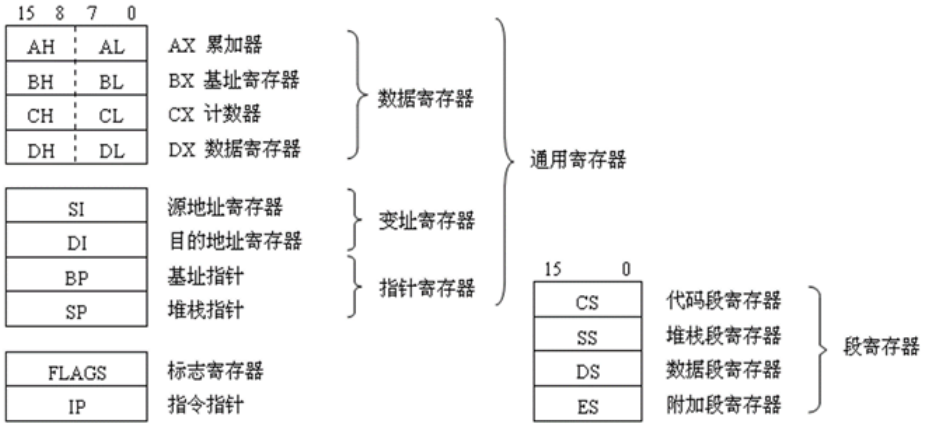
微机指令



数据传送类指令

数据传送是计算机中最基本、最重要的一种操作、最常使用的一类指令

除标志寄存器传送指令外，**均不影响标志位**

重点掌握：`MOV` `XCHG` `PUSH` `POP` `LEA`

通用数据传送指令

> MOV（move）：传送指令

把一个字节或字的操作数从源地址传送至目的地址

```
MOV reg/mem    , imm           ; 立即数送寄存器或主存
MOV reg/mem/seg, reg           ; 寄存器送（段）寄存器或主存
MOV reg/seg    , mem           ; 主存送（段）寄存器
MOV reg/mem    , seg           ; 段寄存器送寄存器或主存
```

1. 立即数传送

```
MOV                AL ,    4       ; AL ← 4，字节传送
MOV                CX , 0ffh       ; CX ← 00ffh，字传送
MOV                SI , 200h       ; SI ← 0200h，字传送
MOV byte ptr     [SI] ,  0ah       ; byte ptr 说明是字节操作
MOV word ptr [SI + 2] ,  0bh       ; word ptr 说明是字操作

注意 立即数 是 字节量 还是 字量，明确指令是 字节操作 还是 字操作
```

2. 寄存器传送

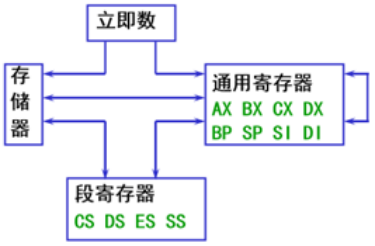
```
MOV  AX , BX          ;   AX ← BX，字传送
MOV  AH , AL          ;   AH ← AL，字节传送
MOV  DS , AX          ;   DS ← AX，字传送
MOV [BX], AL          ; [BX] ← AL，字节传送
```

3. 存储器传送

```
MOV AL, [BX]
MOV DX, [BP]          ; DX ← SS:[BP]
MOV ES, [SI]          ; ES ← DS:[SI]

不存在 存储器 向 存储器 的传送指令
```

4. 段寄存器传送

```
MOV [SI], DS
MOV  AX , ES          ;         AX ← ES
MOV  DS , AX          ; 实际为：DS ← AX ← ES

对 段寄存器 的操作有一些限制
不能给 段寄存器 和 标志寄存器 直接赋予 立即数
```



非法传送

1. 两个操作数的类型不一致：例如源操作数是字节，而目的操作数是字；或相反

```
MOV AL, 050AH        ; 非法指令：050Ah 为字，而 AL 为字节
```

1. 绝大多数双操作数指令，除非特别说明，**目的操作数**与**源操作数**必须类型一致，否则为非法指令
2. 寄存器有明确的**字节**或**字类型**，有寄存器参与的指令其操作数类型就是寄存器的类型
3. 对于存储器单元与立即数同时作为操作数的情况，必须显式指明；`byte ptr`指示字节类型，`word ptr`指示字类型

2. 两个操作数不能都是存储器：传送指令很灵活，但主存之间的直接传送却不允许

```
MOV AX     , buffer1    ; AX ← buffer1（将buffer1内容送AX）
MOV buffer2,      AX    ; buffer2 ← AX
                                                ; 这里buffer1和buffer2是两个字变量
                                                ; 实际表示直接寻址方式
8086指令系统不允许两个操作数都是存储单元（除串操作指令），要实现这种传送，可通过寄存器间接实现
```

3. 段寄存器的操作有一些限制：段寄存器属专用寄存器，对他们的操作能力有限

```
不允许立即数传送给段寄存器
MOV DS, 100H       ; 非法指令：立即数不能传送段寄存器

不允许直接改变CS值
MOV CS, [SI]       ; 不允许使用的指令

不允许段寄存器之间的直接数据传送
MOV DS,   ES       ; 非法指令：不允许段寄存器间传送
```

XCHG：交换指令

两个地方的数据进行互换

```
XCHG reg, reg/mem         ; reg <- reg/mem
```

1. 寄存器与寄存器之间对换数据

```
MOV  AX , 1234h       ; AX = 1234h
MOV  BX , 5678h       ; BX = 5678h
XCHG AX ,    BX       ; AX = 5678h, BX = 1234h
XCHG AH ,    AL       ; AX = 7856h
```

2. 寄存器与存储器之间对换数据

```
XCHG AX, [2000h]      ; 字交换
                                        ; 等同于 XCHG [2000h], AX;
XCHG AL, [2000h]      ; 字节交换
                                        ; 等同于 XCHG [2000h], AL;
```

3. 不能在存储器与存储器之间对换数据

```
...
```

XLAT（translate）：换码指令

将`BX`指定的缓冲区中、`AL`指定的位移处的一个字节数据取出赋给`AL`

```
XLAT       ; AL ← DS:[BX + AL]
```

- 换码指令执行前：

  主存建立一个字节量表格，含要转换成的目的代码

  表格首地址存放于`BX，AL`存放相对表格首的位移量

- 换码指令执行后：

  将`AL`寄存器的内容转换为目标代码

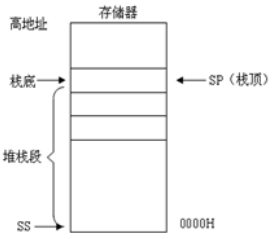
```
例：代码转换

MOV  BX , 100h
MOV  AL ,  03h
XLAT

换码指令没有显式的操作数，但使用了BX和AL；因为换码指令使用了隐含寻址方式—采用默认操作数



换码指令的两种格式完全等效。
第一种格式中，label表示首地址
第二中也可以用XLATB助记符。实际的首地址在BX寄存器中

换码指令用于将BX指定的缓冲区中、AL指定的位移处的数据取出赋给AL，格式为：
XLAT LABEL
XLAT               ;al←ds:[bx+al]

将首地址为100H的表格缓冲区中的3号数据取出
MOV BX,100H
MOV AL,03H
XLAT


因为AL的内容实际上是距离表格首地址的位移量，只有8位，所以表格的最大长度为256，
超过256的表格需要采用修改BX和AL的方法才能转换。
XLAT指令中没有显式指明操作数，而是默认是用BX和AL寄存器，这种方法称为隐含寻址方式。
```

堆栈操作指令

堆栈：按照**后进先出**(`LIFO`)的原则组织的存储器空间（栈）

注意：

- 后进先出`FILO`，位于**堆栈段**；
- `SS`**段寄存器**记录其段地址
- 堆栈只有一个出口，即当前栈顶，用**堆栈指针寄存器** `SP` 指定
- 堆栈操作的单位是**字(16位)**，进栈和出栈只对**字量**
- 字量数据从栈顶压入和弹出时，都是**低地址字节送低字节，高地址字节送高字节**
- 堆栈操作遵循先进后出原则，但可用存储器寻址方式**随机存取**堆栈中的数据



堆栈作用：

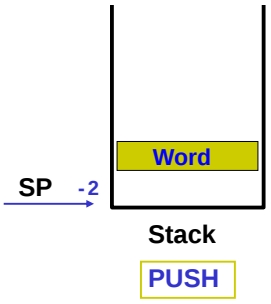
- 临时存放数据
- 传递参数
- 保存和恢复寄存器

例：现场保护恢复

```
PUSH AX          ; 进入子程序后
PUSH BX          ; 将 BX 的数据压入栈中
PUSH DS          ; 将 DS 的数据压入栈中
...
POP  DS          ; 返回主程序前
POP  BX          ; 将 出栈数据 传到 BX 中
POP  AX          ; 将 出栈数据 传到 AX 中（不是将 AX 出栈）
```

