



北京師範大學
BEIJING NORMAL UNIVERSITY

人工智能學院

第9章

转移指令的原理



第9章 转移指令的原理

- 9.1 操作符offset
- 9.2 jmp指令
- 9.3 依据位移进行转移的jmp指令
- 9.4 转移的目的地址在指令中的jmp指令
- 9.5 转移地址在寄存器中的jmp指令
- 9.6 转移地址在内存中的jmp指令
- 9.7 jcxz指令
- 9.8 loop指令
- 9.9 根据位移进行转移的意义
- 9.10 编译器对转移位移超界的检测



引言

- 8086CPU的转移指令分为以下几类：
 - 无条件转移指令（如：`jmp`）
 - 条件转移指令
 - 循环指令（如：`loop`）
 - 过程
 - 中断



引言

- 8086CPU的转移行为分为以下几类：
 - 只修改IP
 - 相对或者绝对IP地址
 - 同时修改CS和IP
 - 绝对地址



引言

- 8086CPU的转移对IP的修改范围，分为：
 - 短转移，IP修改范围为 $-128 \sim 127$
 - 近转移，IP修改范围为 $-32768 \sim 32767$



9.1 操作符offset

- 操作符**offset**在汇编语言中是由编译器处理的符号，其功能是取得标号的偏移地址。

比如下面的程序：

```
assume cs:codesg
```

```
codeseg segment
```

```
start:mov ax,offset start ; 相当于 mov ax,0
```

```
    s:mov ax,offset s ; 相当于 mov ax,3
```

```
codesg ends
```

```
end start
```



9.1 操作符offset

■ 问题9.1

有如下程序段，添写2条指令，使该程序在运行中将s处的一条指令复制到s0处。

```
assume cs:codesg
```

```
codesg segment
```

```
s: mov ax,bx          ; (mov ax,bx 的机器码占两个字节)
```

```
    mov si,offset s
```

```
    mov di,offset s0
```

```
    _____
```

```
    _____
```

```
s0: nop                ; (nop的机器码占一个字节)
```

```
    nop
```

```
codesg ends
```

```
ends
```

思考后看分析。



9.1 操作符offset

■ 问题9.1分析

有如下程序段，添写2条指令，使该程序在运行中将s处的一条指令复制到s0处。

```
assume cs:codesg
codesg segment
s: mov ax,bx
   mov si,offset s
   mov di,offset s0
```

```
_____
_____
s0: nop
    nop
codesg ends
ends
```

(1) s和s0处的指令所在的内存单元的地址是：`cs:offset s` 和 `cs:offset s0`。

(2) 将s处的指令复制到s0处，就是将`cs:offset s` 处的数据复制到 `cs:offset s0`处；



9.1 操作符offset

■ 问题9.1分析

有如下程序段，添写2条指令，使该程序在运行中将s处的一条指令复制到s0处。

```
assume cs:codesg
codesg segment
s: mov ax,bx
    mov si,offset s
    mov di,offset s0
    _____
    _____
s0: nop
    nop
codesg ends
ends
```

(3) 段地址已知在cs中，偏移地址offset s和offset s0已经送入si和di中；

(4) 要复制的数据有多长？

mov ax,bx指令长度为两个字节，即1个字。



9.1 操作符offset

■ 问题9.1分析

有如下程序段，添写2条指令，使该程序在运行中将s处的一条指令复制到s0处。

```
assume cs:codesg
```

```
codesg segment
```

```
s: mov ax,bx          ; (mov ax,bx 的机器码占两个字节)
```

```
    mov si,offset s
```

```
    mov di,offset s0
```

```
    mov ax,cs:[si]
```

```
    mov cs:[di],ax
```

```
s0: nop              ; (nop的机器码占一个字节)
```

```
    nop
```

```
codesg ends
```

```
ends
```



9.1 操作符offset

■ 问题9.1分析

有如下程序段，添写2条指令，使该程序在运行中将s处的一条指令复制到s0处。

```
assume cs:codesg
codesg segment
    s: mov ax,bx
        mov si,offset s
        mov di,offset s0
        mov ax,cs:[si]
        mov cs:[di],ax
s0: nop
    nop
codesg ends
ends
```

想一想：

入果代码段执行前

AX: 0000H

BX: 0001H

代码段执行完后

AX: ?

