

# 汇编语言与接口技术

## 指令系统

# 指令系统

- 一台计算机所拥有的全部指令的集合构成了它的指令系统。
- CPU的每一次升级都伴随着指令集的更新与扩充。
- 一条指令通常由操作码域和操作数域两部分组成，操作码域指示计算机要执行的操作，操作数域则提供与操作数或操作数地址有关的信息。

# 数据寻址方式

- 寻址方式分为与数据有关的寻址方式和与转移地址有关的寻址方式。
- MOV 目标, 源 ;把源操作数传送给目标
- MOV是实现数据传送功能的操作码助记符，简称为操作码。目标和源是操作数，中间用逗号隔开。注释内容从“;”开始。

# 立即寻址方式

- 操作数直接包含在指令中，紧跟在操作码之后的寻址方式称为立即寻址方式。
- MOV BL,9 ;执行结果 (BL) = 9
- MOV EAX,1234H ;执行结果 (EAX) = 1234H
- 只能出现在源操作数的位置

# 寄存器寻址方式

- 操作数直接包含在寄存器中，由指令指定寄存器的寻址方式。
- 寄存器可以是通用寄存器，包括8位、16位、32位通用寄存器。也可以是段寄存器，但目标寄存器不能是CS。
- MOV ECX, 9AH
- MOV BX, AX
- 速度快，数量少，个别操作受限制，比如段寄存器。

# 直接寻址方式

- 从这个寻址方式开始，操作数位于**存储器**中。
- 操作数的有效地址Effective Address（EA）直接包含在指令中的寻址方式称为直接寻址方式。

操作类型	默认段寄存器	允许指定的段寄存器	偏移量
1. 普通变量*	DS	ES、SS、CS、GS、FS	EA
2. 字符串指令的源串地址	DS	ES、SS、CS、GS、FS	SI、ESI
3. <b>字符串</b> 指令的目标串地址	ES	<b>无</b>	DI、EDI
4. BP、EBP用作基址寄存器	SS	DS、ES、CS、GS、FS	EA

\* 普通变量是指除第2～4种操作类型以外的内存变量。

- MOV EAX, [00404011H]
- [00404011H]是直接寻址方式的一种表示形式。
- 注意这里的00404011H用[ ]括起来，它是一个普通变量的有效地址，而不是操作数本身。
- MOV EAX, VAR
- VAR是一个内存变量名，它代表一个内存单元的符号地址。

# 段超越

- 1、2、4寻址类型，可以使用段超越前缀显式地说明。段超越前缀的功能是明确指出本条指令所要寻址的内存单元在哪个段。
- 段寄存器名:存储器寻址方式。
- MOV EBX, ES: MEM



# 寄存器间接寻址方式

- 操作数的有效地址在寄存器而操作数本身在存储器中的寻址方式称为寄存器间接寻址方式。
- 对于16位寻址，这个寄存器只能是基址寄存器BX、BP或变址寄存器SI、DI；对于32位寻址，允许使用任何32位通用寄存器。
- 指令中使用的是BX、SI、DI、EAX、EBX、ECX、EDX、ESI、EDI，则默认操作数在数据段，即它们默认与DS段寄存器配合；若使用的是BP、EBP、ESP，则默认操作数在堆栈段，即它们默认与SS段寄存器配合。

- MOV AL, [BX]
- 运行在实模式下, 若  $(DS) = 3000H$ ,  $(BX) = 78H$ ,  $(30078H) = 12H$ , 则物理地址  $= 10H \times (DS) + (BX) = 30078H$ , 该指令的执行结果是  $(AL) = 12H$ 。
- MOV AX, [BP]
- 运行在实模式下, 若  $(SS) = 2000H$ ,  $(BP) = 80H$ ,  $(20080H) = 12H$ ,  $(20081H) = 56H$ , 则物理地址  $= 10H \times (SS) + (BP) = 20080H$ , 该指令的执行结果是  $(AX) = 5612H$ 。

# 寄存器相对寻址方式

- 操作数的有效地址是一个寄存器的内容和指令中给定的一个位移量（disp）之和。
- 对于16位寻址，这个寄存器只能是基址寄存器BX、BP或变址寄存器SI、DI；对于32位寻址，允许使用任何32位通用寄存器。
- 位移量可以是8位、16位、32位（只适用于32位寻址情况）的带符号数。
- 与段寄存器的配合情况同寄存器间接寻址方式。
- 使用BP、EBP、ESP，则默认与SS段寄存器配合；使用其他通用寄存器，则默认与DS段寄存器配合。
- 允许使用段超越前缀。

- `MOV AL, 8[BX]` ;8是位移量
- 也可以表示为: `MOV AL, [BX+8]`
- 实模式下, 若  $(DS) = 3000H$ ,  $(BX) = 70H$ ,  $(30078H) = 12H$
- 则物理地址  $= 10H \times (DS) + (BX) + 8 = 30078H$ , 该指令的执行结果是  $(AL) = 12H$ 。

- 使用这种寻址方式可以访问一维数组。
- 其中，TABLE是数组起始地址的偏移量（即数组名），寄存器中是数组元素的下标乘以元素的长度（一个元素占用的字节数）
- 下标从0开始计数。

# 基址变址寻址方式

- 对于16位寻址，操作数的有效地址是一个基址寄存器（BX、BP）和一个变址寄存器（SI、DI）的内容之和。
- 对于32位寻址，允许使用变址部分除ESP以外的任何两个32位通用寄存器的组合。
- 默认使用段寄存器的情况由所选用的基址寄存器决定。若使用BP、ESP或EBP，默认与SS配合；若使用BX或其他32位通用寄存器（386以上），则默认与DS配合。
- 允许使用段超越前缀。

- $EA = (\text{基址寄存器}) + (\text{变址寄存器})$
- `MOV AL, [BX][SI] / MOV AL, [BX+SI]`
- `MOV EAX, [EBX][ESI]`

# 相对基址变址寻址方式

- 对于16位寻址，操作数的有效地址是一个基址（BX、BP）和一个变址寄存器（SI、DI）的内容和指令中给定的一个位移量（disp）之和。
- 对于32位寻址，允许使用变址部分除ESP以外的任何两个32位通用寄存器及一个位移量的组合。
- 默认段寄存器与基址变址寻址方式相同。
- 可以访问形如ARY[3][2]的二维数组



- MOV AL, ARY[BX] [SI] / MOV AL, ARY [BX+SI]
- MOV EAX, ARY[EBX] [ESI]

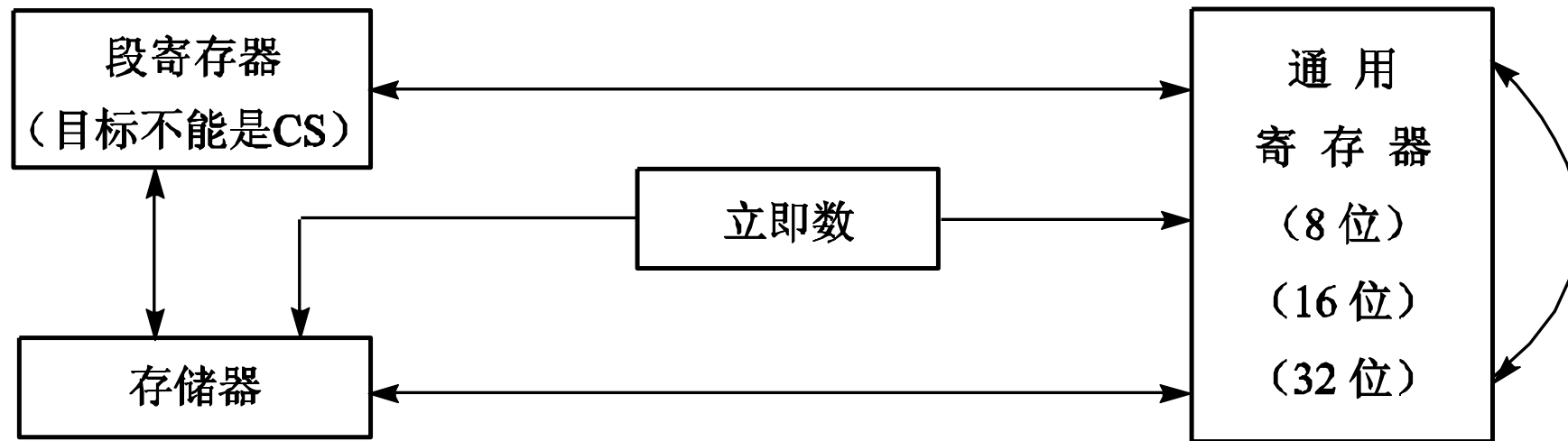
# 比例变址寻址方式

- 80386以上的微处理器才提供的
- 基址部分（8个32位通用寄存器）、变址部分（除ESP以外的32位通用寄存器）乘以比例因子、位移量（disp）。比例因子可以是1（默认值）、2、4或8
- 位移量可以是8位、32位的带符号数。
- 默认使用段寄存器的情况由所选用的基址寄存器决定。若使用ESP或EBP，默认与SS配合；若使用其他32位通用寄存器，默认与DS配合。
- 允许使用段超越前缀。

- 有效地址EA= (基址寄存器) + (变址寄存器) × 比例因子 + disp
- MOV EAX, ARY[EBX][ESI] ; (DS:[ARY+EBX+ESI]) → EAX
- MOV ECX, [EAX+2\*EDX] ; (DS:[EAX+2\*EDX]) → ECX
- MOV EBX, [EBP+ECX\*4+10H] ; (SS:[EBP+ECX\*4+10H]) → EBX
- MOV EDX, ES:ARY[4\*EBX] ; (ES:[ARY+4\*EBX]) → EDX

# 数据运算指令

- 数据传送指令
- 数据传送指令可以实现数据、地址、标志的传送。除了目标地址为标志寄存器的传送指令外，其他指令不影响标志。
- 格式：MOV DST, SRC
- 功能：SRC（源）→DST（目标）
- 源操作数和目标操作数的数据类型，即应同为字节、字或双字型数据。