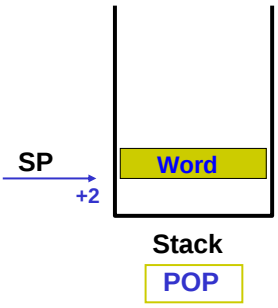
PUSH：标志寄存器进堆栈指令

```
PUSH   ; 进栈指令先使堆栈指针SP减2，然后把一个字操作数存入堆栈顶部
PUSH r16/m16/seg    ; SP ← SP - 2
                                        ; SS:[SP] ← r16/m16/seg

PUSH      AX
PUSH [2000h]
```



POP：标志寄存器出堆栈指令

```
POP    ; 出栈指令把栈顶的一个字传送至指定的目的操作数，然后堆栈指针SP加2
POP r16/m16/seg     ; r16/m16/seg ← SS:[SP]
                                    ; SP ← SP + 2

POP     AX
POP [2000h]

POP     CS   ; 错误
```



标志传送指令

标志寄存器**传送**指令用来**传送标志寄存器** `FLAGS` 的内容

标志位**操作**指令**直接**对 `CF、DF、IF` 标志进行**复位**或**置位**

标志寄存器传送

标志寄存器**传送指令**用来传送标志寄存器 `FLAGS` 的内容，方便进行**对各个标志位**的**直接操作**

有2对4条指令：

低8位传送：`LAHF` 和 `SAHF`

16位传送：`PUSHF` 和 `POPF`

- 标志低字节进出AH指令

```
LAHF      ; AH←FLAGS的低字节
--------------------------------------------------------------------------
LAHF指令将标志寄存器的低字节送寄存器AH
SF/ZF/AF/PF/CF状态标志位分别送入AH的第7/6/4/2/0位，而AH的第5/3/1位任意


SAHF      ; FLAGS的低字节←AH
--------------------------------------------------------------------------
SAHF将AH寄存器内容送FLAGS的低字节
用AH的第7/6/4/2/0位相应设置SF/ZF/AF/PF/CF标志
```

- 标志寄存器进出堆栈指令

```
PUSHF      ; SP ← SP-2
                  ; SS:[SP] ← FLAGS
PUSHF指令将标志寄存器的内容压入堆栈，同时栈顶指针SP减2

POPF       ; FLAGS ← SS:[SP]
                     ; SP ← SP+2
POPF指令将栈顶字单元内容送标志寄存器，同时栈顶指针SP加2
```

```
例：置位单步标志

PUSHF              ; 保存全部标志到堆栈
```

```
POP  AX          ; 堆栈中取出全部标志
OR   AX, 0100h   ; 设置D8 = TF = 1
                                    ; AX其他位不变
PUSH AX          ; 将AX压入堆栈
POPF             ; FLAGS ← AX
                                    ; 将堆栈内容取到标志寄存器
```

标志位操作

标志位**操作指令**直接对 `CF、DF、IF` 标志进行**复位或置位**，常用于特定的情况

对标志位进行设置的指令：`CLC STC CMC CLD STD CLI STI`

- 进位标志操作指令

```
用于任意设置进位标志
CLC    ; 复位进位标志：CF ← 0
STC    ; 置位进位标志：CF ← 1
CMC    ; 求反进位标志：CF ← ~CF
```

- 方向标志操作指令

```
串操作指令中，需要使用
CLD   ; 复位方向标志：DF ← 0  在字串操作中使变址寄存器SI/DI的地址指针自动增加，字串处理由前往后
STD   ; 置位方向标志：DF ← 1  在字串操作中使SI/DI的地址指针自动递减，字串处理由后往前


CLD
MOV SI , 0000H
LODSB ; 将字串中的 SI 指针所指的一个字节装入 AL
; 指令执行后，SI 自动加1，更新为0001H

STD
MOV SI , 0100H
LODSW ; 将字串中的 SI 指针所指的一个字(双字节)装入 AX
; 指令执行后，SI 自动加2，更新为0102H
```

- 中断标志操作指令

```
在编写中断服务程序时，需要控制可屏蔽中断的允许和禁止
CLI    ; 复位中断标志：DF ← 0
STI    ; 置位中断标志：DF ← 1
```

地址传送指令

地址传送指令将存储器单元的逻辑地址送至指定的寄存器

有效地址传送指令：`LEA`

指针传送指令：`LDS` 和 `LES`

注意不是获取存储器单元的内容

LEA（load EA）：有效地址传送指令

将存储器操作数的有效地址传送至指定的16位寄存器中

```
LEA r16, mem                    ; r16 ← mem的有效地址EA
```

```
例：获取有效地址
MOV  BX,          0400h
MOV  SI,            3ch
LEA  BX,  [BX+SI+0f62h]        ; BX = 0400h + 003ch + 0f62h = 139EH

获得主存单元的有效地址；不是物理地址，也不是该单元的内容
可以实现计算功能
```

LDS 和 LES：指针传送指令

```
LDS r16, mem        ; r16 ← mem
                                          ; DS ← mem+2
LDS指令将主存中mem指定的字送至r16，并将mem的下一字送DS寄存器

LES r16, mem        ; r16 ← mem
                                        ; ES ← mem+2
LES指令将主存中mem指定的字送至r16，并将mem的下一字送ES寄存器
```

```
例：地址指针传送

MOV word ptr [3060h],  0100h
MOV word ptr [3062h],  1450h
LES               DI, [3060h]      ; ES = 1450h, DI = 0100h
LDS               SI, [3060h]      ; DS = 1450h, SI = 0100h

mem指定主存的连续4个字节作为逻辑地址（32位的地址指针），送入 DS:r16 或 ES:r16
```

输入输出指令

8086通过输入输出指令与外设进行数据交换；呈现给程序员的外设是端口（`Port`）即`I/O`地址

8086用于寻址外设端口的地址线为16条，端口最多为 $2^{16}=65536(64k)$ 个，端口号为`0000H~FFFFH`

每个端口用于传送一个字节的外设数据

附：输入输出寻址方式

8086的端口有64K个，无需分段，设计有两种寻址方式

- **直接寻址**：只用于寻址`00H~FFH`前256个端口，操作数`i8`表示端口号
- **间接寻址**：可用于寻址全部64K个端口，`DX`寄存器的值就是端口号

大于 `FFH` 的端口只能采用间接寻址方式

IN：输入指令

将外设数据传送给CPU内的`AL/AX`

```
IN AL, i8       ; 字节输入：AL ← I/O端口（i8直接寻址）
IN AL, DX       ; 字节输入：AL ← I/O端口（DX间接寻址）
IN AX, i8       ; 字输入：  AX ← I/O端口（i8直接寻址）
IN AX, DX       ; 字输入：  AX ← I/O端口（DX间接寻址）
```

```
例：输入字量

; 直接寻址，字节量输入
IN   AL, 21h
MOV  AH,  AL
IN   AL, 20h

; 直接寻址，字量输入
IN   AX, 20h

; 间接寻址，字量输入
MOV  DX, 20h
IN   AX,  DX
```

OUT：输出指令

将CPU内的`AL/AX`数据传送给外设

```
OUT i8 , AL      ; 字节输出：I/O端口 ← AL（i8直接寻址）
OUT DX , AL      ; 字节输出：I/O端口 ← AL（DX间接寻址）
OUT i8 , AX      ; 字输出：  I/O端口 ← AX（i8直接寻址）
OUT DX , AX      ; 字输出：  I/O端口 ← AX（DX间接寻址）
```

```
例：输出字节量

; 间接寻址，字节量输出
MOV DX , 3fch

MOV AL ,  80h
OUT DX ,   AL
```

算术运算类指令

四则运算是计算机经常进行的一种操作。

算术运算指令实现二进制（和十进制）数据的四则运算。

请注意算术运算类指令对标志的影响

掌握：`ADD/ADC/INC` `SUB/SBB/DEC/NEG/CMP`

熟悉：`MUL/IMUL` `DIV/IDIV`

理解：`CBW/CWD` `DAA/DAS` `AAA/AAS/AAM/AAD`

ADD：加法指令

`ADD` 指令将源与目的操作数相加，结果**送**到**目的操作数**

`ADD` 指令按**状态标志**的定义相应设置

```
ADD reg , imm/reg/mem              ; reg ← reg + imm/reg/mem
ADD mem ,     imm/reg              ; mem ← mem + imm/reg
```

例：加法运算

```
MOV              AL ,  0fbh        ;     AL =   fbh
ADD              AL ,   07h        ;     AL =(1)02h , CF = 1
MOV word ptr [200h] , 4652h        ; [200h] = 4652h
MOV              BX ,  1feh        ;     BX =  1feh
ADD              AL ,    BL        ;     AL =(2)00h , CF = 1
ADD word ptr [BX+2] ,0f0f0h        ; [200h] = 3742h ; BX + 2 = 200h
```

