



# 第三章 指令系统

张华平 副教授 博士

Email: [kevinzhang@bit.edu.cn](mailto:kevinzhang@bit.edu.cn)

Website: <http://www.nlpir.org/>

@ICTCLAS张华平博士

大数据搜索挖掘实验室 (wSMS@BIT)





# 重点范围

- (1) 【重点讲解】数据寻址方式、数据运算指令、程序控制指令
- (2) 【一般性讲解】处理机控制指令、块操作指令





# 基本概念

指令集和指令系统

指令的构成：操作码 操作数

如何给出操作数或操作数地址？

操作数域的比较复杂，这就是寻址方式要解决的问题。





# 寻址方式/指令系统

寻址方式:

与数据有关的寻址方式

与转移地址有关的寻址方式

指令系统:

数据传送/算术运算/逻辑

程序控制/串操作/. . .





## 3.1 数据寻址方式

与数据有关的寻址方式与操作数有关。

格式：MOV 目标, 源

功能：源→目标

目标和源是操作数域，各自可以有不同的寻址方式。下边以源操作数的寻址方式为例说明。







# CPU操作数寻址：1.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，把该操作数称为立即数。

**注意：**立即寻址方式只能出现在源操作数的位置。其它寻址方式既可以出现在源也可以出现在目标操作数位置。



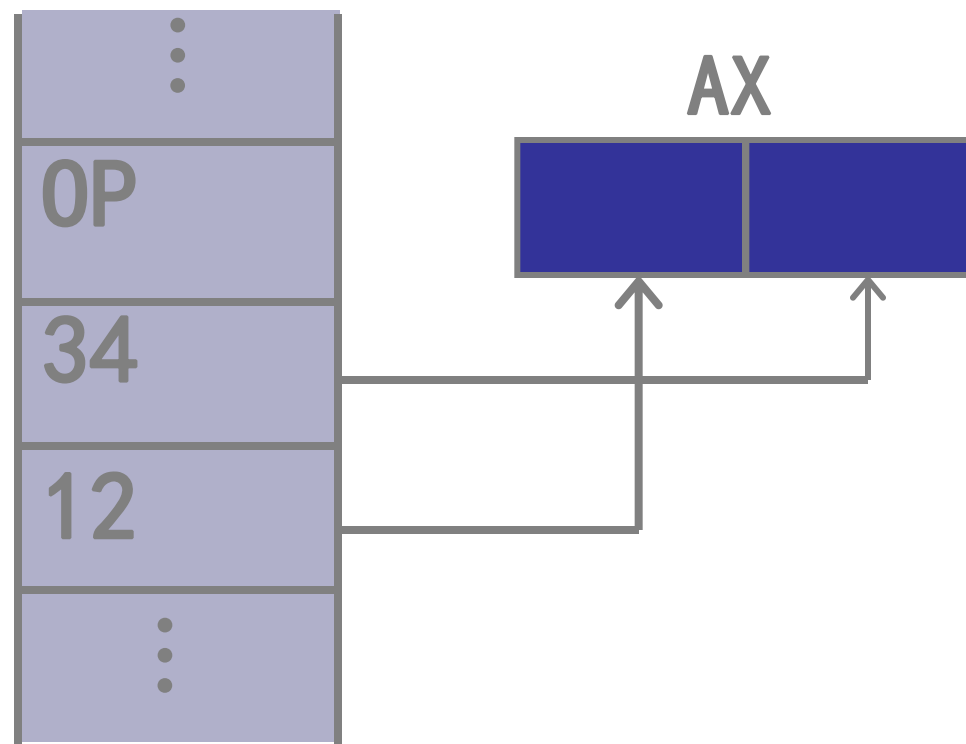


## 存储器

例1.

MOV AX, 1234H

; (AX) = 1234H



立即寻址方式





## CPU操作数寻址： 2. 寄存器寻址方式

操作数直接包含在寄存器中，由指令指定寄存器号的寻址方式。

寄存器可以是8位、16位、32位通用寄存器或16位段寄存器(但CS不能用于目标)。







MOV BX, AX ; (BX) = (AX)

MOV DI, 5678H ; (DI) = 5678H

MOV AL, 78H ; (AL) = 78H

MOV ECX, 7890ABCDH  
; (ECX) = 7890ABCDH

其中BX、AX、DI、AL、ECX均为寄存器寻址方式。





表3-1 段基址和偏移量的约定情况

操作类型	约定段寄存器	允许指定的段寄存器	偏移量
1. 指令	CS	无	IP
2. 堆栈操作	SS	无	SP
3. 普通变量	DS	ES、SS、CS	EA
4. 字符串指令的源串地址	DS	ES、SS、CS	SI
5. 字符串指令的目标串地址	ES	无	DI
6. BP用作基址寄存器	SS	DS、ES、CS	EA





当访内操作类型允许指定段寄存器时，可以使用**段超越前缀**指定。

**功能：**明确指出本条指令所要寻址的内存单元在哪个段中。

**格式：**段寄存器名：

**例.** ES:、CS:、SS:等。

MOV AX, ES:VER





# 存储器操作数寻址：1直接寻址方式

操作数的有效地址直接包含在指令中的寻址方式。

有效地址存放在代码段的指令操作码之后，但操作数本身在存储器中，所以必须先求出操作数的物理地址。普通变量缺省情况是存放在DS所指向的数据段，但允许使用段超越前缀指定为其它段。

这种寻址方式常用于存取简单变量。





## 例7. MOV AL, [78H]

[78H] 为直接寻址方式。由于没有使用段超越前缀, 因此缺省使用 DS 段寄存器。

在实模式中, 若  $DS = 3000H$

$$(30078H) = 12H$$

则 DS:78H 表示物理地址 30078H

该指令的执行结果是  $(AL) = 12H$







由于在汇编语言中用符号表示地址, 所以指令“**MOV AL, VAR**”中的源操作数寻址方式是直接寻址, **VAR**是内存的符号地址。实际上在汇编语言源程序中所看到的直接寻址方式都是用符号表示的, 只有在**DEBUG**环境下, 才有**[78H]**这样的表示。





例8. MOV AL, ES: [78H]

(ES: 78H) → AL

该指令的源操作数前使用了段超越前缀“ES:”，明确表示使用附加数据段中的变量。

注意 [78H] 与 ES: [78H] 表示不同的物理地址，只是段内偏移量相同而已。



# 存储器操作数寻址：2寄存器间接寻址方式

操作数有效地址在基址寄存器BX、BP或变址寄存器SI、DI中，而操作数在存储器中的寻址方式。

对于386以上CPU，这种寻址方式允许使用任何32位通用寄存器。





若指令中使用的是BX、SI、DI、EAX、EBX、ECX、EDX、ESI、EDI，则缺省情况操作数在数据段，即它们默认与DS段寄存器配合。

若使用的是BP、EBP、ESP，则缺省情况默认与SS段寄存器配合。

均允许使用段超越前缀。



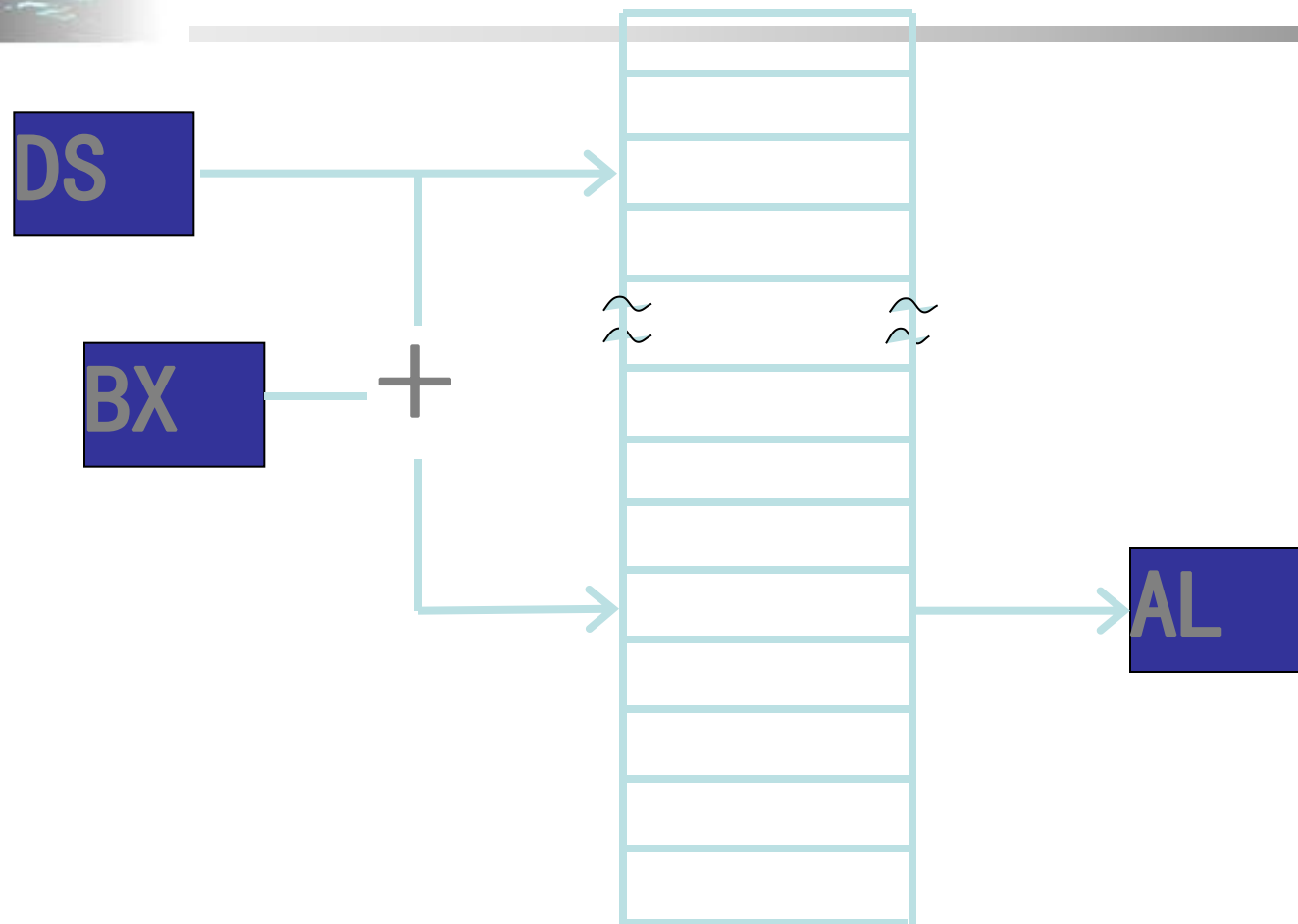


例9. MOV AL, [BX]  
; (DS:[BX]) → AL  
; 如下图所示

其中[BX]为寄存器间接寻址方式，注意它与寄存器寻址方式在汇编格式上的区别。







寄存器间接寻址方式执行情况





对于MOV AL, [BX]