BX: ?



9.2 jmp指令

- jmp为无条件转移，2种用法
 - 只修改IP
 - 同时修改CS和IP;
- jmp指令要给出两种信息：
 - 转移的目的地址
 - 转移的距离
 - 段间转移
 - 段内短转移
 - 段内近转移



jmp指令

■ 转移的目标地址怎么给出？

立即数：转移的目标地址（相对或绝地址）在立即数编码在指令里面。

均可在汇编程序中设置**标号**，由编译器计算目标地址（相对或绝地址）。

寄存器：转移的目标地址（绝对地址）在寄存器中。

内存：转移的目标地址（绝对地址）在内存中。



9.3 依据位移进行转移的jmp指令

- 转移地址为立即数的jmp指令有三种格式：之格式1

- jmp指令格式1

jmp **short** 标号（转到标号处执行指令）

实现段内短转移，IP的修改范围为 -128~127。



9.3 依据位移进行转移的jmp指令

比如：程序9.1

```
assume cs:codesg
codesg segment
    start:mov ax,0
           jmp short s
           add ax,1
    s:inc ax
codesg ends
end start
```

程序执行后ax 等于多少？

因为执行 **jmp short s** 后，
IP 指向了标号 **s** 处的 **inc ax**，
越过了指令 **add ax,1**。

所以执行后， ax= 1，



9.3 依据位移进行转移的jmp指令

- 汇编指令 `jmp short s` 对应的机器码是什么样的呢？

先看汇编指令和其对应的机器指令，（示例）

汇编指令	机器指令
<code>mov ax,0123</code>	<code>B8 23 01</code>
<code>mov ax,ds:[0123]</code>	<code>A1 23 01</code>
<code>push ds:[0123]</code>	<code>FF 36 23 01</code>

以上指令中的idata（立即数）都出现于其机器码中——无论idata是一个数据还是内存单元偏移地址。

猜测： 转换指令机器码中包含目标地址吗？



9.3 依据位移进行转移的jmp指令

- Debug 中将 `jmp short s`
 - ▣ 汇编代码 `jmp 0008`
 - ▣ 机器码为 `EB 03`
- 不包含转移的目的地址，CPU如何知道转移到哪里呢？

```
-u
0BBD:0000 B80000      MOV     AX,0000
0BBD:0003 EB03        JMP     0008
0BBD:0005 050100      ADD     AX,0001
0BBD:0008 40          INC     AX
```



9.3 依据位移进行转移的jmp指令

- 我们把程序9.1改写为程序9.2，如下

```
assume cs:codesg
```

```
codesg segment
```

```
start:mov ax,0
```

```
mov bx,0 ;其他没有改变，只是增加了此句
```

```
jmp short s
```

```
add ax,1
```

```
s:inc ax
```

```
codesg ends
```

```
end start
```

在Debug中将程序9.2翻译为机器码，看看结果



9.3 依据位移进行转移的jmp指令

■ 比较程序1和2用Debug查看的结果

修改后

```
-u
0BBD:0000 B80000      MOV     AX,0000
0BBD:0003 BB0000      MOV     BX,0000
0BBD:0006 EB03        JMP     000B
0BBD:0008 050100      ADD     AX,0001
0BBD:000B 40           INC     AX
```

修改前

```
-u
0BBD:0000 B80000      MOV     AX,0000
0BBD:0003 EB03        JMP     0008
0BBD:0005 050100      ADD     AX,0001
0BBD:0008 40           INC     AX
```