- 立即数不能作为目标操作数；立即数不能直接送段寄存器；目标寄存器不能是CS
- 两个段寄存器间不能直接传送；
- 两个存储单元之间不能直接传送。

- MOV AL, 5
- MOV BL, 'A'
- MOV AX, BX
- MOV BP, DS
- MOV DS, AX
- MOV [EBX], EAX
- MOV ES:VAR, 12
- MOV WORD PTR [EBX], 12
- MOV EAX, EBX

;字符A的ASCII码41H送BL

- MOV DWORD PTR [EBX], 12
- “DWORD PTR”，它明确指出EBX所指向的内存单元为双字型
- 要生成8 / 16位的二进制数，则需要用“BYTE PTR / WORD PTR”

# 对错

- MOV 1000H, EAX; 错误原因: 立即数作为目标操作数
- MOV DS, 1000H ; 错误原因: 立即数直接送段寄存器
- MOV VAR, [EBX] ; 错误原因: 两个存储单元之间直接传送
- MOV CS, AX ; 错误原因: 目标寄存器是CS
- MOV ES, DS ; 错误原因: 两个段寄存器间直接传送



- 借助于OFFSET和SEG操作符，实现地址传送
- MOV AX, SEG TAB ;把TAB的段基址送给AX寄存器
- MOV DI, OFFSET TAB ;把TAB的偏移量送给DI寄存器
- MOV BX, TAB ;与前两条的差异？

# 带符号扩展的数据传送指令MOVSX

- MOV<sup>S</sup>X指令只有80386以上CPU提供。
- 格式：MOVSX DST, SRC
- 功能：SRC→DST，DST空出的位用SRC的符号位填充。
- 说明：DST必须是<sup>16</sup>位或<sup>32</sup>位寄存器操作数，SRC可以是<sup>8</sup>位或<sup>16</sup>位的寄存器或存储器操作数，但不能是立即数。

- MOV DL, 98H
- MOVSX AX, DL ;AX中得到98H的带符号扩展值0FF98H
- MOV CX, 1234H
- MOVSX EAX, CX ;EAX中得到1234H的带符号扩展值  
00001234H
- MOV VAR, 56H
- MOVSX AX, VAR ;AX 中得到56H的带符号扩展值0056H

# 带零扩展的数据传送指令MOVZX

- MOV<sup>Z</sup>X指令只有80386以上CPU提供。
- 格式：MOVZX DST, SRC
- 功能：SRC→DST，DST空出的位用0填充。
- 说明：DST必须是16位或32位寄存器操作数；SRC可以是8位或16位的寄存器或存储器操作数，但不能是立即数。

- MOV DL, 98H
- MOVZX AX, DL ;AX中得到98H的帶零扩展值0098H
- MOV CX, 1234H
- MOVZX EAX, CX ;EAX中得到1234H的帶零扩展值00001234H
- MOV VAR, 56H
- MOVZX AX, VAR ;AX 中得到56H的帶零扩展值0056H

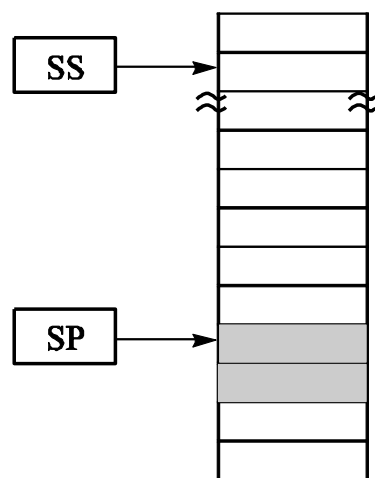
# 堆栈操作指令

- 栈基地址放在SS堆栈段寄存器中，栈顶地址放在SP（16位）或ESP（32位）堆栈指针寄存器中

# 进栈指令PUSH

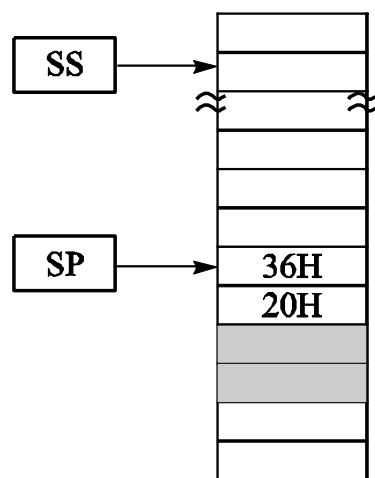
- PUSH SRC
- 功能：先修改堆栈指针使其指向新的栈顶（若SRC是16位操作数则堆栈指针减2，若SRC是32位操作数则堆栈指针减4），然后把SRC压入到栈顶单元。
- 在8086、8088中，SRC只能是16位寄存器操作数或存储器操作数，**不能**是立即数。在80286以上的机器中，SRC可以是16位或32位（80386以上）立即数、寄存器操作数、存储器操作数。

- 设操作数及堆栈地址长度均为16位
- PUSH AX
- PUSH 1234H ;80286以上可用



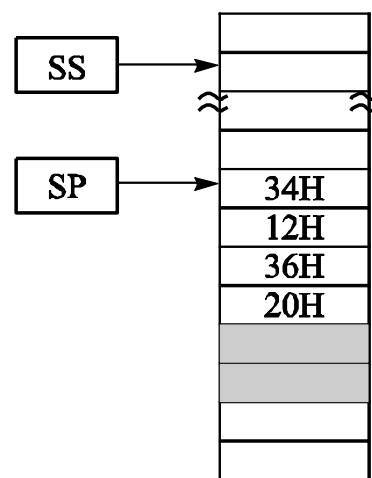
SS=2000H  
SP=0058H  
AX=2036H

(a)



SS=2000H  
SP=0056H  
AX=2036H

(b)



SS=2000H  
SP=0054H  
AX=2036H

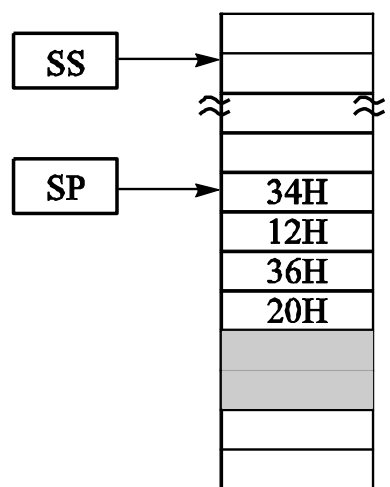
(c)



# 出栈指令POP

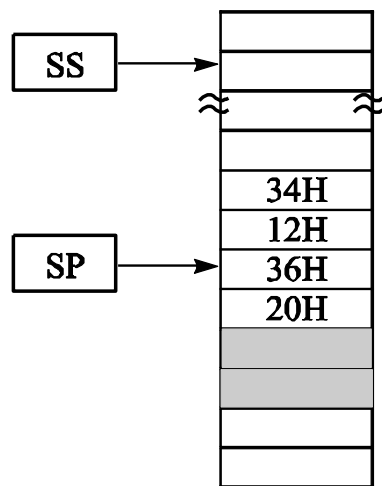
- POP DST
- 功能：先把堆栈指针所指向单元的内容弹出到DST，然后修改堆栈指针以指向新的栈顶（若SRC是16位操作数则堆栈指针加2，若SRC是32位操作数则堆栈指针加4）
- DST可以是16位或32位（80386以上）的寄存器操作数和存储器操作数，也可以是除CS寄存器以外的任何段寄存器。

- 设操作数及堆栈地址长度均为16位。
- POP BX
- POP AX



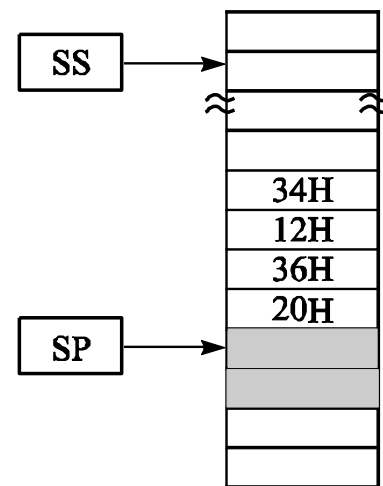
SS=2000H  
SP=0054H

(a)



SS=2000H  
SP=0056H  
BX=1234H

(b)



SS=2000H  
SP=0058H  
AX=2036H

(c)

# 交换指令XCHG

- 格式：XCHG OPR1, OPR2
- 功能：交换两个操作数。
- 说明：OPR是操作数，可以是8位、16位、32位（80386以上），两个操作数均不能是立即数，可以是寄存器操作数和存储器操作数，并且其中之一**必须**是寄存器操作数。
- XCHG AX, BX
- XCHG ECX, WORD PTR [EBX]

# 输入输出指令

- 外设端口独立编址，所以指令系统应提供专门的输入/输出指令。
- IN, OUT
- 结合IOPL和IO位图

# 输入指令IN

- 格式：IN ACR, PORT
- 功能：把外设端口（PORT）的内容传送给累加器（ACR）。
- 说明：可以传送8位、16位、32位（80386以上）的数据，相应的累加器选择AL、AX、EAX。若端口号在0～255之间，则端口号直接写在指令中；若端口号大于255，则端口号通过DX寄存器间接寻址，即端口号应先放入DX中。

- IN AL, 61H ;把61H端口的字节内容输入到AL
- IN AX, 20H ;把20H端口的字内容输入到AX
- MOV DX, 3F8H
- IN AL, DX ;把3F8H端口的字节内容输入到AL
- IN EAX, DX ;把DX所指向的端口双字内容输入到EAX

# 输出指令OUT

- 格式：OUT PORT, ACR
- 功能：把累加器的内容传送给外设端口。
- 说明：对累加器和端口号的选择限制与IN指令相同

- OUT        61H, AL        ;把AL寄存器的内容输出到61H端口
- OUT        20H, AX        ;把AX内容输出到20H端口
- MOV        DX, 3F8H
- OUT        DX, AL        ;把AL寄存器的内容输出到3F8H端口
- OUT        DX, EAX        ;把EAX内容输出到DX所指向的端口



# 地址传送指令

- 传送有效地址指令LEA (Load Effective Address)，指令传送的是操作数的地址（逻辑），而**不是**操作数本身。
- 格式：LEA          REG, SRC
- 功能：把源操作数的有效地址送给指定的寄存器。
- 说明：源操作数**必须**是存储器操作数。