若：(DS) = 3000H, (BX) = 78H

(30078H) = 12H

则：

物理地址 =  $10\text{H} \times (\text{DS}) + (\text{BX}) = 30078\text{H}$

该指令的执行结果是 (AL) = 12H





例10. `MOV AX, [BP] ; (SS:[BP]) → AX`

若  $(SS) = 2000H$ ,  $(BP) = 80H$

$(20080H) = 12H$ ,  $(20081H) = 56H$

物理地址:  $10H \times (SS) + (BP) = 20080H$

执行结果:  $(AX) = 5612H$ 。

利用这种寻址方式再配合修改寄存器内容的指令可以方便地处理一维数组。



# 存储器操作数寻址：3寄存器相对寻址方式

操作数的有效地址是一个基址(BX、BP)或变址寄存器(SI、DI)的内容和指令中给定的一个位移量(displacement)之和。

386以上允许使用任何32位通用寄存器。位移量可以是一个字节、一个字、一个双字(386以上)的带符号数。

$$EA = (\text{基址} < \text{或变址} > \text{寄存器}) + \text{disp}$$

$$\text{或: } EA = (\text{32位通用寄存器}) + \text{disp}$$





与段寄存器的配合情况：若指令中寄存器相对寻址方式使用BP、EBP、ESP，则默认与SS段寄存器配合。使用其它通用寄存器，则默认与DS段寄存器配合。均允许使用段超越前缀。







例11. `MOV AL, 8 [BX]`

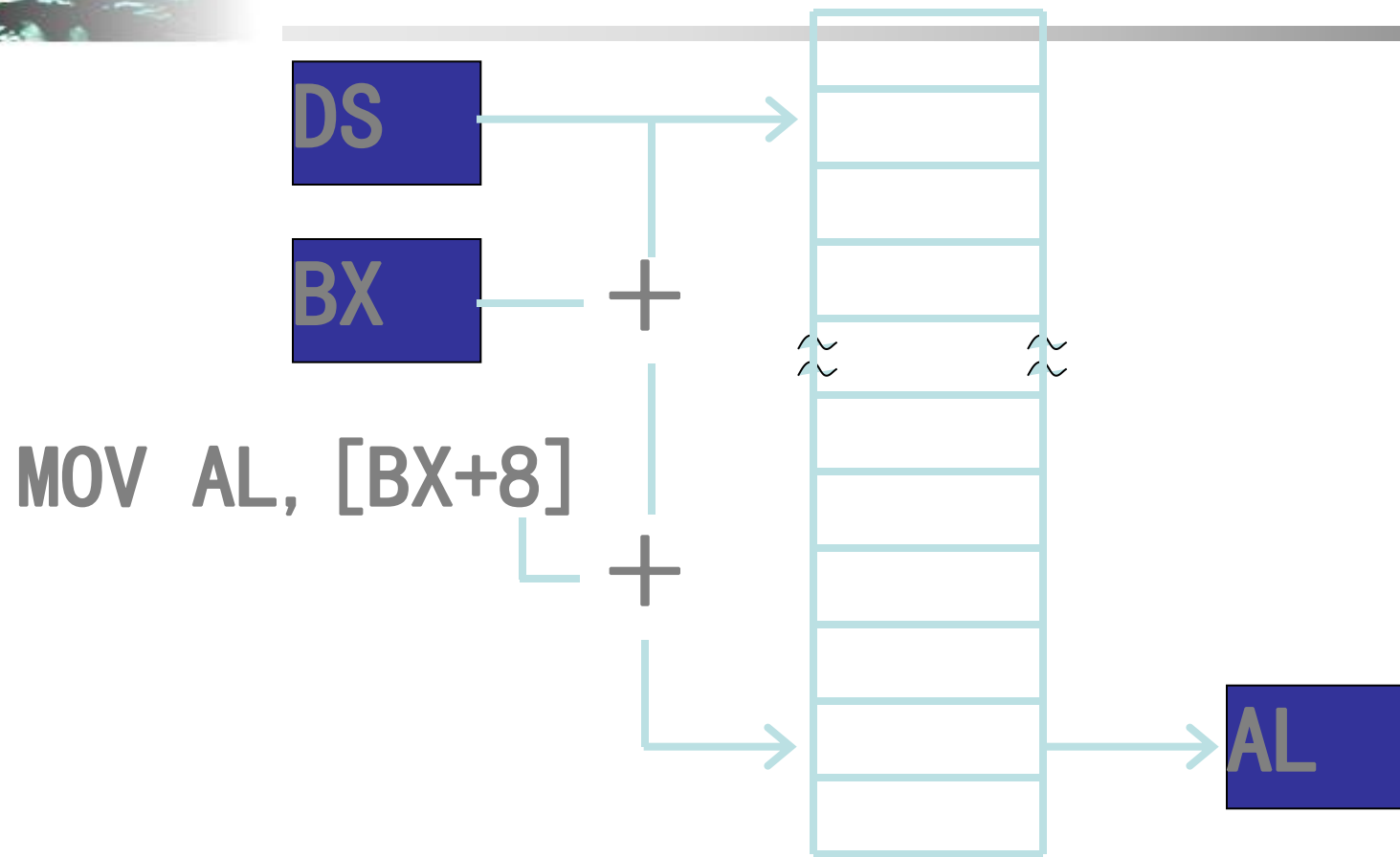
可以表示为: `MOV AL, [BX+8]`

若  $(DS) = 3000H$ ,  $(BX) = 70$   
 $(30078H) = 12H$

则物理地址=30078H

该指令的执行结果是  $(AL) = 12H$ 。

其中8 [BX]为寄存器相对寻址方式, 该指令执行情况如下图所示。



寄存器相对寻址执行情况





例12. `MOV AL, TABLE[BX]`

也可以表示为: `MOV AL, [BX+TABLE]`

其中TABLE为位移量的符号表示, 指令执行结果是  $(DS:[BX+TABLE]) \rightarrow AL$ 。

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, 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。





# 存储器操作数寻址：4基址变址寻址方式

操作数的有效地址是一个基址寄存器和一个变址寄存器的内容之和。

386以上允许使用变址部分除ESP以外的任何两个32位通用寄存器组合。





缺省使用段寄存器的情况由基址寄存器决定。若使用BP、ESP或EBP，缺省与SS配合；若使用BX或其它32位通用寄存器，则缺省与DS配合。允许使用段超越前缀。  
即： $EA = (\text{基址寄存器}) + (\text{变址寄存器})$

80386以上支持的32位基址变址寻址方式组合如下图。







$$EA = \left\{ \begin{array}{c} \text{基址} \\ \text{EAX} \\ \text{EBX} \\ \text{ECX} \\ \text{EDX} \\ \text{ESP} \\ \text{EBP} \\ \text{ESI} \\ \text{EDI} \end{array} \right\} + \left\{ \begin{array}{c} \text{变址} \\ \text{EAX} \\ \text{EBX} \\ \text{ECX} \\ \text{EDX} \\ \\ \text{EBP} \\ \text{ESI} \\ \text{EDI} \end{array} \right\}$$

80386以上32位基址变址寻址方  
式组合





例13. MOV AL, [BX] [SI]

