数据处理的两个基本问题

第8章 数据处理的两个基本问题

- 8.1 bx、si、di、bp
- 8.2 机器指令处理的数据所在位置
- 8.3 汇编语言中数据位置的表达
- 8.4 寻址方式
- 8.5 指令要处理的数据有多长？
- 8.6 寻址方式的综合应用
- 8.7 div 指令
- 8.8 伪指令 dd
- 8.9 dup

引言

- 本章总结之前所学所有内容。

- 任何计算机进行数据处理、运算时，要知道：
  （1）处理的数据在什么地方？
  （2）要处理的数据有多长？

- 本章针对8086CPU对这两个基本问题进行讨论。

引言

- 为了描述简介，我们将使用两个描述性的符号
  - reg表示**寄存器**

  - sreg表示**段寄存器**
    - 包括：ds、ss、cs、es。

8.1 bx、si、di、bp

- 总结：
  - （1）在8086CPU 中，只有这4个寄存器（bx、bp、si、di）可以用在“[…]”中来进行内存单元的寻址。

正确的指令

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

错误的指令

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

8.1 bx、si、di、bp

- （2）在“[…]”中，这4个寄存器（bx、bp、si、di）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

正确的指令

```
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]
```

错误的指令

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

8.1 bx、si、di、bp

- （3）只要在[...]中使用寄存器bp，而指令中没有显性的给出段地址，**则段地址默认在ss中**。

  - 如：

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所在位置

- 大部分机器指令是数据处理的指令，大致分为三类：
  - 读取
  - 写入
  - 运算

8.2 机器指令处理的数据所在位置

- 机器指令要区分待处理数据所在位置（三个地方）：
  - CPU内部
  - 内存
  - 端口 （将在后面的课程中进行讨论）

- 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？

汇编语言中用三个概念来表达数据的位置。

1、立即数（idata）

2、寄存器

3、段地址（SA）和偏移地址（EA）

8.3 汇编语言中数据位置的表达

- 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），称为立即数（idata），在汇编指令中直接给出。例如：

