第4章　寻址方式

- 茫茫内存，何处寻觅操作数?
- CPU如何知道操作数的地址?
- 日常生活中，有哪些给出地址的方式?
- C程序中，有哪些给出地址的方式?


```
int   i, j;
int   a[10];
int   *p;
int   *q[10];
int   b[20][10];
```

```
struct point {
    int px;
    int py;
};
```

```
point  original;
```

```
point  lists[10];
```

思考：如何得到要访问单元的地址？

```
a[i] =5;
b[i][j] =10;
original.px=12;
lists[5].px=15;
```

第4章　寻址方式

一、本章的学习内容

立即寻址

寄存器寻址

直接寻址

寄存器间接寻址

变址寻址

基址加变址寻址

寻址方式的综合举例

x86机器指令编码规则

第4章　寻址方式

二、学习重点

6种寻址方式的使用格式、语法规定；

6种寻址方式地址表示的含义及应用。

三、学习的难点

直接寻址、寄存器间接寻址、变址寻址、

基址加变址寻址的使用格式及功能。

四、学习方法

比较共性与差异、与C语言对应，多用

4.1 寻址方式概述

对一条指令，关注的焦点有哪些？

- 执行什么操作？
- 操作数在哪里？

操作数的存放位置即存放地址
- CPU内的寄存器
- 主存
- I/O设备端口

操作数在主存时：关注段址/段选择符、段内偏移

- 操作数的类型　字节/字/双字？

寻找操作数存放地址的方式称为寻址方式。

4.1 寻址方式概述

双操作数的指令格式　　操作符 OPD, OPS

intel

ADD　EAX，EBX

目的操作数地址　　源操作数地址

(OPD) + (OPS) → OPD

(EAX) + (EBX) → EAX

Question：操作结束后，运算结果保存在哪？

源操作数是否变化？

4.2 立即寻址

- 操作数直接放在指令中，在指令的操作码后；
- 操作数是指令的一部分，位于代码段中；
- 指令中的操作数是8位、16位或32位二进制数。

使用格式： n

操作码及目的操作数寻址方式码
立即操作数 n

立即操作数只能作为源操作数。

例： MOV EAX , 12H

机器码： B8 12 00 00 00

4.2 立即寻址

- 立即操作数只能作为源操作数

  右值表达式
- 立即数值的大小应在另一个操作数的类型限定的取值范围内
- 字符、字符串是立即数

  MOV AL, ’1’　　MOV AL, 31H

  MOV AX, ’12’　　MOV AX, 3132H
- 由常量组成的数值表达式是立即数

  MOV EAX, 3*4+5*6　　MOV EAX, 42

外在的表象（表达式）-» 编译-» 内部的本质（立即数）

4.2 立即寻址


```
int   global;   // 注：global 是全局变量
global = 2 * 5 ;
00521A17      C7 05 B8 A5 52 00 0A 00 00 00
                mov    dword ptr [global (052A5B8h)],0Ah
```

4.3 寄存器寻址

使用格式： R

功　　能： 寄存器R中的内容即为操作数。

说　　明： 除个别指令外，R可为任意寄存器。

例1：DEC　BL

执行前 ： (BL)=43H

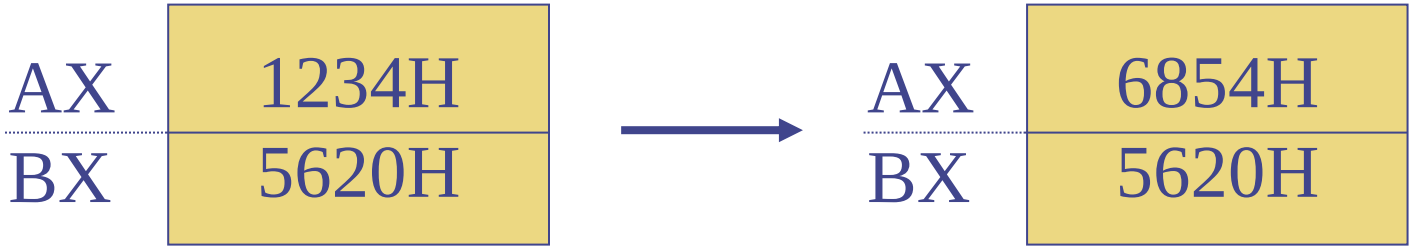
执　行： (BL) –1 = 43 H –1 = 42H → BL

执行后： (BL)=42H

Question： 操作数在哪？操作数类型是什么？

4.3 寄存器寻址

例2：ADD AX , BX



执行前： (AX)=1234H, (BX)=5620H

执行： (AX)+(BX) = 6854H → AX

结果： (AX)=6854H , (BX)=5620H

例3：MOV AX，BX

4.3 寄存器寻址

例4：ADD EAX , EDX

Question： 指令 MOV AX, BH 正确吗？为什么？

Question： 如何实现 (AX)+(BX)-> CX ？

```
MOV  CX,  AX
ADD  CX,  BX
```

```
ADD  AX,  BX
MOV  CX,  AX
```

有缺陷：

改变了（AX）

4.4 直接寻址

- 操作数在内存中；
- 操作数的偏移地址EA紧跟在指令操作码后面。

格式：变量 ；等同 [变量]

变量 + 常量 ；等同 变量[常量]

；等同 [变量 +常量]

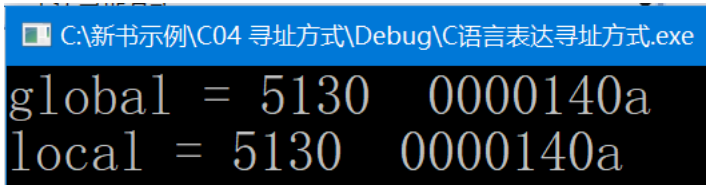
要点：编译器对 变量+常量 进行编译时，取变量的地址，将该地址与常量运算，得到一个新地址存放在机器码中。