; (DS: [BX+SI]) → AL

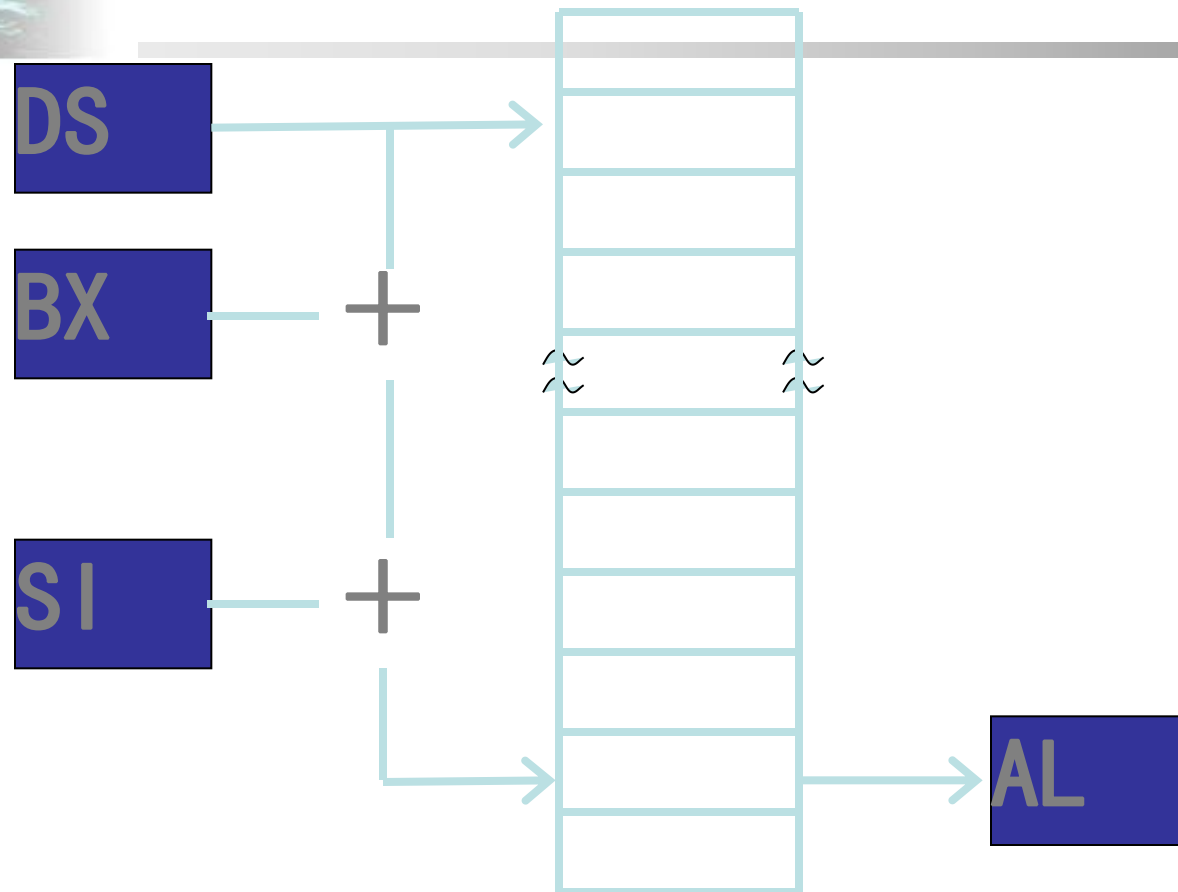
该指令可以表示为：

MOV AL, [BX+SI]

动画演示

执行情况如下图所示。

使用这种寻址方式可以访问一维数组，其中BX存放数组起始地址的偏移量，SI存放数组元素的下标乘以元素的长度，下标从0开始计数。



## 基址变址寻址方式执行情况





# 存储器操作数寻址： 5相对基址变址寻址方式

操作数的有效地址是一个基址和一个变址寄存器的内容和指令中给定的一个位移量之和。

386以上CPU允许使用变址部分除ESP以外的任何两个32位通用寄存器组合。位移量可以是一个字节、一个字、一个双字（386以上）的带符号数。





缺省使用段寄存器的情况由基址寄存器决定。

若使用BP、EBP或ESP，缺省与SS配合；若使用BX或其它32位通用寄存器，缺省与DS配合。

允许使用段超越前缀。

即： $EA = (\text{基址寄存器}) + (\text{变址寄存器}) + \text{disp}$

80386以上支持的32位相对基址变址寻址方式组合如下图。







$$EA = \left\{ \begin{array}{c} \text{基址} \\ \text{EAX} \\ \text{EBX} \\ \text{ECX} \\ \text{EDX} \\ \text{ESP} \\ \text{EBP} \\ \text{ESI} \\ \text{EDI} \end{array} \right\} + \left\{ \begin{array}{c} \text{变址} \\ \text{EAX} \\ \text{EBX} \\ \text{ECX} \\ \text{EDX} \\ \text{EBP} \\ \text{ESI} \\ \text{EDI} \end{array} \right\} + \text{disp}$$

80386以上32位相对基址变址  
寻址方式组合





例14. `MOV AL, ARY[BX][SI]`  
; `(DS:[BX+SI+ARY]) → AL`

可表示为:

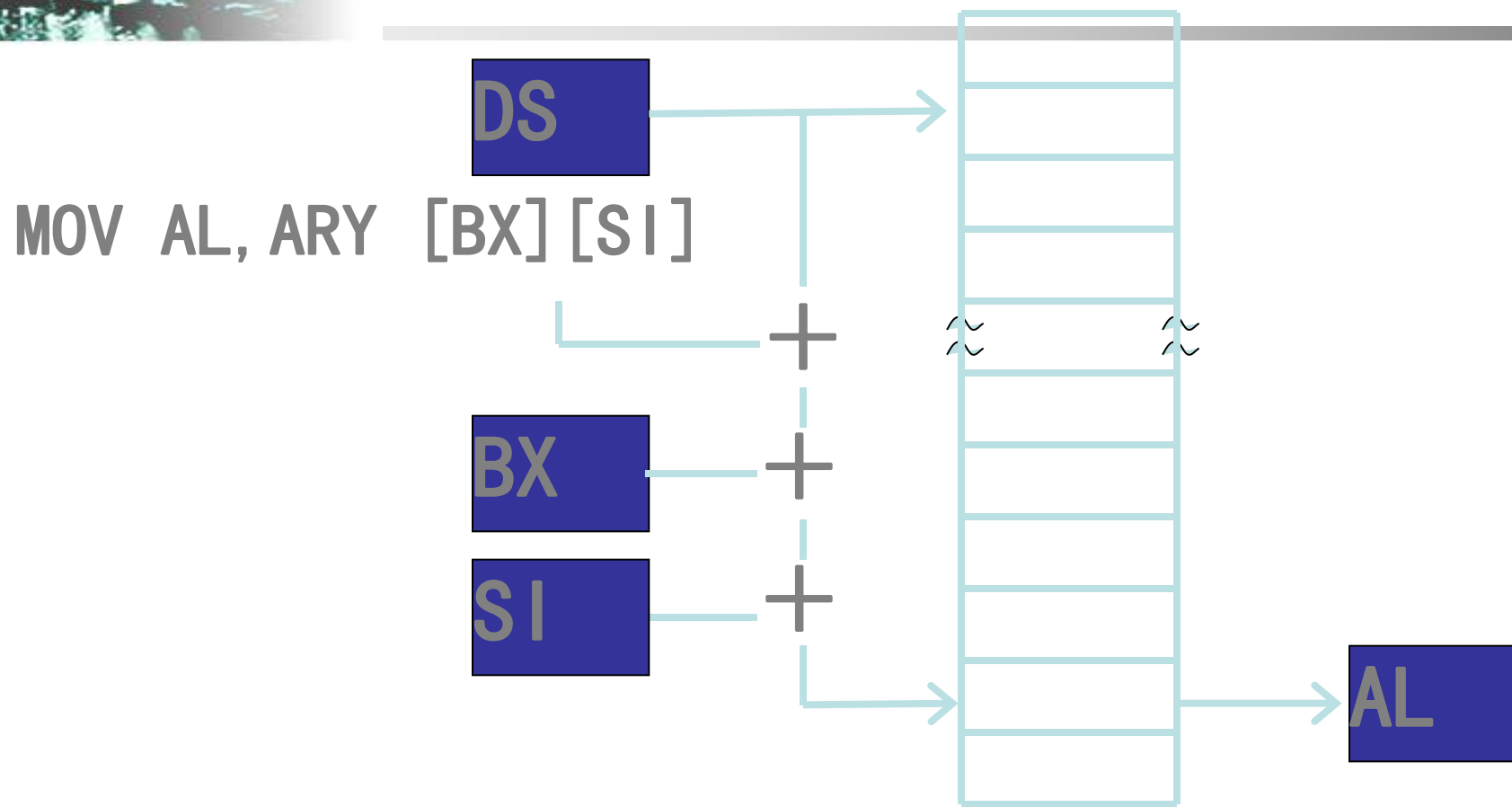
`MOV AL, ARY[BX+SI]`

或 `MOV AL, [BX+SI+ARY]`

动画演示

执行情况如下图所示。





## 相对基址变址寻址方式





使用这种寻址方式可以  
访问形如`ARY[3][3]`的二维数组，下标  
从0开始计数。





# 存储器操作数寻址： 6比例变址寻址方式

这种寻址方式是80386以上的微处理器才提供的。  
操作数的有效地址由以下几部分相加组成：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(缺省值)、2、4或8，1可用来寻址字节数组，2可用来寻址字数数组，4可用来寻址双字数数组，8可用来寻址4字数数组。位移量可以是一个字节、一个双字的带符号数。缺省使用段寄存器的情况由基址寄存器决定。

即： $EA = (\text{基址寄存器}) + (\text{变址寄存器}) \times \text{比例因子} + \text{disp}$





比例变址寻址方式组合如下图所示。可以看出，它实际上是386以上CPU存储器操作数寻址方式的通用公式。除比例因子不能单独使用外，其它各项都可以独立存在或以组合形式出现。

例如若只含有第一列或第二列，就变成寄存器间接寻址方式。若含有第一列和第二列，就变成基址变址寻址方式。





$$EA = \left\{ \begin{array}{l} \text{EAX} \\ \text{EBX} \\ \text{ECX} \\ \text{EDX} \\ \text{ESP} \\ \text{EBP} \\ \text{ESI} \\ \text{EDI} \end{array} \right\} + \left\{ \begin{array}{l} \text{EAX} \\ \text{EBX} \\ \text{ECX} \\ \text{EDX} \\ \text{EBP} \\ \text{ESI} \\ \text{EDI} \end{array} \right\} \times \left\{ \begin{array}{l} 1 \\ 2 \\ 4 \\ 8 \end{array} \right\} + \left\{ \begin{array}{l} \text{无} \\ 8\text{位} \\ 32\text{位} \end{array} \right\}$$

比例变址寻址方式组合





例15. `MOV EAX, ARY[EBX][ESI]`  
;  $(DS: [ARY + EBX + ESI]) \rightarrow EAX$

例16. `MOV CX, [EAX+2*EDX]`  
;  $(DS: [EAX + 2*EDX]) \rightarrow CX$

例17. `MOV EBX, [EBP+ECX*4+10H]`  
;  $(SS: [EBP + ECX*4 + 10H]) \rightarrow EBX$

例18. `MOV EDX, ES:ARY[4*EBX]`  
;  $(ES: [ARY + 4*EBX]) \rightarrow EDX$

使用这种寻址方式可以方便地访问数组，其中变址寄存器的内容等于数组下标，比例因子为元素长度。



- 立即寻址方式; MOV AX, 5
- 寄存器寻址方式; MOV AX, 5
- 直接寻址方式; MOV AX, [78H]
- 寄存器间接寻址方式 MOV AX, [BX]
- 寄存器相对寻址 MOV AX, 8[BX]
- 基址变址寻址 MOV AX, [BX][SI]
- 相对基址变址: MOV AX, ARRAY[BX][SI]
- 比例变址寻址方式: MOV EAX, ES:ARRAY[4\*BX]