这两jmp目标地址不同，但机器码相同。可知CPU在执行jmp指令时并非直接使用转移的目标地址。

jmp到底如何实现跳转的





9.3 依据位移进行转移的jmp指令

- jmp到底如何实现跳转的？

- 回忆一下

CPU执行指令的过程（参见2.10节）

(1) 从CS:IP指向内存单元读取指令，读取的指令进入指令缓冲区；

(2) $(IP) = (IP) + \text{所读取指令的长度}$ ，从而指向下一条指令；

(3) 执行指令。转到1，重复这个过程。



9.3 依据位移进行转移的jmp指令

- 我们参照程序9.2 中 `jmp short s` 指令的读取和执行过程：
 1. $(CS)=0BBDH$, $(IP)=0006$, $CS:IP$ 指向 `EB03`;
 2. 读取指令码 `EB03` 进入指令缓冲器;
 3. $(IP)=(IP)+\text{指令EB03长度}=(IP)+2=0008$, $CS:IP$ 指向 `add ax,1`;
 4. CPU 执行指令缓冲器中的指令 `EB03`;
 5. 指令 `EB03` 执行后, 修改 $(IP)=000BH$, $CS:IP$ 指向 `inc ax`。

```
-u
0BBD:0000 B80000      MOV     AX,0000
0BBD:0003 BB0000      MOV     BX,0000
0BBD:0006 EB03       JMP     000B
0BBD:0008 050100      ADD     AX,0001
0BBD:000B 40          INC     AX
```



9.3 依据位移进行转移的jmp指令

- 我们参照程序9.2 中 `jmp short s` 指令的读取和执行过程：
 1. $(CS)=0BBDH$, $(IP)=0006$, $CS:IP$ 指向 `EB03`;
 2. 读取指令码 `EB03` 进入指令缓冲器;
 3. $(IP)=(IP)+\text{指令EB03长度}=(IP)+2=0008$, $CS:IP$ 指向 `add ax,1`;
 4. CPU 执行指令缓冲器中的指令 `EB03`;
 5. 指令 `EB03` 执行后, 修改 $(IP)=000BH$, $CS:IP$ 指向 `inc ax`。


```
-u
0BBD:0000 B80000    MOV     AX,0000
0BBD:0003 BB0000    MOV     BX,0000
0BBD:0006 EB03      JMP     000B
0BBD:0008 050100    ADD     AX,0001
0BBD:000B 40        INC     AX
```

注意: CPU 在执行 `EB03` 的时候是根据什么修改 `IP`, 使其指向目标指令呢?
——根据指令码中的 `03`



9.3 依据位移进行转移的jmp指令

- 我们参照程序9.2 中 `jmp short s` 指令的读取和执行过程：
 1. $(CS)=0BBDH$, $(IP)=0006$, $CS:IP$ 指向 `EB03`;
 2. 读取指令码 `EB03` 进入指令缓冲器;
 3. $(IP)=(IP)+\text{指令EB03长度}=(IP)+2=0008$, $CS:IP$ 指向 `add ax,1`;
 4. CPU 执行指令缓冲器中的指令 `EB03`;
 5. 指令 `EB03` 执行后, 修改 $(IP)=000BH$, $CS:IP$ 指向 `inc ax`。



```
-u
0BBD:0000 B80000      MOV     AX,0000
0BBD:0003 BB0000      MOV     BX,0000
0BBD:0006 EB03        JMP     000B
0BBD:0008 050100      ADD     AX,0001
0BBD:000B 40          INC     AX
```



执行 `EB03` 时, CPU 根据什么修改的 `IP`, 使其指向目标指令?

答: 根据指令码中的 `03`—— 执行 `EB03` 时的 $(IP)=0008$, 若当前的 `IP` 值加3, 即转移目的地址是 `CS:000B`



9.3 依据位移进行转移的jmp指令

■ 比较程序1和2用Debug查看的结果

程序9.1

```
-u
0BB0:0000 B8000000 MOV AX,0000
0BB0:0003 BB000000 MOV BX,0000
0BB0:0006 EB03 JMP 000B
0BB0:0008 050100 ADD AX,0001
0BB0:000B 40 INC AX
```

跳转

程序9.2

```
-u
0BB0:0000 B8000000 MOV AX,0000
0BB0:0003 EB03 JMP 0008
0BB0:0005 050100 ADD AX,0001
0BB0:0008 40 INC AX
```