Q：操作数的类型是什么？

段地址和偏移地址怎样得到？

4.4 直接寻址

```
int global;
void address_compare()
{
    int local;
    global = 2*5;
    *((char*)&global + 1) = 20;
    printf("global = %d  %08x\n", global,global);

    local = 10;
    *((char*)&local + 1) = 20;
    printf("local = %d  %08x\n", local, local);
}
```



虽然全局/局部变量的访问形式相同，

但是它们对应的寻址方式是不同的。

4.4 直接寻址

- 操作数在内存中；
- 操作数的偏移地址EA紧跟在指令操作码后面。

全局变量的访问：直接寻址

```
    global = 2*5;
C7 05 44 A1 E9 00 0A 00 00 00 mov         dword ptr ds:[00E9A144h],0Ah
    *((char*)&global + 1) = 20;
C6 05 45 A1 E9 00 14 mov         byte ptr ds:[00E9A145h],14h
```

局部变量的访问：变址寻址

```
    local = 10;
C7 45 F4 0A 00 00 00 mov         dword ptr [ebp-0Ch],0Ah
    *((char*)&local + 1) = 20;
C6 45 F5 14          mov         byte ptr [ebp-0Bh],14h
```

4.4 直接寻址

Q：下面三种写法各自的含义是什么？

```
*((char*)&global + 1) = 20;
*((char*)(&global + 1)) = 20;
*(int *)((char*)&global + 1) = 20;
```

```
    global = 2*5;
C7 05 CC A5 99 00 0A 00 00 00 mov         dword ptr ds:[0099A5CCh],0Ah
    *((char*)&global + 1) = 20;
C6 05 CD A5 99 00 14 mov         byte ptr ds:[0099A5CDh],14h
    *((char*)(&global + 1)) = 20;
C6 05 D0 A5 99 00 14 mov         byte ptr ds:[0099A5D0h],14h
    *(int *)((char*)&global + 1) = 20;
C7 05 CD A5 99 00 14 00 00 00 mov         dword ptr ds:[0099A5CDh],14h
```

4.4 直接寻址


```
X    DD   10,   20,   30,   40
MOV    EAX,   X          ;   (EAX)=10
MOV    EAX,   [X]        ;   (EAX)=10
MOV    EAX,   X+4        ;   (EAX)=20
MOV    EAX,  X[4]        ;   (EAX)=20
MOV    EAX,   [X+8]      ;   (EAX)=30
MOV    EAX,   X+1        ;   (EAX)= 14000000H
```

提醒：注意C语言的写法与汇编源程序编写法的差异！

Q：直接寻址与C语言中一维数组的访问有何异同点？

Q：如何将 x中第 0 个双字数据 +2-> EAX?

```
MOV   EAX,   X
ADD   EAX,   2
```

4.4 直接寻址

提醒：注意C语言的写法与汇编源程序中写法的差异！

- 汇编语言源程序中，x[n]是从x处开始，再偏移n个字节。

- C 程序中， x[n] 是从 x 处开始，再偏移 n 个元素
  编译后生成的结果是：
  从 x处开始，再偏移 (n* x的数据类型长度)个字节。

- 汇编语言源程序中，x[n] 等同 x+n。
- C 程序中， x[n] 与 x + n 完全不同

- C 程序中，实现与汇编源程序中 x + n 的等同表达是：
  (x的类型 *)((char *)&x + n)

4.4 直接寻址

直接寻址另一种写法

格式： 段寄存器名：[n]

功能：操作码的下一个字（或双字）单元的内容为操作数的偏移地址EA。

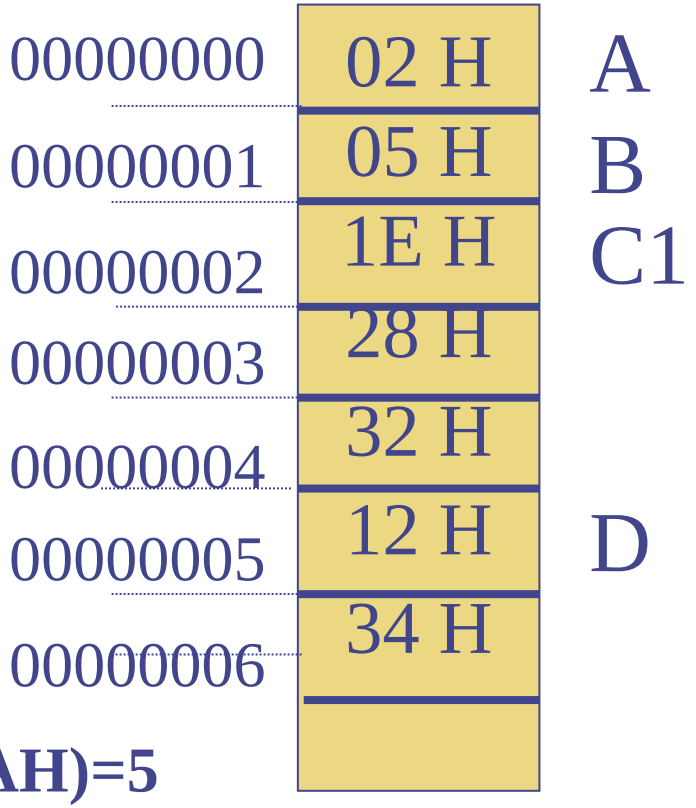
- 操作数所在的段：由段寄存器名指示
- 操作数的类型： 未知

变量 或者 变量[常量] 编译后对应的值即为 n.

