基础知识

汇编语言的产生

> 每一种CPU都有自己的汇编指令集

- 汇编语言的**主体**是汇编指令。(汇编语言的核心是汇编指令，它决定了汇编语言的特性。)
- 汇编指令和机器指令的差别在于指令的表示方法上。汇编指令是机器指令便于记忆的书写格式。
- 汇编指令是机器指令的助记符。

例：*机器指令：1000100111011000*

操作：寄存器BX的内容送到AX中

汇编指令：MOV AX,BX

指令和数据

> 指令和数据是应用上的概念。

在内存或磁盘上，指令和数据没有任何区别，都是二进制信息。

1000100111011000 —> 89D8H （数据）

1000100111011000 —> MOV AX,BX （程序）

存储器和存储单元

CPU可以直接使用的信息在存储器中存放。

- 存储单元从零开始顺序编号。
- 一个存储单元可以存储 8 个 bit （用作单位写成“b”），即 8 位二进制数。
  1B = 8b　　1KB = 1024B　　1MB = 1024KB　　1GB = 1024MB

总线

> 专门有连接CPU和其他芯片的导线，通常称为总线。

逻辑上划分为：

- **地址总线**：决定CPU的寻址能力
- **数据总线**：决定CPU与其它器件进行数据传送时的一次数据传送量
- **控制总线**：决定CPU对系统中其它器件的控制能力

地址总线

定义

- CPU是通过地址总线来指定存储单元的。
- 地址总线上能传送多少个不同的信息，CPU就可以对多少个存储单元进行寻址。

特性

- 由 CPU 输出地址信息到地址总线上
- 单向传输

说明

- 地址总线的位宽决定了 CPU 的寻址能力
- 在电子计算机中，一根导线可以传送的稳定状态只有两种，高电平或者低电平。用二进制表示就是 1 或 0，即一个导线能传输一位二进制数。那么 10 根地址线就能传输 10 位二进制数。那么就能表示 2^10 种二进制数据（十进制表示就是 0-1023），即能表示 2^10 个地址
- 若一个 CPU 有 N 根地址线，则可以说这个 CPU 的地址总线的位宽为 N 。这样的 CPU 最多可以寻找 2 的 N 次方个地址

8086有20根地址总线，寻址能力达到1M

数据总线

定义

- CPU与内存或其它器件之间的数据传送是通过数据总线来进行的。
- 数据总线的宽度决定了CPU和外界的数据传送速度。

特性

- 是双向传输的
- 其位宽与机器字长、存储字长有关，一般为 8 位、16 位或 32 位

说明

- 数据总线的位宽决定了 CPU 和外界的数据传输速度
- 多少位宽的数据总线就能传输多少个二进制数

8086有16根数据总线。

控制总线

定义

CPU对外部器件的控制是通过控制总线来进行的。在这里控制总线是个总称，控制总线是一些不同控制线的集合。

有多少根控制总线，就意味着CPU提供了对外部器件的多少种控制。因此，控制总线的宽度决定了CPU对外部器件的控制能力。

寄存器

> 8086CPU所有的寄存器都是16位的，可以存放两个字节。

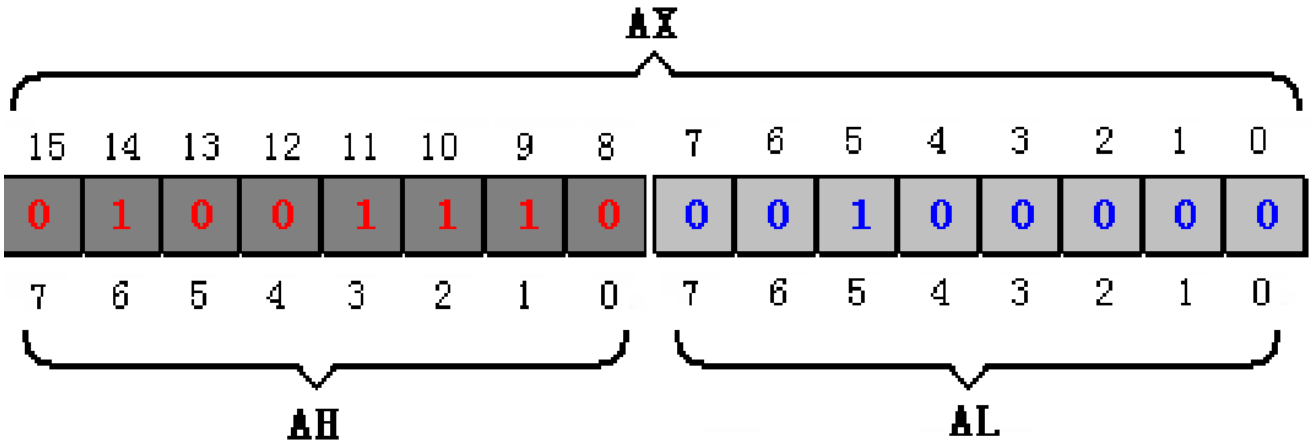
- 数据寄存器（通用寄存器）： **AX BX CX DX**
- 地址寄存器： **SI DI SP BP**
- 段寄存器： **CS DS ES SS**
- 控制寄存器： **IP PSW(FLAGS)**