## 3.2 数据运算指令

指令告诉CPU执行什么样的操作及操作数从哪里得到。指令可以用大写、小写或大小写字母混合的方式书写。

- ☞ 指令格式和功能
- ☞ 指令所影响的标志位
- ☞ 指令执行周期 \*
- ☞ 指令机器长度 \*

汇编语言是面向机器的，指令和机器码基本上是一一对应的，所以它们的实现取决于硬件。





- 数据传送指令
- 算术运算指令[2进制/10进制]
- 逻辑指令
- 程序控制指令
- 处理机控制指令
- 串操作指令
- 条件字节设置指令





## 3.2.1 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，本组的其它指令不影响标志。

1. 通用数据传送指令；
2. 堆栈操作指令；
3. 输入输出指令；
4. 查表转换指令；
5. 地址传送指令；
6. 标志传送指令；





# 1. 通用数据传送指令

## (1). 传送指令 **MOV**

格式: **MOV DST, SRC**

功能: **SRC (源) → DST (目标)**

说明: **MOV**指令可以实现一个字节、一个字、一个双字的数据传送,

注意: 源操作数和目标操作数的数据类型匹配问题, 即应同为字节、字或双字型数据。

**MOV** 指令可实现的数据传送方向如下图所示。



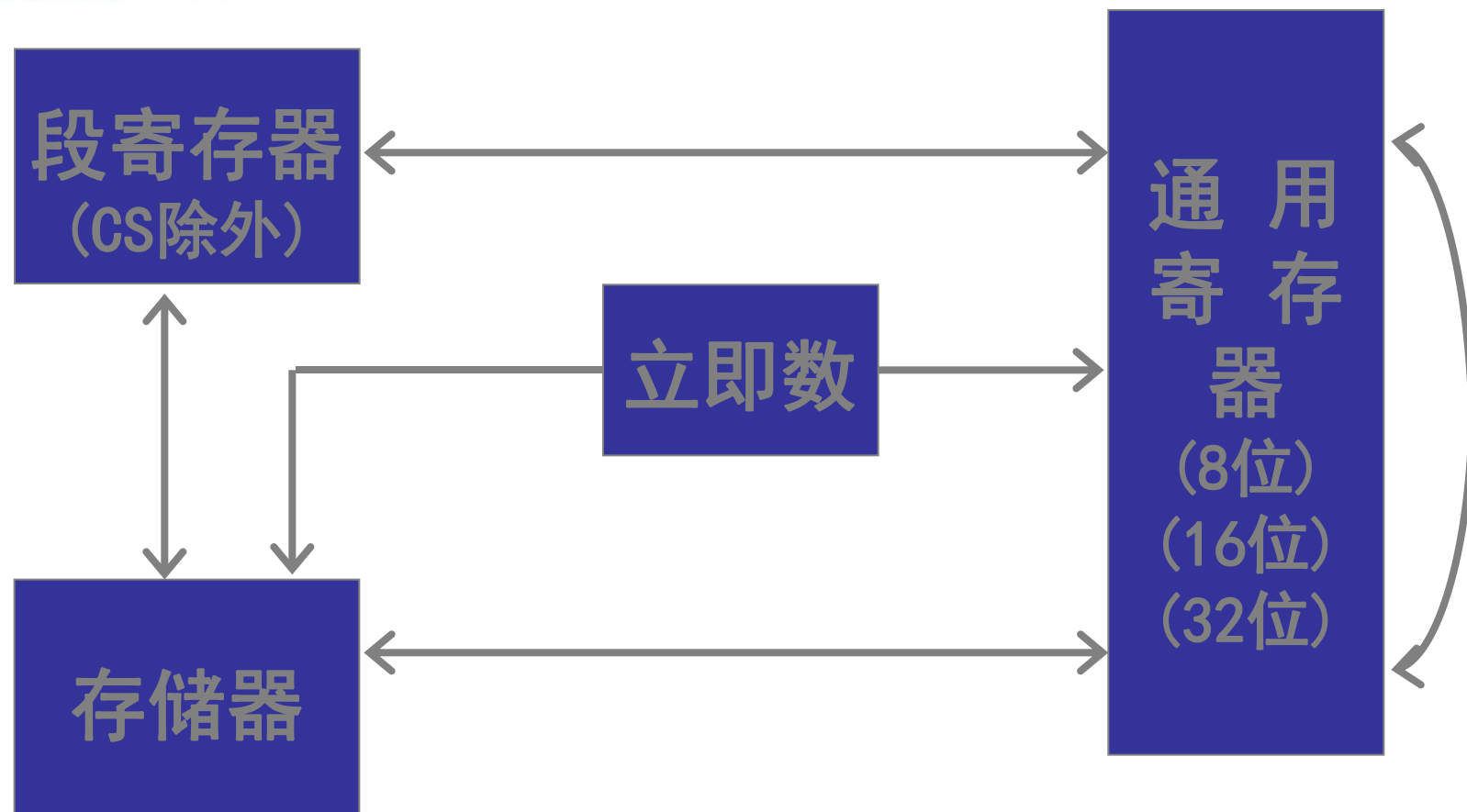


图3-7 MOV指令数据传送方向示意图







## 数据传送规则：

- 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。





例：

```
MOV    AL, 5
MOV    DS, AX
MOV    [BX], AX
MOV    ES:VAR, 12
MOV    WORD PTR [BX], 10
MOV    EAX, EBX
```

“WORD PTR”，它明确指出BX所指向的内存单元为字型，立即数12被汇编为16位的二进制数。若要生成8位的二进制数，需要用

“BYTE PTR”，这里的类型显式说明是必须的

。





(2). 带符号扩展的数据传送指令 **MOVSX** (  
386以上)

格式: **MOVSX** DST, SRC

(3). 带零扩展的数据传送指令 **MOVZX**  
(386以上)

格式: **MOVZX** DST, SRC





## (4) 堆栈操作指令

堆栈数据的存取原则是“LIFO”。

堆栈段段基址→SS

堆栈栈顶地址→SP/ESP

堆栈用途：

对现场数据的保护与恢复、子程  
地址的保护与恢复等。

序与中断服务返回



## 1). 进栈指令 **PUSH**

格式: **PUSH SRC**

功能:  $SP = SP - 2$

$SS:SP = (SRC)$

说明: SRC可以是16位或32位(386以上)的寄存器操作数或存储器操作数。在80286以上的机器中, SRC还可以是立即数。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。





## 2). 出栈指令 **POP**

格式: POP DST

功能:  $(DST) = SS:SP$

$SP = SP + 2$

说明: DST可以是16位或32位(386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。若DST是16位,则堆栈指针加2;若DST是32位,则堆栈指针加4。





## 常见指令序列:

PUSH AX

PUSH BX

.....

PUSH 1234H ; 80286以

上可用

POP DX

.....

POP BX

POP AX

注意堆栈的初始设置/堆栈异常





## (5). 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, 操作数可以是8位、16位、32位。该指令可能的组合是:

XCHG 寄存器操作数, 寄存器操作数

XCHG 寄存器操作数, 存储器操作数

XCHG 存储器操作数, 寄存器操作数





例：

设：(AX) = 1234H, (BX) = 4567H

则：XCHG AX, BX

执行后 (AX) = 4567H, (BX) = 1234H

由于系统提供了这个指令，因此，采用其他方法交换时，速度将会较慢，并需要占用更多的存储空间，编程时要避免这种情况。





## 2. 输入输出指令

### (1). 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, 相应的累加器选择AL、AX、EAX。







例.

IN AL, 61H ; AL ← (61H端口)

IN AX, 20H ; AX ← (20H端口)

MOV DX, 3F8H

IN AL, DX ; AL ← (3F8H端口)

IN EAX, DX

; EAX ← (DX所指向的端口)





## (2). 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同IN指令。





例.

MOV AL, 0

OUT 61H, AL ;61H端口 ← (AL)

;关掉PC扬声器

MOV DX, 3F8H

OUT DX, AL ;3F8H端口 ← (AL)

;向COM1端口输出一个字

符





### 3. 地址传送指令

这类指令传送的是操作数的地址，而不是操作数本身。





## 传送有效地址指令 **LEA**

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。

例.

LEA SI, TAB

LEA BX, TAB [SI]

LEA DI, ASCTAB [BX] [SI]

**动画**





## 4. 标志传送指令

指令的操作过程

考虑这些指令对标志位的影响！





(1). 16位标志进栈指令 **PUSHF**

格式: PUSHF

功能: SP减2; FLAGS→栈

顶单元。

(2). 16位标志出栈指令 **POPF**

格式: POPF





(3). 32位标志进栈指令 **PUSHFD**

格式: PUSHFD

功能: ESP减4; EFLAGS→栈顶

。

(4). 32位标志出栈指令 **POPFD**

格式: POPFD

(5). 标志送AH指令 **LAHF** [低8位]

格式: LAHF

(6). AH送标志寄存器指令 **SAHF**

格式: SAHF





## 3.2.2 算术运算指令

⑩ 二进制算术运算指令

⑩ 十进制算术运算指令

对于其中的双操作数指令，其两个操作数寻址方式的限定同MOV指令。



1. 实现**二进制**算术运算。
2. 操作数和运算结果为**二进制**。
3. 操作数及计算结果:8位、16位、32位无符号或带符号二进制数(在书写指令时可以用十进制形式表示)。
4. 带符号数在机器中用**补码**形式表示, 最高位为符号位, 0表示正数, 1表示负数。





# 1、类型转换指令

类指令实际上是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。





## (1). 字节扩展成字指令 **CBW**

格式:**CBW**

功能:把AL寄存器中的符号位值扩展到AH中

```
MOV    AL, FFH
```

例. 

```
MOV    AL, 5
```

```
CBW                ; (AH) = 0, AL值不变
```

```
MOV    AL, 98H
```

```
CBW                ; (AX) = 0FF98H, AL值
```