4.4 直接寻址

```
.DATA
A    DB   2
B    DB   5
C1   DB   30, 40, 50
D    DW   3412H
```

C 不能作为变量名



(1) MOV AH，B　　(AH)=5

(2) MOV CX, D　　(CX)=3412H

(3) MOV AL, C1+1　　(AL)=40

(4) MOV AX, WORD PTR A　　(AX)=0502H

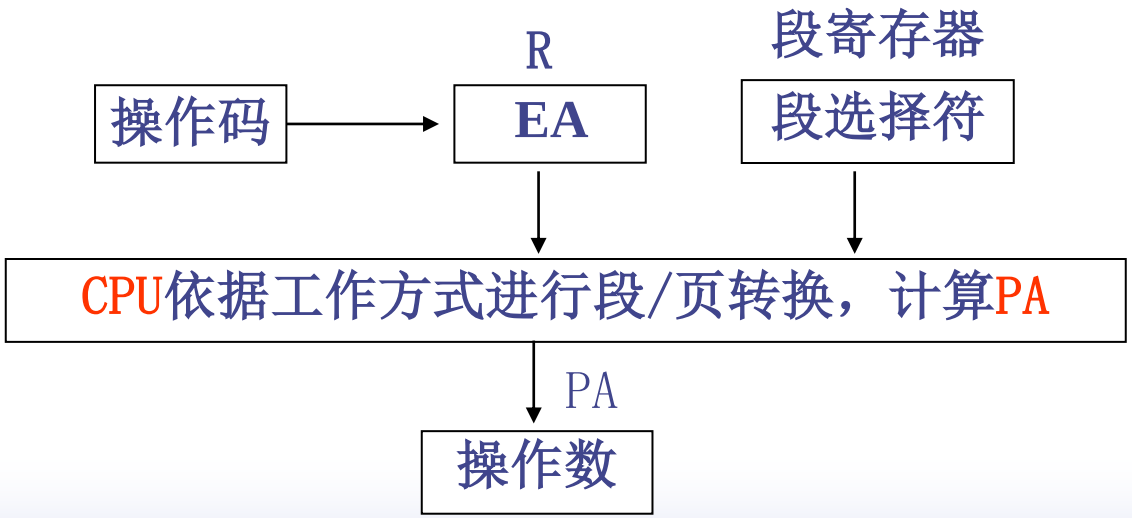
(5) MOV EAX,DWORD PTR A　　(EAX)=281E0502H

4.5 寄存器间接寻址

格式：[R]

功能：操作数在内存中，操作数的偏移地址在寄存器R中。即（R)为操作数的偏移地址.

例如：MOV　AX, [ESI]



寄存器间接寻址方式的寻址过程

4.5 寄存器间接寻址

- R 可以是:
  8个32位通用寄存器中的任意一个
  EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP

- 操作数的偏移地址在指令指明的寄存器中

- 操作数所在的段是?
  扁平内存管理模式下，（DS）=（SS）

- 操作数的类型：未知

4.5 寄存器间接寻址

例1：MOV AX, [ESI]

执行前 (AX)=0005H

(ESI) =00000020H

DS:(00000020H)=1234H

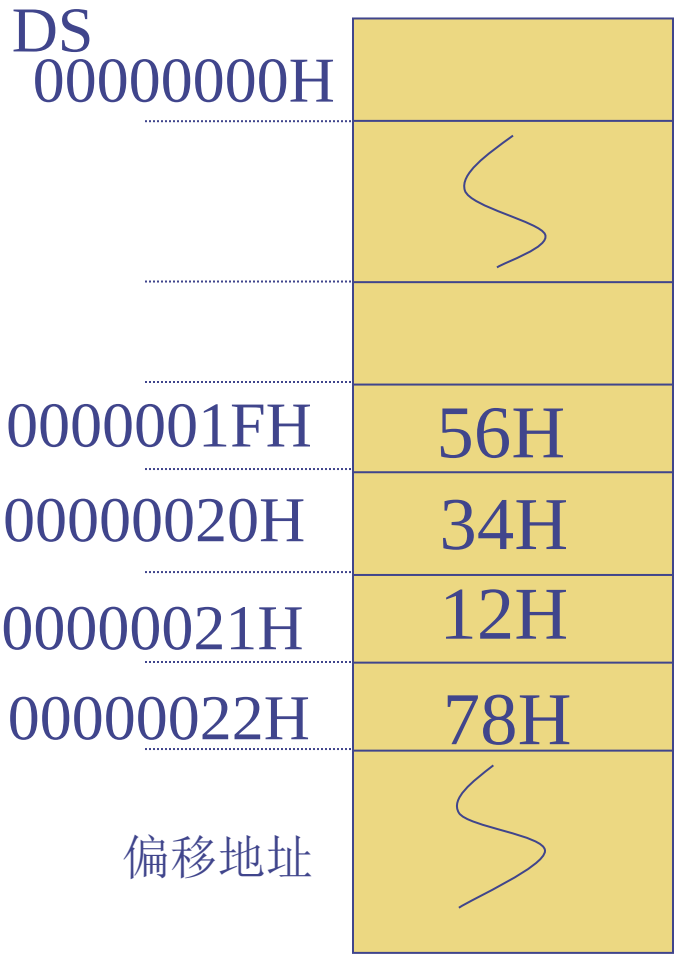
执行后 (AX)= 1234H

(ESI) = 00000020H

问：MOV CL, [ESI]

(CL) = ?

操作数的类型是如何确定的?



4.5 寄存器间接寻址

例2：MOV AH, [EBP]