- LEA BX, ASC ;同MOV BX, OFFSET ASC指令
- LEA BX, ASC[SI] ;把DS:[ASC+SI]单元的16位偏移量送给BX
- LEA DI, ASC[BX][SI];把DS:[ASC+BX+SI]单元的16位偏移量送给DI
- LEA EAX, 6[ESI] ;把DS:[6+ESI]单元的32位偏移量送给EAX

# 标志传送指令

- POPF、POPFD、SAHF指令影响标志位，其他不影响。
- 16位标志进栈指令PUSHF (Push Flags Register onto the Stack)
- 格式：PUSHF
- 功能：先使堆栈指针寄存器SP减2，然后压入标志寄存器FLAGS的内容到栈顶单元。

- 16位标志出栈指令POP<sup>F</sup> (Pop Stack into Flags Register)
- 格式: POPF
- 功能: 先把堆栈指针所指向的字弹出到FLAGS, 然后使堆栈指针寄存器SP加2。
- 标志: 影响FLAGS中的所有标志。

- 32位标志进栈指令PUSHFD (Push Eflags Register onto the Stack)
- 格式： PUSHFD
- 功能： 先使堆栈指针寄存器ESP减4， 然后压入标志寄存器EFLAGS的内容到栈顶单元。
- 32位标志出栈指令POPFD (Pop Stack into Eflags Register)
- 格式： POPFD
- 功能： 先把堆栈指针所指向的双字弹出到EFLAGS， 然后使堆栈指针寄存器ESP加4。
- 标志： 影响EFLAGS中的所有标志。

# 算术运算指令

- 包括二进制和十进制算术运算指令
- 两个操作数寻址方式的**限定**同MOV指令，即目标操作数不允许是立即数和CS段寄存器，两个操作数不能同时为存储器操作数等。
- 除类型转换指令外，其他指令均影响某些运算结果特征标志。

# 类型转换指令

- 把操作数的最高位进行扩展，用于处理带符号数运算的操作数类型匹配问题。
- 这类指令均不影响标志。

- 字节扩展成字指令CBW (Convert Byte to Word)

- 格式: CBW

- 功能: 把AL寄存器中的符号位值扩展到AH中。

- MOV AL, 5

- CBW ;执行结果为 (AX) = 0005H

- MOV AL, 98H

- CBW ;执行结果为 (AX) = 0FF98H



- 字扩展成双字指令CWD (Convert Word to Doubleword)
- 格式: CWD
- 功能: 把AX寄存器中的符号位值扩展到DX中。
- MOV AX, 5
- CWD ;执行结果为 (DX) = 0, AX值不变
- MOV AX, 9098H
- CWD ;执行结果为 (DX) = 0FFFFH, AX值不变

- 双字扩展成四字指令CDQ (Convert Doubleword to Quad-Word)
- 格式: CDQ
- 功能: 把EAX寄存器中的符号位值扩展到EDX中。
- 说明: 80386以上CPU支持此指令。
- MOV EAX, 5
- CDQ ;执行结果为 (EDX) = 0, EAX值不变
- MOV EAX, 90980000H
- CDQ ;执行结果为 (EDX) = 0FFFFFFFFH, EAX值不变

- AX符号位扩展到EAX指令CWDE (Convert Word to Doubleword Extended)
- 格式: CWDE
- 功能: 把AX寄存器中的符号位值扩展到EAX的高16位。
- 说明: 80386以上CPU支持此指令。
- MOV AX, 5
- CWDE ;执行结果为  $(EAX_{31} \sim EAX_{16}) = 0$ , AX值不变
- MOV AX, 9098H
- CWDE ;执行结果为  $(EAX_{31} \sim EAX_{16}) = 0FFFFH$ , AX值不变

# 二进制加法指令

- 每一条均适用于带符号数和无符号数运算。
- 加法指令ADD (Add)
- 格式：ADD DST, SRC
- 功能：  $(DST) + (SRC) \rightarrow DST$
- 说明：对于操作数的限定同MOV指令。即源和目标均可以是8位、16位、32位的操作数；要注意源和目标操作数的类型匹配，即它们的长度要一致；目标不能是立即数和CS段寄存器，两个操作数不能同时为存储器操作数等。
- 标志：影响OF、SF、ZF、AF、PF、CF标志。

- ADD AX, 535
- ADD AL, '0'
- ADD WORD PTR[BX], 56
- ADD EDX, EAX

- 带进位加法指令ADC (Add with Carry)
- 格式: ADC DST, SRC
- 功能:  $(DST) + (SRC) + CF \rightarrow DST$
- 说明: 除了执行时要加进位标志CF的值外, 其他要求同ADD。因为它考虑了CF, 所以可用于数值是多字节或多字的加法程序。
- 标志: 影响OF、SF、ZF、AF、PF、CF标志。
- ADC AX, 35 ;执行后  $(AX) = (AX) + 35 + CF$
- 多字节加法

- 加1指令INC (Increment)
- 格式: INC DST
- 功能:  $(DST) + 1 \rightarrow DST$
- 说明: 使用本指令可以很方便地实现地址指针或循环次数的加1修改。
- 标志: 不影响CF标志, 影响其他5个算术运算特征标志。
- INC BX

- 互换并加法指令XADD (Exchange and Add)
- 格式: XADD DST, SRC
- 功能:  $(DST) + (SRC) \rightarrow TEMP$ ,  $(DST) \rightarrow SRC$ ,  $TEMP \rightarrow DST$ 。
- 说明: TEMP是临时变量。该指令执行后, 原DST的内容在SRC中, 和在DST中。只有80486以上CPU才支持XADD指令。
- 标志: 影响OF、SF、ZF、AF、PF、CF标志。



- XADD AL, BL
- 若 (AL) = 16H, (BL) = 35H,
- 则本条指令执行后的结果是?
- (AL) = 4BH, (BL) = 16H。

# 关于溢出

- 同一个加法运算，分别解释为无符号数和带符号数，溢出的情况不一定一致。
- $1100 + 0101$
- 无符号数相加结果若使CF置1，则表示溢出；带符号数相加结果若使OF置1，则表示溢出。

类别	二进制数加法	解释为无符号数	解释为带符号数
1. 带符号数和无符号数都不溢出	$\begin{array}{r} 00000100 \\ +00001011 \\ \hline 00001111 \end{array}$	$\begin{array}{r} 4 \\ +11 \\ \hline 15 \\ CF=0 \end{array}$	$\begin{array}{r} +4 \\ +(+11) \\ \hline +15 \\ OF=0 \end{array}$
2. 无符号数溢出	$\begin{array}{r} 00000111 \\ +11111011 \\ \hline 100000010 \\ CF \leftarrow 1 \end{array}$	$\begin{array}{r} 7 \\ +251 \\ \hline 258(\text{错}) \\ CF=1 \end{array}$	$\begin{array}{r} +7 \\ +(-5) \\ \hline +2 \\ OF=0 \end{array}$
3. 带符号数溢出	$\begin{array}{r} 00001001 \\ +01111100 \\ \hline 10000101 \end{array}$	$\begin{array}{r} 9 \\ +124 \\ \hline 133 \\ CF=0 \end{array}$	$\begin{array}{r} +9 \\ +(+124) \\ \hline -123(\text{错}) \\ OF=1 \end{array}$
4. 带符号数和无符号数都溢出	$\begin{array}{r} 10000111 \\ +11110101 \\ \hline 101111100 \\ CF \leftarrow 1 \end{array}$	$\begin{array}{r} 135 \\ +245 \\ \hline 124(\text{错}) \\ CF=1 \end{array}$	$\begin{array}{r} -121 \\ +(-11) \\ \hline +124(\text{错}) \\ OF=1 \end{array}$

# 二进制减法指令

- 每一条均适用于带符号数和无符号数运算
- 减法指令SUB (Subtract)
- 格式: SUB DST, SRC
- 功能:  $(DST) - (SRC) \rightarrow DST$
- 说明: 除了是实现减法功能外, 其他要求同ADD。
- 标志: 影响OF、SF、ZF、AF、PF、CF标志。

- SUB AX, 35
- SUB AL, '0'
- SUB WORD PTR[BX], 56
- SUB EDX, EAX

- 带借位减法指令SBB (Subtract with Borrow)
- 格式: SBB DST, SRC
- 功能:  $(DST) - (SRC) - CF \rightarrow DST$
- 说明: 除了操作时要减进位标志CF的值外, 其他要求同ADC。因为它考虑了CF, 所以可用于数值是多字节或多字的减法程序。
- 标志: 影响OF、SF、ZF、AF、PF、CF标志。
- SBB AX, 35 ;执行后  $(AX) = (AX) - 35 - CF$
- 多字节减法

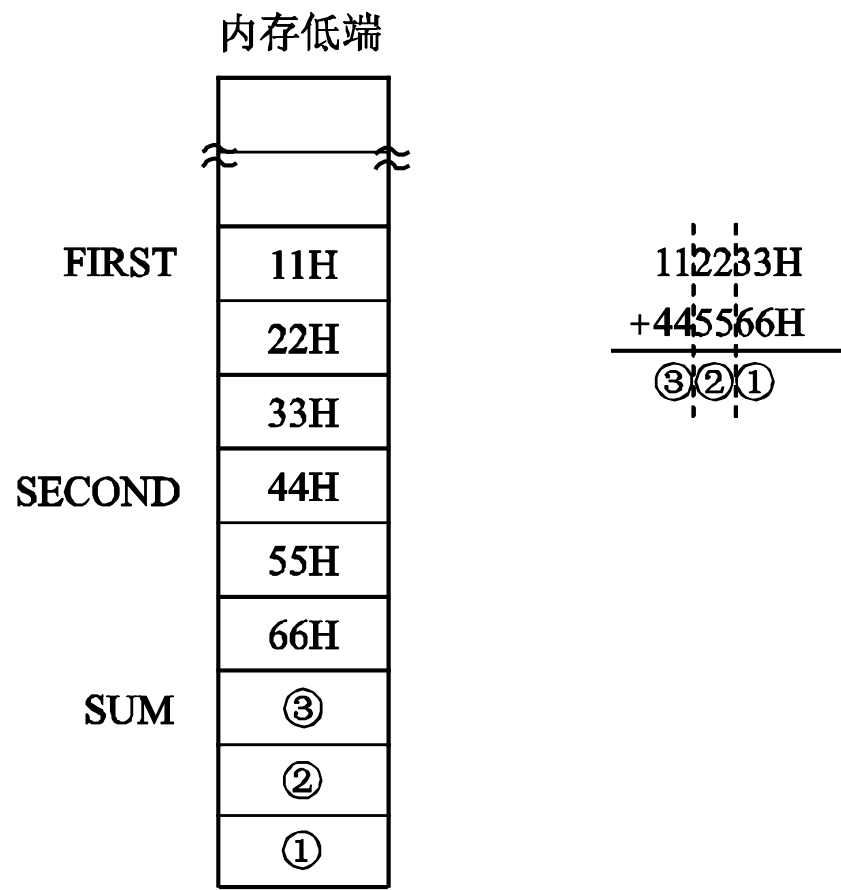
- 减1指令DEC (Decrement)
- 格式: DEC DST
- 功能:  $(DST) - 1 \rightarrow DST$
- 说明: 使用本指令可以很方便地实现地址指针或循环次数的减1修改。
- 标志: 不影响CF标志, 影响其他5个算术运算特征标志。
- DEC BX