通用寄存器（AX、BX、CX、DX）

为保证兼容性（8086上一代CPU中的寄存器都是8位的），这四个寄存器都可以分为两个独立的8位寄存器使用。

例：AX 可分为 AH 和 AL

(如果对al, bl, cl等8位寄存器进行指令操作时产生了进位，该进位并不会保存到ah, bh, ch这些高位寄存器中)



寄存器	寄存器中的数据	所表示的值
AX	0100111000100000	20000（4E20H）
AH	01001110	78（4EH）
AL	00100000	32（20H）

段寄存器（CS、DS、SS、ES）

> 段寄存器就是提供段地址的。

特性及寻址

8086CPU有4个段寄存器：

CS、DS、SS、ES

当8086CPU要访问内存时，由这4个段寄存器提供内存单元的段地址。

8086CPU是16位结构的CPU，具有一下几方面的结构特性：

- 运算器一次最多可以处理16位的数据
- 寄存器的最大宽度为16位

- 寄存器和运算器之间的通路为16位

但**8086有20位地址总线**，可以传送20位地址，达到1MB的寻址能力。8086CPU在内部又只能一次性处理、传输、暂时存储16位地址，实际上寻址时CPU中的相关部件会提供两个16位的地址，通过**地址加法器**合成为20位的物理地址：

物理地址 = 段地址*16+偏移地址

段地址*16也被成为基地址，因此物理地址 = 基地址 + 偏移地址

每段最大64K字节，最少16个字节

- 为什么段最小为16byte?

  分段的依据是段地址，即不同的段的段地址是不同的。 当偏移地址相同时，两个相邻的段的最短距离为16个byte。

- 为什么段最大为64byte?

  一个段的最大空间决定于一个段的偏移地址：一个4位的16进制数，也就是一共可以表示2^16= 65536 个byte= 64byte。

段的类型

8086汇编语言中把逻辑段分为四种类型，分别是代码段、数据段、附加段和堆栈段。

段名	段寄存器	偏移地址
代码段	CS	IP
数据段	DS	BX、SI、DI等地址寄存器
附加段	ES	BX、SI、DI等地址寄存器
栈段	SS	SP

CS和IP

> **CPU将CS:IP指向的内存单元中的内容看作指令。**

通过改变CS、IP中的内容来控制CPU执行目标指令。

8086PC工作过程：（重点）

IP先转移再执行

（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 = IP + 所读取指令的长度，从而指向下一条指令；

（3）执行指令。 转到步骤 （1），重复这个过程。

jmp:修改CS,IP的指令

- 同时修改CS、IP的内容： 用指令中给出的段地址修改CS，偏移地址修改IP。

```
jmp 段地址：偏移地址
    jmp 2AE3:3
    jmp 3:0B16
```

- 仅修改IP的内容： 用寄存器中的值修改IP

```
jmp 某一合法寄存器
        jmp ax    (类似于 mov IP,ax)
        jmp bx
```

8086对字的存储方式

> 8086的一个内存单元为8位，1字节。对于一个字，8086采用小端模式（little-endian）在内存中存储，即**一个字的高字节放到高地址，低字节放到低地址。**

- 16位的字存储在一个16位的寄存器中，如何存储?

  高8位放高字节，低8位放低字节

- 16位的字在内存中需要2个连续字节存储，怎么存放?

  低位字节存在低地址单元，高位字节存在高地址单元

例：20000D(4E20H)存放0、1两个单元，18D(0012H)存放在2、3两个单元

任何两个地址连续的内存单元，N号单元和 N+1号单元，可以将它们看成两个内存单元 ，也可以看成一个地址为N的字单元中的高位字节单元和低位字节单元。

字单元：由两个地址连续的内存单元组成，存放一个字型数据（16位）

原理：在一个字单元中，低地址单元存放低位字节，高地址单元存放高位字节。

DS和[address]

1. CPU要读取一个内存单元的时候，必须先给出这个内存单元的地址；
2. 在8086PC中，内存地址由段地址和偏移地址组成。
3. 8086CPU中有一个 **DS寄存器**，通常用来存放要访问的数据的段地址。

读取10000H单元的内容可以用如下程序段进行:

(将10000H(1000:0)中的数据读到al中。)

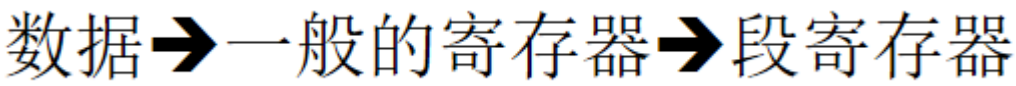
```
mov bx,1000H
mov ds,bx
mov al,[0]
```

段地址：执行指令时，8086CPU自动取DS中的数据为内存单元的段地址。

偏移地址：mov指令中的[]说明操作对象是一个内存单元，[]中的0说明这个内存单元的偏移地址是0

思考：为什么不直接mov ds,1000H?