ADC：带进位加法指令

`ADC` 指令将源与目的操作数相加，再加上进位 `CF` 标志，结果送到目的操作数

`ADC` 指令按**状态标志**的定义相应设置

`ADC` 指令主要与 `ADD` 配合，实现多精度加法运算

```
ADC reg , imm/reg/mem       ; reg ← reg + imm/reg/mem + CF
ADC mem , imm/reg           ; mem ← mem + imm/reg     + CF
```

例：双字加法

```
MOV AX ,  4652h       ; AX = 4652h
ADD AX , 0f0f0h       ; AX = 3742h, CF = 1
MOV DX ,  0234h       ; DX = 0234h
ADC DX , 0f0f0h       ; DX = f325h, CF = 0(f325h + 0)
```

```
                                                ; DX.AX = 0234 4652H + F0F0 F0F0H
                                                          = F325 3742H
```

INC（increment）：增量指令

`INC` 指令对操作数加1（增量）

`INC` 指令**不影响**进位 `CF` 标志，按定义设置其他状态标志

```
INC reg/mem         ; reg/mem ← reg/mem + 1
```

```
INC           BX
INC byte ptr [BX]
```

SUB（subtract）：减法指令

`SUB` 指令将目的操作数减去源操作数，结果送到目的操作数

`SUB` 指令按照定义相应设置**状态标志**

```
SUB reg , imm/reg/mem     ; reg ← reg - imm/reg/mem
SUB mem , imm/reg         ; mem ← mem - imm/reg
```

```
例：减法运算
MOV              AL ,   0fbh         ;     AL =   fbh
SUB              AL ,    07h         ;     AL =   f4h, CF = 0
MOV word ptr [200h] ,  4652h         ; [200h] = 4652h
MOV              BX ,   1feh         ;     BX =  1feh
SUB              AL ,     BL         ;     AL =   f6h
SUB word ptr [bx+2] , 0f0f0h         ; [200h] = 5562h, CF = 1
```

SBB：带借位减法指令

`SBB` 指令将目的操作数减去源操作数，再减去借位 `CF`（进位），结果送到目的操作数。

`SBB` 指令按照定义相应设置状态标志

`SBB` 指令主要与 `SUB` 配合，实现多精度减法运算

```
SBB reg , imm/reg/mem      ; reg ← reg - imm/reg/mem - CF
SBB mem , imm/reg          ; mem ← mem - imm/reg     - CF
```

```
例：双字减法
MOV AX ,  4652h       ; ax = 4652h
SUB AX , 0f0f0h       ; ax = 5562h, CF = 1
MOV DX ,  0234h       ; dx = 0234h
```

```
SBB DX , 0f0f0h       ; dx = 1143h, CF = 1(1144h - 1)

                                          ; DX.AX = 0234 4652H - F0F0 F0F0H
                                                  = 1143 5562H
```

DEC（decrement）：减量指令

`DEC`指令对操作数减1（减量）

`DEC`指令**不影响**进位`CF`标志，按定义设置其他状态标志

```
DEC reg/mem        ; reg/mem ← reg/mem - 1
```

`INC`指令和`DEC`指令都是**单操作数指令，**主要用于对计数器和地址指针的调整。

NEG（negative）：求补指令

`NEG`指令对操作数执行求补运算：用零减去操作数，然后结果返回操作数

求补运算也可以表达成：将操作数按位取反后加1

`NEG`指令对标志的影响与用零作减法的`SUB`指令一样

```
NEG reg/mem         ; reg/mem ← 0 - reg/mem

CF = 1  不为 0的操作数 求补
CF = 0  为   0的操作数 求补
(想象成用0去减一个数，减的如果是非零数CF位肯定改变)

OF = 1  当求补运算的操作数为-128（字节运算）或操作数为-32768（字运算）
OF = 0  当求补运算的操作数不为-128（字节）或操作数为-32768（字运算）
```

```
例：求补运算

MOV AX , 0ff64h
NEG AL                ; AX = ff9ch, OF=0、SF=1、ZF=0、PF=1、CF=1
SUB AL ,    9dh       ; AX = ffffh, OF=0、SF=1、ZF=0、PF=1、CF=1
NEG AX                ; AX = 0001h, OF=0、SF=0、ZF=0、PF=0、CF=1
DEC AL                ; AX = 0000h, OF=0、SF=0、ZF=1、PF=1、CF=1
NEG AX                ; AX = 0000h, OF=0、SF=0、ZF=1、PF=1、CF=0

-------------------------------------------------
MOV AL , -128         ; AL = 1000 0000
NEG AL                ; AL = 1000 0000, OF = 1
  +128，超出了8位数的补码的范围，就是溢出了，OF = 1

MOV AL , -127         ; AL = 1111 1111
NEG AL                ; AL = 1000 0001, OF = 0
  -127的补码，不超出范围，没有溢出，OF = 0

-------------------------------------------------
MOV AL , 64h
```

```
NEG AL          ; AL = 9ch
64h  补码表示  0110 0100b
     按位取反  1001 1011b
     末位加1   1001 1100b = 9ch


MOV AL , -8
NEG AL          ; AL = 08h
-8  原码表示：1000 1000b
    反码表示：1111 0111b
    补码表示：1111 1000b
    按位取反：0000 0111b
    末位加1： 0000 1000b = 8 = 08h

-------------------------------------------------
操作数 > 0：加负号，求反码
操作数 < 0：直接求绝对值
```

CMP（compare）：比较指令

`CMP`指令将目的操作数减去源操作数，按照定义相应设置状态标志

`CMP`指令执行的功能与SUB指令，但结果不回送目的操作数

```
CMP reg , imm/reg/mem     ; reg - imm/reg/mem
CMP mem , imm/reg         ; mem - imm/reg
```

```
例：比较AL与100

                CMP AL , 100        ; AL - 100
                JB below            ; AL < 100, 跳转到 below 执行
                SUB AL , 100        ; AL ≥ 100, AL ← AL - 100
                INC AH              ; AH ← AH + 1
below:  ...

执行比较指令之后，可以根据标志判断两个数是否相等、大小关系等
```

乘法指令

```
MUL r8/m8         ; 无符号字节乘法
                                ; AX ← AL × r8/m8

MUL r16/m16       ; 无符号字乘法
                                  ; DX.AX ← AX × r16/m16

IMUL r8/m8            ; 有符号字节乘法
```

```
                                    ; AX ← AL × r8/m8

IMUL r16/m16        ; 有符号字乘法
                                    ; DX.AX ← AX × r16/m16
```

功能：

乘法指令分无符号和有符号乘法指令

乘法指令的源操作数显式给出，隐含使用另一个操作数`AX`和`DX`

字节量相乘：`AL`与`r8/m8`相乘，得到16位的结果，存入`AX`

字量相乘：`AX`与`r16/m16`相乘，得到32位的结果，其高字存入`DX`，低字存入`AX`

乘法指令利用`OF`和`CF`判断乘积的高一半是否具有有效数值

对标志的影响：

乘法指令如下影响`OF`和`CF`标志：

`MUL`指令——若乘积的高一半（`AH`或`DX`）为0，则OF = CF = 0；否则OF = CF = 1

`IMUL`指令——若乘积的高一半是低一半的符号扩展，则OF = CF = 0；否则均为1

乘法指令对其他状态标志没有定义

对标志没有定义：指令执行后这些标志是任意的、不可预测（就是谁也不知道是0还是1）

对标志没有影响：指令执行不改变标志状态

```
例：乘法运算

MOV  AL ,  b4h      ; AL =   b4h = 180
MOV  BL ,  11h      ; BL =   11h = 17
MUL  BL             ; AX = 0bf4h = 3060
                                ; OF = CF = 1 , AX高8位不为0

MOV  AL ,  b4h      ; AL =   b4h (          1011 0100) =   -76 (     0100 1100)
MOV  BL ,  11h      ; BL =   11h (          0001 0001) =    17
IMUL BL             ; AX = faf4h (1111 1010 1111 0100) = -1292 (0101 0000 1100)
                            ; OF = CF = 1 , AX高8位含有效数字
```

除法指令

```
DIV r8/m8           ; 无符号字节除法：
                                    ; AL ← AX ÷ r8/m8的商 , AH ← AX ÷ r8/m8的余数

DIV r16/m16         ; 无符号字除法：
                                    ; AX ← DX.AX ÷ r16/m16的商 , DX ← DX.AX ÷ r16/m16的余

IDIV r8/m8          ; 有符号字节除法：
                                    ; AL ← AX ÷ r8/m8的商 , AH ← AX ÷ r8/m8的余数

IDIV r16/m16        ; 有符号字除法：
                                    ; AX ← DX.AX ÷ r16/m16的商 , DX ← DX.AX ÷ r16/m16的余
```