- 比较指令CMP (Compare)
- 格式: CMP DST, SRC
- 功能:  $(DST) - (SRC)$ , 影响标志位。
- 说明: 这条指令执行相减操作后只根据结果设置标志位, 并不改变两个操作数的原值。其他要求同SUB。CMP指令常用于比较两个数的大小。
- 标志: 影响OF、SF、ZF、AF、PF、CF标志。
- CMP AX, [BX]



- 求补指令NEG (Negate)
- 格式: NEG DST
- 功能: 对目标操作数 (含符号位) 求反加1, 并且把结果送回目标。即实现 $0 - (\text{DST}) \rightarrow \text{DST}$ 。
- 说明: 利用NEG指令可实现求一个数的相反数。
- 标志: 影响OF、SF、ZF、AF、PF、CF标志。

- 编写两个3字节长的二进制数加法程序，加数FIRST、SECOND及和SUM的分配情况如图所示。



- LEA DI, SUM ;建立和的地址指针DI
- ADD DI, 2 ;DI指向和的低字节
- MOV BX, 2
- MOV AL, FIRST[BX] ;取FIRST的低字节（本例为33H）
- ADD AL, SECOND+2 ;两个低字节相加，和①在AL中，进位反映在CF中
- MOV [DI], AL ;把低字节和存到DI指向的单元（本例为SUM+2单元）
- DEC DI ;修改和指针，使其指向中字节
- DEC BX ;修改加数指针，使其指向中字节
- MOV AL, FIRST[BX] ;取FIRST的中字节（本例为22H）
- ADC AL, SECOND+1 ;两个中字节相加且加CF，和②在AL中,进位反映在CF中
- MOV [DI], AL ;把中字节和存到DI指向的单元（本例为SUM+1单元）
- DEC DI ;修改和指针，使其指向高字节
- DEC BX ;修改加数指针，使其指向高字节
- MOV AL, FIRST[BX] ;取FIRST的高字节（本例为11H）
- ADC AL, SECOND ;两个高字节相加且加CF，和③在AL中,进位反映在CF中
- MOV [DI], AL ;把高字节和存到DI指向的单元（本例为SUM单元）

- 两个ADC指令能否换为ADD?
- 否。因为在对高字节计算时要考虑到低字节的进位，这个进位在执行上一条加法指令时已反映在CF中。
- DEC DI和SUB DI, 1指令是否可以互换?
- 否。因为上一条加法指令对CF的影响后边要用到，所以不能破坏CF值，使用DEC指令正好不影响CF。
- 等价替换指令
- PUSHF ;保存包括CF的 FLAGS值
- SUB DI,1 ;修改DI
- POPF ;恢复原FLAGS值

# 多字节运算溢出情况判断

- 若是两个无符号数相加，则当最后一次的CF被置1时，表示溢出，结果不正确；若是两个带符号数相加，则当最后一次的OF被置1时，表示溢出，结果不正确。

# 二进制乘法指令

- 分无符号数和带符号数二进制乘法指令
- 无符号数乘法指令MUL (Unsigned Multiple)
- 格式: MUL SRC<sub>reg / m</sub>
- 功能: 实现两个无符号二进制数乘。
- 说明: 该指令只含一个源操作数, 必须注意这个源操作数只能是寄存器 (reg) 或存储器操作数 (m), 不能是立即数。另一个乘数必须先放在累加器中。该指令可以实现8位、16位、32位无符号数乘。若源操作数是8位的, 则与AL中的内容相乘, 乘积在AX中; 若源操作数是16位的, 则与AX中的内容相乘, 乘积在DX:AX这一对寄存器中; 若源操作数是32位 (80386以上) 的, 则与EAX中的内容相乘, 乘积在EDX:EAX这一对寄存器中。

- 字节型乘法:  $(AL) \times (SRC)_8 \rightarrow AX$
- 字型乘法:  $(AX) \times (SRC)_{16} \rightarrow DX:AX$
- 双字型乘法:  $(EAX) \times (SRC)_{32} \rightarrow EDX:EAX$
- 标志: 若乘积的高半部分 (例: 字节型乘法结果的AH) 为0, 则对CF和OF清0, 否则置CF和OF为1。其他标志不确定。

- MOV AL, 8
- MUL BL ;  $(AL) \times (BL)$  , 结果在AX中
- MOV AX, 1234H
- MUL WORD PTR [BX] ;  $(AX) \times ([BX])$  , 结果在DX:AX中
- MOV EAX, 0F901H
- MUL EBX ;  $(EAX) \times (EBX)$  , 结果在EDX:EAX中



- 带符号数乘法指令IMUL (Signed Multiple)
- 功能：实现两个带符号二进制数乘。
- IMUL SRC<sub>reg / m</sub>
- 说明：这种格式的指令除了是实现两个带符号数相乘且结果为带符号数外，其他均与MUL指令相同。所有的80x86 CPU都支持这种格式。

- 字节型乘法:  $(AL) \times (SRC)_8 \rightarrow AX$
- 字型乘法:  $(AX) \times (SRC)_{16} \rightarrow DX:AX$
- 双字型乘法:  $(EAX) \times (SRC)_{32} \rightarrow EDX:EAX$
- 标志: 若乘积的高半部分 (例: 字节型乘法结果的AH) 为低半部分的符号扩展, 则对CF和OF清0, 否则置CF和OF为1。其他标志不确定。

- MOV AL, 8
- IMUL BL ; (AL) × (BL) , 结果在AX中
- MOV AX, 1234H
- IMUL WORD PTR [BX] ; (AX) × ([BX]) , 结果在DX:AX中
- MOV AL, 98H
- CBW ; AL中的符号扩展至字
- IMUL BX ; (AX) × (BX) , 结果在DX:AX中
- MOV AX, 1234H
- CWDE ; AX中的符号扩展至EAX
- IMUL ECX ; (EAX) × (ECX) , 结果在EDX:EAX中

- $\text{IMUL REG, SRC}_{\text{reg} / \text{m}}$
- 说明：REG和SRC的长度必须相同，目标操作数REG必须是16位或32位通用寄存器，源操作数SRC可以是寄存器或存储器操作数。
- 具体操作为：
  - $(\text{REG})_{16} \times (\text{SRC})_{16} \rightarrow \text{REG}_{16}$
  - $(\text{REG})_{32} \times (\text{SRC})_{32} \rightarrow \text{REG}_{32}$
- 标志：若乘积完全能放入目标寄存器（例如：若目标REG是16位的，则结果的有效数字不超过16位），则对CF和OF清0，否则置CF和OF为1。其他标志不确定。
- $\text{IMUL CX, WORD PTR [BX]} ; (\text{CX}) \times ([\text{BX}])$ ，结果在CX中
- $\text{IMUL ECX, EBX} ; (\text{ECX}) \times (\text{EBX})$ ，结果在ECX中

- IMUL REG, imm<sub>8</sub>
- 说明：目标操作数REG可以是16位或32位通用寄存器，源操作数imm<sub>8</sub>只能是8位立即操作数，计算时系统自动对其进行符号扩展。
- 具体操作为：
  - (REG)<sub>16</sub> × imm<sub>8</sub> 符号扩展 → REG<sub>16</sub>
  - (REG)<sub>32</sub> × imm<sub>8</sub> 符号扩展 → REG<sub>32</sub>
- 标志：对标志位的影响同格式2。
- IMUL CX, 98H ; (CX) × 0FF98H, 结果在CX中
- IMUL CX, 68H ; (CX) × 0068H, 结果在CX中

- $\text{IMUL REG, SRC}_{\text{reg} / \text{m}}, \text{imm}_8$
- 说明：REG和SRC的长度必须相同，目标操作数REG必须是16位或32位通用寄存器；源操作数SRC可以是寄存器或存储器操作数；操作数 $\text{imm}_8$ 正如它的英文缩写那样，只能是8位立即操作数。
- 具体操作为：
  - $(\text{SRC})_{16} \times \text{imm}_8$  符号扩展  $\rightarrow \text{REG}_{16}$
  - $(\text{SRC})_{32} \times \text{imm}_8$  符号扩展  $\rightarrow \text{REG}_{32}$
- 标志：对标志位的影响同格式2。
- $\text{IMUL CX, BX, 98H}$  ;  $(\text{BX}) \times 0\text{FF}98\text{H}$ , 结果在CX中
- $\text{IMUL ECX, DWORD PTR [EBX], 68H}$ ;  $([\text{EBX}]) \times 0068\text{H}$ , 结果在ECX中

# 二进制除法指令

- 与乘法类似，对无符号数和带符号数分别提供了二进制除法指令。
- 无符号数除法指令DIV (Unsigned Divide)
- 格式：DIV SRC<sub>reg</sub> / m
- 功能：实现两个无符号二进制数除法。
- 说明：该指令只含一个源操作数，该操作数作为除数使用，注意它只能是寄存器或存储器操作数，不能是立即数。被除数必须事先放在隐含的寄存器中。可以实现8位、16位、32位无符号数除。若源操作数是8位的，则被除数在AX中，商在AL中，余数在AH中；若源操作数是16位的，则被除数在DX:AX一对寄存器中，商在AX中，余数在DX中；若源操作数是32位（80386以上）的，则被除数在EDX:EAX一对寄存器中，商在EAX中，余数在EDX中。