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

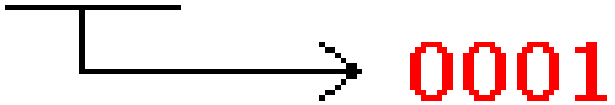
8.3 汇编语言中数据位置的表达

- 1、立即数（idata）

mov ax,1

对应机器码：

B80100



执行结果：(ax) = 1

8.3 汇编语言中数据位置的表达

- 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
```

8.3 汇编语言中数据位置的表达

- 2、寄存器

  mov ax,bx

  对应机器码：89D8

  执行结果：(ax) = (bx)

8.3 汇编语言中数据位置的表达

- 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。

  - 默认的段地址的寄存器
    - bx ~ ds
    - bp ~ ss
  - 存放段地址的寄存器也可显性给出。

8.3 汇编语言中数据位置的表达

- 存放段地址的寄存器是默认(ds或ss段寄存器)的
  - 示例：

mov ax,[0]

mov ax,[bx]

mov ax,[bx+8]

mov ax,[bx+si]

mov ax,[bx+si+8]

段地址默认在**ds**中

mov ax,[bp]

mov ax,[bp+8]

mov ax,[bp+si]

mov ax,[bp+si+8]

段地址默认在ss中

8.3 汇编语言中数据位置的表达

- 显性的给出存放段地址的寄存器
  - 示例

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)

8.3 汇编语言中数据位置的表达

- 3、段地址（SA）和偏移地址（EA）

如：　　mov ax,[bx]

对应机器码：8B07

执行结果: (ax) = ((ds)×16+(bx))

8.4 寻址方式

- 当数据存放在内存中的时候，可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的5类寻址方式
  - 直接寻址
  - 寄存器间接寻址
  - 寄存器相对寻址
  - 基址变址寻址
  - 相对基址变址寻址

BX、BP是基址寄存器(Base)
SI、DI叫变址寄存器(Index)

8.4 寻址方式

- 8086CPU的各种寻址方式

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] [si] [di] [bp]	EA=(bx);SA=(ds) EA=(si);SA=(ds) EA=(di);SA=(ds) EA=(bp);SA=(ss)	寄存器间接寻址	[bx]
[bx+idata] [si+idata] [di+idata] [bp+idata]	EA=(bx)+idata;SA=(ds) EA=(si)+idata;SA=(ds) EA=(di)+idata;SA=(ds) EA=(bp)+idata;SA=(ss)	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]
[bx+si] [bx+di] [bp+si] [bp+di]	EA=(bx)+(si);SA=(ds) EA=(bx)+(di);SA=(ds) EA=(bp)+(si);SA=(ss) EA=(bp)+(di);SA=(ss)	基址变址寻址	用于二维数组：[bx][si]
[bx+si+idata] [bx+di+idata] [bp+si+idata] [bp+di+idata]	EA=(bx)+(si)+idata; SA=(ds) EA=(bx)+(di)+idata; SA=(ds) EA=(bp)+(si)+idata; SA=(ss) EA=(bp)+(di)+idata; SA=(ss)	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]

扩展内容：386以后的寻址方式

- 386后的寻址方式更灵活

  A=基址+（变址 x 比例因子）+位移量

  - 位移量（displacement）：即 idata。
  - 基址（base）：扩展到通用寄存器
  - 变址（index）：扩展到通用寄存器