不变





## (2). 字扩展成双字指令 CWD

格式: **CWD**

功能: 把AX的符号位值扩展到DX中

## (3). 双字扩展成四字指令 CDQ

格式: **CDQ** (386以上)

功能: EAX符号位扩展到EDX中

## (4). AX符号位扩展到EAX指令 CWDE

格式: **CWDE** (386以上)

功能: AX寄存器符号位扩展到EAX高16位





## 2、二进制加法指令

任何一条二进制加、减法指令均适用于带符号数和无符号数运算。





## (1). 加法指令 **ADD**

格式: `ADD DST, SRC`

功能:  $(DST) + (SRC) \rightarrow DST$

说明: 对操作数的限定同MOV指令。

标志: 影响**OF**、SF、ZF、AF、PF、**CF**标志

例. `ADD AL, BL`

`ADD CL, 6`

`ADD WORD PTR[BX], 56`

`ADD EDX, EAX`

[动画](#)







## ADD运算的说明:

对于两个二进制数进行加法运算，如果把数解释为无符号数，其结果可能是溢出的，而如果把数解释为带符号数，其结果可能是不溢出的，反之也一样。

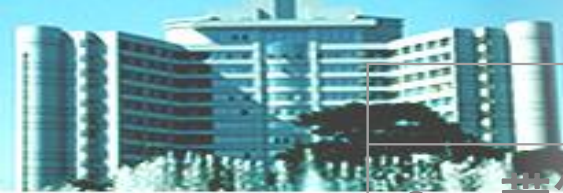
## 判定条件:

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。

一旦发生溢出，结果就不正确了。[表3-2](#)以8位数为例说明了这种情况。





	二进制数加法	解释为无符号数	解释为带符号数
a. 带符号数 和无符号 数不溢出	$\begin{array}{r} 00000100 \\ + 00001011 \\ \hline 00001111 \end{array}$	$\begin{array}{r} 4 \\ + 11 \\ \hline 15 \\ CF=0 \end{array}$	$\begin{array}{r} +4 \\ + (+11) \\ \hline +15 \\ OF=0 \end{array}$
b. 无符号 数溢出	$\begin{array}{r} 00000111 \\ + 11111011 \\ \hline 1\ 00000010 \end{array}$ CF ← 1	$\begin{array}{r} 7 \\ + 251 \\ \hline 258 \text{ (错)} \\ CF=1 \end{array}$	$\begin{array}{r} +7 \\ + (-5) \\ \hline +2 \\ OF=0 \end{array}$
c. 带符号 数溢出	$\begin{array}{r} 00001001 \\ + 01111100 \\ \hline 10000101 \end{array}$	$\begin{array}{r} 9 \\ + 124 \\ \hline 133 \\ CF=0 \end{array}$	$\begin{array}{r} +9 \\ + (+124) \\ \hline -123 \text{ (错)} \\ OF=1 \end{array}$
d. 带符号数 和无符号 数都溢出	$\begin{array}{r} 10000111 \\ + 11110101 \\ \hline 1\ 01111100 \end{array}$ CF ← 1	$\begin{array}{r} 135 \\ + 245 \\ \hline 124 \text{ (错)} \\ CF=1 \end{array}$	$\begin{array}{r} -121 \\ + (-11) \\ \hline +124 \text{ (错)} \\ OF=1 \end{array}$

表3-2 对二进制数加法结果的解释





对于情况b:

最高位向前有进位, 该进位记录在CF中, 7加251应该等于258, 但在有效的8位结果中, 只看到2, 这是因为丢掉了 $2^8=256$ , 所以对于无符号数来讲, 通过判断 $CF=1$ , 便知该结果是错的。

[返回](#)





对于情况c:

因为两同号数相加，和的符号却与加数符号相反. 所以对于带符号数来讲，该结果是错的，发生这种情况后系统会置OF=1，所以可通过OF来判断。

[返回](#)





## (2). 带进位加法指令 **ADC**

格式: **ADC** DST, SRC

功能:  $(DST) + (SRC) + \text{CF} \rightarrow DST$

说明: 对操作数的限定同**MOV**指令, 该指令  
适用于多字节或多字的加法运

算。

标志: 影响**OF**、**SF**、**ZF**、**AF**、**PF**、**CF**标志

例. **ADC** AX, 35

;  $(AX) = (AX) + 35 + CF$







### (3). 加1指令 **INC**

格式: **INC DST**

功能:  $(DST) + 1 \rightarrow DST$

标志: 除**不影响CF标志**外, 影响其它五个算术运算特征标志。

例. **INC BX**

例. 实现+2操作: **ADD AX, 2**

**INC AX**

**INC AX**





#### (4). 互换并加法指令 **XADD** (486以上)

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

。

标志: 影响OF、SF、ZF、AF、PF、CF。



### 3、二进制减法指令

#### (1). 减法指令 SUB

格式: SUB DST, SRC

功能:  $(DST) - (SRC) \rightarrow DST$

标志: 影响OF、SF、ZF、AF、PF、CF标志。

例. SUB AX, 35

SUB WORD PTR[BX], 56

减法指令执行后若使 $CF=1$ ，则对无符号数而言发生了溢出。若使 $OF=1$ ，则对带符号数而言发生了溢出。





## (2). 带借位减法指令 **SBB**

格式: SBB DST, SRC

功能:  $(DST) - (SRC) - CF \rightarrow DST$

说明: 除了操作为减外, 其它要求同ADC, 该指令适用于多字节或多字的减法运算。

标志: 影响OF、SF、ZF、AF、PF、CF标志

例. SBB AX, 35

;  $(AX) = (AX) - 35 - CF$





### (3). 减1指令 **DEC**

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: 除不影响CF标志外, 影响其它五个算术运算特征标志。

例. DEC BX







#### (4). 比较指令 **CMP**

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, 其它要求同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

例. **CMP** AX, [BX]





## (5). 求补指令 **NEG**

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  $0 - (DST) \rightarrow DST$

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。其中

**对CF和OF的影响如下:**

- a. 对操作数所能表示的最小负数 (例若操作数是8位则为-128) 求补, 原操作数不变, 但OF被置1
- b. 当操作数为0时, 清0 CF。
- c. 对非0操作数求补后, 置1 CF。





例. 实现  $0 - (AL)$  的运算

NEG AL

例. EAX 中存放一负数, 求该数的绝对值

NEG EAX





例. 试编写  
两个三字节  
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  
。

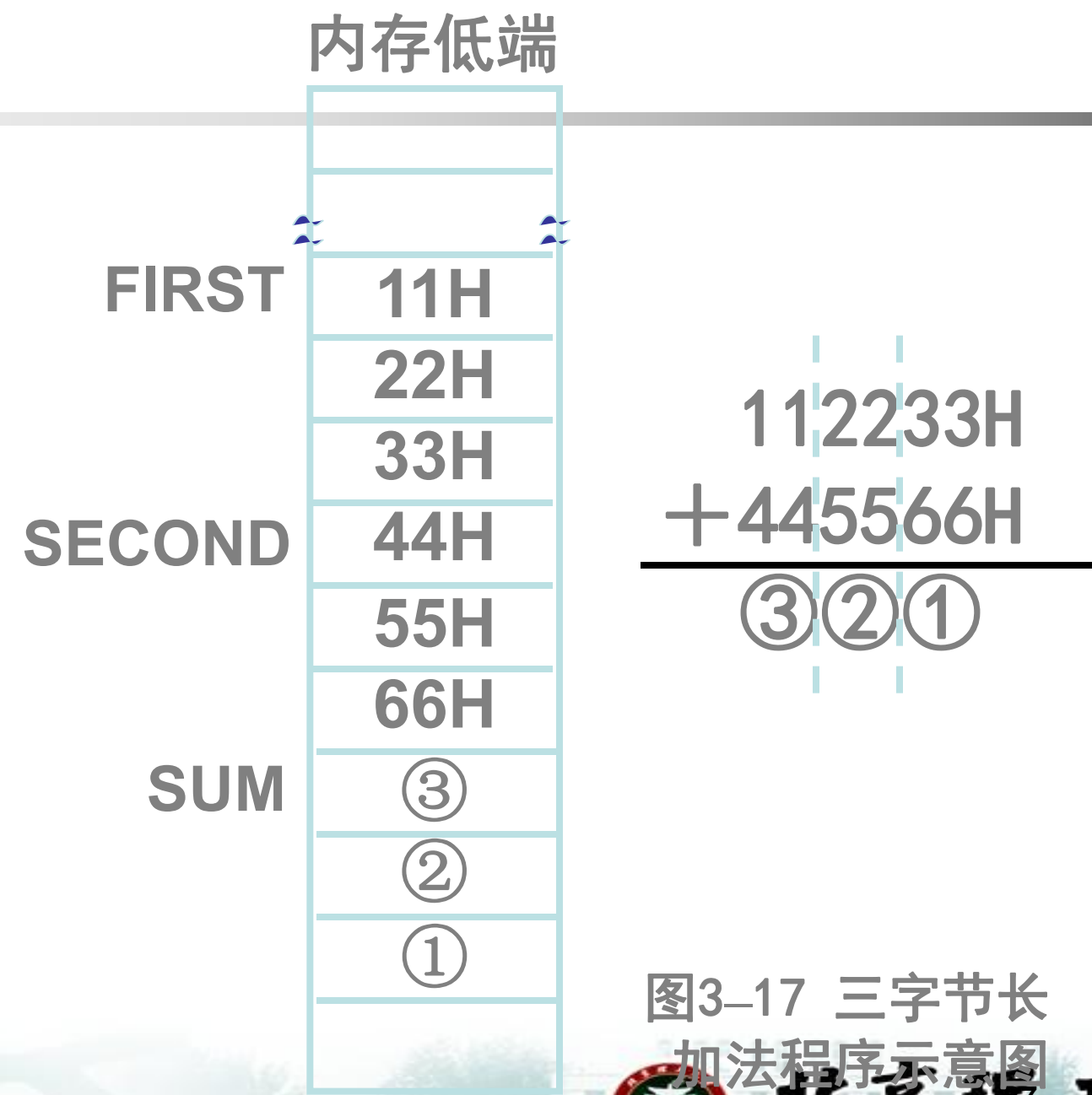


图3-17 三字节长  
加法程序示意图





分析：为便于理解和方便讨论，设(FIRST) = 112233H, (SECOND) = 445566H, 存放顺序如上页图所示。

算法类似于手工计算, 从最低字节数据开始加起, 计算高字节时要考虑低字节的进位问题。为了简化讨论, 假设和不会超过三字节, 即最高位不会有进位产生。

为了最大限度地用到我们所学的指令和寻址方式, 本程序不使用循环结构, 不考虑优化问题。因为是三字节数据, 所以不能用字或双字加法指令实现。







经过分析，编程如下：

```
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）
```





讨论:

- ① 两个ADC指令能否换为ADD?
- ② DEC DI和SUB DI, 1指令的功能从表面上看是等价的, 是否可以互换?