字节型除法:  $(AX) \div (SRC)_8 \rightarrow \begin{cases} \text{商: AL} \\ \text{余数: AH} \end{cases}$

字型除法:  $(DX:AX) \div (SRC)_{16} \rightarrow \begin{cases} \text{商: AX} \\ \text{余数: DX} \end{cases}$

双字型除法:  $(EDX:EAX) \div (SRC)_{32} \rightarrow \begin{cases} \text{商: EAX} \\ \text{余数: EDX} \end{cases}$

标志: 不确定。



- 实现 $1000 \div 25$ 的无符号数除法。
- `MOV       AX, 1000`
- `MOV       BL, 25`
- `DIV       BL`           ;  $(AX) \div (BL)$  , 商在AL中, 余数在AH中
- 实现 $1000 \div 512$ 的无符号数除法。
- `MOV       AX, 1000`
- `SUB       DX, DX`       ; DX清0
- `MOV       BX, 512`
- `DIV       BX`           ;  $(DX:AX) \div (BX)$  , 商在AX中, 余数在DX中

- 带符号数除法指令IDIV (Signed Divide)
- 格式: IDIV SRC<sub>reg</sub> / m
- 功能: 实现两个带符号二进制数除。
- 说明: 除了是实现两个带符号数相除且商和余数均为带符号数外, 其他均与DIV指令相同。余数符号与被除数相同。

- 实现  $(-1000) \div (+25)$  的带符号数除法。
- `MOV AX, -1000`
- `MOV BL, 25`
- `IDIV BL` ;  $(AX) \div (BL)$  , 商在AL中, 余数在AH中
- 实现  $1000 \div (-512)$  的带符号数除法。
- `MOV AX, 1000`
- `CWD` ; AX的符号扩展到DX
- `MOV BX, -512`
- `IDIV BX` ;  $(DX:AX) \div (BX)$  , 商在AX中, 余数在DX中

# 位运算指令

- 逻辑运算指令
- 逻辑运算指令包括逻辑非（NOT）、逻辑与（AND）、逻辑测试（TEST）、逻辑或（OR）和逻辑异或（XOR）指令。这些指令的操作数可以是8位、16位、32位，其寻址方式与MOV指令的限制相同。
- 逻辑非指令可用于把操作数的每一位均**变反**的场合；逻辑与指令用于把某位**清0**（与0相与，也可称为屏蔽某位）、某位保持**不变**（与1相与）的场合；逻辑测试指令可用于只测试其值而**不改变**操作数的场合；逻辑或指令用于把某位**置1**（与1相或）、某位保持**不变**（与0相或）的场合；逻辑异或指令用于把某位**变反**（与1相异或）、某位保持**不变**（与0相异或）的场合。

名 称	格 式	功 能	标 志
逻辑非	NOT DST	(DST)按位变反送DST	不影响
逻辑与	AND DST, SRC	$DST \leftarrow (DST) \wedge (SRC)$	CF和OF清0，影响SF、ZF及PF，AF不定
逻辑测试	TEST OPR1, OPR2	$OPR1 \wedge OPR2$	同AND指令
逻辑或	OR DST, SRC	$DST \leftarrow (DST) \vee (SRC)$	同AND指令
逻辑异或	XOR DST, SRC	$DST \leftarrow (DST) \vee (SRC)$	同AND指令

- 对AL中的值按位求反。
- MOVAL, 00001111 B
- NOT AL ; (AL) = 11110000 B
- 将EAX寄存器清0。
- AND EAX, 0
- 把AL中的0 ~ 9二进制值转换成十进制数的ASCII码形式输出。
- OR AL, 30H ;AL中的高4位变成0011 B, 低4位不变
- 使61H端口的D<sub>1</sub>位变反。
- IN AL, 61H
- XOR AL, 2
- OUT 61H, AL

- 将EAX寄存器清0。
- XOR EAX, EAX
- 这种清0方式比用MOV AX, 0指令占用空间少，执行速度快。
- 转换AL中字母的大小写。
- XOR AL, 20H
- 设某并行打印机的状态端口是379H，其D<sub>7</sub>位是忙闲位。若D<sub>7</sub>为0表示忙，为1表示闲，测试该打印机的当前状态，若为忙则继续测试，否则顺序执行下一条指令。
- MOV DX, 379H
- WT: IN AL, DX;读入状态字节
- TEST AL, 80H ;只关心D<sub>7</sub>位，其他位屏蔽，若结果使ZF=1，表示D<sub>7</sub>=0
- JZ WT ;若ZF=1则跳转到WT继续测试，否则顺序执行下一条指令

# 位测试指令

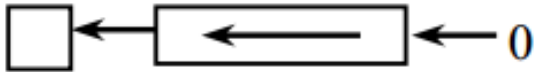
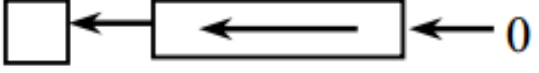
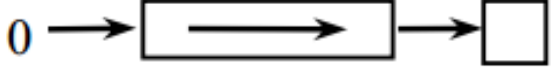
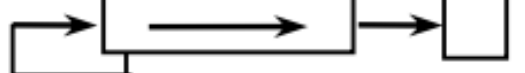
- 80386开始增加了位测试指令
- 包括BT (Bit Test) 、 BTS (Bit Test and Set) 、 BTR (Bit Test and Reset) 和BTC (Bit Test and Complement)
- 这些指令可以直接对一个16位或32位的通用寄存器或存储单元中的指定二进制位进行必要的操作，它们首先把指定位的值送给CF标志，然后对该位按照指令的要求操作。



名 称	格 式	功 能	标 志
位测试	BT DST, SRC	测试由 SRC 指定的 DST 中的位	所选位送 CF，其他标志不定。
位测试并置位	BTS DST, SRC	测试并置 1 由 SRC 指定的 DST 中的位	同上
位测试并复位	BTR DST, SRC	测试并清 0 由 SRC 指定的 DST 中的位	同上
位测试并取反	BTC DST, SRC	测试并取反由 SRC 指定的 DST 中的位	同上

- 目标可以是16位或32位的寄存器或存储器操作数，源可以是8位的立即数、寄存器或存储器操作数
- 若是后两种情况，其长度一定要和目标的长度相同。若源操作数是立即数形式，则其值不应超过目标操作数的长度。目标操作数的位偏移从最右边位开始，从0开始计数。

- MOV EAX, 2357H
- MOV ECX, 3
- BT AX, 0 ;CF=1, (AX) = 2357H
- BT AX, CX ;CF=0, (AX) = 2357H
- BTS EAX, ECX ;CF=0, (EAX) = 235FH
- BTR AX, CX ;CF=0, (EAX) = 2357H
- BTC AX, 3 ;CF=0, (EAX) = 235FH

名 称	格 式	功 能	标 志
逻辑左移	SHL DST,CNT		CF 中总是最后移出的一位, ZF、SF、PF 按结果设置, 当 CNT=1 时, 移位使符号位变化, OF 置 1, 否则清 0。
算术左移	SAL DST,CNT		同上
逻辑右移	SHR DST,CNT		同上
算术右移	SAR DST,CNT		同上

注: 1. 当 CNT>1 时, OF 值不确定。




说明: DST 可以是 8 位、16 位或 32 位的寄存器或存储器操作数, CNT 是移位位数。

2. 对 CNT 的限定是:

当 CNT=1 时, 直接写在指令中; 适用于 8086、8088

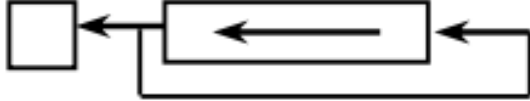
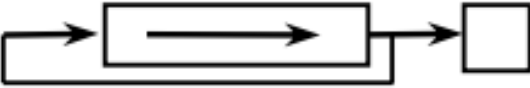
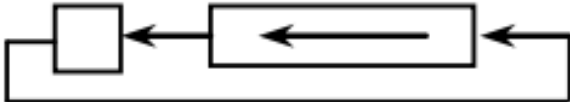
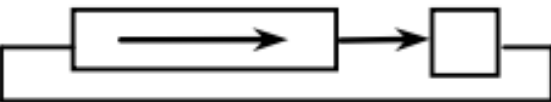
当 CNT>1 时, 由 CL 寄存器给出; 适用于 80x86 系列的所有型号

当 CNT>1 时, 由指令中的 8 位立即数给出; 适用于 80286 以上

3. 功能图中的符号表示:  — CF;  — 数据流向;  — 操作数

- 可以用于对一个数进行 $2^n$ 的倍增或倍减运算，使用这种方法比直接使用乘除法效率要高得多。可以用逻辑移位指令实现无符号数乘除法运算，只要移出位不含1，SHL DST, n执行后是原数的 $2^n$ 倍，SHR DST, n执行后是原数的 $1/2^n$ 。可以用算术右移指令实现带符号数除法运算，只要移出位不含1，SAR DST, n执行后就是原数的 $1/2^n$ 。

- 设无符号数X在AL中， 用移位指令实现 $X \times 10$ 的运算。
- MOV        AH, 0            ;为了保证不溢出， 将AL扩展为字
- SAL        AX, 1            ;求得 $2X$
- MOV        BX, AX          ;暂存 $2X$
- MOV        CL, 2            ;设置移位次数
- SAL        AX, CL           ;求得 $8X$
- ADD        AX, BX          ; $10X = 8X + 2X$

名 称	格 式	功 能	标 志
循环左移	ROL DST,CNT		CF 中总是最后移进的位，当 CNT= 1 时，移位使符号位改变则 OF 置 1，否则清 0，不影响 ZF、SF、PF。
循环右移	ROR DST,CNT		同上
带进位循环左移	RCL DST,CNT		同上
带进位循环右移	RCR DST,CNT		同上
注：当 CNT>1 时，OF 值不确定。 说明：对 DST 和 CNT 的限定同基本移位指令。			

- 把CX:BX:AX一组寄存器中的48位数据左移一个二进制位。
- SHL           AX, 1
- RCL           BX, 1
- RCL           CX, 1
- 在没有溢出的情况下， 以上程序实现了  $2 \times (CX:BX:AX) \rightarrow CX:BX:AX$  的功能。

