汇编语言

汇编语言程序形式

```
;这里展示的是16位汇编，32位需要在头处加  .386 同时每个segment后面需要加use 16
;i.e data segment use 16

;数据段 一般存放变量，指针等等
data segment
T db "lalala",0Dh,0Ah,"$"
;回车，换行
data ends
;代码段
code segment
assume cs:code,ds:data,ss:stk
;可以在code段中进行变量的定义
; t db "XYZ",0Dh,0Ah,"$"
;这时下方如果有指令mov dl,t[bx]在编译后会变成mov dl,cs:[0+bx]
main:
    mov ax,data
    mov ds,ax
    ;......
    mov ah,4Ch
    int 21h
code ends
;堆栈段，可以没有
stk segment stack
db 200h dup(0)
stk ends

end main
```

函数书写规范

```
;输入2个2位的16进制数,输出它们的和.
;例如输入FF02,则输出0101。
.386
code segment use16
assume cs:code
input:
   mov ah, 1
   int 21h; AL=getchar()
   cmp al, 'A'; 'A'==41h, '9'==39h
   jb is_digit
is_alpha:
   sub al, 'A'
   add al, 10
   jmp input_done
```

```
is_digit:
   sub al, '0'; AL是函数的返回值
input_done:
   ret

input_a_number:
   push bx
   call input
   mov bl, al
   call input
   shl bl, 4
   add al, bl; AL是函数的返回值
   pop bx
   ret

output_a_number:
   push cx
   push dx
   mov cx, 4
;1011 0000 1111 0101 循环左移前
;0000 1111 0101 1011 循环左移后
convert_next_digit:
   rol ax, 4
   push ax
   and ax, 0Fh; AX=0000 0000 0000 1011
   cmp al, 10
   jb output_is_digit
   sub al, 10
   add al, 'A'
   jmp output_a_digit
output_is_digit:
   add al, '0'
output_a_digit:
   mov ah, 2
   mov dl, al
   int 21h; putchar(DL)
   pop ax
   sub cx, 1
   jnz convert_next_digit
   pop dx
   pop cx
   ret

main:
   call input_a_number
   mov bh, 0
   mov bl, al; BX=00FF
   call input_a_number
   mov ah, 0 ; AX=0002
   add ax, bx; 计算两数之和
   call output_a_number; 以16进制格式输出AX的值
   mov ah, 4Ch
   int 21h
code ends
end main
```

寄存器

- **段寄存器**

  汇编语言中一共存在4个段寄存器，分别是**cs,ds,es,ss**,

  其中**cs,ss**两个寄存器不需要用户对其进行赋值操作，而是当程序刚开始运行时，由**操作系统**对其进行赋值，

  相反，**ds,es**两个寄存器必须由用户对其进行赋值

  cs: code segment cs决定了程序第一条指令的段地址，assume指令编译后会消失，也就是说，在程序刚开始运行时，当前将要执行的指令地址一定是cs:ip，这也就意味着程序开始时操作系统已经对cs和ip进行过赋值。

  ss: stack segment 操作系统也会自动对ss和sp进行赋值 其中sp为堆栈指针寄存器，表示堆栈的偏移地址

  程序中没有定义堆栈段时，DOS系统会自动设定SS=1000H（首段的段地址）,SP=0并不代表堆栈大小为0，执行Push操作时，sp会先-2，从而成为1000H段中最后两个字节位置

  和C语言中堆栈相同，ss:sp指向堆栈top

  ```
  mov ax ,ss:[sp];语法错误
  mov eax ,ss:[esp] ;正确
  mov bp ,sp
  mov ax ,ss:[bp];正确，可以简化为 mov ax,[bp]
  ;[si]\[di]\[bx]隐含段地址为ds，而[bp]隐含段地址是ss，
  ```

  es: extra segment,和ds类似，可以用来表示一个数据段的段地址

  需要注意的一个地方：变量虽然一般在data segment中进行定义，但我们也允许在code段中进行变量的定义

- **FL标志寄存器**
  - 考纲上列出考点：标志位：CF、ZF、OF、SF、DF、IF、TF
  - **mov 指令不会影响任何标志位 mov指令不会影响任何标志位 mov 指令不会影响任何标志位!**
  - ***CF ZF SF OF AF PF: 这6个称为状态标志*** 被动的反映状态

    DF TF IF: 这3个称为控制标志 主动改变他们的值，影响CPU的行为
  - CF： carry flag 进位标志

    ```
    mov ah, 0FFh
    add ah, 1; AH=0, CF=1产生了进位
    add ah, 2; AH=2, CF=0
    sub ah, 3; AH=0FFh, CF=1产生了借位

    ;关于CF的跳转指令 jc 和jnc (jmp if not carry),其中jc与jb指令等价
    ;注意，不能通过cmp指令获取cf值，即 cmp cf,1是错误的

    clc ;clear carry flag|cf = 0
    stc ;set carry flag |cf = 1
    cmc ;使cf反转
    ```

  - ZF: 零标志位(zero flag)

```
sub ax, ax; AX=0, ZF=1
add ax, 1; AX=1, ZF=0
add ax, 0FFFFh; AX=0, ZF=1, CF=1
jz is_zero; 会发生跳转, 因为当前ZF==1
          ;与jz相反的指令是jnz, jnz是根据ZF==0作出跳转

;关于ZF的跳转指令 jz , jnz (jmp if not zero),其中je与jz指令等价
;cmp指令内部会产生减法运算,->cmp ax,bx -> ax-bx==0?ZF = 1:ZF = 0;
```

- OF :溢出标志位 (overflow flag) !!!OF默认符号数

```
;这里需要注意，正数+负数永远不会发生溢出
;正+正或负+负时可能会发生溢出
;如果是非符号数作算术运算CF位代表溢出位

mov ah, 7Fh
add ah, 1; AH=80h, OF=1, ZF=0, CF=0, SF=1
mov ah,  80h
add ah, 0FFh; AH=7Fh, OF=1, ZF=0, CF=1, SF=0
mov ah, 80h
sub ah, 1; AH=7Fh, OF=1, ZF=0, CF=0, SF=0

;关于OF的跳转指令 jo , jno (jmp if not overflow)
```

- SF: 符号标志位(sign flag)

```
;SF用来反映正负
;copy运算结果的最高位
mov ah, 7Fh
add ah, 1; AH=80h=1000 0000B, SF=1
sub ah, 1; AH=7Fh=0111 1111B, SF=0
jns positive; 会发生跳转, 因为SF==0

;关于SF的跳转指令 js, jns (jmp if not overflow)
```

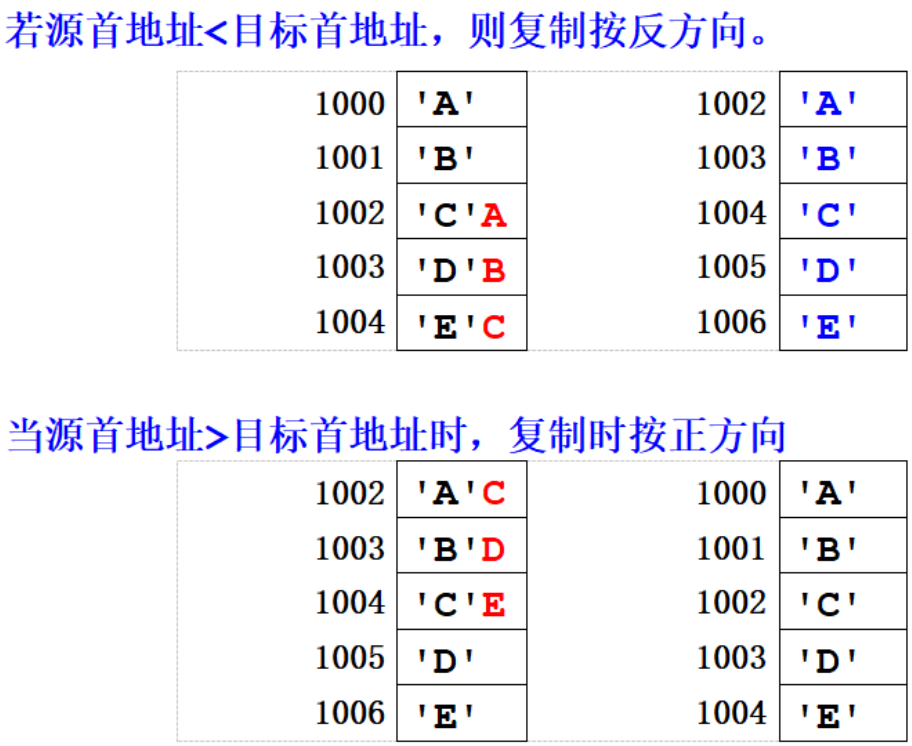
- ~~***不考*** PF: 奇偶标志(parity flag)~~

```
mov ah, 4
add ah, 1; AH=0000 0101B, PF=1表示有偶数个1
mov ax, 0101h
add ax, 0004h; AX=0105h=0000 0001 0000 0101B
             ; PF=1只统计低8位中1的个数
            ;要是低8位中1的个数是奇数时，PF=0
;关于PF的跳转指令 jp ,jnp, jpe, jpo (jmp if parity odd)
```