跳转

因为程序9.1、9.22中的**jmp** 指令转移的位移相同，都是向后3个字节，所以它们的机器码都是**EB03**。



9.3 依据位移进行转移的jmp指令

- 在“**jmp short 标号**”指令所对应的机器码中包含的不是目的地址，而是转移的**位移**——目标地址与当前IP的差值。

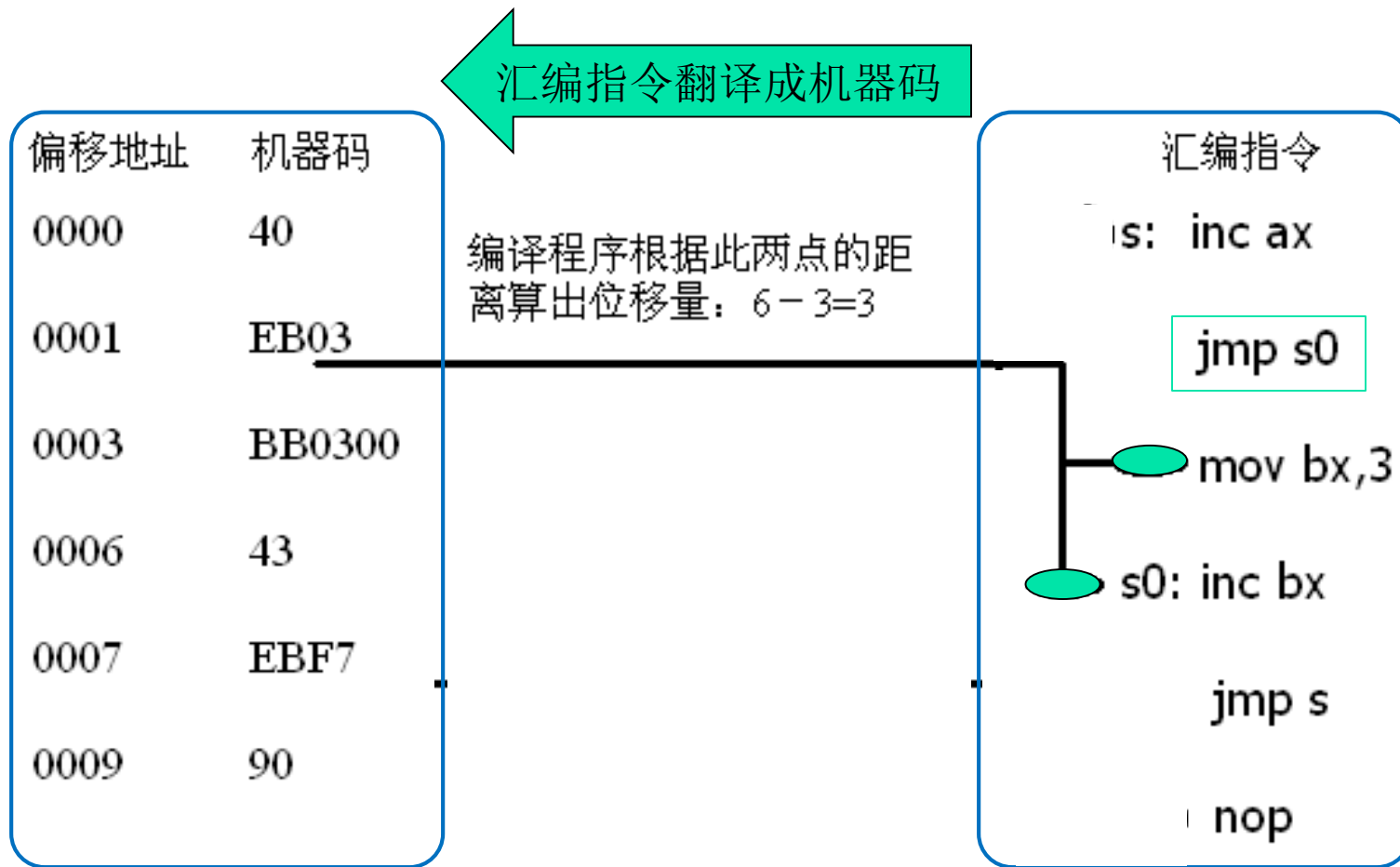
位移怎么得到的？

编译器根据汇编指令中的“标号”计算出来的。



9.3 依据位移进行转移的jmp指令

- 转移位移具体的计算方法如下图：

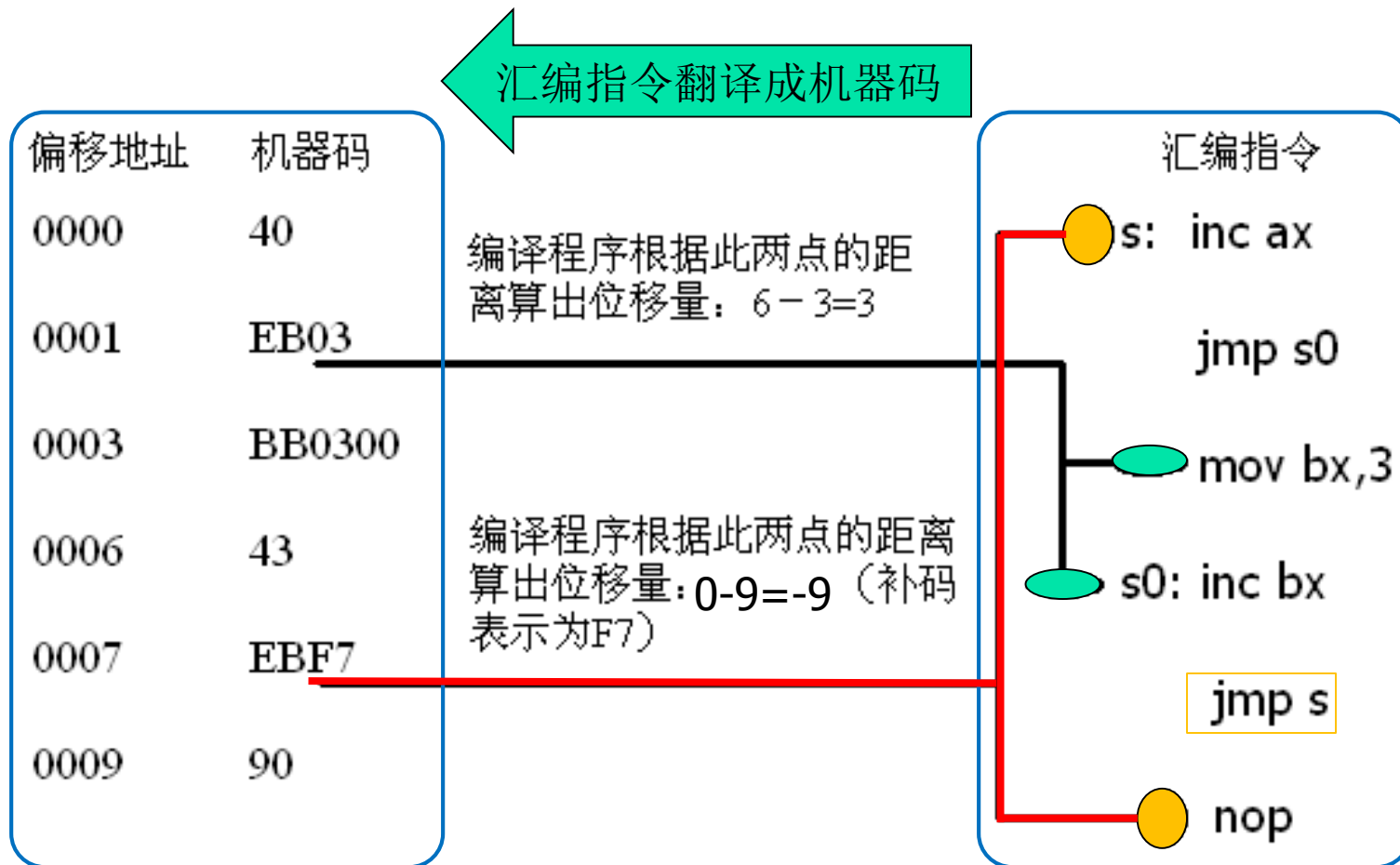


转移位移的计算方法



9.3 依据位移进行转移的jmp指令

- 转移位移具体的计算方法如下图：



转移位移的计算方法



9.3 依据位移进行转移的jmp指令

■ 结论：

CPU执行 `jmp short 标号` 指令时并不需要转移的目的地址，只需知道转移的位移。



9.3 依据位移进行转移的jmp指令

- 指令“**jmp short 标号**”的功能:

$(IP) = (IP) + 8\text{位位移}$ 。

- (1) 8位位移=“标号”处的地址-**jmp**指令后的第一个字节的地址;
- (2) **short**指明此处的位移为8位位移;
- (3) 8位位移的范围为-128~127, 用**补码**表示
(如果你对补码还不了解, 请阅读附注2)
- (4) 8位位移由编译程序在编译时算出。



9.3 依据位移进行转移的jmp指令

- 转移地址为立即数的jmp指令有三种格式：之格式2

jmp near ptr 标号

它实现段内近转移。