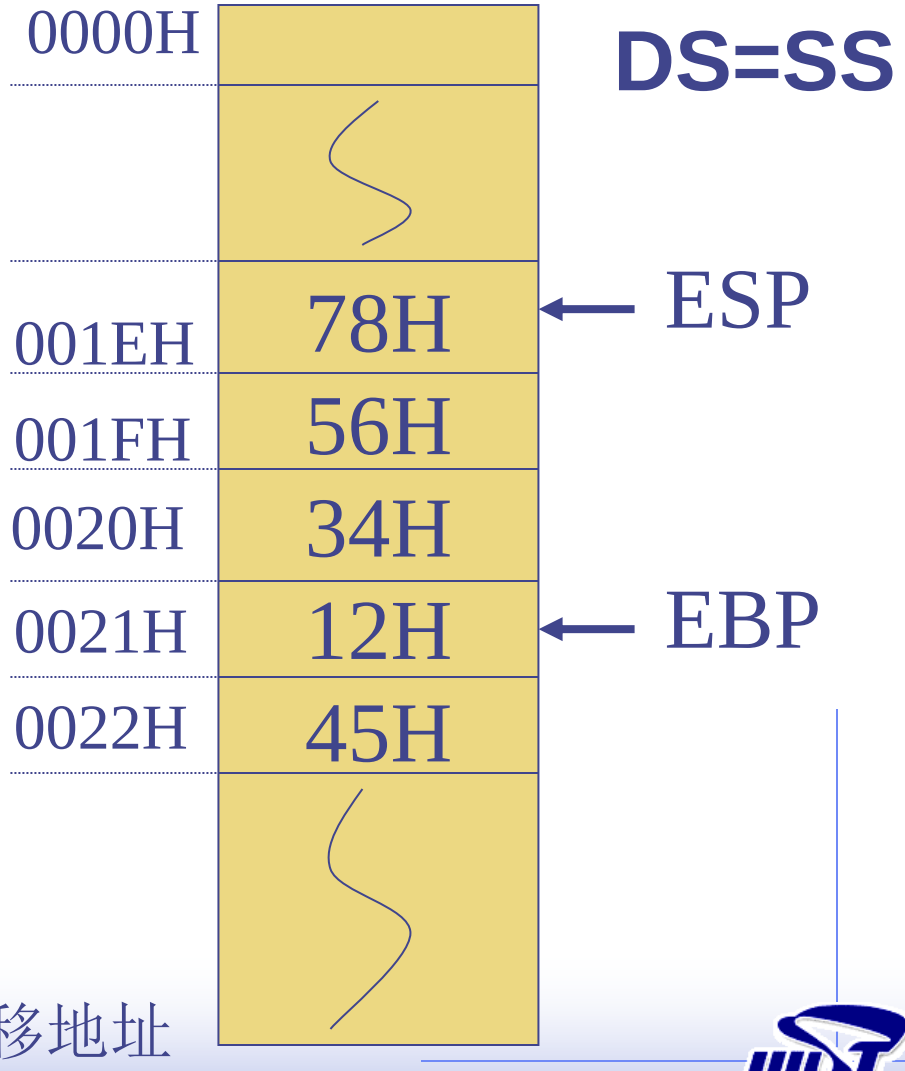
执行前 (AX)=0005H

(EBP) =21H

SS:(EBP) =12H

执行后 (AX) =1205H

(EBP) =21H

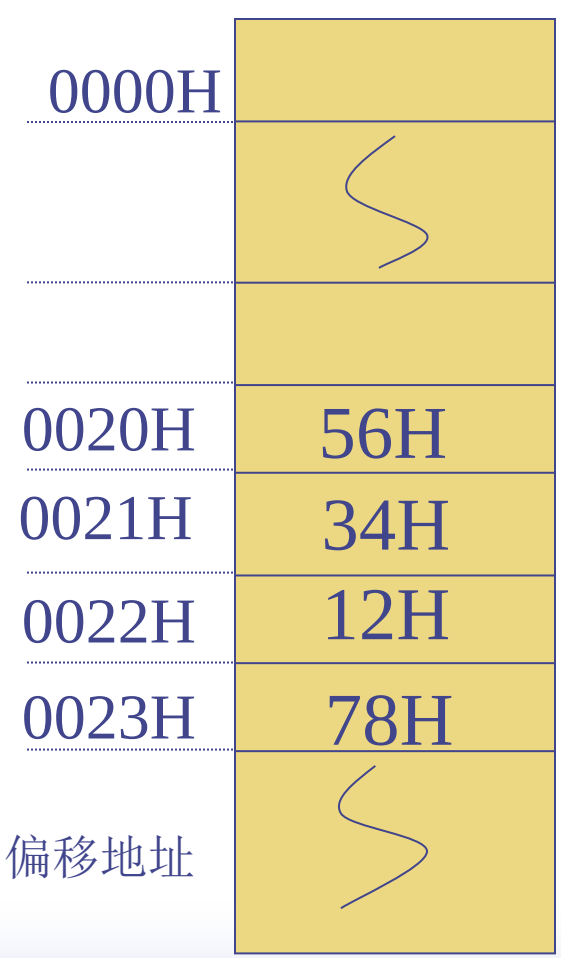


4.5 寄存器间接寻址

比较：

```
MOV    EAX，EBX
MOV    EAX, [EBX]
```

EAX	78123456H
EBX	00000020H



比较：

~~MOV AX，EBX~~

```
MOV   AX, [EBX]
```

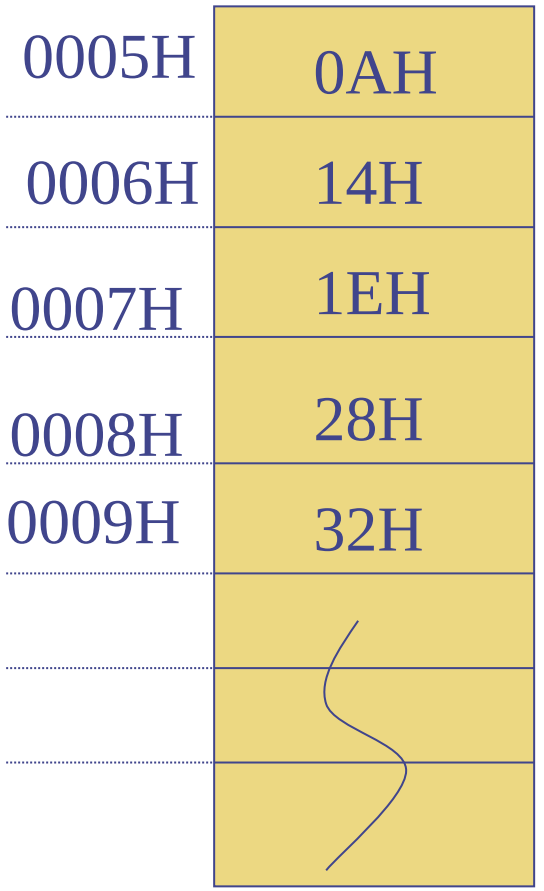
4.5 寄存器间接寻址

例：设 BUF DB 10,20,30,40,50
即以BUF为首址的字节区中存放有5个数据，求它们的和。

算法分析：……

和？ AL
循环次数？ ECX
数据位置？ EBX

EBX 0005 ([EBX])



