



北京师范大学
BEIJING NORMAL UNIVERSITY

信息科学与技术学院

《汇编语言》 课件



第11章 标志寄存器

- 11.1 ZF标志
- 11.2 PF标志
- 11.3 SF标志
- 11.4 CF标志
- 11.5 OF标志
- 11.6 adc指令
- 11.7 sbb指令
- 11.8 cmp指令
- 11.9 检测比较结果的条件转移指令
- 11.10 DF标志和串传送指令
- 11.11 pushf和popf
- 11.12 标志寄存器在Debug中的表示



引言

- 8086CPU的标志寄存器有16位~ 程序状态字（PSW）。
- 已经使用过8086CPU的ax、bx、cx、dx、si、di、bp、sp、ip、cs、ss、ds、es等13个寄存器了。
- 本章学习：标志寄存器（简称为flag）



引言

■ 8086CPU的flag寄存器的结构：

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

- 标志位表示一些指令执行的结果
 - ▣ OF、DF、IF、TF、SF、ZF、AF、PF、CF
 - 未标记的位在8086CPU中未使用到
-
- flag 和其他寄存器不同：
 - flag寄存器是按位起作用的
 - 其他寄存器是整个寄存器具有一个含义。



11.1 ZF标志

- ZF (zero flag), 零标志位。

相关指令执行后,

- 若结果为0, 则ZF = 1 (~真)
- 若结果非0, 则ZF = 0 (~假)

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



11.1 ZF标志

■ 示例

指令	mov ax,1 sub ax,1	mov ax,2 sub ax,1	mov ax,1 and ax,0	mov ax,1 or ax,0
执行后的 ZF	1	0	1	0



11.1 ZF标志

- 影响ZF的指令，**运算指令**（算术和逻辑运算）：
add、sub、mul、div、inc、or、and等；
- 不影响ZF的指令，**传送指令**：
mov、push、pop等
- 注意：使用指令时，要注意指令的全部功能——
包括对标记寄存器的哪些标志位的影响。



11.2 PF标志

- PF (Partial flag), 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。

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



11.2 PF标志

■ 示例

■ 指令: `mov al,1`
`add al,10`

结果为00001011B, 其中有3 (奇数) 个1, 则PF=0;

■ 指令: `mov al,1`
`or al,10`

结果为00000011B, 其中有2 (偶数) 个1, 则PF=1;



11.3 SF标志

- SF (sign flag), 符号标志位。

它记录指令执行后,

- 结果为负, $SF = 1$;
- 结果为非负, $SF = 0$ 。

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



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（用补码表示），也可以看成是无符号数。

如：

	二进制数	无符号数	有符号数
al	00000001B	1	+1
bl	10000001B	129	-127
add al, bl	10000010B	130	-126
说明	二进制加法	无符号数加法	有符号数加法

- CPU执行add等本质是二进制加法，同时可表示为有/无符号加法——取决于程序需要哪一种结果。



11.3 SF标志

- CPU在执行 `add` 等指令时，无论数据本身是**有符号数**或者**无符号数**，CPU按照总会有符号数加法结果来设影响到SF标志位的值的。
 - 对于**有符号数**，可以通过**SF**来得知结果的正负。
 - 对于**无符号数**，则**SF**的值则没有意义
 - ▣ 虽无符号数的操作指令影响了它的值。



11.3 SF标志

■ 如：

■ `mov al,10000001B`
`add al,1`

执行后，结果为 `10000010B`，`SF=1`，
表示：若指令进行的是有符号数运算，则结果为负；

■ `mov al,10000001B`
`add al,01111111B`

执行后，结果为 `00000000B`，`SF=0`，
表示：若指令进行的是有符号数运算，则结果为非负。



11.3 SF标志

- 某此指令将影响标志寄存器中的多个标志位，这些被影响的标记位比较全面地记录了指令的执行结果，为相关的处理提供了所需的依据。

■ 比如指令 `sub al,al`
执行后，
 $ZF = \underline{\quad 1 \quad}$
 $PF = \underline{\quad 1 \quad}$
 $SF = \underline{\quad 0 \quad}$