功能：

除法指令分无符号和有符号除法指令

除法指令的除数显式给出，隐含使用另一个操作数`AX`和`DX`作为被除数

字节量除法：`AX`除以`r8/m8`，8位商存入`AL`，8位余数存入`AH`

字量除法：`DX.AX`除以`r16/m16`，16位商存入`AX`，16位余数存入`DX`

除法指令对标志没有定义

除法指令会产生结果溢出

除法错中断

当被除数远大于除数时，所得的商就有可能超出它所能表达的范围。如果存放商的寄存器`AL/AX`不能表达，便产生溢出，8086CPU中就产生编号为**0的内部中断**——除法错中断

对`DIV`指令，除数为0，或者在字节除时商超过8位，或者在字除时商超过16位

对`IDIV`指令，除数为0，或者在字节除时商不在`-128~127`范围内，或者在字除时商不在`-32768~32767`范围内

例：除法运算

```
MOV  AX , 0400h      ; AX = 400h = 1024
MOV  BL , 0b4h       ; BL =  b4h =  180
DIV  BL              ; 商   AL = 05h =   5
                                       ; 余数 AH = 7ch = 124

MOV  AX , 0400h      ; AX = 400h = 1024
MOV  BL , 0b4h       ; BL =  b4h =  -76
IDIV BL              ; 商   AL = f3h = -13
                                       ; 余数 AH = 24h =  36
```

符号扩展指令

```
CBW       ; AL 的符号扩展至 AH
                    ; eg.AL的最高有效位是0 , 则 AH = 00
                    ;    AL的最高有效位为1 , 则 AH = FFH。AL不变

CWD       ; AX 的符号扩展至 DX
                    ; eg.AX的最高有效位是0 , 则 DX = 00
                    ;    AX的最高有效位为1 , 则 DX = FFFFH。AX不变
```

符号扩展指令常用于获得倍长的数据

符号扩展是指用一个操作数的符号位（即最高位）形成另一个操作数，后一个操作数的各位是全0（正数）或全1（负数）。符号扩展不改变数据大小。

对于数据64H（表示数据100），其最高位D7为0，符号扩展后高8位都是0，成为`0064H`（仍表示数据100）

对于数据`FF00H`（表示有符号数-256），其最高位D15为1，符号扩展后高16位都是1，成为`FFFFFF00H`（仍表示有符号数-256）

```
例：符号扩展
MOV AL , 80h     ; AL =   80h
CBW              ; AX = ff80h

ADD AL , 255     ; AL =   7fh
CBW              ; AX = 007fh
```

```
例：AX ÷ BX

CWD              ; DX.AX ← AX
IDIV BX          ; AX ← DX.AX ÷ BX
对有符号数除法，可以利用符号扩展指令得到倍长于除数的被除数
对无符号数除法，采用直接使高8位或高16位清0，获得倍长的被除数。这就是零位扩展
```

十进制调整指令

十进制数调整指令对二进制运算的结果进行十进制调整，以得到十进制的运算结果

分成压缩BCD码和非压缩BCD码调整

压缩BCD码就是通常的8421码，它用4个二进制位表示一个十进制位，一个字节可以表示两个十进制位，即00～99

非压缩BCD码用8个二进制位表示一个十进制位，只用低4个二进制位表示一个十进制位0～9，高4位任意，通常默认为0

BCD码（Binary Coded DecimAL）

二进制编码的十进制数：一位十进制数用4位二进制编码来表示

8086支持压缩BCD码和非压缩BCD码的调整运算

```
真值            8      64
二进制编码     08H     40H
压缩BCD码      08H     64H
非压缩BCD码    08H    0604H

直接看成十进制运算
```

压缩BCD码加、减调整指令

```
 (ADD AL , i8/r8/m8)
 (ADC AL , i8/r8/m8)
DAA                  ; AL ← 将AL的加和调整为压缩BCD码

 (SUB AL , i8/r8/m8)
 (SBB AL , i8/r8/m8)
DAS                  ; AL ← 将AL的减差调整为压缩BCD码
```

使用DAA或DAS指令前，应先执行以AL为目的操作数的加法或减法指令

DAA和DAS指令对OF标志无定义，按结果影响其他标志。例如CF反映压缩BCD码相加或减的进位或借位状态

例：压缩BCD加法

```
MOV AL , 68h                    ; AL = 68h , 压缩BCD码表示真值68
MOV BL , 28h                    ; BL = 28h , 压缩BCD码表示真值28
ADD AL ,  BL          ; 二进制加法：AL = 68h + 28h = 90h
DAA                   ; 十进制调整：AL = 96h
                      ; 实现压缩BCD码加法：68 + 28 = 96
```

例：压缩BCD减法

```
MOV  AX , 1234h
MOV  BX , 4612h
SUB  AL , BL
DAS              ; 34 - 12 = 22 , CF = 0
XCHG AL , AH
SBB  AL , BH
DAS              ; 12 - 46 = 66 , CF = 1
XCHG AL , AH     ; 1234 - 4612 = 6622
```

非压缩BCD码加、减调整指令

```
 (ADD AL , i8/r8/m8)
 (ADC AL , i8/r8/m8)
AAA                 ; AL ← 将AL的加和调整为非压缩BCD码
                    ; AH ← AH＋调整的进位

 (SUB AL , i8/r8/m8)
 (SBB AL , i8/r8/m8)
AAS                 ; AL ← 将AL的减差调整为非压缩BCD码
                    ; AH ← AH－调整的借位
```

使用AAA或AAS指令前，应先执行以AL为目的操作数的加法或减法指令

AAA和AAS指令在调整中产生了进位或借位，则AH要加上进位或减去借位，同时CF = AF = 1，否则CF = AF = 0；它们对其他标志无定义

例：非压缩BCD加法

```
MOV AX , 0608h        ; AX = 0608h , 非压缩BCD码表示真值68
MOV BL ,   09h        ; BL =   09h , 非压缩BCD码表示真值9
ADD AL ,    BL        ; 二进制加法：AL = 08h + 09h = 11h
AAA                   ; 十进制调整：AX = 0707h
                      ; 实现非压缩BCD码加法：68 + 9 = 77
```

例：非压缩BCD减法

```
MOV AX , 0608h      ; AX = 0608h , 非压缩BCD码表示真值68
MOV BL ,   09h      ; BL =   09h , 非压缩BCD码表示真值9
```

```
SUB AL ,    BL      ; 二进制减法：AL = 08h - 09h = ffh
AAS                 ; 十进制调整：AX = 0509h
                                    ; 实现非压缩BCD码减法：68 - 9 = 59
```

非压缩BCD码乘、除调整指令

```
（MUL r8/m8）
AAM       ; AX ← 将AX的乘积调整为非压缩BCD码

AAD       ; AX ← 将AX中非压缩BCD码扩展成二进制数
（DIV r8/m8）
```

AAM指令跟在字节乘MUL**之后**，将乘积调整为非压缩BCD码

AAD指令跟在字节除DIV**之前**，先将非压缩BCD码的被除数调整为二进制数

AAM和AAD指令根据结果设置SF、ZF和PF，但对OF、CF和AF无定义

例：非压缩BCD乘法

```
MOV AX , 0608h      ; AX = 0608h , 非压缩BCD码表示真值68
MOV BL ,   09h      ; BL = 09h , 非压缩BCD码表示真值9
MUL BL              ; 二进制乘法：AL = 08h × 09h = 0048h
AAM                 ; 十进制调整：AX = 0702h
                    ; 实现非压缩BCD码乘法：8 × 9 = 72
```

例：非压缩BCD除法

```
MOV AX , 0608h          ; AX = 0608h , 非压缩BCD码表示真值68
MOV BL ,   09h          ; BL = 09h , 非压缩BCD码表示真值9
AAD                     ; 二进制扩展：AX = 68 = 0044h
DIV BL                          ; 除法运算：商AL = 07h , 余数AH = 05h
                                  ; 实现非压缩BCD码初法：68 ÷ 9 = 7（余5）
```

例：算术运算1

```
MOV  AX ,  X
IMUL  Y             ; DX.AX = X × Y
MOV  CX , AX
MOV  BX , DX        ; BX.CX = DX.AX = X × Y
MOV  AX ,  Z
CWD
ADD  CX , AX
ADC  BX , DX        ; BX.CX = X × Y + Z
```

例：算术运算2

```
SUB  CX , 540
SBB  BX ,   0       ; BX.CX = X × Y + Z - 540
```

```
MOV  AX ,   V
CWD
SUB  AX ,  CX
SBB  DX ,  BX       ; DX.AX = V - (X×Y+Z-540)
IDIV  X             ; DX.AX =  (V - (X×Y+Z-540) ) ÷X
```

位操作类指令

位操作类指令以二进制位为基本单位进行数据的操作

这是一类常用的指令，都应该掌握

注意这些指令对标志位的影响

1、逻辑运算指令：`AND OR XOR NOT TEST`

2、移位指令：`SHL SHR SAR`

3、循环移位指令：`ROL ROR RCL RCR`

AND：逻辑与指令

对两个操作数执行逻辑与运算，结果送到目的操作数

```
AND reg , imm/reg/mem    ; reg ← reg ∧ imm/reg/mem
AND mem , imm/reg        ; mem ← mem ∧ imm/reg
```