③ 多字节或多字加减时，CF始终有意义(前边的反映进 / 借位，最后一次反映溢出情况)，而OF只有最后一次的才有意义。

④ 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。

从该例可以看出，选取合适的算法、恰当的指令、灵活的寻址方式对汇编语言程序设计人员十分重要，它可以达到事半功倍的效果。





**C5FF0000**  
**+9E000000**  
**163FF0000**  
**ZF=0 SF=0 CF=1 OF=1 AF=0 PF=1**  
**0C5FF0000**  
**-09E000000**  
**027FF0000**  
**ZF=0 SF=0 CF=0 OF=0 AF=0 PF=1**



## 4、二进制乘法指令

### (1). 无符号乘法指令 **MUL**

格式: **MUL**  $\text{SRC}_{\text{reg} / \text{m}}$

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, 另一个乘数必须事前放在累加器中。可以实现8位、16位、32位无符号数乘。







具体操作为：

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：影响CF、OF、SF、ZF、AF、PF，而只有CF、OF有意义，其它标志不确定。

对CF和OF的影响是：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。





例.

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 AL, 80H

SUB AH, AH ; 清0 AH

MUL BX ;  $(AX) \times (BX)$ , 结果在DX:AX中





## (2). 带符号乘法指令 **IMUL**

功能：实现两个带符号二进制数乘。

格式1: **IMUL SRC<sub>reg</sub> / m**

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，其它与MUL指令相同。所有的80X86 CPU都支持这种格式。





具体操作为：

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：

影响CF、OF、SF、ZF、AF、PF，而只有CF、OF有意义，其它标志不确定。对CF和OF的影响是：若乘积的高半部分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。





格式2: **IMUL REG, SRC<sub>reg / m</sub>** (286以上)

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:

$$(\text{REG})_{16} \times (\text{SRC})_{16} \rightarrow \text{REG}_{16}$$

$$(\text{REG})_{32} \times (\text{SRC})_{32} \rightarrow \text{REG}_{32}$$

格式3: **IMUL REG, imm<sub>8</sub>** (286以上)

格式4: **IMUL REG, SRC<sub>reg / m</sub>, imm<sub>8</sub>** (286以上)





## 5、二进制除法指令

### (1). 无符号除法指令 **DIV**

格式: **DIV**  $\text{SRC}_{\text{reg} / \text{m}}$

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, 注意它不能是立即数。被除数必须事前放在隐含的寄存器中。可以实现8位、16位、32位无符号数除。





具体操作为：

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。



例. 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

例. 实现 $1000 \div 512$ 的无符号数除法。

```
MOV  AX, 1000
```

```
SUB  DX, DX      ; 清0 DX
```

```
MOV  BX, 512
```

```
DIV  BX
```

; (DX:AX)  $\div$  (BX)、商在AX中、余数在DX中





## (2). 带符号除法指令 **IDIV**

格式: **IDIV**  $\text{SRC}_{\text{reg} / \text{m}}$

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、余数符号与被除数相同外, 其它与 **DIV** 指令相同。

具体操作同无符号数除法。



例. 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

例. 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







注意: 若除数为0或商超出操作数所表示的范围 (例如字节型除法的商超出8位) 会产生除法错中断, 此时系统直接进入0号中断处理程序, 为避免出现这种情况, 必要时在程序中应事先对操作数进行判断。



### 3.2.3 位运算指令

位运算指令提供了对二进制位的控制。该系统提供的逻辑指令包括：

逻辑运算指令

位测试指令

位扫描指令

基本移位指令

循环移位指令

双精度移位指令





# 1、逻辑运算指令

逻辑运算指令见下表。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。





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) $\wedge$ (SRC) ⑦ DST	清0 CF和OF, 影响SF、ZF及PF, AF不定
测 试	TEST OPR1, OPR2	OPR1 $\wedge$ OPR2	同AND指令
逻辑或	OR DST, SRC	(DST) $\vee$ (SRC) ⑦ DST	同AND指令
逻辑异或	XOR DST, SRC	(DST) $\vee$ (SRC) ⑦ DST	同AND指令

表3-3 逻辑运算指令





**NOT**指令可用于把操作数的每一位均变反。

**AND**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）。

**TEST**指令可用于只测试其值而不改变操作数。

**OR**指令用于把某位置1（与1相或）、某位保持不变（与0相或）。

**XOR**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）。







## XOR指令的用法:

$$0 \oplus 0 = 0 \quad 1 \oplus 1 = 0$$

$$1 \oplus 0 = 1 \quad 0 \oplus 1 = 1$$

$$X \oplus \text{key} = X' \quad X' \oplus \text{key} = X$$

1. 逻辑值相同，运算结果为0；

将某个寄存器清零操作；

2. 与1异或，结果相反；

将输入字符进行大小写转换；





## 加密解密

设数据的原始形式为A，密码为B。加密的操作为：

$$C = A \oplus B$$

加密后的数据为C。在不知道密码的情况下，从C不能推断出原始数据A，从而达到保密的目的。

而知道密码B后，从C可以求出A。如何解密？

不需要借助中间数，使用“异或”运算就能交换两个变量的值？





## 2、位测试指令

从386开始增加了位测试指令, 它们包括**BT**、**BTS**、**BTR**和**BTC**。这些指令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	所选位值送CF, 其它标志不定
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

表3-4 位测试指令





### 3、基本移位指令

这类指令实现对操作数移位, 包括SHL、SAL、SHR和SAR指令。表3-5给出了这组指令。





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, 移位使符号位变化置1 0F*, 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: 当CNT > 1时, 0F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。

对CNT的限定是:

当CNT = 1时, 直接写在指令中;

当CNT > 1时, 由CL寄存器给出;

当CNT > 1时, 由指令中的8位立即数给出;

功能图中的符号表示:

CF 

数据流向 

操作数 

适用于8086 / 8088

适用于80X86系列的所有型号

适用于80286以上





可以用逻辑移位指令实现无符号数乘法运算, 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 $1/2^n$

可以用算术移位指令实现带符号数乘法运算, 只要移位操作不改变符号位:

**SAL** **DST, n** 执行后是原数的 $2^n$ 倍

**SAR** **DST, n** 执行后是原数的 $1/2^n$

(只要移出位不含1)





例. 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

SHL AX, 1 ;求得2X

MOV BX, AX ;暂存2X

MOV CL, 2 ;设置移位次数

SHL AX, CL ;求得8X

ADD AX, BX ; $10X = 8X + 2X$

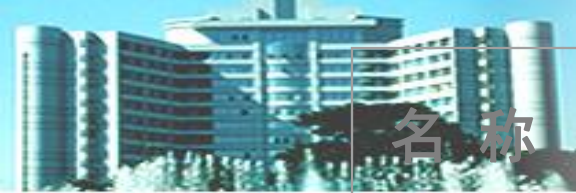




## 4、循环移位指令

这类指令实现循环移位操作，  
包括**ROL**、**ROR**、**RCL**、**RCR**指令。表3-6给出了基本移位指令。





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置1 OF*, 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。			

表3-6 基本移位指令







例. 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。





## 3.3 程序控制指令

本节提供的指令可以改变程序执行的顺序，控制程序的流向。它们均不影响标志位。

转移指令；

循环指令；

子程序调用及返回指令；

中断调用及返回指令；





# 1、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8位、16位位移量之和，该位移量是一个相对于指令指针的带符号数。





即要转向的有效地址为：

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

EA就是要转向的本代码段内指令地址的偏移量。它是通过把IP的当前值加上指令中给出的位移量disp得到的，从而使IP指向下一条要执行的指令，实现段内转移。





若位移量是8位的，则称为短转移，可以实现在距离下条指令的 $+127 \sim -128$ 字节范围之内转移

段内无条件短转移指令格式：

**JMP    SHORT    LAB**

字节0      字节1

操作码	disp
-----	------

段内短转移方式







若位移量是16位的，则称为近转移，可以实现在距  
离下条指令的 $+32767 \sim -32768$  (以下简称 $\pm 32K$ ) 字节范  
围之内转移。

段内无条件近转移指令格式：JMP LAB

或

JMP NEAR PTR LAB

该格式的机器指令

其位移量占两个字节

字节0 字节1 字节2



段内直接寻址方式



## 2、段内间接寻址方式

转向的有效地址在一个寄存器或内存单元中，该寄存器号或内存地址按上节介绍的与操作数有关的寻址方式（立即寻址方式除外）获得。所得到的有效地址送给IP，于是实现转移。指令格式及举例见下表。





表3-1 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
JMP 内存单元	JMP WORD PTR VAR JMP WORD PTR[BX]	16位转向地址在VAR字型内存变量 中 16位转向地址在BX所指向的内存 变量中