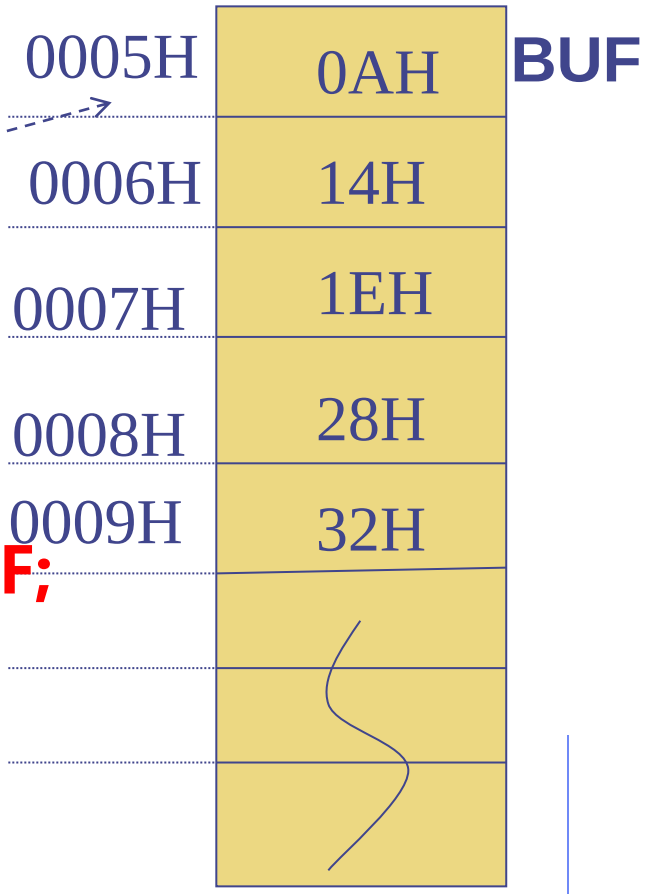
共同特点：单元中的内容无规律，
但单元之间的地址有规律。

4.5 寄存器间接寻址


```
char  buf[5]={10,20,30,40,50} ;

 char  *p;

  0005
```

EBX

```
 p=buf;        MOV EBX, OFFSET BUF;

 AL=AL+*p;     ADD AL, [EBX];
 p=p+1;        ADD EBX, 1;
```



C 用指针实现存储单元内容的间接访问；

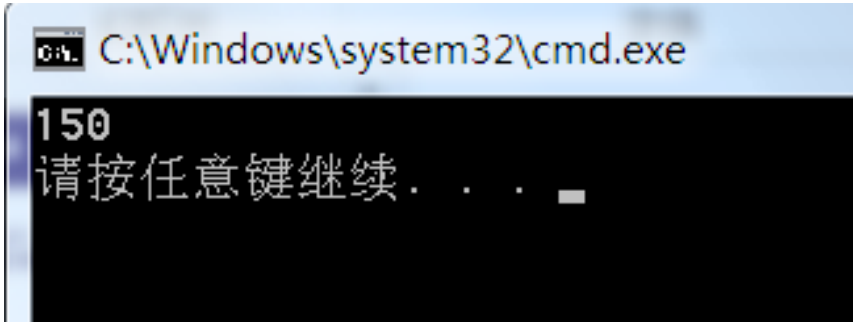
本质：寄存器间接寻址

4.5 寄存器间接寻址


```
; 程序功能：求一个数组缓冲区中 5个字节数据的和，输出和
.686P
.model flat, c
  ExitProcess PROTO STDCALL :DWORD
  includelib  kernel32.lib
  printf      proto c :vararg
  includelib  libcmt.lib
  includelib  legacy_stdio_definitions.lib

.DATA
  lpFmt   db  "%d",0ah, 0dh, 0
  buf     db  10, 20, 30, 40, 50

.STACK 200
```

4.5 寄存器间接寻址

```
.CODE
 main    proc
    mov   eax, 0
    mov   ebx, offset buf
    mov   ecx, 0
lp: cmp    ecx, 5
    jge    exit
    add   al, [ebx]
    add   ebx, 1
    inc   ecx
    jmp   lp
exit:
    invoke printf,offset lpFmt,eax
    invoke ExitProcess, 0
main  endp
end
```



Question:

```
 add  al, [ebx]
```

能否改为：

```
 add al, ebx
```

不能，语法错误
操作数类型不匹配

Question:

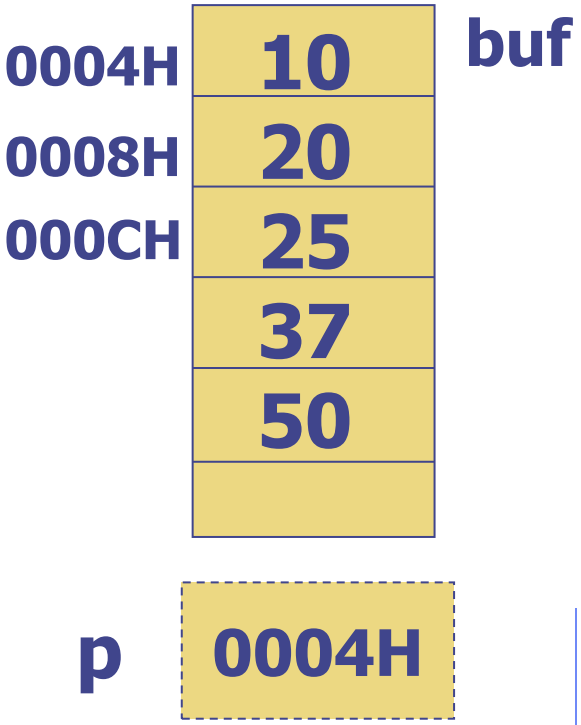
能否将 add al,[ebx] 处的两条语句改为：

```
   add al, buf
   add buf, 1
```