# 程序控制指令

- 转移指令的寻址方式
- 以无条件转移指令为例来说明。
- 无条件转移指令格式：JMP 目标
- 该指令的功能是**无条件**转移到目标处，这里的**目标**有各种寻址方式，这些寻址方式可以被分为**段内**转移和**段间**转移两类，每一类又可分为**直接**转移和**间接**转移。
- 段内转移只影响指令指针IP或EIP的值；段间转移既要影响IP或EIP的值，也要影响代码段寄存器CS的值。



# 段内直接寻址方式

- 转向的有效地址是当前指令指针寄存器的内容和指令中指定的位移量（disp）之和，该位移量是一个相对于指令指针的带符号数。
- 这种寻址方式是一种相对寻址方式，指令代码不会因为内存的不同区域运行而发生变化

$$EA = (IP) + \left\{ \begin{array}{c} 8 \text{ 位} \\ 16 \text{ 位} \end{array} \right\} \text{disp} \quad (16 \text{ 位})$$

$$EA = (EIP) + \left\{ \begin{array}{c} 8 \text{ 位} \\ 32 \text{ 位} \end{array} \right\} \text{disp} \quad (32 \text{ 位})$$

字节



(a)

字节 0

字节 1



(b)

字节 0

字节 1

字节 2

字节 3



(c)

- 短转移
- 若位移量是8位的，则称为短转移，短转移可以实现在距离下条指令的-128 ~ +127字节范围内转移。其汇编格式为：
- `JMP SHORT LAB`
- `JMP SHORT L1`
- `MOV BL, CL`
- ...
- `L1: ADD AL, 61H`

# 标号/行号

- “L1:”是一条指令地址的符号表示，叫做标号，标号名由程序员确定，所包含字符应符合名字的要求，并必须以冒号结束。

- 近转移
- 若位移量是16位或32位的，则称为近转移。当位移量是16位时，将其加到当前IP值中形成转向的目标地址，其跳转范围为 $\pm 32$  KB，当位移量是32位时，将其加到当前EIP值中形成目标地址，其跳转范围为 $\pm 2$  GB。
- `JMP LAB` 或 `JMP NEAR PTR LAB`
- `JMP L2` ;转向同段内的L2标号处
- `JMP NEAR PTR L3` ;转向同段内的L3标号处

# 段内间接寻址方式

格 式	举 例	注 释
JMP 通用寄存器	JMP BX	16 位转向地址在 BX 中，其值送给 IP
	JMP EAX	32 位转向地址在 EAX 中，其值送给 EIP
JMP 内存单元	JMP WORD PTR VAR	16 位转向地址在 VAR 字型内存变量中
	JMP WORD PTR [BX]	16 位转向地址在 BX 所指向的内存变量中
	JMP DWORD PTR DVAR	32 位转向地址在 DVAR 双字型内存变量中
	JMP DWORD PTR [EBX]	32 位转向地址在 EBX 所指向的内存变量中

- 实模式下, 设  $(DS) = 2000H$ ,  $(BX) = 300H$ ,  $(IP) = 100H$ ,  $(20300H) = 0$ ,  $(20301H) = 05H$ , 则:
- $JMP\ BX$  ;执行后  $(IP) = (BX) = 0300H$
- $JMP\ WORD\ PTR\ [BX]$
- 这条指令执行时, 先按照操作数寻址方式得到存放转移地址的内存单元:  $10H \times (DS) + (BX) = 20300H$ , 再从该单元中得到转移地址, 即  $EA = (20300H) = 0500H$ , 于是,  $(IP) = EA = 0500H$ , 下一次便执行  $CS:0500H$  处的指令, 实现了段内间接转移。

# 段间直接寻址方式

字节 0

字节 1

字节 2

字节 3

偏移量 0~7

偏移量 8~15

段基址 0~7

段基址 8~15

(a)

字节 0

字节 1

字节 2

字节 3

字节 4

字节 5

偏移量 0~7

偏移量 8~15

偏移量 16~23

偏移量 24~31

段选择符 0~7

段选择符 8~15

(b)



- 段间转移在实模式下的16位编程时使用比较多，这是因为它的段长限定不能超过64 KB，对于一个大程序可能会包含多个代码段。但对于保护模式下的32位程序设计，由于此时代码段的长度可达4GB，**足够**存放一个应用程序的全部代码，包括该程序调用的所有DLL，因此一般并不需要段间转移的JMP指令，只有在**切换**任务、跳转到调用门指定的程序入口或者确需执行**另一个**代码段内的程序时才使用。

# 段间间接寻址方式

- 用一个双字内存变量（称为32位指针）中的低16位取代IP值，高16位取代CS值，从而实现段间转移。
- 当操作数长度为32位时，则是用一个三字内存变量（称为48位全指针）中的低32位取代EIP值，高16位取代CS值。该双字或三字变量的地址可以由除立即寻址方式和寄存器寻址方式以外的其他与数据有关的寻址方式获得。

- `JMP DWORD PTR [BX]`
- 假设  $(DS) = 2000H$ ,  $(BX) = 0300H$ ,  $(IP) = 0100H$ ,  
 $(20300H) = 0$ ,  $(20301H) = 05H$ ,  $(20302H) = 10H$ ,  
 $(20303H) = 60H$ , 则这条指令执行时?
- $10H \times (DS) + (BX) = 20300H$
- 再把该单元中的低字送给IP, 高字送给CS, 即  $0500H \rightarrow IP$ ,  
 $6010H \rightarrow CS$ , 下一次便执行  $6010:0500H$  处的指令, 实现了段间间接转移。

- `JMP FWORD PTR [EBX]`
- `FWORD PTR [EBX]`表示EBX指向一个三字变量。
- 这条指令执行时，先按照与操作数有关的寻址方式得到存放转移地址的内存单元，然后把低32位的值送给EIP，高16位值送给CS，从而实现段间间接转移。

# 转移指令

- 包括无条件转移指令、条件转移指令、测试CX / ECX值为0转移指令，通过它们可以实现程序的分支转移。
- 无条件转移指令
- JMP DST
- 功能：无条件转移到DST所指向的地址。
- 说明：DST为转移的目标地址（或称转向地址）

# 总结

- 段内直接短转移
- 格式: `JMP SHORT LABEL`
- 段内直接转移
- 格式: `JMP LABEL`或 `JMP NEAR PTR LABEL`
- 段内间接转移
- 格式: `JMP REG / M`

- 把B2的偏移量送给通用寄存器，通过寄存器实现段内间接转移。
- LEA EBX, B2
- JMP EBX ;转向地址B2在EBX中
- 把B2的偏移量送给内存单元，通过内存单元实现段内间接转移。
- VAR DWORD ? ;为存放标号B2的偏移量预留一个双字型内存变量
- MOV VAR, OFFSET B2 ;把B2的偏移量送给变量VAR
- JMP DWORD PTR VAR ;转向地址在VAR变量中
- JMP DWORD PTR [EBX][ESI]
- JMP DWORD PTR disp[EBX][ESI]

# 总结

- 段间直接转移
- `JMP FAR PTR LABEL`
- 段间间接转移
- `JMP DWORD PTR M (16位)` / `JMP FWORD PTR M (32位)`



- 实模式下，把B3的双字长地址指针放在变量VAR1中，即可通过VAR1实现段间间接转移（保护模式采用变量VAR2存储，存储类型FWORD）。
- VAR1      DWORD      B3      ;初始化B3的偏移量在VAR1中
- JMP          DWORD      PTR   VAR1          ;通过VAR2无条件转移到其他段的B3标号处
- VAR2      FWORD      B3                      ;初始化B3的偏移量在VAR2中
- JMP          FWORD      PTR   VAR2          ;通过VAR2无条件转移到其他段的B3标号处

# 条件转移指令

- 当需要满足条件则转移到程序的另一处，不满足条件则顺序执行下一条指令时使用条件转移指令（Jump if Condition is True）
- 这类指令本身并不影响标志
- $J_{CC}$  LABEL
- 功能：如果条件为真，则转向标号处，否则顺序执行下一条指令。
- 其中CC为条件，LABEL是要转向的标号。在8086 ~ 80286中，该地址应在距离当前IP值-128 ~ +127 B范围之内，即只能使用与转移地址有关的寻址方式的段内短转移格式，其位移量是8位带符号数。从80386开始，转移范围扩大到了段内任意位置，它们可以使用段内直接近转移格式。

汇编格式	功    能	测试条件
JC LABEL	有进位转移	CF=1
JNC LABEL	无进位转移	CF=0
JO LABEL	溢出转移	OF=1
JNO LABEL	无溢出转移	OF=0
JP/JPE LABEL	偶转移	PF=1
JNP/JPO LABEL	奇转移	PF=0
JS LABEL	负数转移	SF=1
JNS LABEL	非负数转移	SF=0
JZ/JE LABEL	结果为 0/相等转移	ZF=1
JNZ/JNE LABEL	结果不为 0/不相等转移	ZF=0

注：对于实现同一功能但指令助记符有两种形式的情况，在程序中究竟选用哪一种视习惯或用途而定，例如：对于指令 JZ/JE LABEL，当比较两数相等转移时常使用 JE，当比较某数为 0 转移时常使用 JZ。下同。

- 比较EAX和EBX寄存器中的内容，若相等执行ACT1，不等执行ACT2。

方法 1

CMP EAX,EBX

JE ACT1

ACT2: ...

...

ACT1: ...

方法 2

CMP EAX,EBX

JNE ACT2

ACT1: ...

...

ACT2: ...