  - 比例因子（scale factor）是80386以上CPU新增加的。其值可为1、2、4或8。

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。

8.5 指令要处理的数据有多长？

- 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。
  - （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。
  - （3）其他方法

8.5 指令要处理的数据有多长？

- 以下指令中，寄存器指明了处理数据长度

字操作：

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

字节操作

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

8.5 指令要处理的数据有多长？

- 以下指令中，通过关键字明确指示访存单元。

用word ptr指明访存单位为字：

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

用byte ptr指明访存单元为字节：

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

8.5 指令要处理的数据有多长？

- 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：　　将使内存中的内容变为

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

2000:1000 01 FF FF FF FF FF……

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

2000:1000 01 00 FF FF FF FF……

8.5 指令要处理的数据有多长？

- 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；

  - 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。

8.5 指令要处理的数据有多长？

- 有些指令默认了访问的是字单元还是字节单元，

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

- 以下通过一个问题来讨论各种寻址方式的作用。

- 实际应用

8.6 寻址方式的综合应用

关于DEC公司的一条记录（1982年）：

公司名称：DEC

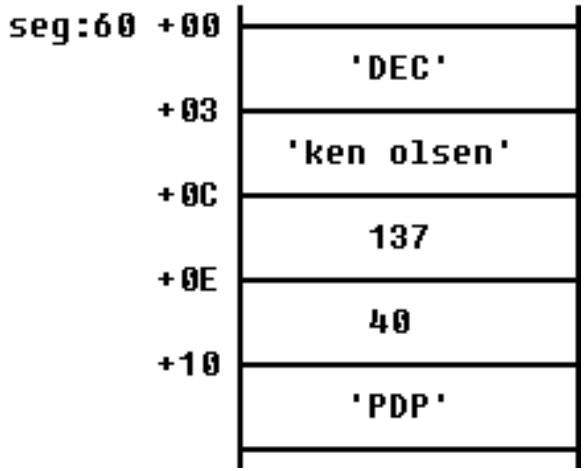
总裁姓名：Ken Olsen

排　　名：137

收　　入：40

著名产品：PDP

在内存中如下存放



1988年DEC公司的信息有了变化：

１、Ken Olsen 在富翁榜上的排名已升至38位；

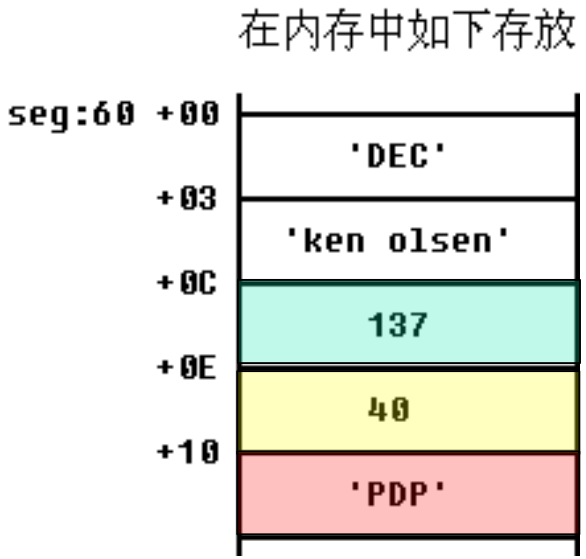
２、DEC的收入增加了70亿美元；

３、该公司的著名产品已变为VAX系列计算机。

任务：编程修改内存中的过时数据。

8.6 寻址方式的综合应用

- 分析：要修改的数据：



排名字段
收入字段
产品字段：（全部3个字符）

8.6 寻址方式的综合应用

- 修改方法：

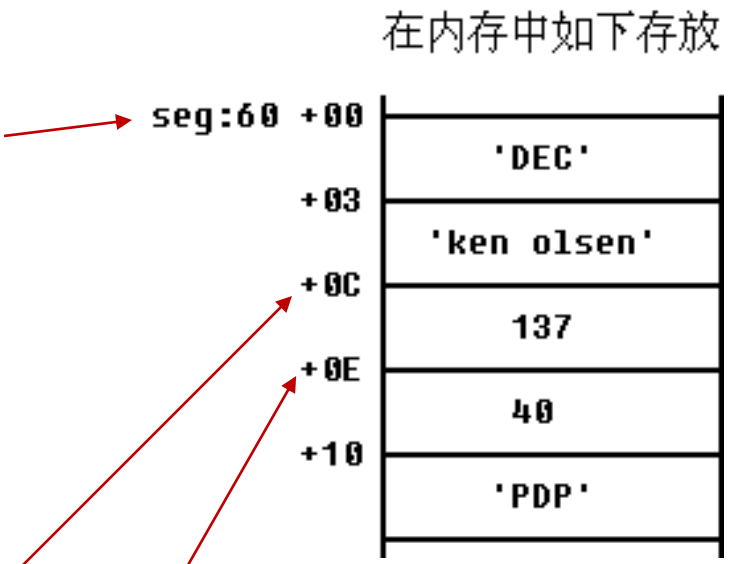
（1）首先确定DEC公司记录的位置：R=seg:60

（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。



8.6 寻址方式的综合应用

- 修改方法：（续）

（6）确定产品字段在记录中的位置：10H。产品字段是一个字符串，需逐个修改其中每个字符。

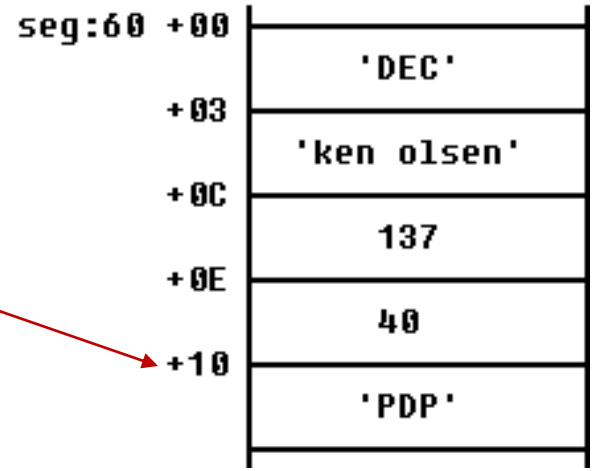
（7）确定第一个字符在产品字段中的位置：P=0。

（8）修改R+10H+P处的数：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。

在内存中如下存放



8.6 寻址方式的综合应用

- 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'
```