特别提示

- 检测点11.1 (p205)
- 没有完成此检测点，请不要向下进行。



11.4 CF标志

CF (carry flag), 进位标志位。

- 在进行无符号数运算的时候，它记录了运算结果的最高有效位向(从)更高位的进位（借位）值。

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



11.4 CF标志

- 位数为N的**无符号数**的二进制信息的**最高有效位**为第N-1位，而假想存在的**更高位**第N位。



- 用**CF**位来记录这个进/借位信息。
 - 当两数相加时，可能产生从向**更高位**的**进位**。
 - 当两数相减时，可能向更高位**借位**



11.4 CF标志

- 用CF位来记录这个进/借位信息。
 - 当两数相加时，可能产生从向更高位的进位。
如，`mov al,98H`
`add al, al` ;执行后 (al)=30H，因有进位CF=1，
 - 当两数相减时，可能向更高位借位
如，`97H-98H`，
将产生借位，借位后，相当于计算197H-98H。
CF位会记录这个借位值。



11.5 OF标志

- **溢出**问题：在进行**有符号数**运算的时候，若结果超过了机器所能表示的范围称为**溢出**。
 - ▣ 8 位有符号数范围：-128~127
 - ▣ 16 位有符号数范围：-32768~32767。

有符号数运算时发生**溢出**会导致运算结果不正确。



11.5 OF标志

- 有符号数运算时发生溢出会导致运算结果不正确。

示例1 `mov al,98`
 `add al,99`

运算后 (al) = -59

	2进制（补码）	16进制	10进制
	0110 0010 B	62H	98
	0110 0011 B	63H	99
求和结果	1100 0101 B	C5H	-59



11.5 OF标志

- 有符号数运算时发生溢出会导致运算结果不正确。

示例2 `mov al,0F0H`
 `add al,88H ;`

运算后 (al) = 120

	2进制 (补码)	16进制	10进制
	1111 0000 B	F0H	-16
	1000 1000 B	88H	-15
求和结果	0111 1000 B	78H	120



11.5 OF标志

■ OF (Overflow flag) 溢出标志

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，
- 如果没有溢出，OF=0。

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



11.5 OF标志

- 一定要注意CF和OF的区别：
 - CF是对无符号数运算有意义的标志位；
 - OF是对有符号数运算有意义的标志位。



11.5 OF标志

- CPU将add等指令既当无符号数运算和又当有符号数运算。
 - CF位记录无符号数运算是否产生了进位；
 - OF位记录有符号数运算是否产生了溢出；
 - SF记录结果正负。

	CF 无符号数	OF 有符号数	SF 有符号数
mov al,0F0H add al,88H	1 (240+135)	1 (-16)+(-120)	0
mov al,0F0H add al,78H	1 (240+120)	0 (-16)+(120)	0



特别提示

■ 检测点11.2 (page219)

写出下面每条指令执行后，ZF、PF、SF、CF、OF 等标志位的值。

	CF	OF	SF	ZF	PF
sub al,al					
mov al,10H					
add al,90H					
mov al,80H					
add al,80H					
mov al,0FCH					
add al,05H					
mov al,7DH					
add al,0BH					

没有完成此检测点，请不要向下进行。



11.6 adc指令

- **adc**是带进位加法指令，它利用了CF位上记录的进位值。
 - 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
 - 助记：**adc** (**ad**d with **c**arry flag)

比如：**adc ax,bx**

实现的功能： $(ax)=(ax)+(bx)+CF$



11.6 adc指令

- adc指令示例：以下各段代码执行后结果

	<pre>mov ax, 2 mov bx, 1 sub bx, ax adc ax, 1</pre>	<pre>mov ax, 1 add ax, ax adc ax, 3</pre>	<pre>mov al, 98H add al, al adc al, 3</pre>
结果	(ax) = ?	(ax) = ?	(al) = ?
解释	$(ax) + 1 + CF$ $= 2 + 1 + 1$ $= 4$	$(ax) + 3 + CF$ $= 2 + 3 + 0$ $= 5$	$(ax) + 3 + CF$ $= 30H + 3 + 1$ $= 34H$



11.6 adc指令

■ adc指令有什么用？

■ 先分析16位数的加法

0198H和0183H相加

```
  01 98
+ 01 83
-----
```

03 1B

add ax , bx

功能等价

```
add al,bl
adc ah,bh
```

加法过程可分两步来进行：

- (1) 低8位相加；
- (2) 高8位相加再加上低8位相加时产生的进位值。



11.6 adc指令

■ adc指令有什么用？

答：adc指令和add指令相配合就可以对更大（数位更宽）的数据进行加法运算。

能否在16位的8086CPU上进行32位宽数据加法？



11.6 adc指令

- 例：编程计算 $1\text{EF}000\text{H} + 201000\text{H}$ ，结果放在 **ax**（高16位）和 **bx**（低16位）中。