- DF： 方向标志(direction flag)

```
;DF 用来控制字符串移动的方向
;当DF=0时为正方向(低地址到高地址)，当DF=1是反方向。
;cld指令使DF=0， std指令使DF=1

cld ;DF=0
std ;DF=1

;在进行rep movsb等字符串复制操作时，如果源数据的首地址>目标数据的首地址，为了保证复制过程不产生意外，一定是正方向复制
```

i.e:



- IF :中断标志 (interrupt flag)

```
;IF = 1 允许中断
cli ;IF = 0
sti ;IF = 1


cli; clear interrupt禁止硬件中断
...; 重要代码
sti; set interrupt允许硬件中断
```

- TF: 跟踪/陷阱标志(trace/trap flag)

```
;TF = 1 ->进入单步模式
;当TF=1时,CPU在每执行完一条指令后,会自动在该条指令与下条指令之间插入一条int 1h指令并执行它。

;改变TF值必须通过pushf 和 popf实现

;要让 TF=1
pushf
pop ax; AX=FL
```

```
or ax, 100000000B; 或or ax, 100h
push ax
popf; FL=AX, 其中第8位即TF=1
;让TF = 0
pushf
pop ax; AX=FL
and ax, not 100000000B; 或and ax, 0FEFFh
push ax
popf; FL=AX, 其中第8位即TF=0
```

指令

- ## 字符串操作指令
  - 字符串传送指令：movsb movsw movsd

```
rep movsb
;strcpy的效果
;其中rep表示repeat，s表示string，b表示byte
;在执行此指令前要做以下准备工作：


;①ds:si源字符串(si就是source index)
;②es:di目标字符串(di就是destination index)
;③CX=移动次数
;④DF=0即方向标志设成正方向(用指令cld)


例子：要把以下左侧4个字节复制到右侧
1000:0000 'A'        2000:0000  'A'
1000:0001 'B'        2000:0001  'B'
1000:0002 'C'        2000:0002  'C'
1000:0003 00         2000:0003  00
则程序可以这样写：
mov ax, 1000h
mov ds, ax
mov si, 0      ; mov si, 3
mov ax, 2000h
mov es, ax
mov di, 0      ; mov di, 3
mov cx, 4
cld             ; std
rep movsb
循环结束时          循环结束时
si=4              si=FFFF
di=4              di=FFFF
cx=0              cx=0
```

- 字符串比较指令： cmpsb cmpsw cmpsd

```
①cmpsb
比较byte ptr ds:[si]与byte ptr es:[di]
当DF=0时，SI++，DI++
当DF=1时，SI--，DI--
②repe cmpsb;(若本次比较相等则继续比较下一个)
again:
if(cx == 0) goto done;
比较byte ptr ds:[si]与byte ptr es:[di]
当DF=0时，SI++，DI++
当DF=1时，SI--，DI--
cx--
若本次比较相等则 goto again
done:
③repne cmpsb
(若本次比较不等则继续比较下一个)


总结：同样的，字符串比较也需要ds:si,es:di,cx,DF的前期准备
```

- 字符串扫描指令: scasb scasw scasd

```
scasb:
  cmp al, es:[di]
  di++; 当DF=1时，为di--

repne scasb:
next:
  if(cx == 0) goto done;
  cmp al, es:[di]
      di++; 当DF=1时，为di--
      cx--
  je done
  goto next
done:


repe scasb: 可用于跳过与al相等的字符
repne scasb: 可用于查找字符串长度
```

- stosb | lodsb

```
字符串填充al : stosb

注意：！！！ stosb操作对象是 es:[di]
令 es:[di] = al

例：把从地址1000:10A0开始共100h个字节的
内存单元全部填0
mov ax, 1000h
mov es, ax; ES=1000h
mov di, 10A0h                          mov di,10A0h
```

```
mov cx, 100h      mov cx, 80h        mov cx,40h
cld                 cld              cld
xor al, al        xor ax, ax         xor eax, eax
rep stosb         rep stosw          rep stosd


lodsb 与上面相反，从ds:[si]中读取，放到al中

例: 设DS:SI "##AB#12#XY"
且ES:DI指向一个空的数组, CX=11
通过编程过滤#最后使得ES:DI "AB12XY"
   cld
again:
   lodsb; AL=DS:[SI], SI++
        ; mov al, ds:[si]
        ; inc si
   cmp al, '#'
   je   next
   stosb; ES:[DI]=AL, DI++
        ; mov es:[di], al
        ; inc di
next:
   dec cx
   jnz again
```

- # 通用数据传送指令
  - mov
    1. 不能mov 内存变量，内存变量
    2. 不能mov 两个不等宽的单元
  - push | pop
    1. push | pop 后可跟变量，要声明长度

       i.e push word ptr ds:[bx+2]

    2. 8086中, push不能跟常数, 但80386及以后的cpu允许push一个常数。
    3. **push/pop后面不能跟一个8位的寄存器或变量。**

       在8086中push的实现过程为 sp-**2**,然后存入数据，
  - xchg： 交换两个寄存器单元

```
mov ax, 1
mov bx, 2
xchg ax, bx; 则ax=2, bx=1
```

- # 地址传送指令
  - lea : **Load effective address(取偏移地址)**

```
lea dx, ds:[1000h] ; DX=1000h
mov dx, 1000h; 上述lea指令的效果等同于mov指令

;分析
lea dx, abc ; 效果等价于以下指令
```

```
mov dx, offset abc
;两者存在差距，假设abc=1000h
;lea dx,abc编译后 lea dx,[1000h]
;mov dx,offset abc 编译后 mov dx,1000h
;实际上mov操作更加高效
;变量名之前+offset ,可以得到偏移地址

;lea指令的优越性
;个人认为是用lea指令简化算术运算
lea dx, ds:[bx+si+3]; dx=bx+si+3
;mov dx, bx+si+3  错误
mov dx, bx; \
add dx, si;  | 效果等同于上述lea指令
add dx, 3 ; /

lea eax, [eax+4*eax]; EAX=EAX*5 用lea做乘法
lea eax, [eax+eax*2]; EAX=EAX*3
```

- lds | les

```
;假定要把1000:0000中存放的远指针取出来，存放到es:bx中，则
mov ax, 1000h
mov ds, ax
les bx, dword ptr ds:[0000h]
            ; es=1234h, bx=5678h
            ; es要的是高位
lds si, dword ptr ds:[0000h]
            ;ds = 1234h,si = 5678h
;此指令造成了ds值的改变

;如果是32位远指针
les ebx, fword ptr cs:[p]; es=18h
;ebx=12345678h


;总结：les/lds指令用于远指针
;根本原因在于16位汇编中远指针需要32位的空间
;所以转换到寄存器中必须是ES:XX或DS:XX
```

