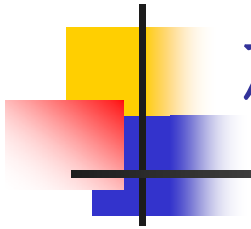
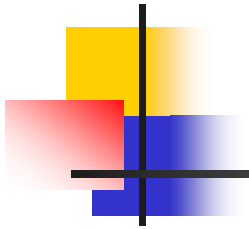


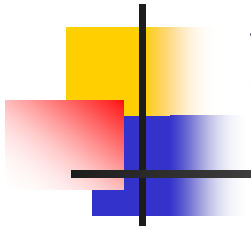
标志寄存器

- ZF标志
- PF标志
- SF标志
- CF标志
- OF标志
- adc指令
- sbb指令
- cmp指令
- 检测比较结果的条件转移指令
- DF标志和串传送指令
- pushf和popf
- 标志寄存器在Debug中的表示



引言

- **8086CPU的标志寄存器有16位，其中存储的信息通常被称为程序状态字（PSW）。**
- **我们已经使用过8086CPU的ax、bx、cx、dx、si、di、bp、sp、ip、cs、ss、ds、es等13个寄存器。**
- **本章中的标志寄存器（简称为flag）是最后一个寄存器。**

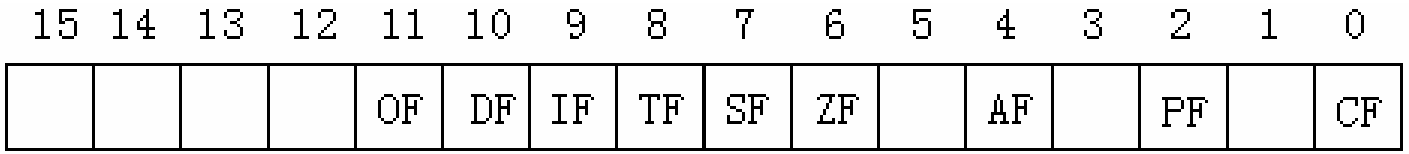


引言

- **flag** 和其他寄存器不一样，其他寄存器是用来存放数据的，整个寄存器具有一个含义。
- 而**flag**寄存器是按位起作用的，它的每一位都有专门的含义，记录特定的信息。

引言

- **8086CPU的flag寄存器的结构：**

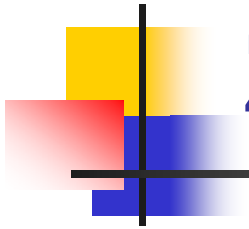


- **flag的1、3、5、12、13、14、15位在8086CPU中没有使用，不具有任何含义。而0、2、4、6、7、8、9、10、11位都具有特殊的含义。**

ZF标志(Zero Flag)

- flag的第6位是ZF，零标志位（全零标志）
- 如果ZF=1，表示操作结果全为零，否则ZF=0

mov ax,1
sub ax,1
结果为0
则ZF = 1
	mov ax,1
	and ax,0
	结果为0
	则ZF = 1
		mov ax, 2
		sub ax, 1
		结果为1
		则ZF = 0
			mov ax,1
			or ax,0
			结果为非零
			则ZF = 0



ZF标志

- 在8086CPU指令集中，影响标志寄存器的大都是运算指令（进行逻辑或算术运算），有：add、sub、mul、div、inc、or、and等
- 对标志寄存器没有影响的大都是传送指令，有：mov、push、pop等

PF标志(Parity Flag)

- flag的第2位是PF，奇偶标志位
- 如果PF = 1，表示操作结果中“1”的个数为偶数，否则PF = 0
- 该指令主要用于检查数据传送过程中的错误

```
mov al,1
add al,10
```
结果为00001011B

有3个1，则PF=0

```
mov al,1
or al,10
```
结果为00000011B

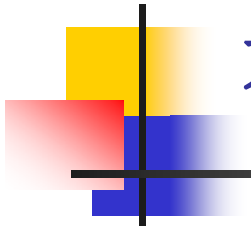
有2个1，则PF=1

SF标志(Sign Flag)

- flag的第7位SF，符号标志位
- 如果SF=1，表示符号数运算后的结果为负，否则SF=0
- mov al,10000001B

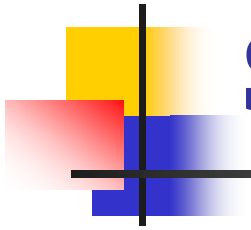
  add al,1

  结果： (al)=10000010B



有符号数与补码

- 计算机中的一个数据可以看作是有符号数，也可以看成是无符号数。
- 如：
  - **00000001B**，可以看作为无符号数 **1**，或有符号数＋**1**；
  - **10000001B**，可以看作为无符号数**129**，也可以看作有符号数**-127**。