功能为： $(IP) = (IP) + 16\text{位位移}$



9.3 依据位移进行转移的jmp指令

- 指令“`jmp near ptr 标号`”的说明：
 - `near ptr`指明此处的位移为16位位移，进行的是段内近转移；
 - 16位位移的范围 -32769~32767，用补码表示；
 - 位移计算方法：
 $\text{16位位移} = \text{“标号”处的地址} - \text{jmp指令后的第一个字节的地址}$ ；
 - 16位位移由编译程序在编译时算出



9.4 转移的目的地址在指令中的jmp指令

- 前面讲的jmp指令，其对应的机器码中并没有转移的目的地址，而是相对于当前IP的转移位移。
- 转移地址为立即数的jmp指令有三种格式：之格式3

- jmp指令格式3

jmp far ptr 标号

实现的是段间转移(远转移)。功能如下：

- (CS)=标号所在段的段地址；
- (IP)=标号所在段中的偏移地址（绝对地址）。
- far ptr指明了指令用标号的段地址和偏移地址修改CS和IP。



9.4 转移的目的地址在指令中的jmp指令

■ 示例，程序9.3：

```
assume cs:codesg
codesg segment
start:mov ax,0
      mov bx,0
      jmp far ptr s
      db 256 dup (0) ; 256个字节被填0
s: add ax,1
   inc ax
codesg ends
end start
```



9.4 转移的目的地址在指令中的jmp指令

■ 9.3翻译成为机器码的结果如图：



为什么目标地址是这么多？

```
-u