一个例子，看看最好

```
;这个例子贴在这里的目的:
;1.远指针定义可以不用必须是dd类型，自己按小端规则
;2.les di,dword ptr video_addr[bx]中dword ptr一定要有
;3.在这个程序架构中bx 每次加4


data segment
video_addr dw 0000h, 0B800h, 160, 0B800h
;0000h,0B800h可以想象成远指针
;远指针不受类型影响
;上述定义也可以写成:
;video_addr dd 0B8000000h, 0B80000A0h
data ends
code segment
assume cs:code, ds:data
```

```
main:
   mov ax, data
   mov ds, ax
   mov bx, 0
   mov cx, 2
next:
   les di, dword ptr video_addr[bx]
;dword ptr 必须存在，强制类型转换
   mov word ptr es:[di], 1741h
   add bx, 4
   sub cx, 1
   jnz next
   mov ah, 1
   int 21h
   mov ah, 4Ch
   int 21h
code ends
end main
```

- **PUSHF POPF ｜32位：PUSHFD,POPFD**
  - 针对标志寄存器FL的操作

```
;pushf/popf配合起来除了可以刻意改变FL中的某些位外，也可以用来保护/恢复FL的值
;寄存器FL及IP在编程时都是不能直接引用的:
mov ip, ax;\
mov ax, ip; \ 都是错误的
mov bx, fl; /
mov fl, bx;/
;ip,fl不可以直接改变！！

pushf;
pop ax; ax=fl   √
or ax,1
push ax
popf;fl=ax

pushfd; 将efl压入堆栈
pop eax;eax = efl
```

- **符号扩充指令：CBW CWD CDQ**
  - **注意：符号扩充 符号扩充 符号扩充!**
  - cbw : convert byte to word

```
mov al, 0FEh
cbw; 把AL扩充成AX, AX=0FFFEh
```

  - cwd : convert word to double word

```
mov ax, 8000h
cwd; 把AX扩充成DX:AX, DX=FFFFh, AX=8000h
```

- cdq : convert double word to quadruple word

```
mov eax, 0ABCD1234h
cdq; 把EAX扩充成EDX:EAX
    ; EDX=0FFFFFFFFh, EAX=0ABCD1234h
```

- movsx

```
movsx ax, al; sx:sign extension符号扩充
            ; 效果等同于cbw
```

- **总结：** 符号扩充指令的用途：

  1. 主要用途：为除法服务

```
;符号数相除时，al应扩充到16位,这时符号扩充指令就派上了用场
mov al ,-2;或 mov al,0FEh
cbw; AX=FFFE
mov bl,2
idiv bl;AL=-1,AH=0
```

零扩充指令： movzx

```
;movzx ax,al <=> mov ah,0
movzx ax, al; zx:zero extension
movzx eax, al
movzx ebx, cx
```

换码指令: XLAT

```
.386 ; 表示程序中会用32位的寄存器
data segment use16; use16表示偏移使用16位
t db "0123456789ABCDEF"
x dd 2147483647
data ends

code segment use16
assume cs:code, ds:data
main:
   mov ax, data    ;\
   mov ds, ax      ; / ds:bx->t[0]
   mov bx, offset t;/
   mov ecx, 8
   mov eax, x
next:
   rol eax, 4
   push eax
   and eax, 0Fh
   xlat
   mov ah, 2
   mov dl, al
   int 21h
   pop eax
```

```
    sub ecx, 1
    jnz next
    mov ah, 4Ch
    int 21h
code ends
end main
```

- **总结:**

  XLAT指令需要的前置工作： DS:BX AL赋值

  从而查找ds:bx对应数组中第al位

  执行xlat指令后，结果存储在al中

- ## 循环指令 (loop)

关于loop

重要! CX = 0时loop循环最大次数

```
Jcxz done ;如果不希望cx=0时还进行10000H次循环，加jmp
;loop自动对cx -1
;循环次数由cx初始值决定
; cx = 0时 可以使loop循环最大次数
;先-1,再判断

i.e
例: 求1+2+3的和
mov ax, 0
mov cx, 3
next:
add ax, cx; ax +3, +2, +1
loop next; cx=2, 1, 0
           ; dec cx
           ; jnz next
done:
```

- ## 函数相关指令 CALL | RET

```
;我的一些看法
call 时，先将下一个地址传入堆栈，再jmp到对应函数
jmp ->有去无回
call ->return
return 相当于 pop ip 将ip改成了要跳回的地址


重要! 汇编语言中的三种参数传递方式
① 寄存器传递
f:
   add ax, ax; ax=2*ax
   ret         ; 返回时ax就是函数值
main:
   mov ax, 3; ax就是f()的参数
   call f
next:
   mov ah, 4Ch
```

```
    int 21h
```

② 变量传递

```
f:
    mov ax, var
    add ax, ax; ax就是函数值
    ret
main:
    mov var, 3; var是一个16位的变量，用作参数
    call f
```

在汇编语言中，用db、dw等关键词定义的变量均为全局变量。在堆栈中定义的变量才是局部变量。

寄存器和变量传递存在以下问题：
1.函数不能递归
2.函数不能重入（多线程）

③ 堆栈传递

```
;堆栈状态相当于:
    sp->|bp          在f中push进来的bp
        |ip          call时传进来的ip
        |variable    我们需要的参数
f:
    push bp
    mov bp, sp
    mov ax, [bp+4]; 从堆栈中取得参数
    ;注意这里采用的是bp+4，因为在call时，目标ip值已经被压入堆栈
    add ax, ax
    pop bp
    ret
main:
    mov ax, 3
    push ax; 参数压入到堆栈
    call f
    add sp, 2
;Add sp,2目的：清空堆栈，使堆栈回到初始状态

;C语言的任何一个函数都有义务将4个寄存器压入堆栈：
;Bx,bp,si,di:16位
;Ebx,ebp,esi,edi：32位
;指针寄存器重要~~
```

建议：自己想一想堆栈布局
push的时候是 sp先动，然后往里面放数据
pop时候相反

最后别忘记add sp,用来清空

C语言实现参数可变也是通过堆栈传递的，将参数从右到左压入堆栈中，然后通过计算sizeof来改变堆栈指针

```
;把这个例子贴过来，觉得蛮有参考价值
(2) C语言函数调用y=f(2,3)求两数之和转化成汇编语言
f:
    push bp;(4)
    mov bp, sp
    mov ax, [bp+4]
```

```
    add ax, [bp+6]
    pop bp; (5)
    ret; (6)
main:
    mov ax, 3
    push ax; (1)
    mov ax, 2
    push ax; (2)
    call f; (3)
here:
    add sp, 4;(7)
上述程序运行过程中的堆栈布局如下:
ss:1FF8 old bp <- bp (4)
ss:1FFA here <- (3)(5)
ss:1FFC 02   <- (2)(6)
ss:1FFE 03   <- (1)
ss:2000 ??   <- (7)


;关于堆栈传递参数放在零碎整理中
```

- **跳转指令　(jx)**
  - 有一部分 je,jz等等在cmp和FL中提到
    - 其中有一些显而易见的等价关系，比如je 和 jz, jb 和 jc 等
  - 当然也会有很猥琐的并不等的

```
jmp指令还是很重要的
①jmp short target        ; 短跳
②jmp near ptr target     ; 近跳
③jmp far ptr target      ; 远跳
一般情况下，编译器会自动度量跳跃的距离，因此我们在
写源程序的时候不需要加上short、near ptr、far ptr等类型修饰。即上述三种写法一律可以
简化为jmp target。

也就是说，我们平时经常用的jmp next

;关于短跳
②短跳指令的机器码
       地址              机器码    汇编指令
    1D3E:0090      ...
    1D3E:0100      EB06      jmp  108h
短跳指令的机器码由2字节构成:
第1个字节=EB
第2个字节=
Δ=目标地址-下条指令的偏移地址=108h-102h=06h
    1D3E:0102      B402      mov  ah, 2
    1D3E:0104      B241      mov  dl, 41h
    1D3E:0106      CD21      int  21h
    1D3E:0108      B44C      mov  ah, 4Ch
    1D3E:010A      CD21      int  21h

;近挑，机器码位数增多，从而可以跳到更远的地方
(3)近跳指令
①近跳指令的3种格式
```

```
jmp 偏移地址或标号; 如jmp 1000h
jmp 16位寄存器       ; 如jmp bx
jmp 16位变量         ; 如jmp word ptr [addr]
```