具体说明。

设：  $(DS) = 2000H$ ,  $(BX) = 0300H$   
 $(IP) = 0100H$ ,  $(20300H) = 000BH$   
 $(20301H) = 0005H$

则： `JMP BX`  
; 执行后  $(IP) = (BX) = 0300H$

动画演示





## 例2. JMP WORD PTR [BX]

说明：式中**WORD PTR [BX]**表示BX指向一个字型内存单元。

1. 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：

$$10H \times (DS) + (BX) = 20306H$$

2. 再从该单元中得到转移地址：

$$EA = (20306H) = 050BH$$

3. 于是， $(IP) = EA = 050BH$ ，下一次便执行CS:50BH处的指令，实现了段内间接转移。

$$10H \times (CS) + (050bh) = 20306H$$

动画演示







### 3、段间直接寻址方式

指令中直接给出转向的4字节的偏移量和段基址, 只需把它们分别送给IP和CS后即可实现段间转移。

指令格式:

JMP    FAR    PTR    LAB





下图给出的是段间直接转移指令机器码示意图。执行时把偏移量送给IP，段基址送给CS，即可实现段间直接转移。



图3-12 段间直接寻址方式

动画演示



北京理工大学  
BEIJING INSTITUTE OF TECHNOLOGY



## 4、段间间接寻址方式

用一个双字内存变量中的低16位取代IP值，高16位取代CS值，从而实现段间转移。该双字变量的地址可以由除立即寻址和寄存器寻址方式以外的其它与数据有关的寻址方式获得。





具体数据说明：

设： (DS) = 2000H, (BX) = 0300H

(IP) = 0100H, (20300H) = 0

(20301H) = 05H

(20302H) = 10H

(20303H) = 60H

则： JMP DWORD PTR [BX]

**说明：**式中DWORD PTR [BX]表示BX指向一个双字变量。





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：

$$10\text{H} \times (\text{DS}) + (\text{BX}) = 20300\text{H}$$

再把该单元中的低字送给IP，高字送给CS，即0500H

→ IP，6010H → CS，下一次便执行6010:0500H处的指令，实现了段间间接转移。

动画演示



**北京理工大学**  
BEIJING INSTITUTE OF TECHNOLOGY





上边介绍的与转移地址有关的寻址方式完全适用于386以上的实模式环境。保护模式下的虚拟8086方式转移地址的形成与此类似，只是32位的偏移量送给EIP，但高16位清0，这是因为段长不能超过64K的缘故。





对于32位的保护模式，其段内转移的目标地址的形成与以上所介绍的16位机相比较没有太大变化，只是偏移量为32位、并把该偏移量送给EIP指令指针寄存器而已。

而段间转移的目标地址采用48位全指针形式，即32位的偏移量和16位的段选择子。段间转移的目标地址的形成比较复杂，需要涉及到任务门、调用门等知识。



## 3.3.2、转移指令

这类指令包括无条件转移指令和条件转移指令。





## 1. 无条件转移指令 **JMP**

格式：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），使用与转移地址有关的寻址方式可以形成目标地址。





# 无条件转移指令 **JMP**

## 段内转移 (IP)

段内直接短转移      **JMP   SHORT   LABEL**

段内直接转移   **JMP   NEAR   PTR   LABEL**

段内间接转移   **JMP   REG / M**

## 段间转移 (CS:IP)

段间直接转移   **JMP   FAR   PTR   LABEL**

段间间接转移   **JMP   DWORD   PTR   M**







## ① 段内直接短转移

格式: **JMP SHORT LABEL**

例.

**JMP SHORT B1** ;无条件转移到

B1标号处

A1: **ADD AX, BX**

B1: **...**





## ② 段内直接转移

格式: **JMP LABEL**

或: **JMP NEAR PTR**

**LABEL**

例.

**JMP B2**

**A2: ADD AX, CX**

...

**B2: SUB AX, CX**





### ③ 段内间接转移

格式: **JMP** REG / M

例.

```
LEA    BX, B2
```

```
JMP    BX
```

```
A2:    ADD    AX, CX
```

```
...
```

```
B2:    SUB    AX, CX
```

动画演示



北京理工大学  
BEIJING INSTITUTE OF TECHNOLOGY



## ④ 段间直接转移

例.

B3

CODE1

SEGMENT

...

JMP FAR PTR

...

CODE1

ENDS

CODE2

SEGMENT

...

B3:

SUB AX, BX

...

CODE2

ENDS





## ⑤ 段间间接转移

例.        V     DD   B3

C1    SEGMENT

...

JMP    DWORD PTR V

...

C1    ENDS

C2    SEGMENT

...

B3:   SUB    AX, CX

...

C2    ENDS

动画演示







## 2. 条件转移指令

执行这类指令时通过检测由前边指令已设置的标志位确定是否转移，所以它们通常是跟在影响标志的指令之后。这类指令本身并不影响标志。

条件转移指令的通用汇编格式：

$J_{cc}$  LABEL





## $J_{cc}$ 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。

说明：其中cc为条件，LABEL是要转向的标号。在8086~80286中，该地址应在与当前IP值的-128~+127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，其位移量占用一个字节。





## (1) 检测单个标志位实现转移的 条件转移指令

这组指令根据一个标志位的设置情况决定是否转移。





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：对实现同一功能但指令助记符有两种形式时，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JZ助记符，当比较某数为0转移时常使用JE指令。下同。		

表3-8 检测单个条件标志位转移指令





例. 比较AX和BX寄存器中的内容，若相等执行ACT1，不等执行ACT2。

CMP AX, BX

JE ACT1

ACT2: .

.

.

ACT1: ...



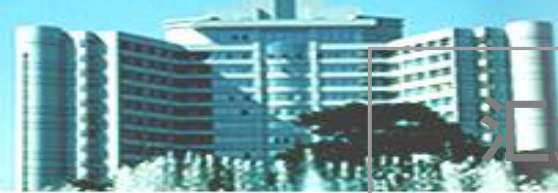




## (2) 根据两个带符号数比较结果实现转移的条件转移指令

利用上表中提供的指令，可以实现两个带符号数的比较转移。





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=0F
JNG/JLE LABEL	不大于/小于等于转移	ZF=1 or SF≠0F
JL/JNGE LABEL	小于/不大于等于转移	SF≠0F
JNL/JGE LABEL	不小于/大于等于转移	SF=0F
注1. G=Greater, L=Less, E=Equal, N=Not 注2. 显然, 表3-7中的指令JZ/JE LABEL和 JNZ/JNE LABEL同样可以用于两个带符号数的比较转移。		

表3-9 根据两个带符号数比较结果实现转移的条件转移指令





### (3) 根据两个无符号数比较结果实现转移的 条件转移指令

利用上表中提供的指令，可以实现两个无符号数的比较转移。





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
<p>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。</p> <p>注2. 显然, 表3-7中的指令JZ/JE LABEL和 JNZ/JNE LABEL同样可以用于两个无符号数的比较转移。</p>		

表3-10 根据两个无符号数比较结果实现转移的条件转移指令





### 3. 测试CX / ECX值为0转移指令

这类指令不同于以上介绍的条件转移指令, 因为它们测试的是CX或ECX寄存器的内容是否为0, 而不是测试标志位。这类指令只能使用与转移地址有关的寻址方式的段内短转移格式, 即位移量只能是8位的。







格式: **JCXZ LABEL**; 适用于16位

操作数

**JECXZ LABEL**; 适用于32位操作

数

功能: 测试CX (或ECX) 寄存器的内容, 当CX (或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。





例. 设  $M = (EDX:EAX)$ ,  $N = (EBX:ECX)$ ,  
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:  ...
```





### 3.3.3、循环指令

循环指令可以控制程序的循环。它们的特点是：

1. 用CX或ECX（操作数长度为32位时）作为循环次数计数器。
2. 不影响标志。





## 1. 循环指令 LOOP

格式: **LOOP LABEL**

功能:  $(CX) - 1 \rightarrow CX$ , 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: 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 (或ECX)。

## 2. 相等循环指令 LOOPE/LOOPZ

## 3. 不等循环指令 LOOPNE/LOOPNZ







例. 用累加的方法实现 $M \times N$ ，并把结果保存到RESULT单元。