8086CPU不支持将数据直接送入段寄存器的操作，ds是一个段寄存器。



数据段

我们可以将一组长度为N（N≤64K）、地址连续、**起始地址为16的倍数**的内存单元当作专门存储数据的内存空间，从而定义了一个数据段。

比如我们用123B0H~123B9H这段空间来存放数据：

段地址：123BH 长度：10字节

栈

> Last In First Out，后进先出

栈的格式：

- 栈顶：低地址
- 栈底：高地址

SS:SP

> **任意时刻，SS:SP指向栈顶元素。**

- 段寄存器SS　存放栈顶的段地址
- 寄存器SP　存放栈顶的偏移地址

找栈顶的起始位置：

我们将10000H~1000FH 这段空间当作栈段，SS=1000H，栈空间大小为16 字节 ，栈最底部的字单元地址为1000:000E。

当栈为空的时候，栈中没有元素，也就不存在栈顶元素，所以SS:SP 只能指向栈的最底部单元下面的单元，该单元的偏移地址为栈最底部的字单元的偏移地址+2，栈最底部字单元的地址为1000:000E，所以栈空时，SP=0010H

push&pop

> 以字为单位操作，因此转移长度为2，注意push和pop的操作顺序

push ax

1. SP=SP-2，栈顶指针空出一个字单位
2. 将AX的内容放到SS:SP指向的单元（一个字）中，SS:SP重新指向栈顶

pop ax

1. 将SS:SP指向的单元（一个字）内容送到AX中
2. SP=SP+2，栈顶指针向下移动一个字单位

栈顶超界的问题

问题情景：栈满的时候再使用push指令入栈，栈空的时候再使用pop指令出栈

栈空间之外的空间里可能存放了具有其他用途的数据代码，超界时很可能将其改写，引发不少错误

原因：8086CPU的工作机理，只考虑当前的情况：

- 当前栈顶在何处；
- 当前要执行的指令是哪一条。

我们要根据可能用到的最大栈空

间来安排栈的大小

栈段

我们将10000H~1FFFFH这段空间当作栈段，SS=1000H，栈空间大小为64KB，栈最底部的字单元地址为1000:FFFE。

任意时刻，SS:SP指向栈顶，当栈中只有一个元素的时候，SS=1000H，SP=FFFEH。

栈为空，就相当于栈中唯一的元素出栈，出栈后，SP=SP+2。SP原来为FFFEH，加2后SP=0，所以，当栈为空的时候，SS=1000H，SP=0

一个栈段的容量最大为64KB：0~FFFFH（2的16次方）

程序的基本结构

汇编程序从写出到执行的过程：

编程 -> 1.asm -> 编译 -> 1.obj -> 连接 -> 1.exe -> 加载 -> 内存中的程序 -> 运行

程序返回：

```
MOV AX, 4C00H
INT 21H
```

这是由CPU执行的中断指令，可以让程序结束返回到DOS系统

寻址

描述性符号“()”

“() ”来表示一个寄存器或一个内存单元中的内容。

比如：

（1）ax中的内容为0010H，我们可以这样来描述：(ax)=0010H；

（2）2000:1000 处的内容为0010H，我们可以这样来描述：(21000H)=0010H；

（3）对于mov ax,[2]的功能，我们可以这样来描述：(ax)=((ds)*16+2)；

（4）对于mov [2],ax 的功能，我们可以这样来描述：((ds)*16+2)=(ax)；

约定符号idata表示常量

比如：

- mov ax,[idata]就代表mov ax,[1]、mov ax,[2]、mov ax,[3]等。
- mov bx,idata就代表mov bx,1、mov bx,2、mov bx,3等。
- mov ds,idata就代表mov ds,1、mov ds,2等，它们都是非法指令。

寄存器间接寻址[bx]

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))；

loop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0则转至标号处执行程序, 如果为0则向下执行

要求：

(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

一段安全的空间

在一般的PC机中，DOS方式下，DOS和其他合法的程序一般都不会使用0:200~0:2FF（0:200h~0:2FFh）的256 个字节的空间。所以，我们使用这段空间是安全的。

and和or指令

and 指令：逻辑与指令，按位进行与运算。(置0)

```
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。

or 指令：逻辑或指令，按位进行或运算。(置1)

```
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。

大小写转换的问题

一个字母，我们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。

寄存器相对寻址

> [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）。

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
```

利用[bx（si或di）+idata]的方式，来使程序变得简洁。

```
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
```

基址变址寻址

[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）。

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

与寻址有关的寄存器：bx,si,di,bp

基址变址寻址方式的4种组合： bx+si,bx+di,bp+si,bp+di

相对基址变址寻址

mov ax,[bx+si+idata]: (ax)=((ds)*16+(bx)+(si)+idata)

寻址方式小结:

几种定位内存地址的方法（可称为寻址方式）：

（1）[iata] 用一个常量来表示地址，可用于直接定位一个内存单元；(**直接寻址**)

（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；(**寄存器间接寻址**)

（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；(**寄存器相对寻址**)

（4）[bx+si]用两个变量表示地址；(**基址变址寻址**)

（5）[bx+si+idata] 用两个变量和一个常量表示地址。(**相对基址变址寻址**)

二重循环处理方式

将四个字符串转换为大写

用多个寄存器实现：