②近跳指令的机器码

```
  地址          机器码        汇编指令
1D3E:0100    E9FD1E       jmp  2000h
```

近跳指令的第1个字节=E9

第2个字节=Δ=目标地址-下条指令的偏移地址

=2000h-103h=1EFDh

```
1D3E:0103    B44C          mov  ah，4Ch
1D3E:0105    CD21          int  21h
...
1D3E:2000    ...
```

```
;远跳 ->32位
;20位地址任意跳
```

①远跳指令的2种格式

```
jmp 段地址:偏移地址
jmp dword ptr 32位变量
```

②远跳指令的机器码

```
jmp 1234h:5678h; 机器码为0EAh,78h,56h,34h,12h
```

远跳到某个常数地址时,在源程序中不能直接用jmp指令，而应该改用机器码0EAh定义，如:

```
db 0EAh
dw 5678h
dw 1234h
```

上述3行定义合在一起表示jmp 1234h:5678h

例: jmp dword ptr 32位变量 的用法

```
data segment
addr dw 0000h, 0FFFFh
;或写成addr dd 0FFFF0000h
data ends
code segment
assume cs:code, ds:data
main:
mov ax, data
mov ds, ax
jmp dword ptr [addr]
;相当于jmp FFFF:0000
code ends
end main

;这个例子很有趣的样子!  建议看看哈哈哈

code segment
assume cs:code
main:
   jmp next; jmp short next
exit:
   mov ah, 4Ch
   int 21h
next:
   mov ah, 2
   mov dl, 'A'
   int 21h
   jmp abc; jmp near ptr abc
   db 200h dup(0)
```

```
abc:
   jmp far ptr away; jmp far ptr away
code ends

fff segment
assume cs:fff
away:
   mov ah, 2
   mov dl, 'F'
   int 21h
   jmp far ptr exit; jmp far ptr exit
fff ends
end main
```

- ## int | iret

```
;个人感觉有点像调用函数，只不过中间多了一步查表
0:0~0:3FF之间共400h个字节的内存，这块区域称为
中断向量表。
int n  ->在进行好准备工作，该push的push掉之后会去中断向量表对应位置 即 0：n*4处拿到该函数的指针，再到相应位置进行函数操作

int指令做了四件事情
pushf
push cs即1000h
push 下条指令的偏移地址即2006h
jmp dword ptr 0:[84h]; jmp 1234h:5678h

iret在执行时，cpu做了以下3件事情:
pop ip即ip=2006h
pop cs即cs=1000h
popf

||这里留个问题？？？ 先pop ip的话不会导致pop cs无法进行么？
```

- ## 运算指令：算术运算+逻辑运算
  - ### 算数运算

    add | sub

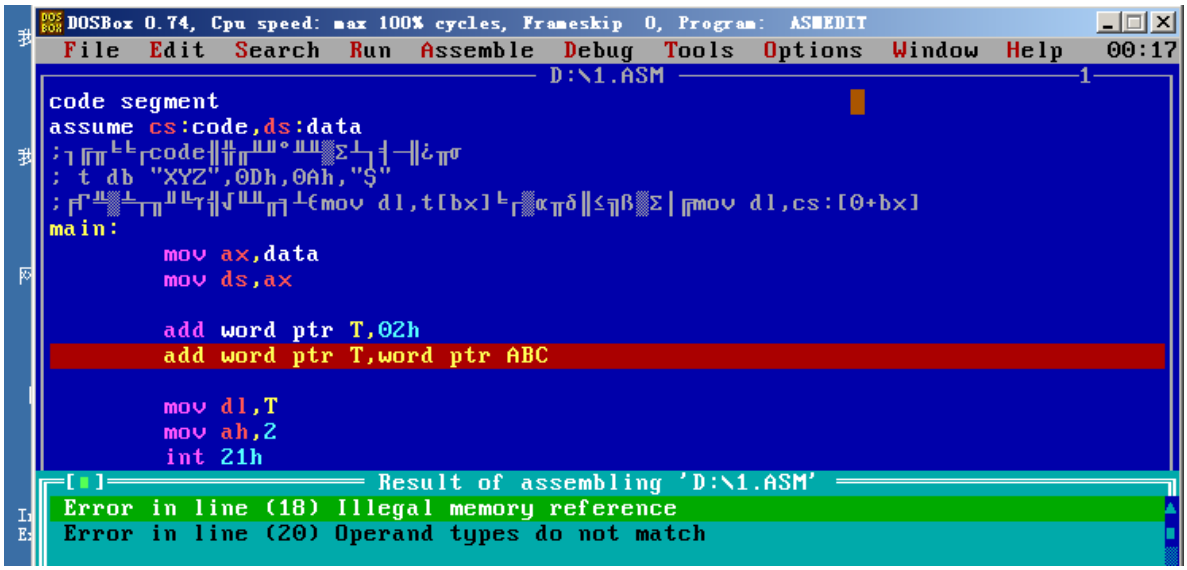
```
;在这里给出相关合理例子
add ax,bx
add ax,2
add ax,t ;t为变量，在编译后会变为ds:[]
add t,ax
add word ptr t,2
;不能实现两个变量相加
;我的理解:
;add 操作必须知道长度才可以进行，两个变量不能知道其长度，因此不能作加法,但事实上即使规定了变量长度也会报错
```

```
;add 参照mov指令 内存变量不能完全等同于寄存器

;老师在课件中给出两个变量可以实现间接相加
mov ax,a
add ax,b
mov a,ax

;总结一下,add sub 指令必须要等宽才可以进行
;! ! mov 也同理, mov bh,ax报错

adc ah,0 ;=>AH = AH +0+CF
        ;adc带进位加
```

inc | adc | sbb | dec | neg | cmp

```
inc ax ; ax = ax + 1
;Inc 指令不影响CF标志位
;i.e
again:
add ax, cx
inc cx
jnc again  ; ->检测ax+cx,而不是inc cx
done:

adc dx,5678h ;DX = DX + 5678h + CF

;dec指令不影响CF标志位! ! !
mov ax ,3
dec ax ; ax = ax - 1

;neg :negate 求相反数,会影响 CF,ZF,SF等标志位
mov ax,1
neg ax ; ax = -1 = 0FFFFh ,相当于减法 0-ax

mov ax ,0FFFEh
neg ax ; AX = 2, CF = 1, SF = 0, ZF = 0
;CF = 1: 0-ax 一定会产生carry
;SF = 0: AX为正数
;ZF = 0: AX != 0

; sbb: subtract with borrow 带借位减
;假设CF = 1
mov dx,5678h
sbb dx,1111h ; DX = 5678h-1111h-CF

;cmp
;cmp与sub的区别是抛弃两数之差, 仅保留标志位状态
mov ax, 3
mov bx, 3
cmp ax, bx; 内部是做了减法ax-bx, 但
;是抛弃两数之差, 只影响标志位。
je they_are_equal; 当ZF=1时则跳
jz they_are_equal; 当ZF=1时则跳
;因此je=jz

;ja,jb,jae,jbe -> 非符号数比较相关指令
;jb : CF = 1 -> jb <=> jc
```

```
;ja : CF = 0 且 ZF = 0
;注意机器指令并不智能
stc
jb below ;一定会跳转到below

;jg, jl, jge, jle -> 符号数比较相关的跳转指令
;jg <=> SF == OF
;jl <=> SF != OF
mov ah,0FFh
mov bh,1
cmp ah,bh   ;CF = 0,SF = 1,OF = 0(OF为溢出标志位)
;compare 进行减法运算，如果是负数，则SF置1
;OF=0表示没有溢出，SF=1(结果为负)是正确的
jg greater ;不满足，不跳
jl less ;满足

;以下是关于 jl为什么等价于SF!=OF 的讨论
;方法：分类讨论
mov ax, 3
mov bx, 2
cmp ax, bx; AX-BX=1, SF=0, OF=0->AX > BX
mov ah, 7Fh
mov bh, 80h
cmp ah, bh; AH-BH=0FFh, SF=1, OF=->AX>BX
mov ax, 2
mov bx, 3
cmp ax, bx; AX-BX=FFFFh, SF=1, OF=0->AX<BX
mov ah, 80h
mov bh, 7Fh
cmp ah, bh; AH-BH=1, SF=0, OF=1->AX<BX
```



