分析（续）：
(3) 以下三条指令
 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，我们可以利用cx来控制add ax,ax的执行次数。

编程计算 2^{12} 。

程序代码：

```
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
```

5.2 Loop指令

- 程序的执行过程：

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

- (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；

5.2 Loop指令

- 程序的执行过程:

`mov cx,11`

`s: add ax,ax`

`loop s`

(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;

5.2 Loop指令

- 程序的执行过程：

mov cx,11

s: add ax,ax

loop s

(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;

5.2 Loop指令

- 程序的执行过程：

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

- (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;

5.2 Loop指令

- 程序的执行过程：

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

- (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;

5.2 Loop指令

- 程序的执行过程：

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

(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)

5.2 Loop指令

- 用**cx**和**loop** 指令相配合实现循环功能的三个要点：
 - （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

5.2 Loop指令

■ 问题5.2

- 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 $(ax)=(ax)+123$ 。
- 程序代码

5.2 Loop指令

- 问题5.2程序代码

```
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
```

5.2 Loop指令

■ 问题5.3

改进问题5.2程序，提高 123×236 的计算速度。

- 思考后看分析。

- 分析：

问题5.2程序做了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。

- 程序代码请自行实现。

5.3 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。思考：

- （1）运算后的结果是否会超出 dx 所能存储的范围？
- （2）我们是否将 ffff:0~ffff:b中的数据直接累加到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中的目标？
- （4）我们到底怎样将用ffff:0~ffff:b中的8位数据，累加到16位寄存器dx中？

5.5 loop和[bx]的联合应用

- **关键问题：**类型的匹配和结果的不超界。
- 具体的说，就是在做加法的时候，我们有两种方法：

$(dx) = (dx) + \text{内存中的8位数据}:$

$(di) = (di) + \text{内存中的8位数据};$

- 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。
- 目前的方法：得用一个**16位**寄存器来做中介。
(在后面的课程中我们还有别的方法)

5.5 loop和[bx]的联合应用

■ 问题分析：（续）

而这些不同的偏移地址是可在 $0 \leq X \leq 0bH$ 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：

$$\text{sum} = \sum_{x=0}^{0bH} (0ffffh \times 10h + x)$$

5.5 loop和[bx]的联合应用

- 问题分析：（续）

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 $(dx) = 0ffffh$ ， $X = 0$ ， $ds:X$ 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。
- 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。
- 写出程序

5.5 loop和[bx]的联合应用

- 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。

5.5 loop和[bx]的联合应用

- 这时，我们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。

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

7.1 and和or指令

- (1) **and** 指令：逻辑与指令，按位进行与运算。

如 `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。

- (2) **or**指令：逻辑或指令，按位进行或运算。

如 `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。

7.2 关于ASCII码

- **ASCII编码**（一种编码方案，在计算机系统中通常被采用的）。

编码	字符	编码	字符	编码	字符	编码	字符
0	NUL (null)	32	Space	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	\$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(72	H	104	h
9	TAB (horizontal tab)	41)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[123	{
28	FS (file separator)	60	<	92	\	124	
29	GS (group separator)	61	=	93]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

- 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。

- 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 'unlX'
    db 'foRK'
data ends
code segment
    start:mov al,'a'
           mov bl,'b'
           mov ax,4c00h
           int 21h
code ends
end start
```


7.3 以字符形式给出的数据

- 上面的源程序中：
 - “db ‘unlX’ ” 相当于 “db 75H,6EH,49H,58H”, “u”、“n”、“l”、“X”的ASCII码分别为75H、6EH、49H、58H;
 - “db ‘foRK’ ” 相当于 “db 66H,6FH,52H,4BH”, “u”、“n”、“l”、“X”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 al,’b’”相当于 “mov al,62H”, “b”的ASCII码为62H。

7.4 大小写转换的问题

- 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，进行对比，从中找到规律。

大写	二进制	小写	二进制
A	01000001	a	01100001
B	01000010	b	01100010
C	01000011	c	01100011
D	01000100	d	01100100

- 通过对比，我们可以看出来，小写字母的**ASCII**码值比大写字符的**ASCII**码值大20H。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”。

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,21H
      mov [bx],al
      inc bx
      loop s
      :
codesg ends
end start
```