不能，add buf,1 是将buf中的内容加了1

4.5 寄存器间接寻址

```
int   buf[5]={10,20,25,37,50} ;
int   i;
int  *p;
int   result=0;
p=buf;
for  (i=0;i<5;i++)
{
    result+=*p;
    p=p+1;
}
```



P与EBX对应：

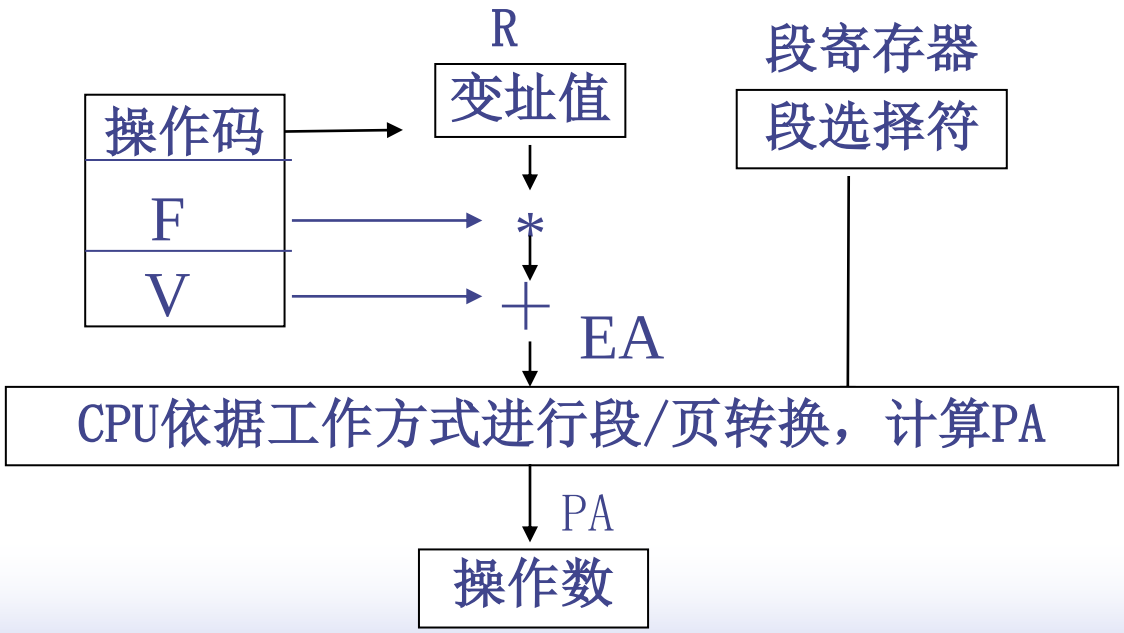
```
MOV   EBX,   OFFSET buf
ADD   EAX,   [EBX]
ADD   EBX,    4
```

4.6 变址寻址

格式：V[R×F] 或[R×F+V] 或 [R×F] +V

功能：R中的内容×F + V 为操作数的偏移地址。

例如：MOV AL, [EBX*2]+5



4.6 变址寻址

格式：[R×F+V]　或　[R×F]+V，　V[R×F]

- R 可以是：

  8个32位通用寄存器

  EAX，EBX，ECX，EDX,ESI,EDI,ESP,EBP

- F 可为 1，2，4，8
- V 可为数值常量，也可以为一个变量。
- 当V 为全局变量时，取该变量对应单元的有效地址参与运算。

4.6 变址寻址


```
19  short gsa[5];
20  int main(int argc, char* argv[])
21  {
22      short lsa[5];
23      int i = 2;
24      gsa[1] = 20;
25      lsa[1] = 20;
26      gsa[i] = 30;
27      lsa[i] = 30;
```

```
    gsa[1] = 20;
007D1A19 B8 02 00 00 00       mov         eax,2
007D1A1E C1 E0 00             shl         eax,0
007D1A21 B9 14 00 00 00       mov         ecx,14h
007D1A26 66 89 88 C0 A5 7D 00 mov         word ptr [eax+007DA5C0h],cx
```

全局变量 编译后对应一个数值化的地址

4.6 变址寻址

全局变量数组访问：比较常量下标与变量下标的差异

```
    gsa[1] = 20;
007D1A19 B8 02 00 00 00       mov         eax,2
007D1A1E C1 E0 00             shl         eax,0
007D1A21 B9 14 00 00 00       mov         ecx,14h
007D1A26 66 89 88 C0 A5 7D 00 mov         word ptr [eax+007DA5C0h],cx
```

```
    gsa[i] = 30;
007D1A3F B8 1E 00 00 00       mov         eax,1Eh
007D1A44 8B 4D E0             mov         ecx,dword ptr [ebp-20h]
007D1A47 66 89 04 4D C0 A5 7D 00 mov         word ptr [ecx*2+007DA5C0h],ax
```

全局变量 编译后对应一个数值化的地址

4.6 变址寻址

局部变量数组访问：比较常量下标与变量下标的差异

```
    lsa[1] = 20;
007D1A2D B8 02 00 00 00       mov         eax,2
007D1A32 C1 E0 00             shl         eax,0
007D1A35 B9 14 00 00 00       mov         ecx,14h
007D1A3A 66 89 4C 05 EC       mov         word ptr [ebp+eax-14h],cx

    lsa[i] = 30;
007D1A4F B8 1E 00 00 00       mov         eax,1Eh
007D1A54 8B 4D E0             mov         ecx,dword ptr [ebp-20h]
007D1A57 66 89 44 4D EC       mov         word ptr [ebp+ecx*2-14h],ax
```

局部变量 编译后对应一个地址为 [ebp - n]