分析：

因两个数据的位数都大于16，分两步计算：

(1) 先将低16位相加，进位值记录在 **CF** 中；

(2) 将高16位及 **CF** 相加，进位值记录在 **CF** 中。

程序代码

```
mov ax,001EH  
mov bx,0F000H
```

```
add bx,1000H
```

```
adc ax,0020H
```



11.6 adc指令

- 能否实验32位以上位宽数据的加法 

答：可以。因为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。因此，可以对任意大的数据进行加法运算。



11.6 adc指令

- 例：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：

计算分3步进行：

(1) 将低16位相加，进位值记录在**CF**中；

(2) 将次高16位及 **CF**相加，进位置记录在**CF**中；

(3) 高16位及**CF**相加，进位置记录在**CF**中。

程序代码

```
mov ax,001EH  
mov bx,0F000H  
mov cx,1000H
```

```
add cx,1EF0H
```

```
adc bx,1000H
```

```
adc ax,0020H
```




11.6 adc指令

- 编写一个子程序，对两个128位数据进行相加。
 - 名称：add128
 - 功能：两个128位数据进行相加
 - 参数：
 - ▣ **ds:si**指向存储第一个数的内存空间，因数据为128位，所以需要8个字单元，由低地址单元到高地址单元依次存放128位数据由低到高的各个字。运算结果存储在第一个数的存储空间中。
 - ▣ **ds:di**指向存储第二个数的内存空间



11.6 adc指令

■ 程序代码

```
add128: push ax
        push cx
        push si
        push di
```



思考：

能否用 `add si,2`
`add di,2`

取代这4条指令

```
sub ax,ax ;将 CF 设置为 0
```

```
mov cx,8
s: mov ax,[si]
  adc ax,[di]
  mov [si],ax
  inc si
  inc si
  inc di
  inc di
  loop s
```

提示： inc和loop指令不影响CF位

```
pop di
pop si
pop cx
pop ax
ret
```



11.7 sbb指令

- **sbb**是带借位（CF位）的减法指令。
 - 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
 - 助记：**sbb**（**s**ubtract with **b**orrow flag）

如：**sbb ax,bx**

实现功能： $(ax) = (ax) - (bx) - CF$

- **sbb**和**adc**的用法类似。



11.7 sbb指令

- 可以在16位的8086上实现32位以上数据的减法吗？

答：可以。利用sbb指令可对任意大的数做减法运算。

- 例：计算003E1000H-00202000H，结果放在ax，bx中

程序如下：

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



11.8 cmp指令

■ 比较指令cmp

- 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，根据计算结果设置标志寄存器。

注意： 结果不会保存到操作对象1中。

	General Register	Memory Location	Constant
General Register	yes	yes	yes
Memory Location	yes	no	yes