乘除

```
;乘法溢出后可以继续运行,除法溢出后程序会挂掉
;乘法!
;8位乘法: 被乘数一定是AL, 乘积一定是AX
;例如mul bh表示AX=AL*BH
;16位乘法: 被乘数一定是AX, 乘积一定是DX:AX
;例如mul bx表示DX:AX=AX*BX
```

```
;32位乘法: 被乘数一定是EAX, 乘积一定是EDX:EAX
;例如mul ebx表示EDX:EAX=EAX*EBX
;imul 符号数乘法 mul 非符号数乘法
imul eax,ebx,1234h
    ;imul 寄存器 寄存器或变量  只能是常数
;eax = ebx * 1234h
imul eax,ebx
; 含义: EAX=EAX*EBX
        ;imul 寄存器  寄存器或变量
imul eax, dword ptr [esi]

imul eax, dword ptr [ebx+2], 1234h
;含义: eax = dword ptr [ebx+2] * 1234h


;除法!  cnmmmmmm
div bh
;首先分析bh 8位，所以是ax/bh = al .. ah
;商放在al中,余数放在ah中
div bx
;bx 16位,  dx:ax / bx = ax .. dx
;商放在ax中,余数放在dx中
div ebx
;ebx 32位 edx:eax / ebx = eax .. edx

;假定要把一个32位整数如7FFFFFFFh转化成十进制格式
;则一定要采用(3)这种除法以防止发生除法溢出

;关于除法溢出:
```

除法溢出:

(1) 除法溢出的两种情形:

①

```
mov ax, 1234h
mov bh, 0
div bh; 此时因为除以 0，所以会发生除法溢出
```

②

```
mov ax, 123h
mov bh, 1
div bh; 此时由于商无法保存到 AL 中，因此也会发生溢出。
```

```
(2) 除法溢出时会发生什么?
mov ax, 123h
mov bh, 1
;除法溢出时会在此处插入 int 00h 并执行
;在 dos 系统下, int 00h 会显示溢出信息并终止程序运行
div bh; 此处发生除法溢出
mov ah, 4Ch; \ 这 2 条指令将不可能被执行到
int 21h     ; /
```

~~小数运算:~~

```
fadd  fsub  fmul fdiv
;小数的+-*/运算指令

;小数变量的定义:
pi dd 3.14; 32位小数,相当于float
r  dq  3.14159; 64位小数, 相当于double
; q:quadruple 4倍的
s  dt  3.14159265; 80位小数, 相当于long double
  ;在C语言中要输出long double的值需要使用"%Lf"格式

; CPU内部一共有8个小数寄存器, 分别叫做
;st(0)、st(1)、…、st(7)
;其中st(0)简称st
;这8个寄存器的宽度均达到80位, 相当于C语言中的 long double类型。
```

- **逻辑运算**
- ！TEST指令
  - test ax, 8000h; ZF=0, AX=9234h
  - test ax,bx ->对ax,bx作与运算，但结果不会放到ax中，只会对标志位产生影响

```
 &     |     ^     ~     <<    >>
and   or    xor   not   shl   shr

_rotl()   _rotr()
循环左移   循环右移
rol       ror
```

- 提一下循环左移的实现:

```
Unsigned int rol(unsigned int x,int n)
{
//思路：先左移一位
//1011 0110
//0110 1100
//如何把1 放到末尾
//原数右移7位变成：0000 0001
//最后求或
Return x << n | x >> (sizeof(x)*8-n);
}
```

- **这里新加了四个指令** sal , sar , rcl , rcr

  注意：每一个移位指令都会影响CF，最后移出去的位一定会存放在CF中

  rcr | rcl :带进位的循环右移 | 左移

```
mov ah, 0B6h
stc; CF=1
rcl ah, 1; CF=1 AH=1011 0110 移位前
          ; CF=1 AH=0110 1101 移位后
mov ah, 0B6h
stc; CF=1
rcr ah, 1; AH=1011 0110  CF=1移位前
          ; AH=1101 1011  CF=0移位后
```

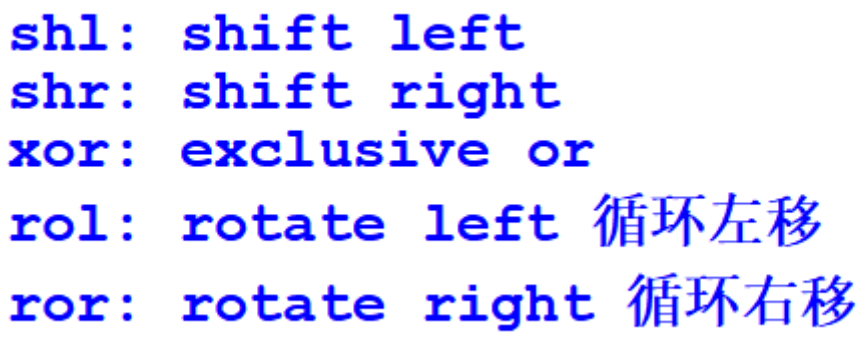
  sal | sar : 算数左移 | 右移

  sal <=> shl

  sar 不等价于 shr

  负数右移补1

- 方便记忆:




```
;    80386允许 shl ax,4 808不允许
;   一种安全的做法为
    mov cl,4
    shl ax,cl

    ;这个例子个人感觉非常好
    ;利用了rol实现16位整数转换为16进制
    data segment
    abc dw 32767
    s db 4 dup(0),0Dh,0Ah,'$'
```

```
    ;4 dup(0)相当于0,0,0,0
    ;s[0]='7'; s[1]='F'; s[2]='F'; s[3]='F'
    data ends
    code segment
    assume cs:code, ds:data
    main:
       mov ax, data
       mov ds, ax
       mov ax, abc
       mov cx, 4
       mov di, 0; 目标数组的下标,可以引用s[di]
    again:
       push cx
       mov cl, 4 ; 设ax的原值=7A9Dh
       rol ax, cl; AX=A9D7, 9D7A, D7A9, 7A9D
       ;这里的push操作
       push ax
       and ax, 0000000000001111B; 000Fh
       cmp ax, 10
       jb is_digit
    is_alpha:
       sub al, 10
       add al, 'A'
       jmp finish_4bits
    is_digit:
       add al, '0'
    finish_4bits:
       mov s[di], al
       pop ax
       pop cx
       add di, 1
       sub cx, 1
       jnz again
       mov ah, 9
       mov dx, offset s
       int 21h
       mov ah, 4Ch
       int 21h
    code ends
    end main
```

变量定义的类型：db , dw , dd , dq , dt

```
unsigned char      == 汇编的byte(字节) 8位
unsigned short int== 汇编的word(字) 16位
unsigned long int==汇编的double word(双字)32位
这三种类型定义的关键词分别为:db dw dd, 例如:
//define type
a db 12h; unsigned char a = 0x12;
//define word
b dw 1234h; unsigned short int b=0x1234;
//define doubleword
c dd 12345678h; unsigned long int c=0x12345678;
```

dd 也可以用来定义一个 32 位的小数即 float 类型的小数，例如：

```
pi dd 3.14; float pi=3.14;
```

dq 定义 64 位整数 quadruple word 或 double 类型小数例如：

```
x dq 123456788765
4321h;  __int64 x=...;
y dq 3.14; double y = 3.14;
```