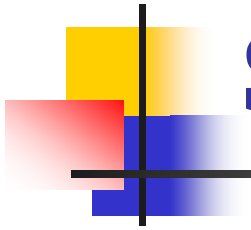


SF标志

- **add指令进行无符号数运算**

  129+1=130（10000010B）
- **进行有符号数的运算**

  -127+1=-126（10000010B）
- **CPU对有符号数运算结果的一种记录，它记录数据的正负**
- **如果将数据当作无符号数来运算，SF的值则没有意义**



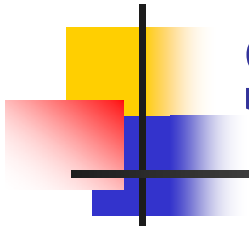
SF标志

- 比如：

  mov al,10000001B
  add al,1

  执行后，结果为**10000010B**，
  SF=1，

  表示：如果指令进行的是有符号数运算，那么结果为负；



SF标志

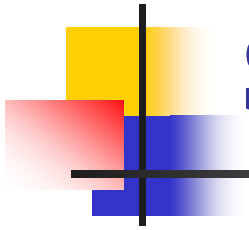
- 再比如：

  mov al,10000001B

  add al, 01111111B

  执行后，结果为**0**，**SF=0**，

  表示：如果指令进行的是有符号数运算，那么结果为非负。

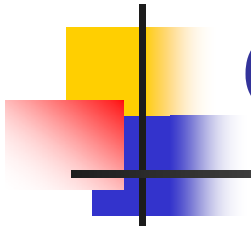


SF标志

- 某此指令影响多个标志位，为相关的处理提供了所需的依据
- 指令**sub al, al**执行后，**ZF**、**PF**、**SF**等标志位分别为：**1**、**1**、**0**

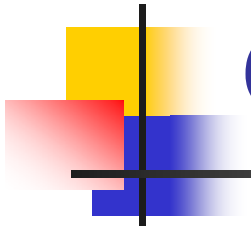
CF标志(Carry Flag)

- flag的第0位CF，进位标志位
- 如果CF=1，表示算术运算时产生进位或借位，否则CF=0
- 进行无符号数运算的时候，它记录运算结果向更高位的进位值，或从更高位的借位值



CF标志

- 如，两个8 位数据：98H+98H，将产生进位
- 8086CPU 用flag的CF位来记录这个进位值
- 如，两个 8 位数据：97H-98H，将产生借位，相当于计算197H-98H
- flag的CF位也可以用来记录这个借位值



CF标志

- 如指令：

```
mov al,98H
add al,al  ;执行后： (al)=30H，CF=1，
           ;CF记录了最高有效位向更高位的进位值
add al,al  ;执行后： (al)=60H，CF=0，
           ;CF记录了最高有效位向更高位的进位值
```

示例1：下列指令执行后，

```
MOV AL,0FFH
ADD AL,1
;AL= 0              CF= 1
```

AL的最高有效位想高位的进位值被复制到进位标志位中

示例2：下列指令执行后，

```
MOV AX,0FFH
ADD AX,1
;AX= 0100H        CF= 0
```

OF标志(Overflow Flag)

- 如果OF=1，表示进行算术运算时，结果超过了最大范围，否则OF=0
- 在进行有符号数运算的时候，如果结果超过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 位有符号数，机器所能表示的范围是-32768~32767

```
mov al, 98
add al, 99
```

CF=0，OF=1

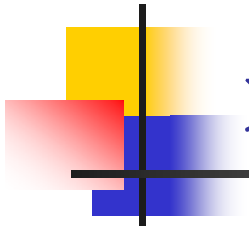
OF标志(Overflow Flag)

- 有符号数算术运算结果上溢（太大）或下溢（太小）以至于目的操作数无法容纳时溢出标志置位

```
mov al, +127
add al, 1
OF= 1
```

```
mov al, -128
sub al, 1
OF= 1
```