11.8 cmp指令

■ 示例：

■ 如： `cmp ax,ax`

因 $(ax)-(ax)$ 的等于0，修改flag的相关位：

$ZF=1$, $PF=1$, $SF=0$, $CF=0$, $OF=0$ 。

`ax`中的值保持原值不变。

■ 如，下面的指令： `mov ax,8`

`mov bx,3`

`cmp ax,bx`

执行后 $ZF=0$, $PF=1$, $SF=0$, $CF=0$, $OF=0$ 。

$(ax)=8$,



11.8 cmp指令

- cmp 指令执行后，相关标志位的值反映了比较结果。

如：cmp ax, bx

	ZF	CF
(ax)<(bx)	0	1
(ax)=(bx)	1	0
(ax)>(bx)	0	0

逻辑

检测标志位

$(ax) = (bx) \iff ZF=1$

$(ax) \neq (bx) \iff ZF=0$

$(ax) < (bx) \iff CF=1$

$(ax) \geq (bx) \iff CF=0$

$(ax) > (bx) \iff CF=0 \text{ and } ZF=0$

$(ax) \leq (bx) \iff CF=1 \text{ or } ZF = 1$



11.8 cmp指令

- 类似于 `add`、`sub` 指令，`cmp`指令有两种含义：
 - 无符号数运算
 - 有符号数运算。



11.8 cmp指令

- 如何判断两有符号数的 $=$ 和 \neq 关系？

示例： `cmp ah,bh`,

如果 $(ah)=(bh)$ 则 $(ah)-(bh)=0$ ，则 $ZF=1$ ；

如果 $(ah)\neq(bh)$ 则 $(ah)-(bh)\neq 0$ ，则 $ZF=0$ ；

答：根据 `cmp` 指令执行后 ZF 的值判断：


若 $ZF = 1$ ，两有符号数是相等；

若 $ZF = 0$ ，两有符号数是不等。



11.8 cmp指令

■ 如何判断 $A < B$

$A < B$  等价 $A-B$ 是一个 负数

如何根据cmp指令执行后的标志位的值来判断



能否说 $SF=1$ 即表示 $A < B$?

- SF 记录了运算结果的正负，不是逻辑结果的正负。
 - ▣ 比如 `add ah, al` 执行后， SF 记录的是 $(ah) - (bh)$ 所得到的8位结果数据（此数据不保存）的正负；



11.8 cmp指令

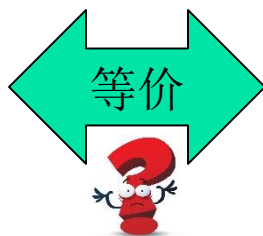
■ 示例: `cmp ah,bh`

代码	进制	(ah)	(bh)	(ah)-(bh)	SF
mov ah, 1 mov bh, 2 com ah, bh	2进制	0000 0001	0000 0010	1111 1111	1
	16进制	01H	02H	0FFH	
	10进制	1	2	-1	
mov ah, -2 mov bh, -1 com ah, bh	2进制	11111110	1111 1111	11111111	1
	16进制	0FEH	0FFH	01H	
	10进制	-2	-1	-1	



是否

SF=1



(ah)<(bh)



11.8 cmp指令

■ 示例: cmp ah,bh

		(ah)	(bh)	(ah)-(bh)	SF
mov ah, 22H mov bh, 0A0H cmp ah, bh	2进制	0010 0010	1010 0000	1000 0010	1
	16进制	22H	0A0H	82H	
	10进制	34	-96	-126 溢出	
mov ah, 08AH mov bh, 070H cmp ah, bh	2进制	1000 1010	0111 0000	0000 1010	1
	16进制	08AH	070H	1AH	
	10进制	-118	112	26 溢出	



是否

SF=1



等价

(ah)<(bh)



11.8 cmp指令

- **运算结果**正负（SF）和**逻辑结果**的正负有何关系？
 - 如果**未发生溢出**，则两者一致。
 - 如果**有发生溢出**，则两者相反。

需要同时考虑SF（运行结果的正负）考察OF（溢出情况）才能知道逻辑结果的正负。



11.8 cmp指令

- 示例：以 `cmp ah, bh` 为例，总结执行 `cmp` 指令后，`SF`和`OF`的值是如何来说明比较的结果的。

SF	OF	比较结果	解释
1	0	$(ah) < (bh)$	无溢出，逻辑结果的正负=运行结果的正负；因SF=1，实际结果为负，那么逻辑结果为负；所以 $(ah) < (bh)$
0	0	$(ah) > (bh)$	无溢出，逻辑结果的正负=运行结果的正负；因SF=0，运行结果非负，那么逻辑结果非负。所以 $(ah) \geq (bh)$ 。
1	1	$(ah) \geq (bh)$	有溢出，逻辑结果的正负 \neq 运行结果的正负；因SF=1，运行结果为负，那么逻辑结果为正；所以 $(ah) > (bh)$ 。
0	1	$(ah) \leq (bh)$	有溢出，逻辑结果的正负 \neq 运行结果的正负；因SF=0，实际结果正，那么逻辑结果为负；所以 $(ah) \leq (bh)$ 。



11.8 cmp指令

- **cmp**指令在进行**有符号数**和**无符号数**比较时，通过设置相关的标志位来表示比较的结果。
- 要注意理解8086CPU这种工作机制——对于各种处理器来说是普遍的。



11.9 检测比较结果的条件转移指令

- 条件转移指令：根据某种条件，决定是否修改IP的指令。
 - 如：jcxz，如果(cx)=0，就修改IP，否则不转移。
 - 其他**条件转移指令**：检测**标志寄存器**的相关标志位，根据检测的结果来决定是否修改IP。
 - 这些条件转移指令通常都和cmp相配合使用。
 - **注意**：8086CPU中，所有条件转移指令的转移位移都是[-128, 127]。