8.6 寻址方式的综合应用

- 用C语言来描述这个程序：

```
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 */
        };
strcut company dec={"DEC","Ken Olsen",137,40,"PDF"};
/* 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;
}
```

8.6 寻址方式的综合应用

- 按照C语言风格，用汇编语言编写这个程序：

```
mov ax,seg
mov ds,ax
mov bx,60h                          ;记录首地址送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';
```

8.6 寻址方式的综合应用

- 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，使得在编程的时候可以从结构化的角度去看待所要处理的数据。

- 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）。

8.6 寻址方式的综合应用

- 一般来说，可以用[bx+idata+si]的方式来访问结构体中的数据。

  bx定位整个结构体，

  idata定位结构体中的某一个数据项，

  si 定位数组项中的每个元素 。

更为贴切的书写方式，如：

[bx].idata

[bx].idata[si]

8.6 寻址方式的综合应用

- 在C语言程序中：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，

  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]

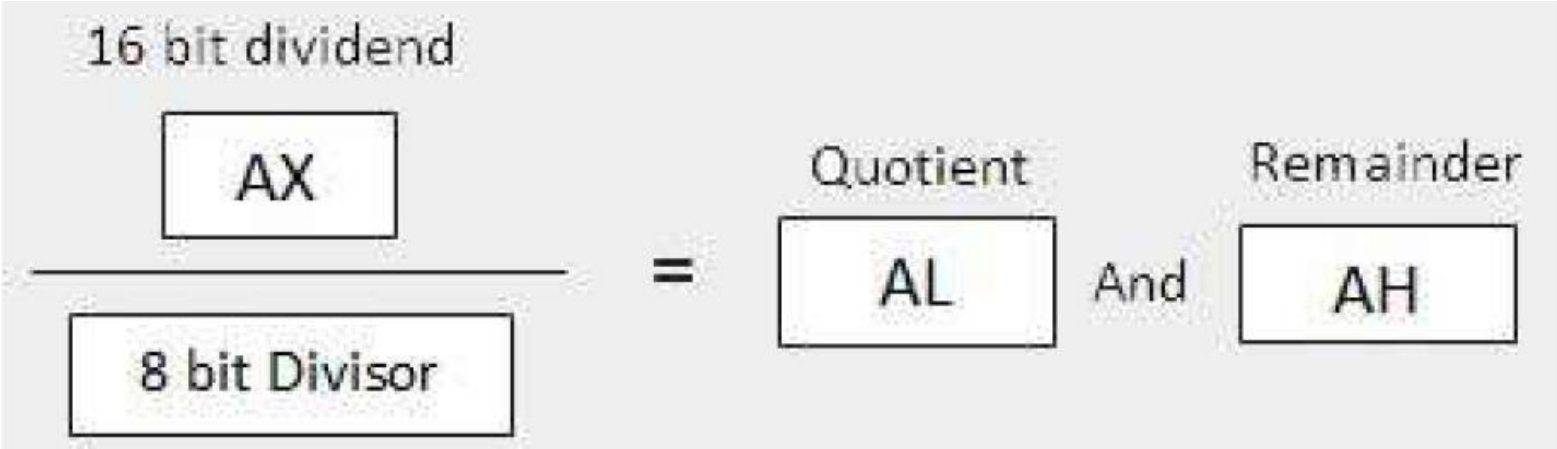
8.7 div 指令

- div指令格式：
  - div reg
  - div 内存单元

- 分为：
  - 8位被除数
  - 16位被除数

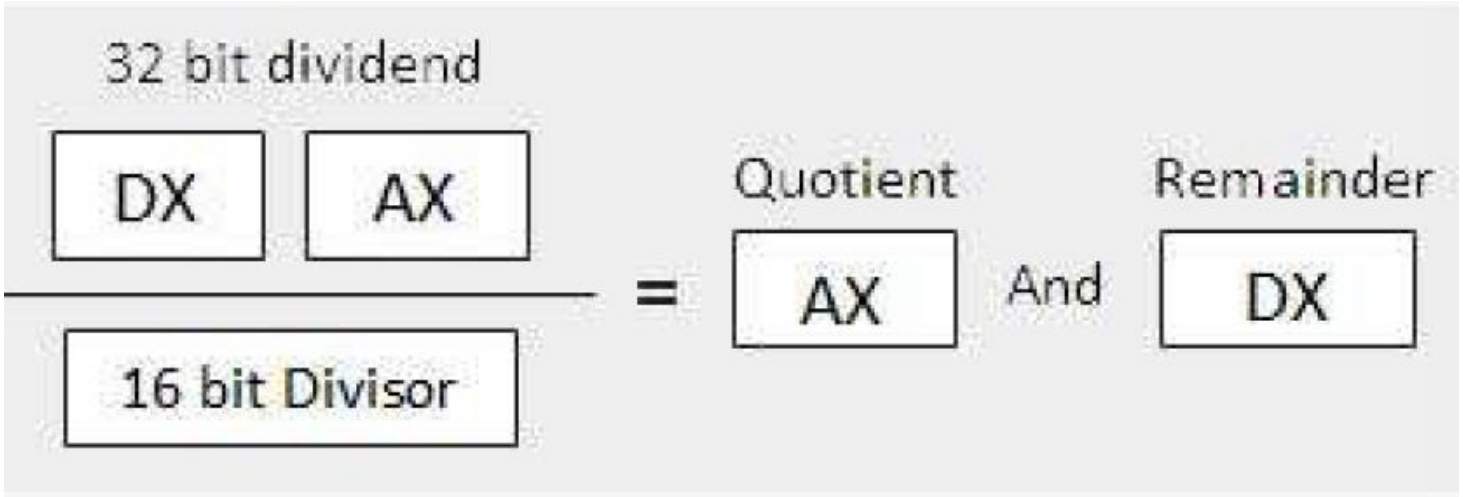
8.7 div 指令

- div
  - 8位被除数的除法指令



8.7 div 指令

- div
  - 16位被除数的除法指令



8.7 div 指令

- div是除法指令，使用div作除法的时候：

div除法		
除数 内存或寄存器	8位	16位
被除数 默认寄存器	AX	DX和AX
商 默认寄存器	AL	AX
余数 默认寄存器	AH	DX

8.7 div 指令

- div指令示例
  - 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

8.7 div 指令

- div指令示例（续）
  - 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

8.7 div 指令

- 编程：利用除法指令计算100001/100。

分析：

- 要用dx和ax两个寄存器联合存放100001
  - 因为被除数 100001 大于65535，不能用ax寄存器存放，即要进行16位的除法。
- 要用一个16位寄存器来存放除数100