一个字节所能存储的最大有符号数整数是+127，再加1将导致上溢，-128从中减1将导致下溢

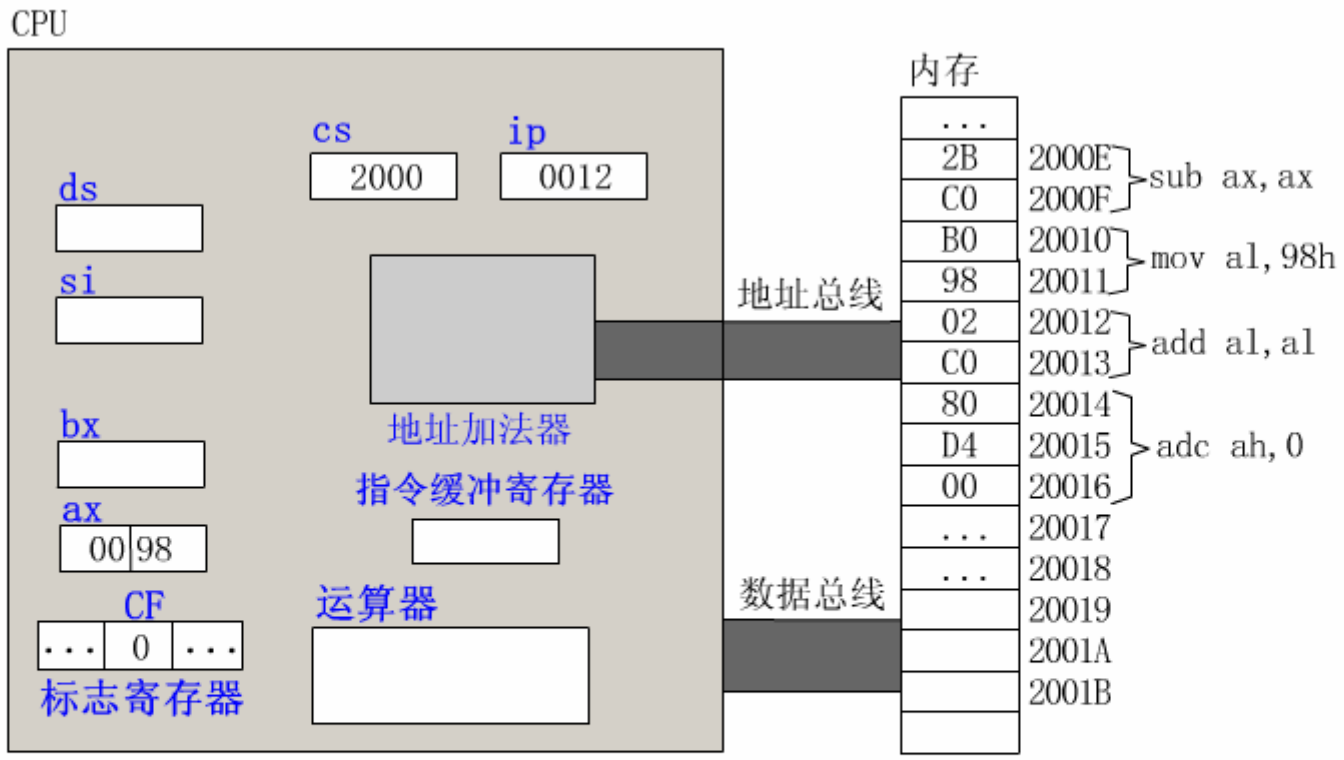


关于标志寄存器

- 无符号数运算：
  - 进位标志位
- 有符号数
  - 符号标志位、溢出标志位

带进位加法指令 adc (P59)

- 格式： **adc 操作对象1，操作对象2**
  - 功能：
    操作对象1=操作对象1+操作对象2+CF
  - 如： **adc ax, bx** 实现的功能是：
    (ax)=(ax)+(bx)+CF
- **adc**指令执行过程演示