只有相“与”的两位都是1，结果才是1; 否则，“与”的结果为0

AND指令设置CF = OF = 0，根据结果设置SF、ZF和PF状态，而对AF未定义

OR：逻辑或指令

对两个操作数执行逻辑或运算，结果送到目的操作数

```
OR reg , imm/reg/mem      ; reg ← reg ∨ imm/reg/mem
OR mem , imm/reg          ; mem ← mem ∨ imm/reg
```

只要相“或”的两位有一位是1，结果就是1; 否则，结果为0

OR指令设置CF = OF = 0，根据结果设置SF、ZF和PF状态，而对AF未定义

XOR：逻辑异或指令

对两个操作数执行逻辑异或运算，结果送到目的操作数

```
XOR reg , imm/reg/mem    ; reg ← reg ⊕ imm/reg/mem
XOR mem , imm/reg        ; mem ← mem ⊕ imm/reg
```

只有相“异或”的两位不相同，结果才是1; 否则，结果为0

XOR指令设置CF = OF = 0，根据结果设置SF、ZF和PF状态，而对AF未定义

NOT：逻辑非指令

对一个操作数执行逻辑非运算

```
NOT reg/mem     ; reg/mem ← ～reg/mem
```

按位取反，原来是“0”的位变为“1”；原来是“1”的位变为“0”

NOT指令是一个单操作数指令

NOT指令不影响标志位

例：逻辑运算

```
MOV AL , 45h      ; 逻辑与 AL = 01h
AND AL , 31h      ; CF = OF = 0 , SF = 0、ZF = 0、PF = 0
MOV AL , 45h      ; 逻辑或 AL = 75h
OR  AL , 31h      ; CF = OF = 0 , SF = 0、ZF = 0、PF = 0
MOV AL , 45h      ; 逻辑异或 AL = 74h
XOR AL , 31h      ; CF = OF = 0 , SF = 0、ZF = 0、PF = 1
MOV AL , 45h      ; 逻辑非 AL = 0bAH
NOT AL            ; 标志不变
```

例：逻辑指令应用

```
; AND指令可用于复位某些位（同0相与）,不影响其他位：将BL中D3和D0位清0,其他位不变
AND BL , 11110110B

; OR指令可用于置位某些位（同1相或）,不影响其他位：将BL中D3和D0位置1,其他位不变
OR  BL , 00001001B

; XOR指令可用于求反某些位（同1相异或）,不影响其他位：将BL中D3和D0位求反,其他不变
XOR BL , 00001001B
```

TEST：测试指令

对两个操作数执行逻辑与运算，结果不回送到目的操作数

```
TEST reg , imm/reg/mem     ; reg ∧ imm/reg/mem
TEST mem , imm/reg         ; mem ∧ imm/reg
```

只有相“与”的两位都是1，结果才是1；否则，“与”的结果为0

TEST指令设置CF = OF = 0，根据结果设置SF、ZF和PF状态，而对AF未定义

例：测试为0或1

```
              TEST AL , 01h   ; 测试AL的最低位D0
                JNZ there       ; 标志ZF = 0 , 即D0 = 1 , 程序转移到 there
                                ; 否则ZF = 1 , 即D0 = 0 , 顺序执行
```

```
                ...
there:  ...
```

TEST指令通常用于检测一些条件是否满足，但又不希望改变原操作数的情况

移位指令（shift）

将操作数移动一位或多位，分成逻辑移位和算术移位，分别具有左移或右移操作（SAL与SHL相同）

```
SHL reg/mem , 1/CL      ; 逻辑左移 , 最高位进入CF , 最低位补0
SHR reg/mem , 1/CL      ; 逻辑右移 , 最低位进入CF , 最高位补0
SAL reg/mem , 1/CL      ; 算术左移 , 最高位进入CF , 最低位补0
SAR reg/mem , 1/CL      ; 算术右移 , 最低位进入CF , 最高位不变
```

移位指令的操作数

移位指令的第一个操作数是指定的被移位的操作数，可以是寄存器或存储单元

后一个操作数表示移位位数，该操作数为1，表示移动一位；当移位位数大于1时，则用`CL`寄存器值表示，该操作数表达为`CL`

移位指令对标志的影响

按照移入的位设置进位标志`CF`

根据移位后的结果影响`SF、ZF、PF`

对AF没有定义

如果进行一位移动，则按照操作数的最高符号位是否改变，相应设置溢出标志OF。如果移位前的操作数最高位与移位后操作数的最高位不同（有变化），则OF = 1；否则OF = 0。

当移位次数大于1时，OF不确定。

例：移位指令

```
MOV CL , 4
MOV AL , 0f0h   ; AL = f0h
SHL AL , 1      ; AL = e0h
                              ; CF =   1 , SF=1、ZF=0、PF=0 , OF=0
SHR AL , 1      ; AL = 70h
                              ; CF =   0 , SF=0、ZF=0、PF=0、OF=1
SAR AL , 1      ; AL = 38h
                                ; CF =   0 , SF=0、ZF=0、PF=0、OF=0
SAR AL , CL     ; AL = 03h
                                ; CF =   1 , SF=0、ZF=0、PF=1
```

例：移位实现乘

```
MOV SI ,  AX
AHL SI ,   1        ; SI ← 2×AX
ADD SI ,  AX        ; SI ← 3×AX
MOV DX ,  BX
```

```
MOV CL , 03h
SHL DX ,  CL          ; DX ← 8×BX  ; 左移二进制3位（CL）
SUB DX ,  BX          ; DX ← 7×BX
ADD DX ,  SI          ; DX ← 7×BX + 3×AX
```

逻辑左移一位相当于无符号数乘以2

逻辑右移一位相当于无符号数除以2

循环移位指令（rotate）

将操作数从一端移出的位返回到另一端形成循环，分成不带进位和带进位，分别具有左移或右移操作

```
ROL reg/mem , 1/CL   ; 不带进位循环左移
ROR reg/mem , 1/CL   ; 不带进位循环右移
RCL reg/mem , 1/CL   ; 带进位循环左移
RCR reg/mem , 1/CL   ; 带进位循环右移
```

循环移位指令对标志的影响

按照指令功能设置进位标志CF，不影响 `SF、ZF、PF、AF`

如果进行一位移动，则按照操作数的最高符号位是否改变，相应设置溢出标志OF。如果移位前的操作数最高位与移位后操作数的最高位不同（有变化），则OF = 1；否则OF = 0。

当移位次数大于1时，OF不确定

```
例： 32位数移位

; 将 DX.AX 中32位数值左移一位
SHL AX , 1
RCL DX , 1
```

```
例：位传送

; 把AL最低位送BL最低位 ， 保持AL不变
ROR BL , 1
ROR AL , 1
RCL BL , 1
ROL AL , 1
```

```
例：BCD码合并

; AH.AL分别存放着非压缩BCD码的两位
; 将它们合并成为一个压缩BCD码存AL
AND AX , 0f0fh        ; 保证高4位为0
MOV CL ,     4
ROL AH ,    CL        ; 也可以用 SHL AH , CL
ADD AL ,    AH        ; 也可以用  OR AL , AH
```

控制转移类指令

控制转移类指令用于实现分支、循环、过程等程序结构，是仅次于传送指令的常用指令

重点掌握：`JMP/JCC/LOOP CALL/RET INT n/IRET（常用系统功能调用）`

一般了解：`LOOPZ/LOOPNZ INTO`

控制转移类指令通过**改变**`IP`（和`CS`）值，实现程序执行顺序的改变。

JMP：无条件转移指令

```
JMP label        ; 程序转向label标号指定的地址
```

只要执行无条件转移指令`JMP`，就使程序转到指定的**目标地址处**，从目标地址处开始执行指令，操作数`label`是要转移到的目标地址（目的地址、转移地址）

`JMP`指令分成4种类型：

1. 段内转移、相对寻址
2. 段内转移、间接寻址
3. 段间转移、直接寻址
4. 段间转移、间接寻址

目标地址的寻址方式：

- **相对寻址**方式（用**标号**表达）

  相对寻址方式以当前IP为基地址，加上位移量构成目标地址
- **直接寻址**方式（用**标号**表达）

  转移地址象立即数一样，直接在指令的机器代码中
- **间接寻址**方式（用**寄存器**或**存储器操作数**表达）

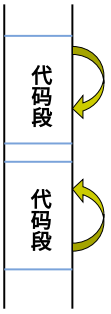
  转移地址在寄存器或主存单元中

目标地址的范围：**段内**

段内转移——近转移（near）

在当前代码段`64KB`范围内转移（`±32KB`范围）

不需要更改`CS`段地址，只要改变`IP`偏移地址



段内转移——短转移（short）

转移范围可以用一个字节表达，在段内`-128 ~ +127`范围的转移

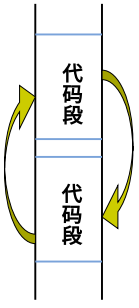
目标地址的范围：**段间**

段间转移——远转移（far）

从当前代码段跳转到另一个代码段，可以在1MB范围

更改`CS`段地址和`IP`偏移地址

目标地址必须用一个32位数表达，叫做32位远指针，它就是**逻辑地址**



实际编程时，MASM汇编程序会根据目标地址的距离，**自动处理**成短转移、近转移或远转移

程序员可用操作符`short`、`near ptr`或`far ptr`强制

- 段内转移、相对寻址

```
JMP label                   ; IP ← IP+位移量
```

位移量是紧接着`JMP`指令后的那条指令的偏移地址，到目标指令的偏移地址的地址位移

当向地址增大方向转移时，位移量为正；向地址减小方向转移时，位移量为负

```
again:    DEC CX            ; 标号again的指令
                    ...
                    JMP again         ; 转移到again处继续执行
                    ...
                    JMP output        ; 转向output
                    ...
output:   MOV result , AL   ; 标号output的指令
```