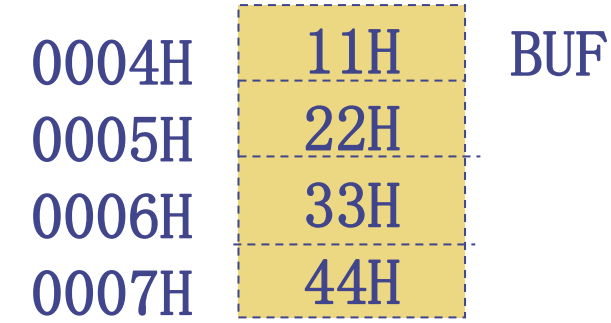
4.6 变址寻址

格式：[R×F+V]　　或　　[R×F]+V

- V 可为数值常量，也可以为一个变量。
- 当V 为变量时，取该变量对应单元的有效地址参与运算。

例：BUF DW 2211H, 4433H

MOV AX, BUF[EBX*2]



Question:

若 （EBX）=0，　（AX）=?　2211H

若 （EBX）=1，　（AX）=?　4433H

4.6 变址寻址

- 操作数所在的段

  (DS) = (SS)

4.6 变址寻址

- 操作数所在的段

  在 VS 2109 编程中，

  直接在 .DATA, .STACK, .CODE 中定义的变量是全局变量。

  在 编译链接后， DS与 SS的值相同。

  在 CODE 中的变量，也会变成 DS：[...]

在 子程序中，可以定义局部变量，其空间分配方式完全不同。

4.6 变址寻址

用变址寻址方式写一段程序，求BUF中数据的和

```
.686
.model flat, stdcall
  ExitProcess PROTO STDCALL :DWORD
  includelib  kernel32.lib

  printf      PROTO C :ptr sbyte, :VARARG
  includelib  libcmt.lib
  includelib  legacy_stdio_definitions.lib

.DATA
  lpFmt   db  "%d",0ah, 0dh, 0
  buf     dd  10,20,30,40,50

.STACK 200
```

4.6 变址寻址

用变址寻址方式写一段程序，求BUF中数据的和

```
.CODE
main  proc c
     MOV   EAX, 0
     MOV   EBX, 0
LP: CMP   EBX, 5
    JGE    EXIT
    ADD    EAX, buf[EBX*4]
    ADD    EBX, 1
    JMP    LP
EXIT:
    invoke printf,offset lpFmt,EAX
    invoke ExitProcess, 0
main  endp
END
```

4.6 变址寻址


```
int   buf[5]={10,20,25,37,50} ;
int   i;
int   result=0;
for  (i=0;i<5;i++)
      result+=buf[i];
```

思考:

(EBX) 中的值就是 变量i的值，

为何C语言中的访问方式是 buf[i]，

而汇编语言中是 buf[EBX*4] ？

4.6 变址寻址

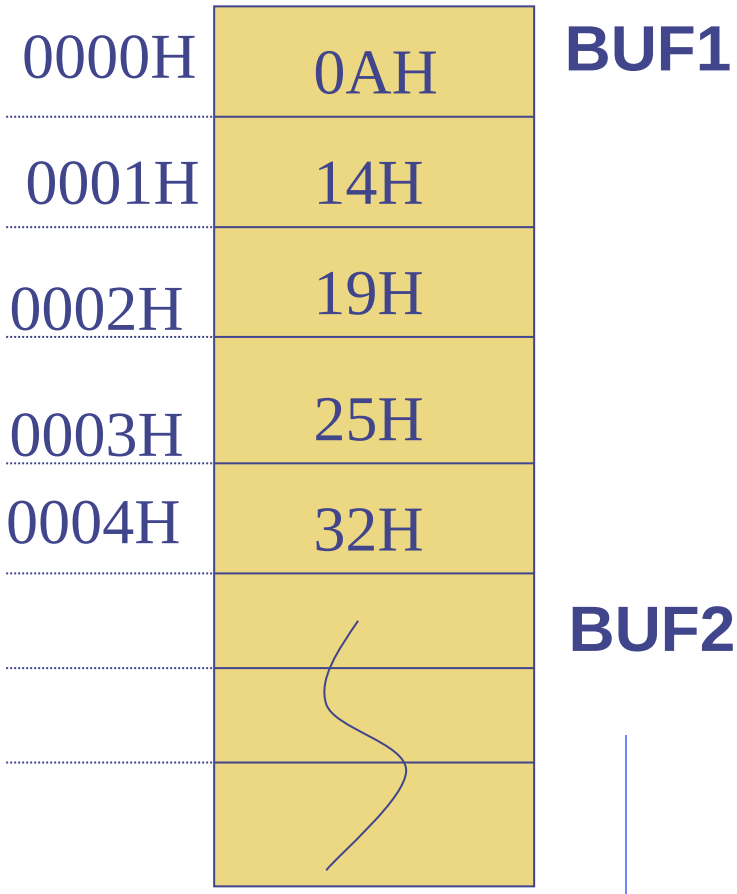
变址寻址和寄存器间接寻址有何异同？

```
        MOV   EBX , 0
        MOV   EAX , 0
LP:     CMP  EBX, 5
        JGE  EXIT
        ADD   EAX,BUF [EBX*4]
        INC    EBX
        JMP    LP
```

```
        MOV   ECX , 0
        MOV   EAX , 0
        MOV   EBX , OFFSET  BUF
LP:     CMP  ECX,5
        JGE   EXIT
        ADD  EAX, [EBX]
        ADD  EBX,4
        INC    ECX
        JMP    LP
```

4.6 变址寻址

例2：设 BUF1 DB 10,20,25,37,50
即以BUF1为首址的字节区中存放有5个数据，将它们拷贝到以BUF2为首址的字节区。



与数组类比：A[0], A[1],A[2],…
B[0],B[1],B[2]

I　　A[I]

EBX　0000　　BUF1[EBX]

BUF2[EBX]

4.6 变址寻址

例2：以BUF1为首址的字节区中存放有5个数据，将它们拷贝到以BUF2为首址的字节区。

```
for (i=0;i<5;i++)  BUF2[i] = BUF1[i];
```

```
          MOV  EBX, 0
MAINP:    CMP  EBX,5
          JGE   EXIT
          MOV  AL,  BUF1[EBX]
          MOV  BUF2[EBX], AL
          INC   EBX
          JMP   MAINP
EXIT:
```