```
      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
```

用栈实现：

```
start: mov ax, data
    mov ds, ax
    mov ax, stack
    mov ss, ax
    mov sp, 8;定义栈顶指针

    mov cx, 4;外循环四次
    mov si, 0;表示处理第几列
    s: push cx;保存外层循环的值
```

```
    mov bx, 0
    mov cx, 3;内循环三次
    s0:
    mov al, ds:[bx+si]
    and al, 11011111b;转换为大写字母，将第5位置0
    mov ds:[bx+si], al
    add bx, 6
    loop s0
    inc si
    pop cx
    loop s
```

标志寄存器

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

注: 8086CPU的指令集中, 有的指令的执行是影响标志寄存器的, 比如

ADD/SUB/MUL/DIV/INC/OR/ADD等

有的指令对标志寄存器没有影响, 比如MOV/PUSH/POP, 它们大都是传送指令

ZF (Zero Flag)

flag的第6位是ZF，零标志位。

它记录相关指令执行后，

- 结果为0 ， ZF = 1
- 结果不为0， ZF = 0

PF (Parity Flag)

flag的第2位是PF，奇偶标志位。

它记录指令执行后，结果的所有二进制位中1的个数：

- 为偶数，PF = 1；
- 为奇数，PF = 0。

SF (Sign Flag)

flag的第7位是SF，符号标志位。

它记录指令执行后，

- 结果为负，SF = 1；
- 结果为正，SF = 0。

我们将数据当作有符号数来运算的时候, 可以通过它来得到结果的正负, 如果当作无符号数来算, SF的值则没有意义

CF (Carry Flag)

flag的第0位是CF, 进位标志位, 一般情况下, 在进行**无符号数运算**的时候, 它记录了运算结果的最高位有效位向更高位的进位值, 或从更高位的借位值

INC和LOOP不影响CF位

OF (Overflow Flag)

溢出:

在进行**有符号数运算**的时候, 如结果超出了机器所能表示的范围称为溢出

对于8位的有符号数据, 机器所能表示的范围就是-128 ~ 127

对于16位的有符号数据, 机器所能表示的范围是-32768 ~ 32767

flag的第11位是OF, 溢出标志位, 一般情况下, OF记录了有符号数运算的结果是否发生了溢出

如果发生溢出, OF = 1

如果没有, OF = 0

DF (Direction Flag)

flag的第10位是DF, **方向标志位**, 在串处理指令中, 控制每次操作后SI, DI的增减

DF = 0, 每次操作后SI, DI递增

DF = 1, 每次操作后SI, DI递减

pushf和popf

- pushf： 将标志寄存器的值压栈；
- popf： 从栈中弹出数据，送入标志寄存器中。

pushf和popf，为直接访问标志寄存器提供了一种方法。

adc指令（OF）

带进位加法指令，下面的指令和add ax , bx具有相同的结果：

- add al,bl
- adc ah,bh(可接收进位信息)

sbb指令（CF）

sbb ax,bx ， 带错位减法指令，它利用了CF位上记录的借位值。

实现功能： (ax) = (ax) – (bx) – CF

cmp指令

CMP是比较指令, 功能相当于减法指令, 只是不保存结果, 但对标志寄存器产生影响

CMP 操作对象1, 操作对象2(操作对象1只能为寄存器/地址访存不能为立即数, 操作对象2可以为立即数/寄存器/地址访存)

对无符号数进行比较时:

- ZF = 1, 说明 1 = 2
- ZF = 0, 说明 1 != 2
- CF = 1, 说明1 < 2
- CF = 0, 说明 1 >= 2
- CF = 0 且 ZF = 0, 说明 1 > 2
- CF = 1 或 ZF = 1, 说明 1 <= 2

对有符号数进行比较时:

由于存在溢出的情况, 仅根据符号标志位SF无法判断1 和 2的相对大小, 因此还要根据OF加以判断

- ZF = 1, 说明 1 = 2
- ZF = 0, 说明 1 != 2
- SF = 1 && OF = 0, 说明 1 < 2
- SF = 0 && OF = 0, 说明 1 >= 2
- SF = 1 && OF = 1, 说明 1 > 2 (有溢出便相反)
- SF = 0 && OF = 1, 说明 1 < 2

串传送指令

movsb(以字节为单位传送)

功能：将 **ds:si** 指向的内存单元中的字节送入 **es:di**中，然后根据标志寄存器DF位的值，将 si和di递增或递减。

(1) ((es)×16 + (di)) = ((ds) ×16 + (si))

(2) 如果DF = 0则： (si) = (si) + 1

(di) = (di) + 1

(3)如果DF = 0则： (si) = (si) - 1

(di) = (di) - 1

使用串传送指令进行数据的传送，需要给它提供一些必要的信息，它们是：

① 传送的原始位置：ds:si；

② 传送的目的位置：es:di；

③ 传送的长度：cx；

④ 传送的方向：DF。

movsw(以字为单位传送)

将 ds:si指向的内存字单元中word送入es:di中，然后根据标志寄存器DF位的值，将si和di递增**2**或递减**2**。

CLD && STD

设置DF标志位的命令:

- CLD 设置DF为0, 正向传送
- STD 设置DF为1, 逆向传送

REP

MOVSB 和 MOVSW 进行的是串传送操作中的一个步骤(即单个指令仅传送一个指令)

通常MOVSB和MOVSW都和REP配合使用, 格式:

`REP MOVSB`

REP的作用是根据CX的值, 重复执行后面的串传送指令, 汇编语言描述如下:

```
s: movsb
   loop s
```

例：保存中断程序

```
;保存中断程序
mov ax,cs
mov ds,ax
mov si,offset do0
mov ax,0
mov es,ax
mov di,200h
mov cx,offset do0end-offset do0
cld
rep movsb
```

转移指令

offset

> 操作符offset在汇编语言中是由**编译器**处理的符号，它的功能是取得标号的**偏移地址**。

```
assume cs:codesg
      codeseg segment
        start:mov ax,offset start ; 相当于 mov ax,0
              s:mov ax,offset s      ; 相当于mov ax,3
      codesg ends
      end start
```

jmp

> jmp为无条件转移，可以只修改IP，也可以同时修改CS和IP；

JMP 段地址：偏移地址； 同时修改CS、IP(仅debug中使用)

JMP 寄存器 :用寄存器中的值仅修改IP（要先MOV到寄存器中）

JMP指令要给出两种信息：

转移的目的地址

转移的距离（段间转移、段内短转移、段内近转移）

段内短转移：

`JMP short 标号（转到标号处执行指令）`

对应的机器码中包含转移的位移, 而不是目的地址

功能为：

(IP) = (IP) + 8位位移

1. 8位位移 = "标号"处的地址 - JMP指令后的第一个字节的地址
2. short指明此处的位移为8位位移
3. 8位位移的范围为-128 ~ 127, 用补码表示
4. 8位位移由编译程序在编译时算出

段内近转移

`JMP near ptr 标号（转到标号处执行指令）`

对应的机器码中包含转移的位移, 而不是目的地址

功能为：

(IP) = (IP) + 16位位移

1. 16位位移 = "标号"处的地址 - JMP指令后的第一个字节的地址
2. near ptr指明此处的位移为16位位移
3. 8位位移的范围为-32769 ~ 32768, 用补码表示
4. 16位位移由编译程序在编译时算出

段间转移

`JMP far ptr 标号`

实现的是段间转移, 又称为远转移:

功能: far ptr指明了指令用标号的段地址和偏移地址修改CS和IP

转移地址在内存中的JMP指令

- JMP word ptr 内存单元地址(段内转移)

```
mov ax,0123H
mov ds:[0],ax
jmp word ptr ds:[0]
执行后，(IP)=0123H
```

- JMP dword ptr 内存单元地址(段间转移)

  内存单元地址处开始存放着两个字，高地址处的字是转移的目的段地址，低地址处是转移的目的偏移地址。

(CS)=(内存单元地址+2)

(IP)=(内存单元地址)

条件转移指令

jcxz

jcxz指令为有条件转移指令，所有的有条件转移指令都是短转移，在对应的机器码中包含转移的位移，而不是目的地址。对IP的修改范围都为-128~127。

指令格式：`jcxz 标号`（如果(cx)=0，则转移到标号处执行。）

用来判断以0结尾的字符串的妙用

```
;子程序功能 将字符串转换为大写
;参数寄存器 si 转换字符串的起始地址
to_upper:
    push cx
    push si
toUpper:
    mov cx,0
    mov cl,ds:[si]
    jcxz upperRet
    and byte ptr ds:[si],11011111B
    inc si
    jmp toUpper
upperRet:
    pop si
    pop cx
    ret
```

根据CMP指令的结果（标志寄存器状态）进行转移的有条件转移指令

因为CMP指令可以进行两种比较，无符号数比较和有符号数比较，所以根据CMP指令的比较结果进行转移的指令也分为两种：

- 根据无符号数的比较结果进行转移的条件转移指令（检测ZF、CF的值）
- 根据有符号数的比较结果进行转移的条件转移指令（检测SF、OF和ZF的值）

无符号数比较的条件转移指令如下：

指令	含义	检测的相关标志位
JE (equal)	等于即转移	ZF=1
JNE (not equal)	不等于即转移	ZF=0
JB (below)	低于即转移	CF=1
JNB (not below)	不低于即转移	CF=0
JA (above)	高于即转移	CF=0 且 ZF=0
JNA (not above)	不高于即转移	CF=1 或 ZF=1

loop

loop 标号 指令操作：

（1）(cx)=(cx)-1;

（2）如果(cx)≠0，(IP)=(IP)+8位位移。.

- - 8位位移="标号"处的地址-loop指令后的第一个字节的地址；
  - 8位位移的范围为-128~127，用补码表示；
- - 8位位移由编译程序在编译时算出。

  当(cx)=0，什么也不做（程序向下执行）

ret 和 retf

- **ret:**用栈中的数据，修改IP的内容，从而实现近转移；

  （1）(IP)=((ss)*16+(sp))

  （2）(sp)=(sp)+2
- **retf:**用栈中的数据，修改CS和IP的内容，从而实现远转移；

  （1）(IP)=((ss)*16+(sp))

  （2）(sp)=(sp)+2

  （3）(CS)=((ss)*16+(sp))

  （4）(sp)=(sp)+2

call

CPU执行call指令，进行两步操作：

（1）将当前的 IP 或 CS和IP 压入栈中；

（2）转移。

call 指令不能实现短转移，除此之外，call指令实现转移的方法和 jmp 指令的原理相同。

依据位移进行转移的call指令

call 标号（将当前的 IP 压栈后，转到标号处执行指令）

CPU执行此种格式的call指令时，进行如下的操作：

1. **(sp) = (sp) – 2**

   ((ss)*16+(sp)) = (IP)

2. **(IP) = (IP) + 16位位移**

16位位移=“标号”处的地址 - call指令后的第一个字节的地址；

CPU 执行指令“call 标号”时，相当于进行：