# 根据两个带符号数比较结果实现转移的条件转移指令

汇编格式	功 能	测试条件
JG/JNLE LABEL	大于/不小于等于转移	ZF=0 and SF=OF
JNG/JLE LABEL	不大于/小于等于转移	ZF=1 or SF≠OF
JL/JNGE LABEL	小于/不大于等于转移	SF≠OF
JNL/JGE LABEL	不小于/大于等于转移	SF=OF
注： 1. G=Greater, L=Less, E=Equal, N=Not。 2. 显然，表 3-8 中的指令 JZ/JE LABEL 和 JNZ/JNE LABEL 同样可以用于实现两个带符号数的比较转移。		

# 根据两个无符号数比较结果实现转移的条件转移指令

汇编格式	功    能	测试条件
JA/JNBE LABEL	高于/不低于等于转移	CF=0 and ZF=0
JNA/JBE LABEL	不高于/低于等于转移	CF=1 or ZF=1
JB/JNAE/JC LABEL	低于/不高于等于转移	CF=1
JNB/JAE/JNC LABEL	不低于/高于等于转移	CF=0

注：1. A=Above, B=Below, C=Carry, E=Equal, N=Not。可以看出，这里的高于相当于带符号数的大于，低于相当于带符号数的小于。

2. 显然，表 3-8 中的指令 JZ/JE LABEL 和 JNZ/JNE LABEL 同样可以用于实现两个无符号数的比较转移。

- 设  $M = (\text{EDX}:\text{EAX})$  ,  $N = (\text{EBX}:\text{ECX})$  , 比较两个64位数。若  $M > N$  , 则转向DMAX, 否则转向DMIN。

若两数为无符号数, 则程序片断为:

```

CMP    EDX,EBX
JA     DMAX
JB     DMIN
CMP    EAX,ECX
JA     DMAX

```

DMIN: ...

...

DMAX: ...

若两数为带符号数, 则程序片断为:

```

CMP    EDX,EBX
JG     DMAX
JL     DMIN
CMP    EAX,ECX
JA     DMAX

```

DMIN: ...

...

DMAX: ...

# 测试CX/ECX值为0的转移指令

- 测试的是CX或ECX寄存器的内容是否为0，而不是测试标志位。这类指令只能使用段内短转移格式，即位移量只能是8位的。
- JCXZ      LABEL      ;适用于16位操作数长度
- JECXZ     LABEL      ;适用于32位操作数长度
- 功能：测试CX（ECX）寄存器的内容，当CX（ECX）= 0时则转移，否则顺序执行。
- 说明：此指令经常用于在循环程序中判断循环计数的情况。



# 循环类指令

- 循环指令可以控制程序的循环。对于80x86系列，所有循环指令的循环入口地址都只能在距离当前IP值的-128 ~ +127B范围之内，即位移量只能是8位的，所以它们只能使用与转移地址有关的寻址方式的段内短转移格式。所有循环指令都用CX或ECX作为循环次数计数器，它们都不影响标志。
- CX/ECX初始值为0时，不是循环零次。

# 循环指令

- LOOP LABEL
- 功能：LOOP (Loop) 循环指令， $(CX) - 1 \rightarrow CX$ ，若  $(CX) \neq 0$ ，则转向标号处执行循环体，否则顺序执行下一条指令。
- 说明：对固定次数的循环，适合用LOOP指令来实现。若操作数长度为32位，则其中的CX应为ECX。在LOOP指令前，应先把循环计数的初始值送给CX (ECX)。

# 相等循环指令

- LOOPE/LOOPZ LABEL
- 功能：LOOPE/LOOPZ (Loop while Equal/Zero) 相等循环指令， $(CX) - 1 \rightarrow CX$ ，若  $(CX) \neq 0$  and  $ZF=1$ ，则转向标号处执行循环体，否则顺序执行下一条指令。
- 说明：若操作数长度为32位，则CX应为ECX。在LOOPE或LOOPZ指令前，应先把循环计数的初始值送给CX (ECX)。
- 该指令常用于比较两个字符串是否相等的情况，若前面的字符相等才有必要继续比较，否则中止比较。

# 不等循环指令

- LOOPNE/LOOPNZ LABEL
- 功能：LOOPNE/LOOPNZ (Loop while Not Equal/ Not Zero) 不等循环指令， $(CX) - 1 \rightarrow CX$ ，若  $(CX) \neq 0$  and  $ZF=0$ ，则转向标号处执行循环体，否则顺序执行下一条指令。
- 说明：若操作数长度为32位，则CX应为ECX。在LOOPNE或LOOPNZ指令前，应先把循环计数的初始值送给CX (ECX)。该指令对在数据块中查找信息很有效，当未找到指定字符时继续查找，找到时退出。

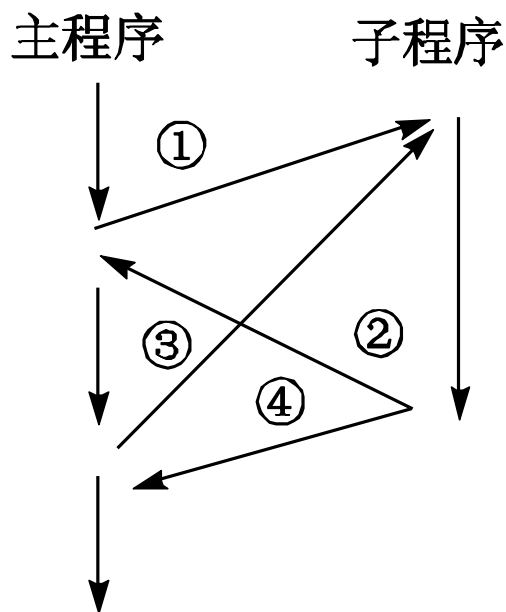
用累加的方法实现 $M \times N$ ，并把结果保存到RESULT单元。假设M、N为32位无符号数。

- 分析：把N个M累加，若用EAX作为累加器并初始化为0，则共累加N次，即把乘数N作为循环次数送给ECX。注意N有可能为0。而当初始化ECX为0时，使用LOOP指令要循环 $2^{32}$ 次而不是0次，因此一定要在循环前判断ECX是否为0，若为0则乘积直接赋为0而循环体一次也不要执行。

- MOV EAX, 0 ;累加器清0
- MOV EDX, 0
- MOV EBX, M
- CMP EBX, 0
- JZ TERM ;被乘数为0则跳转
- MOV ECX, N
- JECXZ TERM ;乘数为0则跳转
- L1: ADD EAX, EBX
- ADC EDX, 0
- LOOP L1 ;累加次数未到则跳转到L1执行循环体, 否则继续
- TERM: MOV RESULT, EAX ;保存结果低32位
- MOV RESULT+4, EDX ;保存结果高32位

# 子程序调用与返回指令

- 汇编语言中的子程序相当于C语言中的函数。为便于模块化程序设计和程序共享，在汇编语言中经常把一些相对独立的程序段组织成子程序的形式。当需要实现该子程序功能时，由调用程序（或泛称为主程序）调用它，当子程序结束后再返回到主程序继续执行。



# 子程序调用指令

- 格式：CALL DST
- 功能：CALL (Call procedure) 调用子程序。执行时先把返回地址压入堆栈，再形成子程序入口地址，最后把控制权交给子程序。
- 其中DST为子程序名或子程序入口地址，其目标地址的形成除了不能使用段内直接短转移格式外，其他与JMP指令相同。它有段内直接 / 间接调用、段间直接 / 间接调用之分。CALL指令的执行结果也是无条件转移到标号处，它与JMP指令的不同之处在于：前者转移后要返回，所以要保存返回地址，而后者转移后不再返回，所以不必保存返回地址。



- 段内调用
- 这类调用指令可实现同一段内的子程序调用，它只改变IP（32位长度时是EIP）值，不改变CS值。
- 首先把CALL之后的那条指令地址的偏移量部分（当前IP或EIP值）压入堆栈；接着根据与转移地址有关的寻址方式形成子程序入口地址的IP（或EIP）值；最后把控制无条件转向子程序。

- 段内直接调用
- 格式：CALL PROCEDURE 或 CALL NEAR PTR PROCEDURE
- 功能：调用PROCEDURE子程序。执行时先把返回地址（当前IP值或EIP值）压入堆栈，再使IP（或EIP）加上指令中的位移量，最后把控制权交给子程序。
- 说明：这种指令使用段内直接寻址方式。
- 设子程序A与CALL指令在同一段内，则调用A子程序的指令是：
- CALL A 或 CALL NEAR PTR A

- 段内间接调用
- 格式：CALL REG / M
- 功能：调用子程序。执行时先把返回地址（当前IP值或EIP值）压入堆栈，再把指令指定的通用寄存器或内存单元的内容送给IP（16位）或EIP（32位），最后把控制权交给子程序。
- 说明：这种指令使用段内间接寻址方式，指令指定的通用寄存器或内存单元中存放段内偏移量。

- 可以把子程序入口地址的偏移量送给通用寄存器或内存单元，通过它们实现段内间接调用。
- `CALL EBX` ;子程序入口地址的32位偏移量在EBX寄存器中
- `CALL WORD PTR [BX]`;子程序入口地址的16位偏移量在数据段的BX所指向的字型内存单元中
- `CALL DWORD PTR [EBX]` ;子程序入口地址的32位偏移量在数据段的EBX所指向的双字型内存单元中

# 段间调用

- 段间直接调用
- 格式：CALL FAR PTR PROCEDURE
- 功能：调用PROCEDURE子程序。执行时先把返回地址（当前IP值和CS值）压入堆栈，再把指令中的偏移量部分送给IP，段基址部分送给CS，最后把控制权交给子程序。对于保护模式，执行时先把当前EIP值压入堆栈，再把当前CS值压入堆栈（注意CS是16位的，但是应以32位形式入栈，其中高16位无意义），然后再把指令中的偏移量部分送给EIP，段选择符部分送给CS，最后把控制权交给子程序。

- 设子程序B与CALL指令不在同一段内，则段间直接调用B子程序的指令是：
- CALL FAR PTR B

- 段间间接调用
- 格式：CALL M
- 功能：调用子程序。执行时先把返回地址（当前指令指针寄存器值和当前CS值）压入堆栈，再把M的16位或32位送给指令指针寄存器，高16位送给CS，最后把控制权交给子程序。

- 说明：段间间接寻址方式，转向地址放在内存变量中。对于8086等16位机，用双字变量的低16位取代指令指针寄存器IP值，高16位取代CS值实现段间转移。在保护模式下，由于偏移量是32位的，则转向地址只能放在内存的3字长（用FWORD定义）变量中，用三字变量的低32位取代指令指针寄存器EIP值，高16位取代CS值实现段间转移。该双字或三字变量的地址由与存储器操作数有关的寻址方式获得。