11.9 检测比较结果的条件转移指令

- 条件转移指令分为两种：
 - 根据**无符号数**的比较结果进行转移的**条件转移指令**，它们检测**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**，它们检测**SF**、**OF**和**ZF**的值。



11.9 检测比较结果的条件转移指令

- 常用根据无符号数的比较结果进行转移的条件转移指令

助记:

j: jump;

e: equal;

ne: not equal;

b: below;

nb: not below;

a: above;

na: not above。

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



11.9 检测比较结果的条件转移指令

- **je**指令：检测 **ZF**位，当 **ZF=1**的时转移，否则不跳转。
 - **说明1**：一般在 **je** 前面使用 **cmp** 指令，这样**je**对**ZF**的检测，实际上就是读取**cmp**对两数是否相等比较结果。



11.9 检测比较结果的条件转移指令

■ 如：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
```

分析

； 如果 $(ah)=(bh)$ ，则 $ZF=1$
； 检测 ZF 为 1 时转移到 s

通过 **cmp** 和 **je** 指令配合实现“相等则转移”的逻辑含义



11.9 检测比较结果的条件转移指令

- **je**指令：检测 **ZF**位，当 **ZF=1**的时转移，否则不跳转。

- **说明2**：**je**指令若没有使用**cmp**指令时也可以运行。

- 比如：下面程序运行之后ax中值为多少？

<code>mov ax,0</code>	分析
<code>add ax,0</code>	
<code>je s</code>	
<code>inc ax</code>	
<code>s: inc ax</code>	

； 结果使得**ZF**=1，
； 将转移，此**je**无“相等则转移”含义。
； (**ax**)=1。



11.9 检测比较结果的条件转移指令

- `cmp`与 `je` 等条件转移指令配合使用，可实现根据比较结果进行转移的功能——类似于高级语言中的IF语句的逻辑功能。
- 程序员可以直接考虑`cmp`和条件转移指令配合使用时，无需考虑具体使用的标志位。
- 其他`jne`、`jb`、`jnb`、`ja`、`jna`等指令和`cmp`指令配合使用的思想和`je`相同。



11.9 检测比较结果的条件转移指令

- 示例： 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。



11.9 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

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。

实现方法1

```
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,8
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
```

用 jne 检测不等于 8 的情况，从而间接地检测等于 8 的情况。



11.9 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

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。

实现方法2

```
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,8
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
```

用je指令检测等于8的情况，没有**方法1**精简



11.9 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。

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。

```
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
   jna 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
```



11.9 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。

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。

```
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
```



11.9 检测比较结果的条件转移指令

- 本小节主要探讨的是 **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。其他条件转移指令工作原理类似。

■ JS	SF=1	Jump if sign
■ JNS	SF=0	Jump if not sign
■ JC	CF=1	Jump if carry
■ JNC	CF=0	Jump if not carry
■ JO	OF=1	Jump if overflow
■ JNO	OF=0	Jump if not overflow
■ JP/JPE	PF=1	Jump if parity/parity even
■ JNP/JPO	PF=0	Jump if no parity/parity odd



11.9 检测比较结果的条件转移指令

- 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。



11.9 检测比较结果的条件转移指令

Opcode	Description	CPU Flags
JA	Above	CF = 0 and ZF = 0
JAE	Above or equal	CF = 0
JB	Bellow	CF
JBE	Bellow or equal	CF or ZF
JC	Carry	CF
JE	Equality	ZF
JG	Greater ^(s)	ZF = 0 and SF = OF
JGE	Greater of equal ^(s)	SF = OF
JL	Less ^(s)	SF ≠ OF
JLE	Less equal ^(s)	ZF or SF ≠ OF
JNA	Not above	CF or ZF
JNAE	Neither above nor equal	CF
JNB	Not bellow	CF = 0
JNBE	Neither bellow nor equal	CF = 0 and ZF = 0

^(s) signed mode

Opcode	Description	CPU Flags
JNC	Not carry	CF = 0
JNE	Not equal	ZF = 0
JNG	Not greater	ZF or SF ≠ OF
JNGE	Neither greater nor equal	SF ≠ OF
JNL	Not less	SF = OF
JNLE	Not less nor equal	ZF = 0 and SF = OF
JNO	Not overflow	OF = 0
JNP	Not parity	PF = 0
JNS	Not negative	SF = 0
JNZ	Not zero	ZF = 0
JO	Overflow ^(s)	OF
JP	Parity	PF
JPE	Parity	PF
JPO	Not parity	PF = 0
JS	Negative ^(s)	SF
JZ	Null	ZF

JCXZ is not dependent on CPU's FLAG but are on register CX (16 bits).



特别提示

■ 检测点11.3 (p219)

(2) 补全下面的程序，统计 F000:0 处 32 个字节中，大小在(32,128)的数据的个数。