```
push IP
jmp near ptr 标号
```

转移的目的地址在指令中的call指令

call far ptr 标号

CPU 执行指令 “call far ptr 标号” 时，相当于进行：

```
push CS
push IP
jmp far ptr  标号
```

转移地址在寄存器中的call指令

- CALL WORD PTR 内存单元地址,相当于执行:
  - PUSH IP
  - JMP WORD PTR 内存单元地址
- CALL DWORD PTR 内存单元地址,相当于执行:
  - PUSH CS
  - PUSH IP
  - JMP DWORD PTR 内存单元地址

call 和 ret 的配合使用

call指令转去执行子程序之前，call指令后面的指令的地址将存储在栈中，所以可以在子程序的后面使用 ret 指令，用栈中的数据设置IP的值，从而转到 call 指令后面的代码处继续执行

- call：将后序指令地址压栈
- ret：将保存的指令地址出栈

div

除数: 8位或16位, 放在寄存器或者内存单元中

被除数: (默认)放在AX(8位除数对应16位被除数)或DX(高16位)和AX(低16位)中(16位除数对应32位被除数)

结果：

除数	8位	16位
商	AL	AX
余数	AH	DX

- **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]**

  (高十六位）+(低十六位)

  含义：

  (ax)=[(dx)*10000H+(ax)]/((es)*16+0) 的商

  (dx)=[(dx)*10000H+(ax)]/((es)*16+0) 的余数

mul

格式：

```
mul reg
mul 内存单元
```

- **mul byte ptr ds:[0]**

  含义为： (ax)=**(al)**x((ds)x16+0)；

- **mul word ptr [bx+si+8]**

  含义为：

  (ax)=(ax)x((ds)x16+(bx)+(si)+8)结果的低16位；

  (dx)=(ax)x((ds)x16+(bx)+(si)+8)结果的高16位；例

例：

（1）计算100*10000

100小于255，可10000大于255，所以必须做16位乘法，程序如下：