adc指令的执行过程

play step step stop

```
mov ax, 2
mov bx, 1
sub bx, ax
adc ax, l
```

执行后，(ax)=4

执行时，相当于：

(ax)+1+CF=2+1+1=4

```
mov al, 98H
add al, al
adc al, 3
```

执行后，(al)=34H

执行时，相当于：

(al)+3+CF=30H+3+1=34H

```
mov ax, 1
add ax, ax
adc ax, 3
```

执行后，(ax)=5

执行时，相当于：

(ax)+3+CF=2+3+0=5

adc指令

- 两个数据：**0198H**和**0183H**如何相加的：

01 98
+ 01 83
1

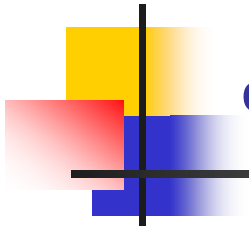
----

03 1B

可以看出，加法可以分两步来进行：

- （1）低位相加；
- （2）高位相加再加上低位相加产生的进位值

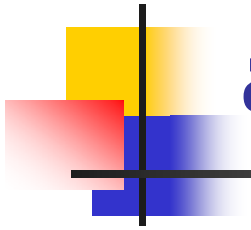
- **adc**指令和**add**指令相配合可以对更大的数据进行加法运算



adc指令

- 编程计算1EF000H+201000H，结果放在ax（高16位）和bx（低16位）中。

  分析

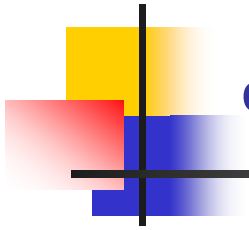


adc指令

- 分析：

  因为两个数据的位数都大于16，用add指令无法进行计算。我们将计算分两步进行，先将低16位相加，然后将高16 位和进位值相加。

  程序如下

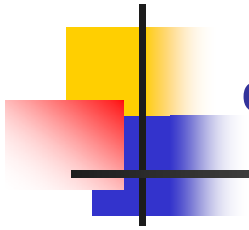


adc指令

- 程序：

```
mov ax, 001EH
mov bx, 0F000H
add bx, 1000H
adc ax, 0020H
```

adc 指令执行后，也可能产生进位值，所以也会对CF位进行设置。



adc指令

- 由于有这样的功能，我们就可以对任意大的数据进行加法运算。

- 看一个例子

adc指令

- 编程计算

  1EF0001000H+2010001EF0H，结果放在ax（高16位），bx（次高16位），cx（低16位）中。

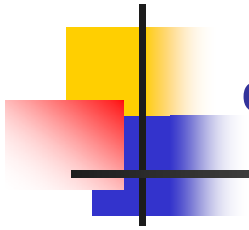
  分析

adc指令

- 分析：

  计算分3步进行：

  - （1）先将低16位相加，完成后，CF 中记录本次相加的进位值；
  - （2）再将次高16位和 CF（来自低16位的进位值）相加，完成后，CF中记录本次相加的进位值；
  - （3）最后高16 位和CF（来自次高16位的进位值）相加，完成后，CF中记录本次相加的进位值。



adc指令

- 程序代码

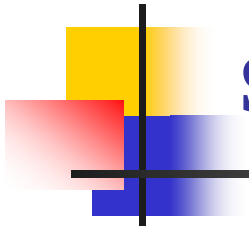
```
mov ax,001EH
mov bx,0F000H
mov cx,1000H
add cx,1EF0H
adc bx,1000H
adc ax,0020H
```

sbb指令 (P62)

- sbb是带借位减法指令
- 格式：**sbb 操作对象1, 操作对象2**
  - 功能：

    操作对象1=操作对象1–操作对象2–CF
  - 比如：**sbb ax,bx**

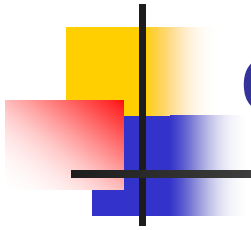
    实现功能： (ax) = (ax) – (bx) – CF



sbb指令

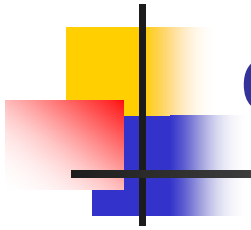
- 比如，计算**003E100OH–00202000H**，结果放在**ax**，**bx**中，程序如下：

```
mov bx,1000H
mov ax,003EH
sub bx,2000H
sub ax,0020H
```



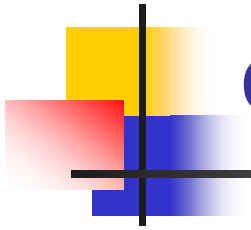
cmp指令 (P62)

- **cmp**：比较指令
- 相当于减法指令，只是不保存结果
- **cmp** 指令执行后，将对标志寄存器产生影响
- 其他相关指令通过识别这些被影响的标志寄存器位来得知比较结果



cmp指令

- cmp指令
  - 格式：**cmp 操作对象1, 操作对象2**
  - 功能：计算操作对象1–操作对象2
  - 并不保存结果，仅根据计算结果对标志寄存器进行设置