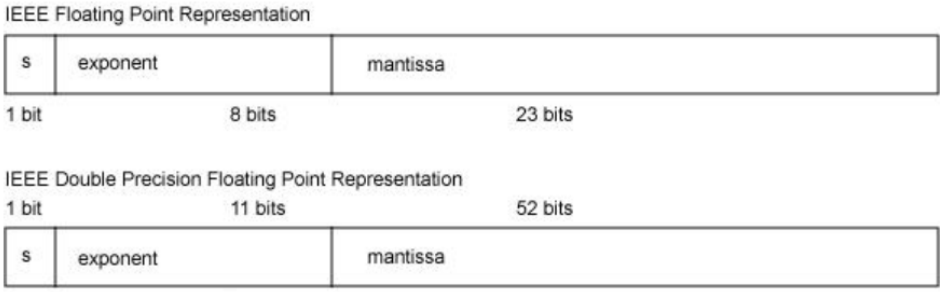
还有一个 dt 可以用来定义一个 80 位的小数即 long double 类型的小数：

```
z dt 3.14; 10 字节的小数，相当于 C 语言的 long double
        ; printf("%Lf", z);
```

- ~~*#这里不考！！*~~

  ~~其中，小数的表示方式为：~~

IEEE754 标准中规定 float 单精度浮点数在机器中表示用 1 位表示数字的符号，用 8 位表示指数，用 23 位表示尾数，即小数部分。对于 double 双精度浮点数，用 1 位表示符号，用 11 位表示指数，52 位表示尾数，其中指数域称为阶码。IEEE754 浮点数的格式如下图所示。



注意，IEE754 规定浮点数阶码E采用"**指数 e 的移码-1**"来表示，请记住这一点。为什么指数移码要减去1，这是 IEEE754 对阶码的特殊要求，以满足特殊情况，比如对正无穷的表示。

2.1单精度浮点数真值

IEEE754标准中，一个规格化32位的浮点数x的真值表示为：

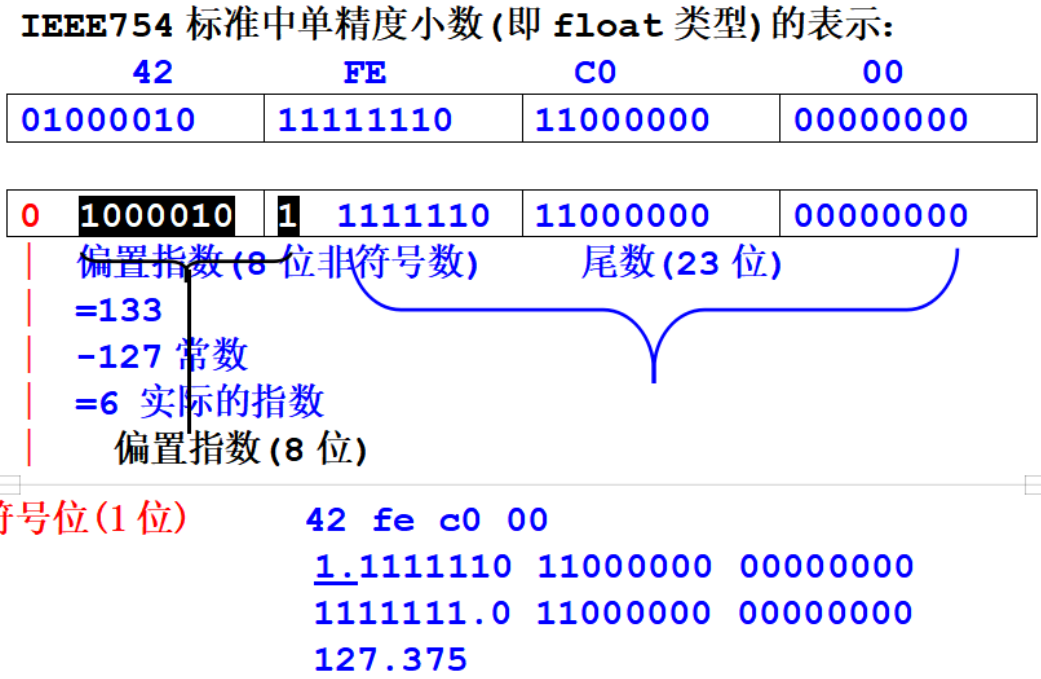
$$x = (-1)^S \times (1.M) \times 2^e$$

$$e = E - 127$$

其中尾数域值是1.M。因为规格化的浮点数的尾数域最左位总是1，故这一位不予存储，而认为隐藏在小数点的左边。

在计算指数e时，对阶码E的计算采用原码的计算方式，因此32位浮点数的8bits的阶码E的取值范围是0到255。其中当E为全0或者全1时，是IEEE754规定的特殊情况，下文会另外说明。

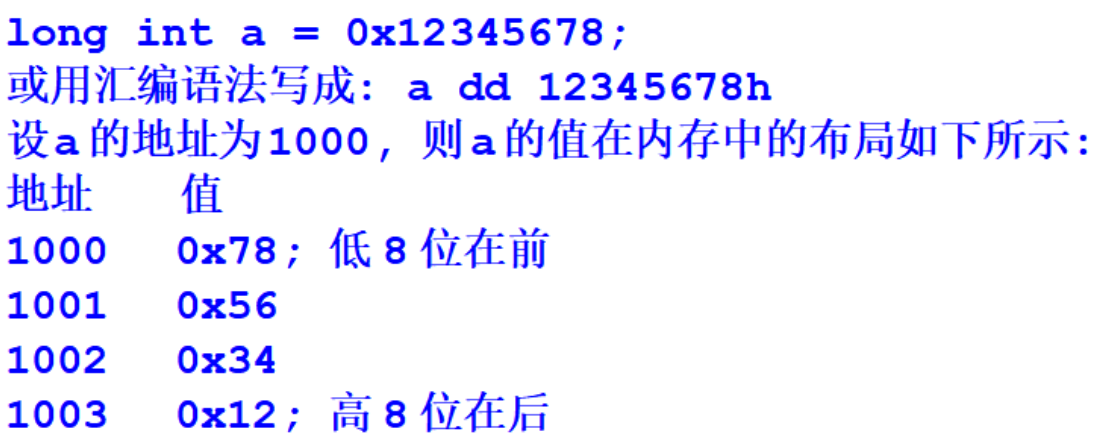
~~所以ppt中有~~



- **这里提到了一个小端规则，小端规则指的是变量存放时在内存中的布局方式**

低八位在前，高八位在后

Example:



零碎整理：

- ***prompt db "The result",0***

  ***Result db 100 dup(0);* ==>等价于 result db 0,0,0,...(100个0)**

  dup:duplicate 重复

- **逻辑地址**表示形式 -> 段地址:偏移地址

  一个段的长度 10000h = 64k

  一个物理地址可表示为多个偏移地址

  ds:[bx+0003]表示指针ds:bx+0003所指的对象,其中ds是变量的段地址, bx+0003是变量的偏移地址。

  段地址不能是常数，必须是寄存器

```
mov al, 1000h:[2000h]; 语法错误: 段地址必须用段
                          ; 寄存器来表示, 不能用常数

;应当写成
mov ax,1000h
mov ds,ax
mov al,ds:[2000h]
```

  assume cs:code, ds:data -> 将data的段地址转化为ds

  注意:assume不能保证ds和data的实际值相等，也不对ds作出任何赋值，只是告诉编译器遇到data替换成ds。因此，上面程序中还需要有mov ax,data 以及mov ds,ax指令来对ds进行赋值操作

  assume的作用:（原话）**帮助编译器建立段寄存器与段的关联, 当源程序中引用了某个段内的变量时，编译器会在编译出来的机器码中把变量的段地址替换成关联的段寄存器。**

```
;关于assume的作用
data segment
abc db 1,2,3,4
data ends

code segment
assume cs:code, es:data
;同一个段与多个段寄存器有关联时:ds > ss > es > cs
main:
   mov ax, data
   mov es, ax
   mov al, abc[1]; 编译后变成mov al, es:[0001]
   ;先替换成mov al, data:[0001]
   ;再替换成mov al, es:[0001]
   ;在替换时编译器并不检查es与data是否相等
   ;假定前面assume ds:data,则这句话就会替换成
   ;mov al, ds:[0001]并简化成mov al, [0001]
   ;因为[]中没有bp时，默认的段地址一定是ds且ds
   ;可以省略。
   mov ah, 4Ch
   int 21h
code ends
end main
```