- 段内转移、间接寻址

```
JMP r16/m16        ; IP ← r16/m16
```

将一个16位寄存器或主存字单元内容送入`IP`寄存器，作为新的指令指针，但不修改`CS`寄存器的内容

```
JMP               AX        ; IP ← AX
JMP word ptr [2000h]        ; IP ← [2000h]
```

- 段间转移、直接寻址

```
JMP far ptr label        ; IP ← label的偏移地址
                             ; CS ← label的段地址
```

将标号所在段的段地址作为新的`CS`值，标号在该段内的偏移地址作为新的IP值；这样，程序跳转到新的代码段执行

```
JMP far ptr otherseg     ; 远转移到代码段2的otherseg
```

- 段间转移、间接寻址

```
JMP far ptr mem          ; IP ← [mem] , CS ← [mem+2]
```

用一个双字存储单元表示要跳转的目标地址。这个目标地址存放在主存中连续的两个字单元中的，低位字送`IP`寄存器，高位字送`CS`寄存器

```
MOV word ptr   [BX] ,     0
MOV word ptr [BX+2] , 1500h
JMP  far ptr   [BX]               ; 转移到1500h:0
```

JCC：条件转移指令

```
JCC label                ; 条件满足 , 发生转移：IP ← IP + 8位位移量
                         ; 条件不满足 , 顺序执行
```

指定的条件`CC`如果成立，程序转移到由标号`label`指定的目标地址去执行指令；条件不成立，则程序将顺序执行下一条指令

操作数`label`是采用**相对寻址方式**的**短转移标号**

表示`JCC`指令后的那条指令的偏移地址，到目标指令的偏移地址的地址位移

距当前`IP`地址`-128 ~ +127`个单元的范围之内

`JCC`指令的分类：

`JCC`指令不影响标志，但要利用标志（见下）。根据利用的标志位不同，16条指令分成3种情况：

1. 判断单个标志位状态
2. 比较无符号数高低
3. 比较有符号数大小

转移条件`CC`：单个标志状态

```
JZ/JE    ZF=1      Jump if Zero/EquAL
JNZ/JNE  ZF=0      Jump if Not Zero/Not EquAL

JS       SF=1      Jump if SIgn
JNS      SF=0      Jump if Not SIgn

JP/JPE   PF=1      Jump if Parity/Parity Even
JNP/JPO  PF=0      Jump if Not Parity/Parity Odd

JO       OF=1      Jump if Overflow
JNO      OF=0      Jump if Not Overflow

JC       CF=1      Jump if Carry
JNC      CF=0      Jump if Not Carry

采用多个助记符 , 只是为了方便记忆和使用
```

转移条件`CC`：两数大小关系

```
JB  /JNAE  CF = 1                  Jump if Below/Not Above  or EquAL
JNB /JAE   CF = 0                  Jump if Not Below/Above  or EquAL
JBE /JNA   CF = 1 或 ZF = 1        Jump if Below/Not Above
JNBE/JA    CF = 0 且 ZF = 0        Jump if Not Below        or EquAL/Above
JL  /JNGE  SF ≠ OF                 Jump if Less/Not Greater or EquAL
JNL /JGE   SF = OF                 Jump if Not Less/Greater or EquAL
JLE /JNG   ZF ≠ OF 或 ZF = 1       Jump if Less             or EquAL/Not Greater
JNLE/JG    SF = OF 且 ZF = 0       Jump if Not Less         or EquAL/Greater
```

判断单个标志位状态

这组指令单独判断5个状态标志之一

- `JZ/JE`和`JNZ/JNE`：利用**零**标志ZF，判断结果是否为零（或相等）

例：JZ/JNZ指令

```
                TEST    AL ,  80h     ; 测试最高位
                JZ   next0            ; D7 = 0（ZF = 1） , 转移
                MOV     AH , 0ffh     ; D7 = 1 , 顺序执行
                JMP   done            ; 无条件转向
next0:  MOV     AH ,    0
done :  ...
-----------------------------------------------------------------
                TEST    AL ,  80h     ; 测试最高位
                JNZ  next1            ; D7 = 1（ZF = 0） , 转移
                MOV     AH ,    0     ; D7 = 0 , 顺序执行
                JMP   done            ; 无条件转向
next1:  MOV     AH , 0ffh
done:   ...
```

- `JS`和`JNS`：利用**符号**标志`SF`，判断结果是正是负

例：JS/JNS指令

```
; 计算|X - Y|（绝对值）
; X和Y为存放于X单元和Y单元的16位操作数
; 结果存入result
                MOV     AX , X
                SUB     AX , Y
                JNS nonneg
                NEG     AX          ; neg：求补指令
nonneg: MOV result , AX
```

- `JO`和`JNO`：利用**溢出**标志`OF`，判断结果是否产生溢出

例：JO/JNO指令

```
; 计算X - Y ;
; X和Y为存放于X单元和Y单元的16位操作数
; 若溢出 , 则转移到overflow处理
                    MOV       AX , X
                    SUB       AX , Y
                    JO  overflow
                    ...               ; 无溢出 , 结果正确
overflow: ...               ; 有溢出处理
```

- `JP/JPE`和`JNP/JPO`：利用奇偶标志`PF`，判断结果中“1”的个数是偶是奇

例：JP/JNP指令

```
; 设字符的ASCII码在AL寄存器中
; 将字符加上奇校验位
; 在字符ASCII码中为“1”的个数已为奇数时
; 则令其最高位为“0” ; 否则令最高位为“1”
            AND   AL , 7fh        ; 最高位置“0” , 同时判断“1”的个数
            JNP next              ; 个数已为奇数 , 则转向next
```

```
            OR    AL , 80h        ; 否则 , 最高位置“1”
next: ...
```

- `JC/JB/JNAE`和`JNC/JNB/JAE`：利用进位标志`CF`，判断结果是否进位或借位

```
例：JC/JNC 指令

; 记录BX中1的个数
                XOR     AL , AL       ; AL = 0 , CF = 0 , 置零（异或）
again:  TEST    BX , 0ffffh   ; 等价于 CMP BX , 0
                JE    next
                SHL     BX , 1
                JNC   again
                INC     AL
                JMP   again
next :  ...                   ; AL保存1的个数
------------------------------------------------------------
                XOR     AL , AL       ; AL = 0 , CF = 0 , 置零（异或）
again:  CMP     BX ,  0
                JZ    next
                SHL     BX ,  1       ; 也可使用 SHR BX , 1
                ADC     AL ,  0
                JMP   again
next :  ...                   ; AL保存1的个数
```

JNB：比较**无符号数**高低

- 无符号数的大小用高（`Above`）低（`Below`）表示
- 利用`CF`确定高低、利用`ZF`标志确定相等（`EquAL`）
- 两数的高低分成4种关系：
  - 低于（不高于等于）：`JB（JNAE）`
  - 不低于（高于等于）：`JNB（JAE）`
  - 低于等于（不高于）：`JBE（JNA）`
  - 不低于等于（高于）：`JNBE（JA）`

```
例：比较无符号数

; AX保存较大的无符号数
            CMP     AX , BX     ; 比较AX和BX
            JNB   next          ; 若AX ≥ BX   ,   转移
            XCHG    AX , BX     ; 若AX < BX   ,   交换
next: ...
```

JNL：比较**有符号数**大小

- 有符号数的大（`Greater`）小（`Less`）需要组合`OF、SF`标志，并利用`ZF`标志确定相等（`EquAL`）
- 两数的大小分成4种关系：
  - 小于（不大于等于）：`JL（JNGE）`