cmp指令

- 如：**cmp ax, ax**

  做**(ax)–(ax)**的运算，结果为**0**，但并不在**ax**中保存，仅影响**flag**的相关各位

  指令执行后：

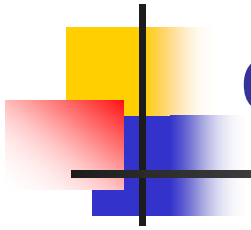
  ZF=1，

  PF=1，

  SF=0，

  CF=0，

  OF=0。



cmp指令

- 下面的指令：

  mov ax, 8

  mov bx, 3

  cmp ax, bx

  执行后： **(ax)=8，**

  ZF=0，

  PF=1，

  SF=0，

  CF=0，

  OF=0。

cmp指令

- **cmp 指令执行后，通过相关标志位的值可以看出比较的结果**
- **cmp ax, bx**

如果(ax)=(bx) 则(ax)-(bx)=0，所以：ZF=1；

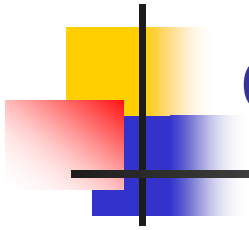
如果(ax)≠(bx) 则(ax)-(bx)≠0，所以：ZF=0；

如果(ax)<(bx) 则(ax)-(bx)将产生借位，所以：CF=1；

如果(ax)≥(bx) 则(ax)-(bx)不必借位，所以：CF=0；

如果(ax)>(bx) 则(ax)-(bx)既不必借位，结果又不为0，所以：CF=0并且ZF＝0；

如果(ax)≤(bx) 则(ax)-(bx)既可能借位，结果可能为0，所以：CF=1或ZF＝1。

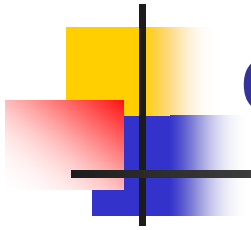


cmp指令

- 比如：

```
mov ah,22H
mov bh,0A0H
sub ah,bh
```

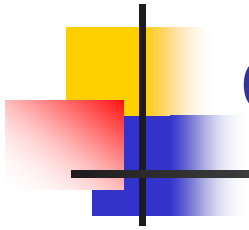
结果SF=1，运算实际得到的结果是(ah)=82H，但是在逻辑上，运算所应该得到的结果是：34-(-96)=130。



cmp指令

- 就是因为130 这个结果作为一个有符号数超出了-128~127这个范围，在ah 中不能表示，而ah中的结果被CPU当作有符号数解释为-126。
- 而SF被用来记录这个实际结果的正负，所以SF=1。

  但SF=1不能说明在逻辑上运算所得的正确结果的正负。



cmp指令

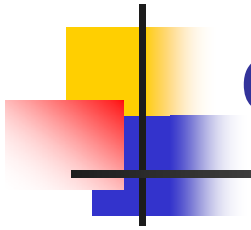
- 又比如：

  mov ah,0A0H

  mov bh,0CBH

  cmp ah,bh

结果SF=1，运算 (ah)-(bh) 实际得到的结果是D5H，但是在逻辑上，运算所应该得到的结果是：160-(-53)=213。



cmp指令

- 但是逻辑上的结果的正负，才是**cmp**指令所求的真正结果，因为我们就是要靠它得到两个操作对象的比较信息。
- 所以**cmp** 指令所作的比较结果，不是仅仅靠**SF**就能记录的，因为它只能记录实际结果的正负。

检测比较结果的条件转移指令

- 因为 **cmp** 指令可以同时进行两种比较，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，即：
  - 根据无符号数的比较结果进行转移的条件转移指令，它们检测**ZF**、**CF**的值；
  - 和根据有符号数的比较结果进行转移的条件转移指令，它们检测 **SF**、**OF**和 **ZF** 的值。

检测比较结果的条件转移指令

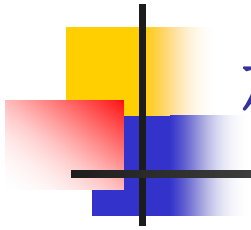
- 下表是常用的根据无符号数的比较结果进行转移的条件转移指令**(P87)**。

条件转移指令小结

指令	含义	检测的相关标志位
je	等于则转移	ZF = 1
jne	不等于则转移	ZF = 0
jb	低于则转移	CF = 1
jnb	不低于则转移	CF = 0
ja	高于则转移	CF = 0，ZF = 0
jna	不高于则转移	CF = 1或ZF = 1

检测比较结果的条件转移指令

- 这些指令比较常用，它们都很好记忆，它们的第一个字母都是j，表示**jump**；后面的：
  - **e**：表示**equal**；
  - **ne**：表示**not equal**；
  - **b**：表示**below**；
  - **nb**：表示**not below**；
  - **a**：表示**above**；
  - **na**：表示**not above**。



检测比较结果的条件转移指令

- 注意观察一下它们所检测的标志位，都是**cmp**指令进行无符号数比较时候，记录比较结果的标志位。
- 比如**je**，检测 **ZF**位，当 **ZF=1**的时候进行转移，如果在 **je** 前面使用了 **cmp** 指令，那么**je**对**ZF**的检测，实际上就是间接地检测**cmp**的比较结果是否为两数相等。

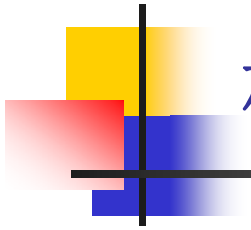
检测比较结果的条件转移指令

- 编程实现如下功能：

如果 (ah) = (bh) , 则 (ah) = (ah) + (ah)，否则
(ah) = (ah) + (bh)。

```
    cmp ah,bh
    je s
    add ah,bh
    jmp short ok
  s: add ah,ah
ok: ret
```

（说明：jmp，无条件跳转指令，P85）



检测比较结果的条件转移指令

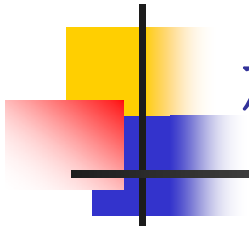
- **je** 的逻辑含义是“相等则转移”，但它进行的操作是，**ZF=1**时则转移。
- “相等则转移”这种逻辑含义，是通过和 **cmp** 指令配合使用来体现的，因为是**cmp** 指令为“**ZF=1**”赋予了“两数相等”的含义。