0BBE:0000 B8000000 MOV     AX,0000
0BBE:0003 BB000000 MOV     BX,0000
0BBE:0006 EA0B01BD0B JMP     0BBE:010B
0BBE:000B 00000000 ADD     [BX+SI],AL
0BBE:000D 00000000 ADD     [BX+SI],AL
0BBE:000F 00000000 ADD     [BX+SI],AL
0BBE:0011 00000000 ADD     [BX+SI],AL
```

机器码中包含转移的目标地址。

高地址的“BD 0B”是转移的段地址：0BBDH，低地址的“0B 01”是偏移地址：010BH



9.5 转移地址在寄存器中的jmp指令

■ jmp指令格式4

jmp 16位寄存器

■ 功能：IP = (16位寄存器) ~ 绝对地址

■ 此指令的应用参见2.11节



9.6 转移地址在内存中的jmp指令

- 转移地址在内存中的jmp指令有两种格式：之格式1

- **jmp word ptr** 内存单元地址（段内转移）

功能：从内存单元地址处开始存放着一个字，是转移的目的偏移地址——只修改**IP**。

内存单元地址可用寻址方式的任一格式给出。

示例



9.6 转移地址在内存中的jmp指令

- 转移地址在内存中的jmp指令有两种格式：之格式1

示例：

```
mov ax,0123H  
mov ds:[0],ax  
jmp word ptr ds:[0]  
执行后，(IP)=0123H
```

```
mov ax,0123H  
mov [bx],ax  
jmp word ptr [bx]  
执行后，(IP)=0123H
```