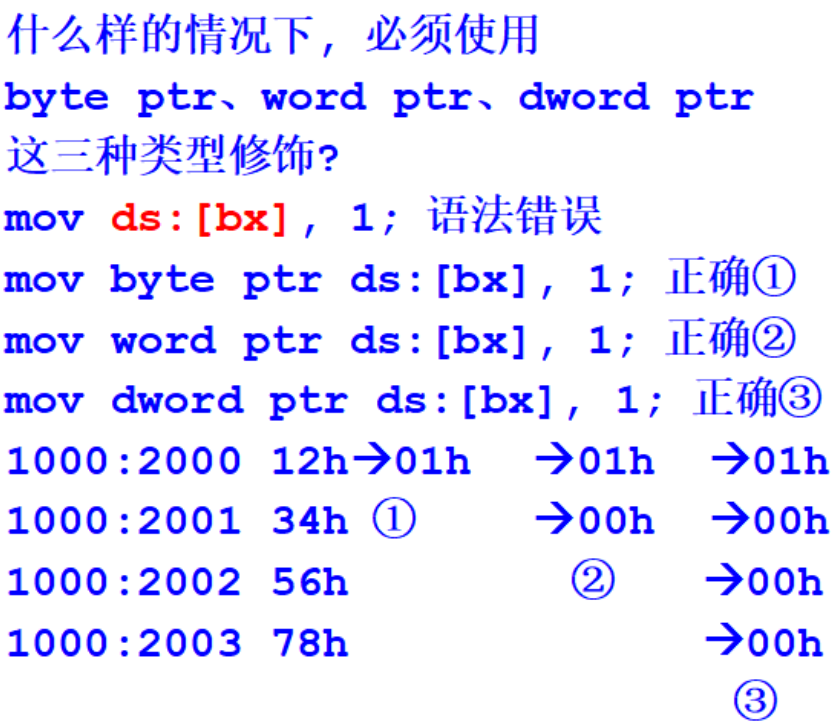
- **寻址**：
  - 直接寻址 -> 用常数表示变量偏移地址

```
mov al,ds:[2000h]
```

  - 间接寻址 -> 用寄存器/寄存器+常数表示变量的偏移地址

```
mov al, ds:[bx]
```

需要注意： 间接寻址所采用的寄存器只能是bx,bp,si,di



总结一下，未定长度的值不允许传入，需要加byte ptr 等限制长度

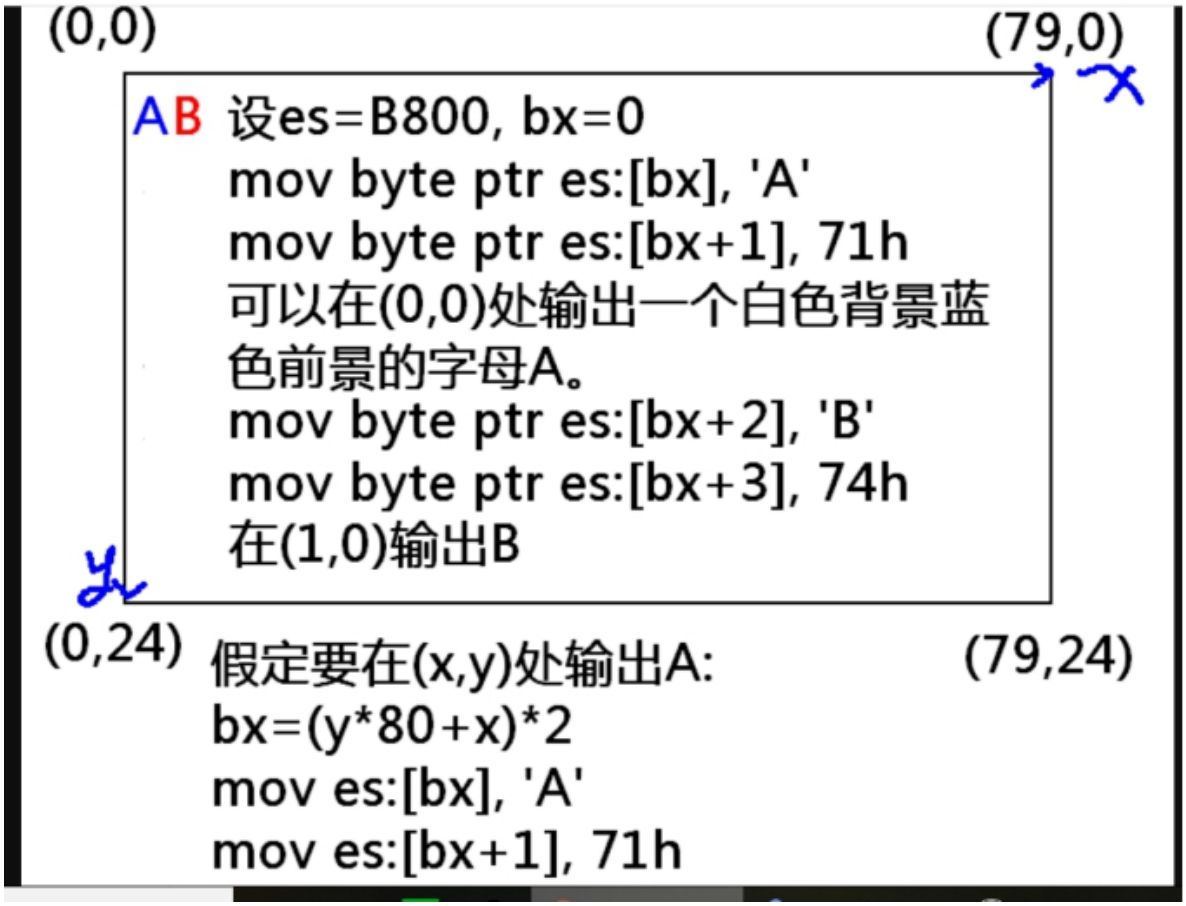
把一个1传入ds:[bx]时，操作系统并不知道要给他分配多大空间才合适，因此报错

间接寻址的形式

```
;形式(1)
mov ah, abc[1]; \编译后变成:
mov ah, [abc+1];/mov ah, ds:[1]; 直接地址
;形式(2)
mov bx, offset abc
mov ah, [bx+1]; 间接地址
;形式(3)
mov bx, 1
mov ah, abc[bx] ;\间接地址
mov ah, [abc+bx];/
;形式(4)
mov bx, offset abc
mov si, 1
mov ah, [bx+si]

;注意：在这里可以有四种变换
 ;①[bx]  [bp]  [si]  [di]就成了最简单的间接寻址方式
 ;使用[ax]  [cx]  [dx]  [sp]语法错误
```

```
    ;②[bx+si] [bx+di] [bp+si] [bp+di]
    ;注意[bx+bp]以及[si+di]是错误的。
    ;③[bx+2] [bp-2] [si+1] [di-1]
    ;④[bx+si+2] [bx+di-2] [bp+si+1]  [bp+di-1]
    ;第4种可用于结构数组中的某一个元素中的成员


    ;两个寄存器相加的间接寻址方式中，bx或bp通常用来表示数组的首地址，而si或di则用来表示下标

   mov ax, xyz[2] ;\编译后变成:
   mov ax, [xyz+2];/mov ax, ds:[6]

   mov ah, abc; 等价于mov ah, abc[0]
   mov ah, [abc]; 效果与上面这句等价
                ;编译后变成mov ah, ds:[0]
   mov ax, xyz
   mov ax, [xyz]; 其完整形式是:
                ; mov ax, word ptr ds:[xyz]
                ; 编译后变成mov ax, ds:[4]


    ;下面介绍32位汇编语言关于间接寻址的灵活性
    ;1.32位汇编语言中多了一种 [寄存器+寄存器*n+常数] 其中n=2、4、8。
    mov eax, [ebx+esi*4+6]
    ;2.32位寻址方式里面，对[]中的两个寄存器几乎不加限制
    ebx, ebp, esi, edi,
    eax, ecx, edx, esp都可以放到[]里面;
    ;包括两个相同的寄存器相加，都是允许的
    mov eax, [ebx+ebx*4]; 两个寄存器可以任意组合
```