7.4 大小写转换的问题

- 从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（位数从0开始计算）为0，小写字母的第5位为1。
- 一个字母，不管它原来是大写还是小写：
 - 我们将它的第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。
 - 用or和and指令，可将数据中的某一位置0或置1。完整的[程序代码](#)

7.5 [bx+idata]

- 在前面，我们可以用[bx]的方式来指明一个内存单元，我们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）。

7.5 [bx+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： $(ax) = ((ds) * 16 + (bx) + 200)$

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]
```


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。

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: .....
codesg ends
end start
```

7.6 用[bx+idata]的方式进行数组的处理

- 按照我们原来的方法，用[bx]的方式定位字符串中的字符。

代码段中的程序代码：

- 我们用[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。

```
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

- 改进的程序:

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

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

- 我们用[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。

;定位第一个字符串的字符

;定位第二个字符串的字符

7.6 用[bx+idata]的方式进行数组的处理

- 程序还可以写成这样：

```
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
```

7.7 SI和DI

- SI和DI是8086CPU中和bx功能相近的寄存器，但是SI和DI不能够分成两个8位寄存器来使用。

下面的三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]`

7.7 SI和DI

- 下面的三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]`

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，我们还可以用更灵活的方式：
 - $[bx+si]$
 - $[bx+di]$
- $[bx+si]$ 和 $[bx+di]$ 的含义相似，我们以 $[bx+si]$ 为例进行讲解。

7.8 $[bx+si]$ 和 $[bx+di]$

- $[bx+si]$ 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）。
- 我们看下指令**mov ax,[bx+si]**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。

7.8 $[bx+si]$ 和 $[bx+di]$

- 指令 `mov ax,[bx+si]` 的数学化的描述为:

$$(ax) = ((ds) * 16 + (bx) + (si))$$

该指令也可以写成如下格式（常用）：

`mov ax,[bx][si]`

7.8 [bx+si]和[bx+di]

■ 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 ax,[bx+di]
```

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**)

7.9 [bx+si+idata]和[bx+di+idata]

- 指令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：
$$(ax) = (ds) * 16 + (bx) + (si) + idata$$

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，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 ax,[bx+2+di]
```

7.10 不同的寻址方式的灵活应用

- 如果我们比较一下前而用到的几种定位内存地址的方法（可称为寻址方式），就可以发现有以下几种方式：
 - （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。
- 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。

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：

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
R→ 00	i	b	m													
10	d	e	c													
20	d	o	s													
30	v	a	x													

我们可以将这4个字符串看成一个4行16列的二维数组。

7.10 不同的寻址方式的灵活应用

- 我们需要进行**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。

7.10 不同的寻址方式的灵活应用

- 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

7.10 不同的寻址方式的灵活应用

- 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，
程序代码：

- 问题7.8

```
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
```

7.10 不同的寻址方式的灵活应用

■ 问题7.8

- ✓ 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，造成在进行内层的时候覆盖了外层循环的循环计数值。
- ✓ 我们可以用寄存器**dx**来临时保存**cx**中的数值。

7.10 不同的寻址方式的灵活应用

改进的程序:

```
    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

- 进一步改进：
- 寄存器数量有限，一般来说，在需要暂存数据的时候，我们可以使用内存保存数据。比如：栈。

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中的数据的数据的存储结构，如图：

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

本章作业

- 完成问题7.9的程序，以“专业-学号-姓名-02”命名文档，文档中包括源程序（要求注释清晰）、运行结果截图。
- 在debug中调试程序，观察loop循环指令执行过程并截图主要过程，回答：（1）循环开始处的标号实质是什么？（2）用什么指令可以直接进入循环？（3）用什么指令可以一次性执行完循环，如何跳出循环？

小结