检测比较结果的条件转移指令

- 至于究竟在**je**之前使不使用**cmp**指令，在于我们的安排。

  je检测的是**ZF**位置，不管**je** 前面是什么指令，只要**CPU**执行**je**指令时，**ZF=1**，那么就会发生转移。

- 比如

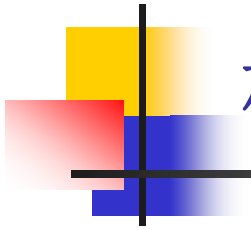


检测比较结果的条件转移指令

- 比如：

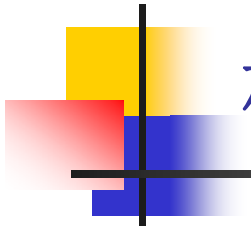
```
    mov ax,0
    add ax,0
    je s
    inc ax
  s: inc ax
```

执行后，(ax)=1。add ax,0使得ZF=1，所以je指令将进行转移。



检测比较结果的条件转移指令

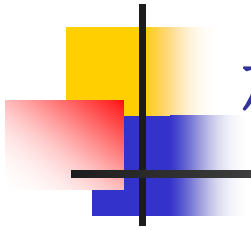
- 可在这个时候发生的转移确不带有“相等则转移”的含义。因为此处的**je**指令检测到的**ZF=1**，不是由**cmp**等比较指令设置的，而是由**add**指令设置的，并不具有“两数相等”的含义。
- 但无论“**ZF=1**” 的含义如何，是什么指令设置的，只要是**ZF=1**，就可以使得**je**指令发生转移。



检测比较结果的条件转移指令

- **CPU提供了cmp指令，也提供了 je 等条件转移指令，如果将它们配合使用，可以实现根据比较结果进行转移的功能。**
- **但这只是“如果”，只是一种合理的建议，和事实上常用的方法。**
- **但究竟是否配合使用它们，完全是你自己的事情。**

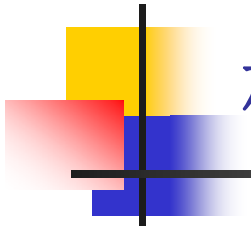
这就好像，call和ret指令的关系一样。



检测比较结果的条件转移指令

- 对于**jne**、**jb**、**jnb**、**ja**、**jna**等指令和**cmp**指令配合使用的思想和**je**相同，可以自己分析一下。
- 虽然我们分别讨论了**cmp**指令和与其比较结果相关的有条件转移指令，但是它们经常在一起配合使用。

所以我们在联合应用它们的时候，不必再考虑**cmp**指令对相关标志位的影响和**je** 等指令对相关标志位的检测。



检测比较结果的条件转移指令

- 因为相关的标志位，只是为**cmp**和**je**等指令传递比较结果。
- 我们可以直接考虑**cmp**和**je**等指令配合使用时，表现出来的逻辑含义。
- 它们在联合使用的时候表现出来的功能有些像高级语言中的**IF**语句。

检测比较结果的条件转移指令

- 我们来看一组程序：
  data段中的8个字节如下：

```
data segment
   db 8,11,8,1,8,5,63,38
data ends
```

  - （1）编程：统计data段中数值为8的字节的个数，用ax保存统计结果。
  - （2）编程：统计data段中数值大于8的字节的个数，用ax保存统计结果。
  - （3）编程：统计data段中数值小于8的字节的个数，用ax保存统计结果。

检测比较结果的条件转移指令

- （1）编程：统计data段中数值为8的字节的个数，用ax保存统计结果。
- 编程思路：初始设置(ax)=0，然后用循环依次比较每个字节的值，找到一个和8相等的数就将ax的值加1。
  - 程序如下
  - 另一种实现方式

检测比较结果的条件转移指令

```
        mov ax,data
        mov ds,ax
        mov bx,0        ;ds:bx指向第一个字节
        mov ax,0        ;初始化累加器
        mov cx,0
    s: cmp byte ptr [bx],8  ;和8进行比较
        jne next        ;如果不相等转到next，继续循环
        inc ax          ;如果相等就将计数值加1
next: inc bx
        loop s          ;程序执行后： (ax)=3
```

检测比较结果的条件转移指令

```
      mov ax,data
      mov ds,ax
      mov bx,0          ;ds:bx指向第一个字节
      mov ax,0          ;初始化累加器
      mov cx,0
   s: cmp byte ptr [bx],8  ;和8进行比较
      je ok              ;如果相等就转到ok，继续循环
      jmp short next ;如果不相等就转到next，继续循环
  ok: inc ax             ;如果相等就将计数值加1
next: inc bx
      loop s
```