- ~~图形模式：我觉得考的意义也不大，可跳过~~



- **端口**： 理解即可

```
CPU<-->端口(port)<-->I/O设备
端口编号就是端口地址。端口地址的范围是:
[0000h, 0FFFFh]，共65536个端口。
对端口操作使用指令in与out实现。

70h及71h端口与cmos内部的时钟有关。

in al, 60h; 从端口60h读取一个字节并存放到AL中
```

- **段地址的隐含** 会考

```
mov ax, [bx]
mov ax, [si]
mov ax, [di+2]
mov ax, [bx+si]
mov ax, [bx+di+2]
mov ax, [1000h]
;上述隐含段地址为ds

mov ax, [bp+2]
;隐含段地址为ss

;总结：偏移地址中只要包含寄存器bp,隐含的段地址一定是ss
;但同时应该注意，默认地址可以人为改变，(类似于强制类型转换)

mov ax, ds:[bp+2]
;这条指令的源操作数段地址从默认的ss改成了ds。

mov ax, ss:[bx+si+2]
```

```
;默认段地址就从ds变成了ss。
```

- **近指针和远指针**

```
16位汇编中，远指针是指16位段地址+16位偏移地址；
32位汇编中，远指针是指16位段地址+32位偏移地址。
48位的远指针在汇编语言中有一个类型修饰词：Fword ptr

近指针(near pointer)：偏移地址就是近指针
16位汇编中，近指针是指16位的偏移地址；
32位汇编中，近指针是指32位的偏移地址；

远指针(far pointer)包括段地址及偏移地址两个部分；
近指针(near pointer)只包括偏移地址，不包含段地址。
```

- # 堆栈传递整理

```
1. 用堆栈传递参数有3种方式：
(1) __cdecl
参数从右到左顺序压入堆栈，由调用者清理堆栈；
是C语言参数传递规范。
(2) __pascal
参数从左到右顺序压入堆栈，由被调用者清理堆栈；
是Pascal语言参数传递规范。
(3) __stdcall
参数从右到左顺序压入堆栈，由被调用者清理堆栈；
是Windows API函数的参数传递规范。


;这里贴一个pascal的例子，其他感觉很类似，不会可以自己去看14课件了
__pascal的例子：
f:
push bp; (4)
mov bp, sp
mov ax, [bp+6]; arg0
; 注意这里一定要push bp,pop bp 是因为要还原bp指针原始状态
;而不使用sp则是因为汇编语言限制sp不能出现在偏移地址[]中
add ax, [bp+4]; arg1
pop bp; (5)
ret 4; (6)
main:
mov ax, 10
push ax; (1) arg0
mov ax, 20
push ax; (2) arg1
call f; (3)
here:
__pascal的堆栈布局：
ss:1FF8 old bp<- bp (4)
ss:1FFA here <- (3)(5)
ss:1FFC 20 <- (2)
ss:1FFE 10 <- (1)
ss:2000    <-(6)
```

- **动态变量！递归！必考**

```
;动态变量这里简单来讲可以看做人为的在调用堆栈
;巧妙的使用了bp-2来挖坑给函数中局部变量
;再通过mov sp,bp的方式还原了堆栈状态
2. 动态变量
int f(int a, int b)
{
   int c; /* c是局部动态变量 */
   c = a+b;
   return c;
}
上述C语言函数可翻译成以下汇编代码:
bp可以看成是一个参照点
f:
push bp; (4)
mov bp, sp
sub sp, 2; (5) 这里挖的坑就是给变量c的
mov ax, [bp+4]
add ax, [bp+6]
mov [bp-2], ax
mov ax, [bp-2]
mov sp, bp; (6)此时变量c死亡
pop bp; (7)
ret; (8)
main:
mov ax, 20
push ax; (1)
mov ax, 10
push ax; (2)
call f; (3)
here:
add sp, 4;(9)此时参数a,b死亡
执行上述代码时, 堆栈布局如下:
ss:1FF6 [30] (5) 变量c
ss:1FF8 old bp<- bp(4)(6)
ss:1FFA here <- (3)(7)
ss:1FFC 10 <- (2)(8)
ss:1FFE 20 <- (1)
ss:2000    <-(9)

;递归递归递归!!!
f:
push bp;(3)(6)(9)
mov bp, sp
mov ax, [bp+4]
cmp ax, 1
je done
dec ax
push ax;(4)(7)
call f;(5)(8)
there:
add sp, 2;(12)(15)
add ax, [bp+4]
done:
pop bp;(10)(13)(16)
ret;(11)(14)(17)
main:
mov ax, 3
```

```
push ax;(1)
call f;(2)
here:
add sp, 2;(18)
```

执行上述代码时的堆栈布局如下：

```
ss:1FEE oldbp<-bp(9)
ss:1FF0 there<-(8)(10)
ss:1FF2 1<-(7)(11)
ss:1FF4 oldbp<-bp(6)12
ss:1FF6 there<-(5)(13)
ss:1FF8 2<-(4)(14)
ss:1FFA oldbp<-bp(3)(15)
ss:1FFC here <-(2)(16)
ss:1FFE 3 <-(1)(17)
ss:2000   <-(18)
```

中断

int 21h 9号功能：输出字符串，字符串以'$'结尾

int 21h 2号功能：输出单个字符 输出DL中的字符

int 21h 4C号功能：汇编程序的结束，al当作返回码

int21h 1号功能： 输入单个字符 AL = getchar();

最后贴一份考纲，查缺补漏！

```
不能带计算器
一、是非题(10个，每题1分，共10分)
二、填空(15个，每空2分，共30分)
三、按要求写出指令(4题，每题5分，共20分)
四、程序填空题(3题，每题10分，共30分)
五、程序阅读(2题，每题5分，共10分)
 进制转换： 例如：16位十六进制数0FFFEh转化为十进制符号数等于_____。-2
十进制数-12转化为8位二进制数等于 1111 0100B
逻辑地址<-->物理地址
例如： 1234h:0058h 转化成物理地址=12340h+0058h=12398h
标志位：CF、ZF、OF、SF、DF、IF、TF

 例如：设AL=00h，则执行指令sub AL，01h后，标志位CF=____。1
 数据在内存中的存放规律：低字节在前，高字节在后。
例如：从地址1000:2000开始顺序存放以下4个字节： 12h，34h，56h，78h。则存放在地址1000:2002
中的字=_____。7856h
寄存器：AX BX CX DX SI DI SP BP CS DS ES SS IP FL
间接寻址： BX BP SI DI
缺省段址：
方括号里有bp就是ss:,否则就是ds:
 mov ax, [bp+2] = mov ax, ss:[bp+2]
mov ax, [bx+si+2]= mov ax, ds:[bx+si+2]
```

```
指令：xchg，push，pop，lea，cbw，cwd，（扩充放大）add，adc，sub，sbb，inc，dec mul，
div，
Xlat
 and，or，xor，not，neg
shl，shr，sal，sar，rol，ror，rcl，rcr cmp，
jxx(条件跳转指令)：ja，jb ，jae，jbe，jg jl jge jle jc jnc je jne jz jnz jcxz js
jns jo jno
loop
clc stc cli sti cld std
call，ret(近调用和近返回)，int，iret 用堆栈传递参数时，如何用[bp+?]实现对参数的引用？ mov
jmp
Call 和jmp区别
字符串指令：repne scasb，repe scasb，rep movsb，lodsb，stosb，rep stosb

编程题涉及的中断调用：  mov ah,1 int 21h  键盘输入
mov ah，2 mov dl，'A' int 21h
mov ah，9 mov dx，offset sth int 21h
mov ah,4Ch int 21h 结束
```

考试顺利！

/*

author : Dreamerryao

date : 2020/1/14

*/