```
mov ax,0f000h
mov ds,ax

mov bx,0
mov dx,0
mov cx,32
s:  mov al,[bx]
    cmp al,32
    _____
    cmp al,128
    _____
    inc dx
s0: inc bx
    loop s
```

■ 没有完成此检测点，请不要向下进行。



11.10 DF标志和串传送指令

- 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。

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

- 对标志寄存器的DF位进行设置的指令
 - cld指令（clear direction flag）：设置DF为0
 - std指令（set direction flag）：设置DF为1



11.10 DF标志和串传送指令

■ 指令： `movsb`

■ 功能：以字节为单位传送，即将 `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。

■ 可描述为：

$$(1) ((\text{es}) \times 16 + (\text{di}))_{\text{byte}} = ((\text{ds}) \times 16 + (\text{si}))_{\text{byte}}$$

$$(2) \text{ 如果 } DF = 0 \text{ 则: } (si) = (si) + 1$$

$$(di) = (di) + 1$$

$$\text{如果 } DF = 1 \text{ 则: } (si) = (si) - 1$$

$$(di) = (di) - 1$$



11.10 DF标志和串传送指令

■ 指令： `movsw`

- 功能：以字为单位传送，即将 `ds:si` 指向的内存字单元中送入 `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。

- 可描述为：

(1) $((\text{es}) \times 16 + (\text{di}))_{\text{word}} = ((\text{ds}) \times 16 + (\text{si}))_{\text{word}}$

(2) 如果 `DF = 0` 则： $(\text{si}) = (\text{si}) + 2$

$$(\text{di}) = (\text{di}) + 2$$

如果 `DF = 1` 则： $(\text{si}) = (\text{si}) - 2$

$$(\text{di}) = (\text{di}) - 2$$



11.10 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，它们 `rep` 配合使用：

	用汇编语法来描述
<code>rep movsb</code>	<code>s : movsb</code> <code>loop s</code>
<code>rep movsw</code>	<code>s : movsw</code> <code>loop s</code>

- `rep`的作用：根据`cx`值，重复执行后面的串传送指令。
 - 每执行一次`movsb`指令`si`和`di`都会递增或递减，从而指向后一个或前一个单元，则`rep movsb`就可以循环实现(`cx`)个字符的传送。



11.10 DF标志和串传送指令

■ 示例：编程1

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



11.10 DF标志和串传送指令

■ **分析**：使用串传送指令进行数据传送，必要信息：

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`。

针对本示例

`ds:si = data:0;`

`es:di = data:16`

`cx = 16;`

`DF = 0` 因正向传送 (`si` 和 `di` 递增) 较方便。



11.10 DF标志和串传送指令

■ 程序代码

```
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
```



11.10 DF标志和串传送指令

■ 示例：编程2

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
```



11.10 DF标志和串传送指令

■ 分析：串传输需要的相关信息如下：

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`

针对编程2

`ds:si = F000:FFFF;`

`Es:di = data:15;`

`cx = 16;`

`DF=1` 因逆向传送(`si` 和 `di` 递减) 较方便。



11.10 DF标志和串传送指令

■ 程序代码

```
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
```



11.11 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。



特别提示

■ 检测点11.4 (p233)

下面的程序执行后: (ax)=?

```
mov ax,0  
push ax  
popf  
mov ax,0fff0h  
add ax,0010h  
pushf  
pop ax  
and al,11000101B  
and ah,00001000B
```

■ 没有完成此检测点, 请不要向下进行。



11.12 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

AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=***** ES=***** SS=***** CS=***** IP=0100

↑	↑	↑	↑	↑	↑
OF	DF	SF	ZF	PF	CF

- Debug对常用标志位的表示

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