检测比较结果的条件转移指令

- 比起第一个程序，它直接的遵循了“等于8则计数值加1”的原则，用je指令检测等于8的情况，但是没有第一个程序精简。
- 第一个程序用 **jne** 检测不等于 **8** 的情况，从而间接地检测等于 **8** 的情况。

要注意在使用 **cmp** 和条件转移指令时的这种编程思想。

检测比较结果的条件转移指令

- （2）编程：统计data段中数值大于8的字节的个数，用ax保存统计结果。
- 编程思路：初始设置(ax)=0，然后用循环依次比较每个字节的值，找到一个大于8的数就将ax的值加1。
  - 程序如下

检测比较结果的条件转移指令

```
        mov ax,data
        mov ds,ax
        mov bx,0        ;ds:bx指向第一个字节
        mov ax,0        ;初始化累加器
        mov cx,0
    s: cmp byte ptr [bx],8  ;和8进行比较
        jne next        ;如果不大于8转到next，继续循环
        inc ax          ;如果大于8就将计数值加1
next: inc bx
        loop s          ;程序执行后： (ax)=3
```

检测比较结果的条件转移指令

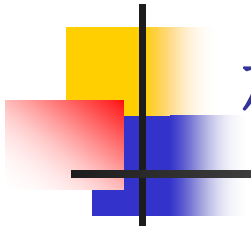
- （3）编程：统计data段中数值小于8的字节的个数，用ax保存统计结果。
- 编程思路：初始设置(ax)=0，然后用循环依次比较每个字节的值，找到一个小于8的数就将ax的值加1。
  - 程序如下

检测比较结果的条件转移指令

```
        mov ax,data
        mov ds,ax
        mov bx,0        ;ds:bx指向第一个字节
        mov ax,0        ;初始化累加器
        mov cx,0
    s: cmp byte ptr [bx],8  ;和8进行比较
        jnb next        ;如果不小于8转到next，继续循环
        inc ax          ;如果小于8就将计数值加1
next: inc bx
        loop s          ;程序执行后：  (ax)=2
```

检测比较结果的条件转移指令

- 上面讲解了根据无符号数的比较结果进行转移的条件转移指令。
- 根据有符号数的比较结果进行转移的条件转移指令的工作原理和无符号的相同，只是检测了不同的标志位。



检测比较结果的条件转移指令

- 我们在这里主要探讨的是**cmp**、标志寄存器的相关位、条件转移指令三者配合应用的原理，这个原理具有普遍性，而不是逐条讲解条件转移指令。
- 对这些指令感兴趣的学习者可以查看相关的指令手册。

DF标志(Direction Flag)

- flag的第10位是DF，方向标志位
- 如果DF=1，表示执行字符串操作时按照从高地址向低地址方向进行，否则DF=0
- 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

DF标志和串传送指令

- 格式1： **movsb** （P76）
- 功能：（以字节为单位传送）
  - **(1) ((es)×16 + (di)) = ((ds) ×16 + (si))**
  - **(2) 如果DF = 0则： (si) = (si) + 1**

    (di) = (di) + 1

    如果DF = 0则： (si) = (si) - 1

    (di) = (di) - 1

DF标志和串传送指令

- 我们可以用汇编语法描述**movsb**的功能如下：

  mov es:[di],byte ptr ds:[si];8086 并不支持这样的指令，这里只是个描述。

  - 如果**DF=0**：**inc si**

    inc di

  - 如果**DF=1**：**dec si**

    dec di

DF标志和串传送指令

- 可以看出，**movsb** 的功能是将 **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。
- 当然，也可以传送一个字， **movsw** 指令

DF标志和串传送指令

- 格式2：**movsw** **(P76)**
- 功能：（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。

DF标志和串传送指令

- 我们可以用汇编语法描述**movsw**的功能如下：

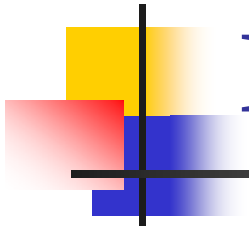
  mov es:[di],word ptr ds:[si];8086 并不支持这样的指令，这里只是个描述。

  - 如果**DF=0**：**add si,2**

    add di,2

  - 如果**DF=1**：**sub si,2**

    sub di,2

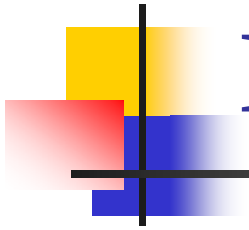


DF标志和串传送指令

- **movsb和movsw进行的是串传送操作中的一个步骤，一般来说，movsb 和 movsw 都和rep配合使用，格式如下：**
  - **rep movsb**
    用汇编语法来描述rep movsb的功能就是：
    s : movsb
    loop s
  - **rep movsw**
    用汇编语法来描述rep movsw的功能就是：
    s : movsw
    loop s