9.6 转移地址在内存中的jmp指令

- 转移地址在内存中的jmp指令有两种格式：之格式2

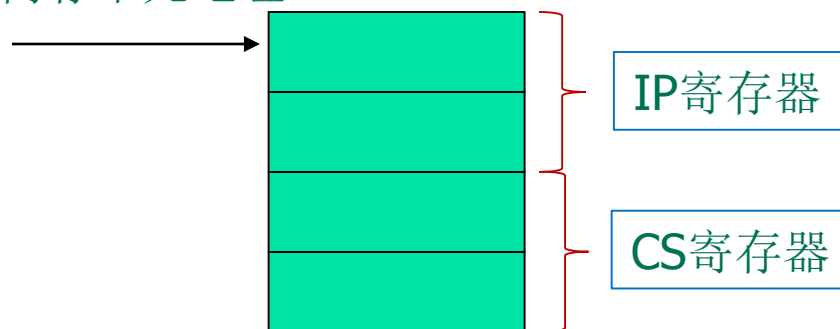
jmp dword ptr 内存单元地址（段间转移）

功能：转移目的地址在内存单元地址处开始两个字中，其中高地址的为**段地址**，低地址的字为**偏移地址（绝对值）**——同时修改**CS**和**IP**。

$(CS) = (\text{内存单元地址} + 2)$

$(IP) = (\text{内存单元地址})$

内存单元地址



内存单元地址可用寻址方式的任一格式给出。 示例



9.6 转移地址在内存中的jmp指令

- (2) `jmp dword ptr` 内存单元地址（段间转移）
示例：

```
mov ax,0123H  
mov ds:[0],ax  
mov word ptr ds:[2],0  
jmp dword ptr ds:[0]
```

执行后，

(CS)= 0

(IP)= 0123H

```
mov ax,0123H  
mov [bx],ax  
mov word ptr [bx+2],0  
jmp dword ptr [bx]
```

执行后，

(CS)= 0

(IP)= 0123H



特别提示

■ 检测点9.1 (p183)

(2) 程序如下。

```
assume cs:code
```

```
data segment
```

```
    dd 12345678H
```

```
data ends
```

```
code segment
```

```
start:mov ax,data
```

```
    mov ds,ax
```

```
    mov bx,0
```

```
    mov [bx],_____
```

```
    mov [bx+2],_____
```

```
    jmp dword ptr ds:[0]
```

```
code ends
```

```
end start
```

补全程序，使 jmp 指令执行后，CS:IP 指向程序的第一条指令。



特别提示

■ 检测点9.1 (p183)

(3) 用 Debug 查看内存, 结果如下:

2000:1000 BE 00 06 00 00 00

则此时, CPU 执行指令:

```
mov ax,2000H  
mov es,ax  
jmp dword ptr es:[1000H]
```

后, (CS)=? , (IP)=?

■ 没有完成此检测点, 请不要向下进行。



jmp 指令总结

- 跳转目标地址为**标号**（**立即数**）
 - **jmp short** 标号 ;相对跳转: 指令用 8位符号整数表示相对位移
 - **jmp near** 标号 ;相对跳转: 指令用 16位符号整数表示相对位移
 - **jmp far ptr** 标号;绝对跳转: 指令包含段内偏移地址和段地址
- 跳转目标地在**寄存器**中
 - **jmp 寄存器** ;绝对跳转: 寄存器中数据修为段内偏移地址
- 跳转目标地址在**内存**中
 - **jmp word ptr 地址** ;绝对跳转: 内存中数据修为段内偏移地址
 - **jmp dword ptr 地址** ;绝对跳转: 内存中数据修为段内偏移地址和段地址