- 不小于（大于等于）：`JNL（JGE）`
- 小于等于（不大于）：`JLE（JNG）`
- 不小于等于（大于）：`JNLE（JG）`

例：比较有符号数

```
; AX保存较大的有符号数
            CMP    AX , BX    ; 比较AX和BX
            JNL  next         ; 若AX ≥ BX  ,  转移
            XCHG   AX , BX    ; 若AX < BX  ,  交换
next: ...
```

LOOP：循环指令

```
JCXZ   label      ; CX = 0            , 转移到标号label

LOOP   label      ; CX ← CX - 1
                                  ; CX ≠ 0            , 循环到标号label

LOOPZ  label      ; CX ← CX - 1
                                  ; CX ≠ 0 且 ZF = 1  , 循环到标号label

LOOPNZ label      ; CX ← CX - 1
                                  ; CX ≠ 0 且 ZF = 0  , 循环到标号label
```

循环指令默认利用CX计数器
label操作数采用相对短转移寻址方式

例：记录空格个数

```
                MOV     CX , count          ; 设置循环次数
                MOV     SI , offset string
                XOR     BX , BX             ; BX = 0 , 记录空格数
                jCXZ  done
                            -- CMP CX , 0
                            -- JZ done
                MOV     AL , 20h            ; 如果长度为0 , 退出
again:  CMP     AL , ES:[SI]
                JNZ   next                  ; ZF = 0非空格 , 转移
                INC     BX                  ; ZF = 1是空格 , 个数加1
next :  INC     SI
                LOOP again                  ; 字符个数减1 , 不为0继续循环
                            -- DEC CX
                            -- JNZ again
```

子程序指令

子程序是完成特定功能的一段程序

当主程序（调用程序）需要执行这个功能时 ， 采用`CALL`调用指令转移到该子程序的起始处执行

当运行完子程序功能后 ， 采用`RET`返回指令回到主程序继续执行

CALL：子程序调用指令

`CALL`指令分成4种类型（类似`JMP`）

```
CALL label              ; 段内调用、相对寻址
CALL r16/m16            ; 段内调用、间接寻址
CALL far ptr label      ; 段间调用、直接寻址
CALL far ptr mem        ; 段间调用、间接寻址
```

CALL指令需要保存返回地址：

段内调用——入栈偏移地址IP

```
SP ← SP - 2  ,  SS:[SP]←IP
```

段间调用——入栈偏移地址IP和段地址CS

```
SP ← SP - 2  ,  SS:[SP]←IP
SP ← SP - 2  ,  SS:[SP]←CS
```

RET：子程序返回指令

根据段内和段间、有无参数，分成4种类型

```
RET          ; 无参数段内返回
RET i16      ; 有参数段内返回
RET          ; 无参数段间返回
RET i16      ; 有参数段间返回
```

需要弹出CALL指令压入堆栈的返回地址：

段内返回——出栈偏移地址IP

```
IP ← SS:[SP]  ,   SP ← SP + 2
```

段间返回——出栈偏移地址IP和段地址CS

```
IP ← SS:[SP]  ,  SP ← SP + 2
CS ← SS:[SP]  ,  SP ← SP + 2
```

返回指令`RET`的参数

```
RET i16    ; 有参数返回
```

RET指令可以带有一个立即数`i16`，则堆栈指针SP将增加 ， 即：

```
SP ← SP + i16
```

这个特点使得程序可以方便地废除若干执行`CALL`指令以前入栈的参数

```
例：子程序

; 主程序
                MOV      AL , 0fh    ; 提供参数AL
                CALL htoasc          ; 调用子程序
                ...

; 子程序：将AL低4位的一位16进制数转换成ASCII码
htoasc: AND      AL , 0fh    ; 只取AL的低4位
                OR       AL , 30h    ; AL高4位变成3
                CMP      AL , 39h    ; 是0~9 , 还是0AH~0Fh
                JBE  htoend
                ADD      AL , 7      ; 是0AH~0Fh , 加上7
htoend: RET                  ; 子程序返回
```

中断指令

中断（Interrupt ）是又一种改变程序执行顺序的方法

中断具有多种中断类型

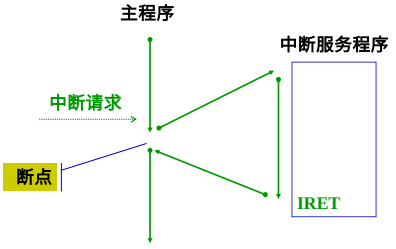
中断的指令有3条：`INT i8` `IRET` `INTO`

本节主要掌握类似子程序调用指令的中断调用指令`INT i8`，进而学习使用DOS功能调用

中断的过程

中断请求可以来自处理器外部的中断源 ， 也可以由处理器执行指令引起：

例如执行`INT i8`指令。



8086的中断

8086可以管理256个中断

各种中断用一个向量编号来区别

主要分成外部中断和内部中断：

外部中断——来自CPU之外的原因引起的中断 ， 又可以分成：

可屏蔽中断：可由CPU的中断允许标志`IF`控制

非屏蔽中断：不受CPU的中断允许标志`IF`控制

内部中断——CPU内部执行程序引起的中断 ， 又可以分成：

除法错中断：执行除法指令 ， 结果溢出产生的 **0 号中断**

指令中断：执行中断调用指令 `INT i8` 产生的 **i8 号中断**

断点中断：用于断点调试（`INT 3`）的 **3 号中断**

溢出中断：执行溢出中断指令 ， `OF = 1` 产生的 **4 号中断**

单步中断：`TF = 1` 在每条指令执行后产生的 **1 号中断**

INT：中断指令

```
INT i8     ; 中断调用指令：产生i8号中断
IRET       ; 中断返回指令：实现中断返回
INTO       ; 溢出中断指令：
                    ; 若溢出标志OF = 1  ,  产生4号中断
                    ; 否则顺序执行
```

串操作类指令

串操作指令是8086指令系统中比较独特的一类指令 ， 采用比较特殊的数据串寻址方式 ， 常用在操作主存连续区域的数据时

主要熟悉：`MOVS STOS LODS CMPS SCAS REP`

一般了解：`REPZ/REPE REPNZ/REPNE`

串数据类型

串操作指令的操作数是主存中连续存放的数据串（String）——即在连续的主存区域中 ， 字节或字的序列

串操作指令的操作对象是以字（W）为单位的字串 ， 或是以字节（B）为单位的字节串

串寻址方式

源操作数用寄存器SI寻址 ， 默认在数据段DS中 ， 但允许段超越：`DS:[SI]`

目的操作数用寄存器DI寻址 ， 默认在附加段ES中 ， 不允许段超越：`ES:[DI]`

每执行一次串操作指令 ， SI和DI将自动修改：

±1（对于字节串）或±2（对于字串）

执行指令CLD后 ， DF = 0 ， 地址指针增1或2

执行指令STD后 ， DF = 1 ， 地址指针减1或2

MOVS（MOVE string）：串传送

把字节或字操作数从主存的源地址传送至目的地址

```
MOVSB      ; 字节串传送：ES:[DI] ← DS:[SI]
                    ; SI ← SI ± 1  ,  DI ← DI ± 1
MOVSW      ; 字串传送   ：ES:[DI] ← DS:[SI]
           ; SI ← SI ± 2  ,  DI ← DI ± 2
```

例：字节串传送

```
; 判断传送次数CX是否为0
; 不为0 , 则到again位置执行指令
; 否则 , 结束
                MOV SI , offset source       ; offset是汇编操作符 , 求出变量的偏移地址
                MOV DI , offset destination
                MOV CX , 100                 ; CX ← 传送次数
                CLD                          ; 置DF = 0  ,  地址增加
again:  MOVSB                        ; 传送一个字节
                DEC CX                       ; 传送次数减1
                JNZ again
```

例：字串传送

```
; 判断传送次数CX是否为0
; 不为0 , 则到again位置执行指令
; 否则 , 结束
                MOV SI , offset source
                MOV DI , offset destination
                MOV CX , 50                   ; CX ← 传送次数
                CLD                           ; 置DF = 0  ,  地址增加
again:  MOVSW                          ; 传送一个字
                DEC CX                        ; 传送次数减1
                JNZ again
```

STOS（store string）：串存储

把AL或AX数据传送至目的地址

```
STOSB          ; 字节串存储：ES:[DI] ← AL
               ; DI ← DI ± 1
STOSW          ; 字串存储：ES:[DI] ← AX
               ; DI ← DI ± 2
```

例：串存储

```
                MOV AX , 0
                MOV DI , 0       ; DI为偶数即可
                MOV CX , 8000h   ; CX ← 传送次数（32×1024）
                CLd              ; DF = 0  ,  地址增加
again:  STOSW            ; 传送一个字
                DEC CX           ; 传送次数减1
                JNZ again        ; 传送次数CX是否为0
```