```
mov ax,100
mov bx,10000
mul bx
```

结果：(ax)=4240H，(dx)=000FH

（F4240H=1000000）

dup

dup是一个操作符，在汇编语言中同db、dw、dd 等一样，也是由编译器识别处理的符号。

它是和db、dw、dd 等数据定义伪指令配合使用的，用来进行数据的重复。

- 例1：

  db 3 dup (0)

  定义了3个字节，它们的值都是0，

  相当于 db 0,0,0
- 例2：

  db 3 dup (0,1,2)

  定义了9个字节，它们是 0、1、2、0、1、2、0、1、2，

  相当于 db 0,1,2,0,1,2,0,1,2
- 例3：

  db 3 dup (‘abc’,’ABC’)

  定义了18个字节，它们是 ‘abcABCabcABCabcABC’，

  相当于db ‘abcABCabcABCabcABC’
- 定义一个容量为 200 个字节的栈段:

  stack segment

  db 200 dup (0)

  stack ends

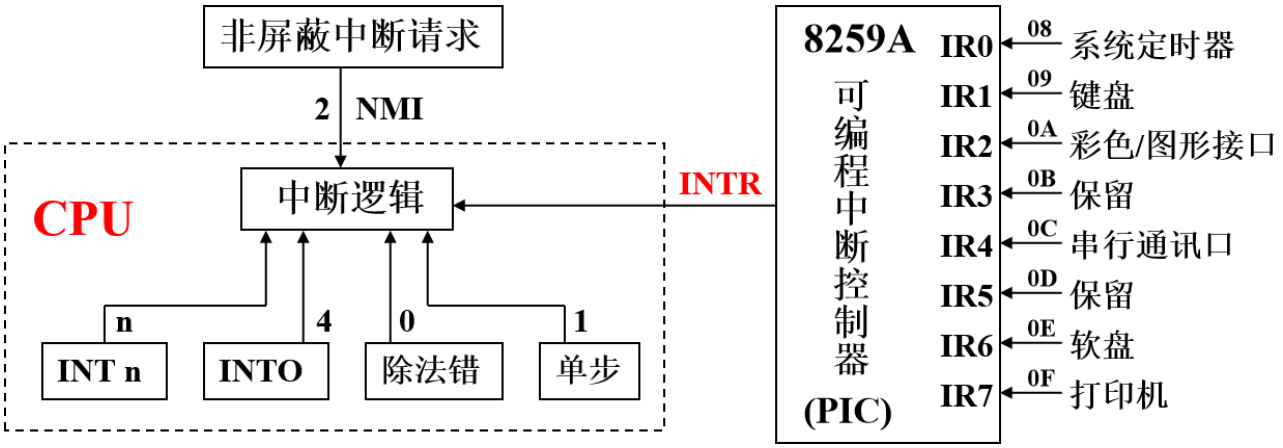
内中断

中断源分为软件中断和硬件中断，软件中断又称为内中断，硬件中断称为外部中断。

软件中断：由CPU内部的某些事件引起的，不受中断允许标志IF的控制。包括以下情况：

- 除法错误, 比如执行DIV指令产生的除法溢出
- 单步执行
- 执行into指令
- 执行INT指令

硬件中断：由输入输出外设产生的中断请求引起的中断。80X86系统的硬件中断分为可屏蔽中断和不可屏蔽中断两大类。所有的中断请求都有对应的中断处理子程序与之对应。



中断处理程序

中断信息中包含有标识中断源的类型码, **中断类型码**的作用就是用来定位中断处理程序

中断向量表

中断处理程序入口地址的列表

中断向量表在内存中保存, 其中存放着**256个**中断源所对应的中断处理程序的入口

对于8086CPU, 中断向量表指定放在内存地址0处. **从内存0000:0000 到 0000:03FF**的1024个单元中存放着中断向量表, 8086CPU用8位的中断类型码通过中断向量表找到对应的中断处理程序的入口地址

1个中断源 = 4个内存单元 = IP(低地址) + CS(高地址)

从内存地址为中断类型码*4 和中断类型码 *4+2 的两个字单元中读取中断处理程序的入口地址设置IP和CS。

256个中断源分布如下:

- 5个专用中断
  - 除法出错(00000H)
  - 单步终端(00004H)
  - NMI00008H)
  - 断点中断(0000CH)
  - 溢出中断(00010H)
- 系统保留中断(27个)

  类型5 ~ 类型31 (00014H ~ 0007FH)
- 用户自定义中断(224个)

  类型32 ~ 类型255 (00080H ~ 003FCH)

中断过程

8086CPU中断过程:

（1）取得中断类型码；

（2）标志寄存器的值入栈

（3）设置标志寄存器的第8位TF 和第9位IF的值为0

（4）CS的内容入栈；

（5）IP的内容入栈；

（6）从内存地址为中断类型码*4 和中断类型码 *4+2 的两个字单元中读取中断处理程序的入口地址设置IP和CS。

编程处理中断

（1）保存用到的寄存器。

（2）处理中断。

（3）恢复用到的寄存器。

（4）用 iret 指令返回。

由于中断处理程序在任意时刻都有可能被调用, 因此中断处理程序应该放在DOS系统和其他应用程序都 不会随便使用的空间中, 例如可以利用中断向量表中的空闲单元**(0000:0200 ~ 0000:02FF这256个字节)**

int指令

int格式： int n，n为中断类型码。它的功能是引发中断过程。

CPU 执行int n指令，相当于引发一个 n号中断的中断过程，执行过程如下：

（1）取中断类型码n；

（2）标志寄存器入栈，IF = 0，TF = 0；

（3）CS、IP入栈；

（4）(IP) = (nx4)，(CS) = (nx4+2)。从此处转去执行n号中断的中断处理程序。

单步中断

CPU在执行完一条指令之后, 如果检测到标志寄存器TF位为1, 则产生单步中断, 引发中断过程, 单步中断 的类型码为1, CPU在执行中断过程时, 有TF=0这个步骤, 就是为了防止在处理中断程序的时候发生单步中断

响应中断的特殊情况

- 在执行完向 ss寄存器传送数据的指令后，即便是发生中断，CPU 也不会响应。这样做的主要原因是，ss:sp联合指向栈顶，而对它们的设置应该连续完成。 所以CPU在执行完设置ss的指令后，不响应中断。 这给连续设置 ss和sp，指向正确的栈顶提供了一个时机。 即，我们应该利用这个特性，将设置ss和sp的指令连续存放，使得设置sp的指令紧接着设置ss的指令执行，而在此之间，CPU不会引发中断过程。
- IRET指令是中断子程序返回指令，它也要求再执行一条后续指令后才能响应中断。这样做的目的是保护系统能够正常运行；

- 当执行到**STI指令时，CPU不会马上响应中断。STI指令是开中断指令，要求在开放中断后再执行后续的一条指令后才能响应中断；

overflow程序