- 若子程序B的入口地址（偏移量和段基址）放在变量VAR中，即可通过VAR实现段间间接调用。如下所示：
- `CALL DWORD PTR VAR`
- `CALL FWORD PTR [EBX][EDI]`
- ;从DS:[EBX+EDI]单元中得到子程序的入口地址实现调用

# 子程序返回指令

- 执行这组指令可以返回到被调用处。有两条返回指令，它们都不影响标志。
- 返回指令RET (Return from procedure)
- 格式：RET
- 功能：按照CALL指令入栈的逆序，从栈顶弹出返回地址送指令指针寄存器IP（或EIP），若子程序是FAR型还需再弹出一个字到CS（若为32位操作数，则弹出一个双字，其中低字送CS，高字丢弃），然后返回到主程序继续执行。
- **无论**子程序是NEAR型还是FAR型，返回指令的汇编格式总是用RET表示。但经汇编后会产生**不同**的机器码。在DEBUG中，段间返回指令被反汇编成RETF。

- 带立即数的返回指令
- 格式：RET imm<sub>16</sub>
- 功能：按照CALL指令入栈的逆序，从栈顶弹出返回地址（偏移量送IP或EIP，若子程序是FAR型还需再弹出一个字到CS），返回到主程序，并修改栈顶指针 $SP = (SP) + imm_{16}$ 。若为32位操作数则使用ESP。
- 注：其中imm<sub>16</sub>是16位的立即数，设通过堆栈给子程序传递了n个字型参数，则imm<sub>16</sub>=2n；若传递了n个双字型参数，则imm<sub>16</sub>=4n。
- 修改堆栈指针是为了废除堆栈中主程序传递给子程序的参数。

- 保护模式的段间调用与返回操作很复杂，好在这些复杂性对在Windows下的32位汇编语言编程人员是**不可见的**，我们只需会使用汇编级指令即可。例如：系统会**自动**为准备要运行的用户程序的代码段、数据段和堆栈段全部定义好段描述符的内容，并且把CS、DS、ES、SS等段选择符指向正确的描述符

# 中断调用与返回指令

- 中断就是使计算机暂时挂起正在执行的进程而转去处理某事件，处理完后再恢复执行原进程的过程。对某事件的处理实际上就是去执行一段例行程序，该程序被称为中断处理例行程序或中断处理子程序，简称为中断子程序。中断分为内中断（或称软中断）和外中断（或称硬中断）

- 中断向量
- 中断向量就是中断处理子程序的入口地址。在PC中规定中断处理子程序为FAR型，所以8086的每个中断向量占用4个字节，其中低两个字节为中断向量的偏移量部分，高两个字节为中断向量的段基址部分。
- 中断类型号
- PC共支持256种中断，相应编号为0 ~ 255，把这些编号称为中断类型号。

- 中断向量表
- 256种中断应该有256个中断处理子程序，显然应有256个中断向量，把这些中断向量按照中断类型号由小到大的顺序排列形成一个中断向量表，该表长为 $4 \times 256 = 1024$ 字节，从内存的0000:0000地址开始存放，占用内存最低端的0 ~ 3FFH单元

- 内中断调用指令
- 内中断调用指令INT是程序员根据需要在程序的适当位置安排的。它完全受程序控制，不像外中断那样由硬件随机产生。
- 格式：INT n
- 其中，n为中断类型号。
- 功能：中断当前正在执行的程序，把当前的FLAGS、CS、IP值依次压入堆栈（保护断点），并从中断向量表的 $4n$ 处取出n类中断向量，其中 $(4n) \rightarrow IP$ ， $(4n+2) \rightarrow CS$ ，然后转去执行中断处理子程序。



- INT 21H
- 21H中断为系统功能调用中断，执行时把当前的FLAGS、CS、IP值依次压入堆栈，并从中断向量表的84H处取出21H类中断向量，其中（84H）→ IP，（86H）→ CS，然后转去执行中断处理子程序。

# 中断返回指令

- 当从中断处理子程序返回时要使用中断返回指令IRET和IRETD，中断返回指令应放在中断处理子程序的末尾。一般将16位实模式代码中的中断返回指令写做IRET，而将32位保护模式代码中的中断返回指令写做IRETD。
- 在实模式下，IRET从栈顶弹出3个字分别送入IP、CS、FLAGS寄存器（按中断调用时的逆序恢复断点），把控制返回到原断点继续执行。在保护模式下，IRETD从栈顶弹出3个双字分别送入EIP、CS、EFLAGS寄存器），返回被中断的程序。

- 保护模式下把这里所称的中断分为中断和异常两大类，系统并不使用中断向量表而是使用中断描述符表IDT，其中每个描述符8个字节，描述符中包含中断子程序16位的段选择符和32位的偏移量等信息。在响应中断或者处理异常时，CPU把中断类型号作为中断描述符表IDT中描述符的索引，取得一个描述符，从中得到中断/异常处理程序的入口地址。

# 处理机控制指令

汇编格式	功    能	影响标志
CLC (Clear Carry)	把进位标志 CF 清 0	CF
STC (Set Carry)	把进位标志 CF 置 1	CF
CMC (Complement Carry)	把进位标志 CF 取反	CF
CLD (Clear Direction)	把方向标志 DF 清 0	DF
STD (Set Direction)	把方向标志 DF 置 1	DF
CLI (Clear Interrupt)	把中断允许标志 IF 清 0	IF
STI (Set Interrupt)	把中断允许标志 IF 置 1	IF

名称	汇编格式	功 能	说 明
空操作	NOP (No Operation)	空操作	CPU 不执行任何操作，其机器码占用 1 个字节。
停机	HLT (Halt)	使 CPU 处于停机状态	只有外中断或复位信号才能退出停机，继续执行。
等待	WAIT (Wait)	使 CPU 处于等待状态	等待 TEST 信号有效后，方可退出等待状态。
锁定前缀	LOCK (Lock)	使总线锁定信号有效	LOCK 是一个单字节前缀，在其后的指令执行期间，维持总线的锁存信号直至该指令执行结束。

# 块操作指令

指令名称	功 能
MOVSB, MOVSW, MOVSD	将一个内存操作数复制到另一个内存操作数
CMPSB, CMPSW, CMPSD	比较两个内存操作数的大小
SCASB, SCASW, SCASD	将内存操作数与 AL, AX, EAX 比较
STOSB, STOSW, STOSD	将 AL, AX, EAX 保存在内存操作数中
LODSB, LODSW, LODSD	读入内存操作数放入 AL, AX, EAX 中

指令名称	源操作数	目标操作数
MOVSB, MOVSW, MOVSD	DS:[ESI]	ES:[EDI]
CMPSB, CMPSW, CMPSD	DS:[ESI]	ES:[EDI]
SCASB, SCASW, SCASD	累加器	ES:[EDI]
STOSB, STOSW, STOSD	累加器	ES:[EDI]
LODSB, LODSW, LODSD	DS:[ESI]	累加器

- 源操作数和目标操作数
- 块操作指令的源操作数是DS:[ESI]所指向的内存单元；目标操作数是ES:[EDI]所指向的内存单元。ESI是源操作数的地址；EDI是目标操作数的地址。

- (1) MOVS<sub>B/W/D</sub>将ESI所指向的字节/字/双字复制到EDI所指向的字节/字/双字。
- (2) CMPS<sub>B/W/D</sub>将ESI和EDI所指向的字节/字/双字进行比较。
- (3) SCAS<sub>B/W/D</sub>将EDI所指向的字节/字/双字和AL/AX/EAX进行比较。
- (4) STOS<sub>B/W/D</sub>将AL/AX/EAX保存到EDI所指向的字节/字/双字中。
- (5) LODS<sub>B/W/D</sub>将ESI所指向的字节/字/双字读入到AL/AX/EAX中。



# 方向标志和地址指针

- 块操作指令会自动地修改ESI和EDI，使它们指向下一个源操作数和目标操作数。CPU的EFLAGS中有一个标志位DF，由DF来决定ESI和EDI是增加还是减小。DF=0时，地址增加；DF=1时，地址减小。

指令名称	源操作数指针 ESI	目标操作数指针 EDI
MOVS <sub>B</sub> , MOVSW, MOVSD	$ESI \leftarrow ESI \pm 1/2/4$	$EDI \leftarrow EDI \pm 1/2/4$
CMPS <sub>B</sub> , CMPSW, CMPSD	$ESI \leftarrow ESI \pm 1/2/4$	$EDI \leftarrow EDI \pm 1/2/4$
SCAS <sub>B</sub> , SCASW, SCASD	AL / AX / EAX	$EDI \leftarrow EDI \pm 1/2/4$
STOS <sub>B</sub> , STOSW, STOSD	AL / AX / EAX	$EDI \leftarrow EDI \pm 1/2/4$
LODS <sub>B</sub> , LODSW, LODSD	$ESI \leftarrow ESI \pm 1/2/4$	AL / AX / EAX

# 重复前缀

- 重复前缀一共有3种形式：REP, REPZ, REPNZ，它放在块操作指令的**前面**。
- 使用重复前缀时，**ECX**的作用是给出内存中连续操作数的个数，也就是块操作指令的最大重复次数。每执行一次块操作指令，ECX就自动**减1**。ESI和EDI自动**修改**。

指令名称	重复前缀	重复执行条件
MOVSB, MOVSW, MOVSD STOSB, STOSW, STOSD LODSB, LODSW, LODSD	REP	ECX!=0
CMPSB, CMPSW, CMPSD SCASB, SCASW, SCASD	REPZ REPNZ	ECX!=0 且 ZF=1 ECX!=0 且 ZF=0

# 块操作指令示例

- 数组复制
- 将数组Array1复制给数组Array2。
- Array1      DWORD   1, 10, 100, 1000, 10000
- Array2      DWORD   5 DUP (0)
- LEA          ESI, Array1
- LEA          EDI, Array2
- CLD
- MOV          ECX, 5
- REP          MOVSD

- 缓冲区初始化
- 将数组Array1的每个元素的初值设为0H。
- dArray      DWORD   7 DUP (?)
- LEA          EDI, dArray              ; EDI指向第1个存储单元
- CLD                                      ; 地址由低至高
- MOV          ECX, 7                    ; 存储7次
- MOV          EAX, 0                    ; 存储内容为0
- REP          STOSD                    ; 以双字为单位存储

# 小结