9.7 jcxz指令

- **jcxz**指令为有条件转移指令，所有的有条件转移指令都是短转移，在对应的机器码中包含转移的位移，而不是目的地址。对**IP**的修改范围都为**-128~127**。

- **jcxz**指令格式：

jcxz 标号

功能：如果(**cx**)=0，则转移到标号处执行。

- 操作



9.7 jcxz指令

■ jcxz 标号 指令操作:

■ 当(cx)=0时, (IP)=(IP)+8位位移

□ 8位位移=“标号”处的地址-jcxz指令后的第一个字节的地址;

□ 8位位移的范围为-128~127, 用补码表示;

□ 8位位移由编译程序在编译时算出。

■ 否则, 程序向下执行。

■ 若用C语言和汇编语言综合描述, 以上功能相当于:

if((cx)==0) jmp short 标号;



特别提示

- 检测点9.2 (p184)
- 没有完成此检测点，请不要向下进行。



9.8 loop指令

- loop指令为循环指令，所有的循环指令都是短转移，在对应的机器码中包含转移的位移，而不是目的地址。对IP的修改范围都为-128~127。

- loop指令格式：

loop 标号

$(cx) = (cx) - 1$ ，如果 $(cx) \neq 0$ ，转移到标号处执行。

- 操作



9.8 loop指令

■ loop 标号 指令操作：

(1) $(cx) = (cx) - 1;$

(2) 如果 $(cx) \neq 0$, $(IP) = (IP) + 8\text{位位移}$ 。

▣ 8位位移 = “标号” 处的地址 - loop指令后的第一个字节的地址；

▣ 8位位移的范围为 -128~127, 用补码表示；

▣ 8位位移由编译程序在编译时算出。

(3) 当 $(cx) = 0$, 程序向下执行。

■ 用c和汇编混合描述，以上功能相当于：

$(cx) --;$

$\text{if}((cx) \neq 0) \text{ jmp short 标号}$



特别提示

■ 检测点9.3 (p185)

补全编程，利用 loop 指令，实现在内存 2000H 段中查找第一个值为 0 的字节，找到后，将它的偏移地址存储在 dx 中。

```
assume cs:code
code segment
    start:mov ax,2000H
           mov ds,ax
           mov bx,0
           s: mov cl,[bx]
              mov ch,0

           _____
           inc bx
           loop s
           ok:dec bx
              mov dx,bx
              mov ax,4c00h
              int 21h
code ends
end start
```

;dec 指令的功能和 inc 相反，dec bx 进行的操作为：(bx) = (bx) - 1

没有完成此检测点，请不要向下进行。



9.9 根据位移进行转移的意义

- 前面我们讲到：

jmp short 标号

jmp near ptr 标号

jcxz 标号

loop 标号

这些指令对 **IP** 的修改是根据转移目的地址和转移起始地址之间的位移来进行——此位移信息包含在机器码中。

- 这样设计，方便了程序段在内存中的浮动装配。



9.9 根据位移进行转移的意义

- 例如：

汇编指令	机器代码
mov cx,6	B9 06 00
mov ax,10	B8 10 00
s: add ax,ax	01 C0
loop s	E2 FC

- 这段程序装在内存中的不同位置都可正确执行，因为 **loop s** 在执行时只涉及到 s 的位移（-4，前移 4 个字节，补码表示为 **FCH**），而不是 s 的地址。
- 类似于 c 程序 `#include` 头文件是用相对路径比用绝对路径好。



9.10 编译器对转移位移超界的检测

- **注意**：根据**位移**进行转移时，转移位移受到移范围受到的限制，否则编译器将报错。

- **示例**

```
assume cs:code
```

```
code segment
```

```
start: jmp short s
```

```
db 128 dup(0)
```

```
s: mov ax,0ffffh
```

```
code ends
```

```
end start
```

← jmp short s的转移范围-128~127，

} 插入数据128（大于127）字节，

程序将引起编译错误。