DF标志和串传送指令

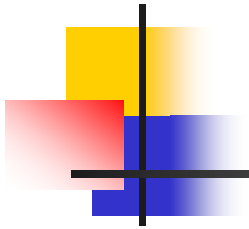
- 可见，**rep**的作用是根据**cx**的值，重复执行后面的串传送指令。
- 由于每执行一次**movsb**指令**si**和**di**都会递增或递减指向后一个单元或前个单元，则**rep movsb**就可以循环实现**(cx)**个字符的传送。



DF标志和串传送指令

- 由于**flag**的**DF**位决定着串传送指令执行后，**si**和**di**改变的方向，

  所以**CPU**应该提供相应的指令来对**DF**位进行设置，从而使程序员能够决定传送的方向。



- **8086CPU提供下而两条指令对DF位进行设置：**
  - **cld指令：将标志寄存器的DF位置0**
  - **std指令：将标志寄存器的DF位置1**
- 我们来看两个程序
  - 编程1
  - 编程2

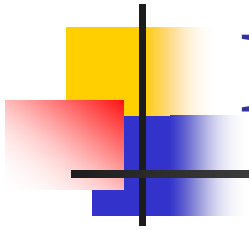
DF标志和串传送指令

- 编程：

  用串传送指令，将data段中的第一个字符串复制到它后面的空间中。

```
data segment
 db ‘Welcome to masm！’
 db 16 dup (0)
data ends
```

- 我们分析一下



DF标志和串传送指令

- 我们分析一下，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**。

DF标志和串传送指令

- 在这个问题中，这些信息如下：

  ① 传送的原始位置：**data:0**；

  ② 传送的目的位置：**data:16**；

  ③ 传送的长度：**16**；

  ④ 传送的方向： 因为正向传送（每次串传送指令执行后，**si**和**di** 递增）比较方便，所以设置**DF=0**。

  明确了这些信息之后，我们来编写程序

DF标志和串传送指令

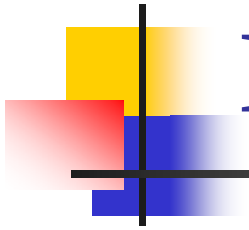
```
mov ax,data
mov ds,ax
mov si,0        ;ds:si指向data:0
mov es,ax
mov di,16       ;es:di指向data:16
mov cx ,16      ;(cx)=16，rep循环16次
cld             ;设置DF=0，正向传送
rep movsb
```

DF标志和串传送指令

- 编程：用串传送指令，将F000H段中的最后16个字符复制到data段中。

```
data segment
    db 16 dup (0)
data ends
```

- 我们分析一下



DF标志和串传送指令

- 我们还是先来看一下应该为串传送指令提供什么样的信息：
  - 要传送的字符串位于**F000H**段的最后**16**个单元中，那么它的最后一个字符的位置：**F000:FFFF**，是显而易见的。
  - 我们可以将**ds:si**指向 **F000H**段的最后一个单元，将**es:di**指向**data**段中的最后一个单元，然后逆向（即从高地址向低地址）传送**16**个字节即可。

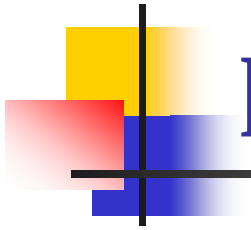
DF标志和串传送指令

- 相关信息如下：
  - ① 传送的原始位置：**F000:FFFF**；
  - ② 传送的目的位置：**data:15**；
  - ③ 传送的长度：**16**；
  - ④ 传送的方向：因为逆向传送（每次串传送指令执行后，**si** 和 **di** 递减）比较方便，所以设置**DF=1**。

程序代码

DF标志和串传送指令

```
mov ax,0f000h
mov ds,ax
mov si,0ffffh ;ds:si指向f000:ffff
mov ax,data
mov es,ax
mov di,15     ;es:di指向data:15
mov cx ,16    ;(cx)=16，rep循环16次
std           ;设置DF=1，逆向传送
rep movsb
```



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。

标志寄存器在Debug中的表示

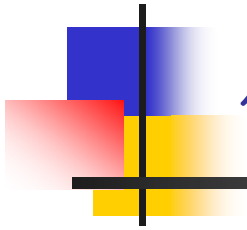
- 在Debug中，标志寄存器是按照有意义的各个标志位单独表示的。
- 在Debug中，我们可以看到下面的信息：

```
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=★★★★ ES=★★★★ SS=★★★★ CS=★★★★ IP=0100  NV UP EI PL NZ NA PO NC
                                           ↑  ↑     ↑  ↑     ↑  ↑
                                          OF DF    SF ZF    PF CF
```

标志寄存器在Debug中的表示

- 下面列出Debug对我们已知的标志位的表示：

标志	值为1的标记	值为0的标记
OF	OV	NV
SF	NG	PL
ZF	ZR	NZ
PF	PE	PO
CF	CY	NC
DF	DN	UP



小结