LODS（load string）：串读取

把指定主存单元的数据传送给AL或AX

```
LODSB     ; 字节串读取：AL ← DS:[SI]
                    ; SI ← SI ± 1
LODSW     ; 字串读取：AX ← DS:[SI]
          ; SI ← SI ± 2
```

```
例：串读取1

MOV SI , offset block
MOV DI , offset dplus
MOV BX , offset dminus
MOV AX , ds
MOV ES , AX               ; 数据都在一个段中  ,  所以设置ES = DS
MOV CX , count            ; CX ← 字节数
CLD
```

```
例：串读取2

go_on:  lodsb             ; 从block取出一个数据
                TEST AL , 80h     ; 检测符号位 , 判断是正是负
                JNZ minus         ; 符号位为1 , 是负数 , 转向minus
                STOSB             ; 符号位为0 , 是正数 , 存入dplus
                JMP again         ; 程序转移到again处继续执行
                JNZ go_on         ; 完成正负数据分离
```

```
例：串读取3
minus:  XCHG  BX , DI
                STOSB           ; 把负数存入dminus
                XCHG  BX , DI
again:  DEC   CX        ; 字节数减1
                JNZ go_on       ; 完成正负数据分离
```

CMPS（compare string）：串比较

将主存中的源操作数减去至目的操作数 , 以便设置标志 , 进而比较两操作数之间的关系

```
CMPSB     ; 字节串比较：DS:[SI] - ES:[DI]
                    ; SI ← SI ± 1 , DI ← DI ± 1
CMPSW     ; 字串比较：DS:[SI] - ES:[DI]
                    ; SI ← SI ± 2 , DI ← DI ± 2
```

```
例：比较字符串

                MOV SI , offset string1
                MOV DI , offset string2
                MOV CX , count
                CLD
again:  CMPSB               ; 比较两个字符
                JNZ  unmat          ; 有不同字符 , 转移
                DEC     CX
                JNZ  again          ; 进行下一个字符比较
```

```
                MOV     AL , 0        ; 字符串相等 , 设置00h
                JMP output            ; 转向output
unmat : MOV     AL , 0ffh     ; 设置ffh
output: MOV result , AL       ; 输出结果标记
```

SCAS（scan string）：串扫描

将`AL/AX`减去至目的操作数 , 以便设置标志 , 进而比较`AL/AX`与操作数之间的关系

```
SCASB     ; 字节串扫描：AL - ES:[DI]
                    ; DI ← DI ± 1
SCASW     ; 字串扫描   ：AX - ES:[DI]
                    ; DI ← DI ± 2
```

例：查找字符串

```
                MOV DI , offset string
                MOV AL , 20h
                MOV CX , count
                CLD
again:  SCASB             ; 搜索
                JZ    found     ; 为0（ZF=1）, 发现空格
                DEC       CX    ; 不是空格
                JNZ   again     ; 搜索下一个字符
                ...             ; 不含空格 , 则继续执行
found:  ...
```

重复前缀指令（repeat）

串操作指令执行一次 , 仅对数据串中的一个字节或字量进行操作。但是串操作指令前 , 都可以加一个重复前缀 , 实现串操作的重复执行。重复次数隐含在`CX`寄存器中

重复前缀分2类 , 3条指令：

- 配合**不影响标志**的`MOVS、STOS`（和`LODS`）指令的REP前缀
- 配合**影响标志**的`CMPS`和`SCAS`指令的`REPZ`和`REPNZ`前缀

REP：重复前缀指令

```
REP    ; 每执行一次串指令   ,   CX减1
             ; 直到CX＝0   ,   重复执行结束
```

REP前缀可以理解为：当数据串没有结束（CX≠0） , 则继续传送

例2.52和例2.53中 , 程序段的最后3条指令 , 可以分别替换为：`REP MOVSB`和`REP STOSW`

例：重复串传送（例2.52）

```
                MOV SI , offset source
                MOV DI , offset destination
```

```
                MOV CX , 100              ; CX ← 传送次数
                CLD
                REP MOVSB
-------------------------
again:  MOVSB       ; 传送一个字节
                DEC CX      ; 传送次数减1
                JNZ again   ; 判断传送次数CX是否为0
                                          ; 不为0（ZF = 0），则转移again位置执行
                                          ; 否则 ，结束
```

例：重复串存储（例2.53）

```
                MOV AX , 0
                MOV DI , 0
                MOV CX , 8000h
                CLD
                REP STOWS
------------------------
again:  STOSB      ; 传送一个字
                DEC CX     ; 传送次数减1
                JNZ again  ; 判断传送次数CX是否为0
```

REPZ：重复前缀指令

```
REPZ        ; 每执行一次串指令 ，CX减1
                        ; 并判断ZF是否为0 ，
                        ; 只要CX = 0或ZF = 0 ，重复执行结束
```

`REPZ/REPE`前缀可以理解为：当数据串没有结束（`CX≠0`），并且串相等（`ZF = 1`），则继续比较

REPNZ：重复前缀指令

```
REPNZ     ; 每执行一次串指令 ，CX减1
                    ; 并判断ZF是否为1 ，
                    ; 只要CX = 0或ZF = 1 ，重复执行结束
```

`REPNZ/REPNE`前缀可以理解为：当数据串没有结束（`CX≠0`），并且串不相等（`ZF=0`），则继续比较

例：比较字符串

```
                MOV     SI , offset string1
                MOV     DI , offset string2
                MOV     CX , count
                CLD
                REPZ cmpsb            ; 重复比较两个字符
                JNZ  unmat            ; 字符串不等 ，转移
                MOV     AL , 0        ; 字符串相等 ，设置00h
                JMP output            ; 转向output
unmat : MOV     AL , 0ffh     ; 设置ffh
output: MOV result , AL       ; 输出结果标记
```

指令`repz cmpsb`结束重复执行的情况

`ZF = 0`，即出现不相等的字符

`CX = 0`，即比较完所有字符：

这种情况下，如果`ZF = 0`，说明最后一个字符不等；而`ZF = 1`表示所有字符比较后都相等，也就是两个字符串相同

所以，重复比较结束后，`jnz unmat`指令的条件成立`ZF = 0`，字符串不相等

例：查找字符串

```
                MOV DI , offset string
                MOV AL , 20h
                MOV CX , count
                CLD
                REPNZ scasb     ; 搜索
                JZ    found     ; 为0（ZF=1），发现空格
                ...             ; 不含空格，则继续执行
found:  ...
```

处理机控制类指令

对CPU状态进行控制的指令

`NOP`

`CS:  SS:  DS:  ES:`

`LOCK  HLT  ESC  WAIT`

NOP：空操作指令

不执行任何操作，但**占用一个字节**存储单元，**空耗**一个指令**执行周期**

`NOP`常用于程序调试：

在**需要预留指令空间**时用`NOP`填充

代码空间多余时也可以用`NOP`填充

还可以用`NOP`实现**软件延时**

事实上，`NOP`和`XCHG AX , AX`的指令代码一样，都是`90H`

段超越前缀指令

在允许段超越的存储器操作数之前，使用段超越前缀指令，将采用指定的段寄存器寻址操作数

```
CS:     ; 使用 代码段 的数据
SS:     ; 使用 堆栈段 的数据
DS:     ; 使用 数据段 的数据
ES:     ; 使用 附加段 的数据
```

LOCK：封锁前缀指令

```
LOCK        ; 封锁总线
```

这是一个指令前缀 ， 可放在任何指令前

这个前缀使得在这个指令执行时间内 ， 8086 处理器的封锁输出引脚有效 ， 即把总线封锁 ， 使别的控制器不能控制总线 ; 直到该指令执行完后 ， 总线封锁解除

HLT：暂停指令

```
HLT  ; 进入暂停状态
```

暂停指令使CPU进入暂停状态 ， 这时CPU不进行任何操作。当CPU发生复位或来自外部的中断时 ， CPU脱离暂停状态

`HLT` 指令可用于程序中等待中断。当程序中必须等待中断时 ， 可用`HLT` ， 而不必用软件死循环。然后 ， 中断使CPU脱离暂停状态 ， 返回执行`HLT` 的下一条指令

ESC：交权指令

```
ESC  6位立即数 , reg/mem        ; 把浮点指令交给浮点处理器执行
```

浮点协处理器8087指令是与8086的整数指令组合在一起的 ， 当8086发现是一条浮点指令时 ， 就利用ESC指令将浮点指令交给8087执行

实际编写程序时 ， 一般采用易于理解的浮点指令助记符格式

```
ESC   6 , [SI]     ; 实数除法指令：FDIV dword ptr [SI]
ESC 20H ,  AL      ; 整数加法指令：FADD ST(0) , ST
```

WAIT：等待指令

```
WAIT        ; 进入等待状态
```

8086利用`WAIT` 指令和测试引脚实现与8087同步运行

浮点指令经由8086处理发往8087 ， 并与8086本身的整数指令在同一个指令序列 ; 而8087执行浮点指令较慢 ， 所以8086必须与8087保持同步

标志位

CBW