```
assume cs:code

code segment
start:
;保存中断程序
mov ax,cs
mov ds,ax
mov si,offset do0
mov ax,0
mov es,ax
mov di,200h
mov cx,offset do0end-offset do0
cld
rep movsb

;设置中断向量表
mov ax,0
mov es,ax
mov word ptr es:[0*4],200h
mov word ptr es:[0*4+2],0

;触发除法错误
call divide_overflow


mov ax,4c00h
int 21h

divide_overflow:
    mov ax,0ffffh
    mov bl,1h
    div bl
    ret

do0:
jmp short do0start
db"overflow!"

do0start:
    mov ax,cs
    mov ds,ax
    mov si,202h;设置ds:si指向字符串

    mov ax,0b800h
    mov es,ax
    mov di,12*160+36*2;设置es：di指向中间显存位置；
```

```
    mov cx,9
  s:mov al,[si]
    mov es:[di],al
    inc si
    add di,2
    loop s
mov ax,4c00h
int 21h

do0end:nop
code ends
end start
```

进制转换显示到屏幕

```
assume cs:codesg,ds:datasg
datasg segment
  BUF db 100
datasg ends

codesg segment    ;将BUF里的数转换成16进制，输出。
start:
mov ax,datasg
mov ds,ax
mov bx,offset BUF
mov ax,ds:[bx]

;ax除16取余数，进栈

  mov bh,16
  mov di,0 ;统计余数个数
s1:
   mov ah,0
   div bh    ;al中存放商，ah存放余数，注意范围<256，要扩大用dx+ax，程序需要稍作改动。
   mov dh,0
   mov dl,ah
   push dx

   inc di
   cmp al,0
   jz showno

 jmp s1

showno:
   mov dl,0ah
   mov ah,02h
   int 21h       ;回车
   mov dl,0dh
   mov ah,02h
   int 21h       ;换行
```

```
                ;回车换行以区别输入输出

showno1:;顺序执行
    pop dx
    dec di

   cmp dl,9
   ja s2
   add dl,30h
   jmp tt
 s2:
   add dl,37h          ;处理>9的余数

tt:
   mov ah,02h
   int 21h           ;显示出栈的余数字符

   cmp di,0
   jnz showno1

 MOV AX, 4C00H
 INT 21H

codesg ends
end start
```

小结

（1）汇编指令是机器指令的助记符，同机器指令一一对应。

（2）每一种CPU都有自己的汇编指令集。

（3）CPU可以直接使用的信息在存储器中存放。

（4）在存储器中指令和数据没有任何区别，都是二进制信息。

（5）存储单元从零开始顺序编号。

（6）一个存储单元可以存储 8 个 bit （用作单位写成“b”），即 8 位二进制数。

（7）1B = 8b 1KB = 1024B 1MB = 1024KB 1GB = 1024MB

（8）每一个CPU芯片都有许多管脚，这些管脚和总线相连。也可以说，这些管脚引出总线。一个CPU可以引出三种总线的宽度标志了这个CPU的不同方面的性能：地址总线的宽度决定了CPU的寻址能力；数据总线的宽度决定了CPU与其它器件进行数据传送时的一次数据传送量；控制总线宽度决定了CPU对系统中其它器件的控制能力。

1、段地址在8086CPU的寄存器中存放。当8086CPU要访问内存时，由段寄存器提供内存单元的段地址。8086CPU有4个段寄存器，其中CS用来存放指令的段地址。

2、CS存放指令的段地址，IP存放指令的偏移地址。 8086机中，任意时刻，CPU将CS:IP指向的内容当作指令执行。

3、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。）

4、8086CPU提供转移指令修改CS、IP的内容。

（1）观察下面的地址，读者有什么发现?

物理地址	段地址	偏移地址
21F60H	2000H	1F60H
	2100H	0F60H
	21F0H	0060H
	21F6H	0000H
	1F00H	2F60H

结论：CPU可以用不同的段地址和偏移地址形成同一个物理地址。

（2）如果给定一个段地址，仅通过变化偏移地址来进行寻址，最多可以定位多少内存单元？结论：偏移地址16位，变化范围为0~FFFFH，仅用偏移地址来寻址最多可寻64K个内存单元。比如：给定段地址1000H，用偏移地址寻址，CPU的寻址范围为：10000H~1FFFFH。

在8086PC机中，存储单元的地址用两个元素来描述。即段地址和偏移地址。

“数据在21F60H内存单元中。”对于8086PC机的两种描述：

- （a）数据存在内存2000:1F60单元中；
- （b）数据存在内存的2000H段中的1F60H单元中。

可根据需要，将地址连续、起始地址为16的倍数的一组内存单元定义为一个段。

（1）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，高位字节存放再高地址单元中。

（2）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。

（3）[address]表示一个偏移地址为address的内存单元。

（4）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。

（5）mov、add、sub是具有两个操作对象的指令。jmp是具有一个操作对象的指令。

（6）可以根据自己的推测，在Debug中实验指令的新格式。

Attenion：

- mov时一定要注意寄存器的容量，转移字节时用AL，转移字时用AX。