```
MOV    AX, 0           ;清0累加器
MOV    BX, M
CMP    BX, 0
JZ     TERM            ;被乘数为0转
MOV    CX, N
JCXZ   TERM            ;乘数为0转
L1:    ADD    AX, BX
      LOOP   L1
TERM:  MOV    RESULT, AX ;保存结果
```

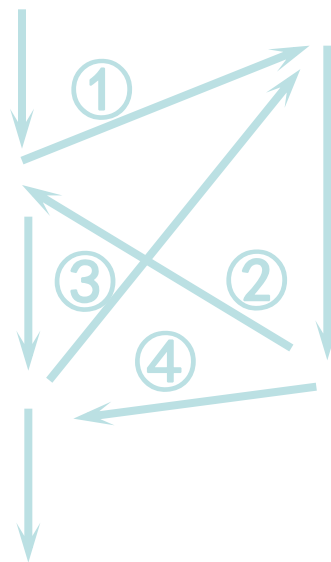




## 3.3.4、子程序调用与返回指令

子程序的作用；  
子程序的执行流程；

主程序 子程序



主子程序关系示意图





## 1. 子程序调用指令 **CALL**

格式：CALL DST

功能：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。





说明：其中DST为子程序名或子程序入口地址，其目标地址的形成与JMP指令有异同。

与JMP指令相同点：

段内直接 / 间接调用、

段间直接 / 间接调用、

只是不能使用段内直接寻址方式的SHORT格式。

指令的执行结果是无条件转到标号处

与JMP指令不同点：

CALL转移后要返回，所以要保存返回地址；

JMP转移后不再返回，所以不必保存返回地址。





## (1) 段内调用

这类调用指令实现同一段内的子程序调用, 它只改变IP值, 不改变CS值。

执行操作:

把返回地址 (CALL之后的那条指令地址的偏移量部分(当前IP值)) 压入堆栈。

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值。

把控制无条件转向子程序, 即执行CS:IP处的指令。







## ① 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压入堆栈, 再使 $IP = (IP) + disp16$ , 最后把控制权交给子程序。

## ② 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压入堆栈, 再把指令指定的16位通用寄存器或内存单元的内容送给IP, 最后把控制权交给子程序。





例. 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 WORD PTR BX**

;子程序入口地址的偏移量在BX中

**CALL WORD PTR [BX]**

;子程序入口地址的偏移量在数据段的BX所指  
;向的内存单元中

**CALL WORD PTR [BX][SI]**

;子程序入口地址的偏移量在数据段的BX+SI  
;所指向的内存单元中





## (2) 段间调用

这类调用指令可以实现段间调用（FAR型调用），执行时即要改变IP值，也要改变CS值。





## ① 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值和当前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。

## ② 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IP值和当前CS值)压入堆栈, 再把M的低字送给IP, 高字送给CS, 最后把控制权交给子程序。





例. 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**

;从VAR变量中得到子程序B的入口地址实现调用。

变量VAR的地址也可以通过寄存器间接寻址方式、基址变址寻址方式等存储器操作数寻址方式得到。

例. **CALL DWORD PTR 8[BX][DI]**







## 2. 子程序返回指令 **RET**

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。以下介绍8086 / 8088的子程序调用指令。





## (1) 返回指令 **RET**

格式: **RET**

功能: 按照CALL指令入栈的逆序, 从栈顶弹出返回地址 (弹出一个字到IP, 若子程序是FAR型还需再弹出一个字到CS), 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**。



## (2) 带立即数的返回指令

格式: **RET**  $\text{imm}_{16}$

功能: 按照CALL指令入栈的逆序, 从栈顶弹出返回地址 (弹出一个字到IP, 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  $\text{SP} = (\text{SP}) + \text{imm}_{16}$ 。

注: 其中  $\text{imm}_{16}$  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  $\text{imm}_{16} = 2n$ 。

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。





### 3.3.5、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种事件，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**。





## 1. 中断向量

中断向量就是中断处理子程序的入口地址。在PC机中规定中断处理子程序为FAR型，所以每个中断向量占用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。

## 2. 中断类型号

IBM PC机共支持256种中断，相应编号为0~255，把这些编号称为中断类型号。







### 3. 中断向量表

256种中断有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，形成中断向量表。

表长为 $4 \times 256 = 1024$ 字节，该表从内存的0000:0000地址开始存放，从内存最低端开始存放。





地址	内容	
00000	0# 偏移量低8位	0# 中断向量
00001	0# 偏移量高8位	
00002	0# 段基址低8位	
00003	0# 段基址高8位	
00004	1# 偏移量低8位	n# 中断向量
≈	≈	
4n	n# 偏移量低8位	
	n# 偏移量高8位	
4n+2	n# 段基址低8位	n# 中断向量
	n# 段基址高8位	
≈	≈	
003FF		

图3-14 中断向量表



## 4. 中断调用指令 **INT**

在8086 / 8088中, 中断分为内中断(或称软中断)和外中断(或称硬中断), 本节只介绍内中断的中断调用指令。

格式: **INT** **n** ; 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  $(4n) \rightarrow IP$ ,  $(4n+2) \rightarrow CS$ , 转去执行中断处理子程序。





例. INT 21H

21H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, 转去执行中断处理子程序。

## 动画演示

例. INT 3 ;断点中断





## 5. 中断返回指令 **IRET**

格式：IRET

功能：从栈顶弹出三个字分别送入IP、CS、FLAGS寄存器，把控制返回到原断点继续执行。







## 3.4 处理机控制指令

这组指令可以控制处理机状态以及对某些标志位进行操作。





## 3.4.1、标志操作指令

这组指令可以直接对CF、DF和IF标志位进行操作，它们只影响本指令所涉及的标志。





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

表3-10 标志操作指令





## 3.4.2、其它处理机控制指令

这组指令可以控制处理机状态，它们均不影响标志，见下表。



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, 其机器码占用一字节
停机	HLT (Ha l t)	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 (Wa i t)	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

表3-11 其它处理机控制指令







## 3.5 块操作指令

利用串操作指令可以**直接**  
**处理两个存储器操作数**，方便地处理字  
符串或数据块。





# 1、块指令的特点

## 1. 指令格式

串指令可以显式地带有操作数，也可以使用隐含格式。例如串传送指令MOVSB，可以有以下几种格式：

显式：MOVSB DST, SRC

隐式：MOVSB ; 字节传送

MOVSW ; 字传送

MOVSD ; 双字传送

经常使用隐含格式。操作数时应先建立地址指针。





## 2. 操作数

串指令可以处理寄存器操作数和存储器操作数。

若为寄存器操作数则只能放在累加器中；

对于存储器操作数应先建立地址指针：若为源操作数，则必须把源串首地址放入SI寄存器，缺省情况寻址DS所指向的段，允许使用段超越前缀；若为目标操作数，则必须把目标串首地址放入ES:DI寄存器，不允许使用段超越前缀。





### 3. 地址指针的修改

串指令执行后系统自动修改地址指针 **SI** (ESI)、**DI** (EDI)。若为字节型操作其修改量为1，若为字型操作其修改量为2，若为双字型操作其修改量为4。

### 4. 方向标志

方向标志 **DF** 决定地址指针的增减方向。

$DF=0$ ，则地址指针增量；

$DF=1$ ，则地址指针减量。

可以用 **CLD** 和 **STD** 指令复位和置位 **DF**

。





## 5. 重复前缀

串指令前可以加重复前缀**REPE / REPZ、 REP或REPNE / REPNZ**, 使后跟的串指令重复执行。重复次数应事先初始化在计数器**CX**中。







## (1) 重复前缀 REP

REP可用在传送类串操作指令之前，其执行的操作为：

若 $(CX) = 0$ ，则结束重复，顺序执行下一条指令。

若 $(CX) \neq 0$ ，则执行后跟的串操作指令，然后修改计数器 $(CX) - 1 \rightarrow CX$ ，继续重复上述操作。

若地址长度为32位的，则使用ECX

。





## (2) 相等重复前缀 REPE / REPZ

REPE / REPZ可以用在比较类串操作指令之前，当比较相等且重复次数未到时重复执行后跟的串指令。其执行的操作为：

若 $(CX)=0$ （计数到）或 $ZF=0$ （不相等），则结束重复，顺序执行下一条指令。

否则执行后跟的串操作指令，修改计数器 $(CX)-1 \rightarrow CX$ 。继续重复上述操作。

若地址长度为32位的，则使用ECX

。

## (3) 不等重复前缀 REPNE / REPNZ





## 二、串指令

### 1. 串传送指令 **MOVS**

显式格式: **MOVS**  $DST_m, SRC_m$

隐含格式: **MOVSB**      **MOVSW**

**MOVSD**

功能: 源  $\rightarrow$  目标, 即

$([SI]) \rightarrow ES:[DI]$ , 且自动修改  $SI$ 、 $DI$  指针。

标志: 不影响标志位。





说明：若 $DF=0$ ，则MOVSB指令使SI、DI各加1；MOVSW指令使SI、DI各加2；MOVSD使SI、DI各加4。若 $DF=1$ ，则

MOVSB使SI、DI各减1；

MOVSW使SI、DI各减2；

MOVSD使SI、DI各减4。

若地址长度是32位的，则SI、DI相应为ESI、EDI。



## 2. 取串指令 LODS

显式格式:  $\text{LODS SRC}_m$

隐含格式:  $\text{LODSB LODSW}$

$\text{LODSD}$

功能: 源  $\rightarrow$  累加器, 即  $([SI])$

$\rightarrow$  AL (或 AX、EAX), 且自动修改 SI 指针。

说明: 若  $DF=0$ , 则  $\text{LODSB}$  (或  $\text{LODSW}$ 、 $\text{LODSD}$ ) 使 SI 加 1 (或 2、4); 若  $DF=1$ , 则  $\text{LODSB}$  (或  $\text{LODSW}$ 、 $\text{LODSD}$ ) 使 SI 减 1 (或 2、4)。若地址长度是 32 位的, 则 SI 相应为 ESI。

标志: 不影响标志位。





### 3. 存串指令 STOS

显式格式: STOS DST<sub>m</sub>

隐含格式: STOSB STOSW

STOSD

功能: 累加器 → 目标, 即 (AL (或 AX  
、EAX))

→ ES: [DI], 且自动修改 DI 指针。

说明: 若 DF=0, 则 STOSB (或 STOSW  
、STOSD) 使 DI 加 1 (或 2、4); 若 DF=1, 则 STOSB  
(或 STOSW、STOSD) 使 DI 减 1 (或 2、4)。若地址长  
度是 32 位的, 则 DI 相应为 EDI。

标志: 不影响标志位。



北京理工大学  
BEIJING INSTITUTE OF TECHNOLOGY



#### 4. 串输入指令 **INS**

显式格式:  $\text{INS } \text{DST}_m, \text{DX}$

隐含格式:  $\text{INSB} \quad \text{INSW}$

$\text{INSD}$

#### 5. 串输出指令 **OUTS**

显式格式:  $\text{OUTS } \text{DX}, \text{SRC}_m$

#### 6. 串比较指令 **CMPS**

显式格式:  $\text{CMPS } \text{DST}_m, \text{SRC}_m$

#### 7. 串扫描指令 **SCAS**

显式格式:  $\text{SCAS } \text{DST}_m$





例：把自AREA1开始的100个字节数据传送到AREA2开始的区域中。

MOV AX, SEG AREA1

MOV DS, AX

MOV AX, SEG AREA2

MOV ES, AX

LEA SI, AREA1

LEA DI, AREA2

MOV CX, 100

CLD

REP **MOVSW**

动画演示

;100个字节数据传送完毕后执行下一

条指令





例：把自NUM1开始的未压缩BCD码字符串转换成ASCII码，并放到NUM2中，字符串长度为8字节。设DS、ES已按要求设置。

```
LEA    SI, NUM1
LEA    DI, NUM2
MOV    CX, 8
CLD

LOP:   LODSB

      OR     AL, 30H

      STOSB

      LOOP  LOP
```





# 感谢关注聆听！



张华平

Email: [kevinzhang@bit.edu.cn](mailto:kevinzhang@bit.edu.cn)

微博: @ICTCLAS张华平博士

实验室官网:

<http://www.nlpir.org>



大数据千人会