  - 虽然除数100小于255，可在一个 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

程序如下

8.7 div 指令

- 编程：利用除法指令计算100001 / 100。

程序：

16进制： 86A1H

```
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）。

8.7 div 指令

- 编程：利用除法指令计算1001/100。

分析:

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：

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

程序执行后　　(al)=0AH（即10），
(ah)=1（余数为1）。

8.8 伪指令 dd

- 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。

示例：在data段中定义了三个数据：

```
data segment
        db 1  ;第一个数据为01H，在data:0处，占1个字节
        dw 1  ;第二个数据为0001H，在data:1处，占1个字
        dd 1  ;第三个数据为00000001H，在data:3处，占1个双字
data ends
```

8.8 伪指令 dd

- **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，商存放在第3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。

8.8 伪指令 dd

- 问题8.1分析

  用div 计算data段中第一个数据除以第二个数据后的结果，商存放在第3个数据的存储单元中。

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

被除数为双字，故在做除法前用dx和ax存储
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。

8.8 伪指令 dd

- 问题8.1分析

程序代码

```
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

- 伪指令dup （duplicate的缩写）配合db、dw、dd 等使用的，进行**数据重复**。

- 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 (重复的双字数据)

8.9 dup

- dup示例
  - db 3 dup (0)

    相当于 db 0,0,0
  - db 3 dup (0,1,2)

    相当于 db 0,1,2,0,1,2,0,1,2
  - db 3 dup (‘abc’,’ABC’)

    相当于db ‘abcABCabcABCabcABC’

8.9 dup

- dup的使用，如：定义一个容量为 200 个字节的栈段。

如果不用dup，

```
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
```

使用dup：

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