4.6 变址寻址

例2：以BUF1为首址的字节区中存放有5个数据，将它们拷贝到以BUF2为首址的字节区。

```
        MOV  ESI, OFFSET BUF1
        MOV  EDI, OFFSET BUF2
        MOV  ECX, 5

MAINP:  MOV  AL,  [ESI]
        MOV  [EDI], AL
        INC   ESI
        INC   EDI
        DEC  ECX
        JNZ   MAINP
EXIT:
```

用寄存器间接寻址的程序段

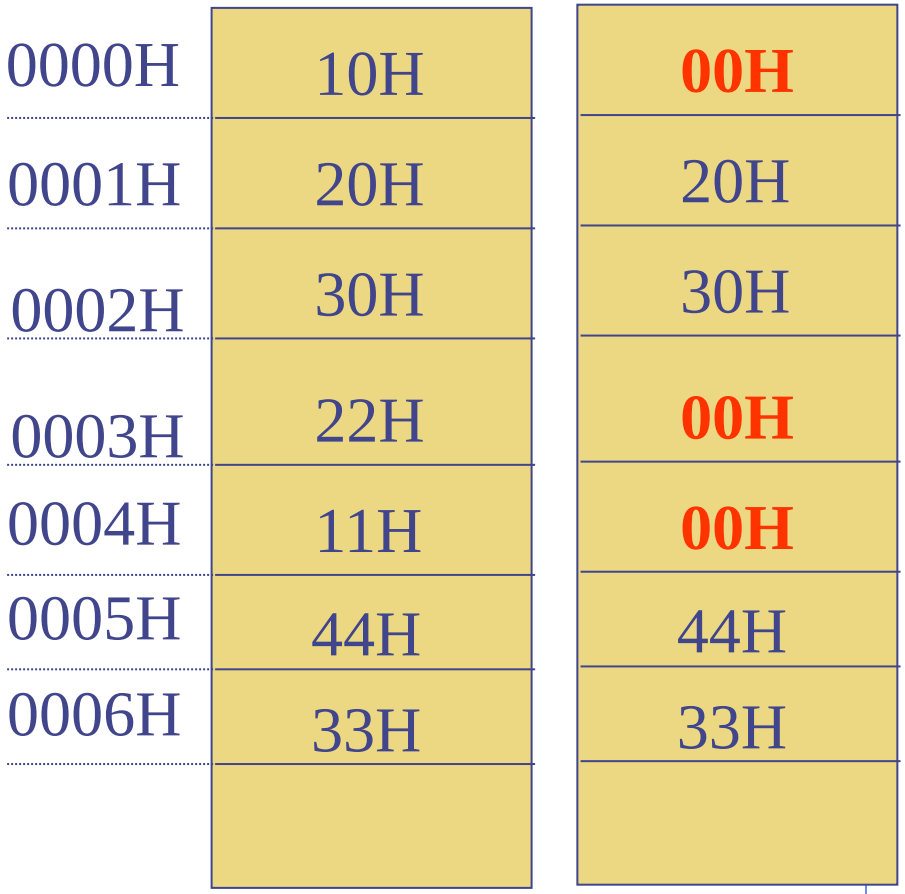
4.6 变址寻址

例3：以BUF1为首址的双字区中存放有5个数据，将它们拷贝到以BUF2为首址的字节区。

```
for (i=0;i<5;i++)  BUF2[i] = BUF1[i];

            MOV  EBX, 0
MAINP:      CMP  EBX,5
            JGE   EXIT
            MOV  EAX,  BUF1[EBX*4]
            MOV  BUF2[EBX*4], EAX
            INC    EBX
            JMP   MAINP
EXIT:
```

4.6 变址寻址

```
.data
x     DB   10H, 20H, 30H
 y    DW   1122H, 3344H
.code
        ……
        MOV    EBX,   0
        MOV  x[EBX], 0
        MOV  y[EBX], 0
        ……
```



翻译出的机器指令

```
mov   byte ptr[EBX],0
mov  word ptr[EBX+0003],0
```

为什么？

4.7 基址加变址寻址

格式： [BR＋IR×F+V]

或 V[BR][IR×F] 或 V [IR×F][BR]

或 V[BR+IR×F]

功 能：操作数的偏移 ＝变址寄存器IR中的内容×比例因子F +位移量V+基址寄存器BR中的内容 。

EA = (IR)*F + V + (BR)

例如：MOV EAX, -6[EDI*2][EBP]

4.7 基址加变址寻址

◆F 可为 1，2，4，8

◆当使用32位寄存器时

BR可以是 EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP 之一；

IR 可以是除ESP外的任一32位寄存器；

未带比例因子的寄存器是 BR；

当没有比例因子时，写在前面的寄存器是BR.

4.7 基址加变址寻址

特别说明：

当V中存在全局变量或标号时，用的段都是 DS。

在 VS2019中， (DS)=(SS)。

CS中的全局变量等同 DS段中的变量。

- 操作数的类型：

  若V为变量，则操作数类型为变量的类型；

  若V为常量，类型未知。

4.8 寻址方式综合举例

寻址方式有6种。
根据操作数的存放位置，寻址方式归为3类：

立即方式
寄存器方式
存储器方式
- 寄存器间接寻址
- 变址寻址
- 基址加变址寻址
- 直接寻址

4.8 寻址方式综合举例

1. 双操作数寻址方式的规定

一条指令的源操作数和目的操作数不能同时用存储器方式。

```
MOV   AX,  BX          MOV   BUF, 0
MOV   BUF, BX          MOV   BX,   0
MOV   BX,   BUF
```

~~MOV BUF, MSG~~

~~MOV BUF, [EBX]~~

~~MOV BUF, 5[EBX]~~

~~MOV BUF, 5 [EAX + EDI *4]~~

4.8 寻址方式综合举例

1. 双操作数寻址方式的规定

Question：

为什么Intel cpu 要规定源操作数和目的操作数不能同时用存储器方式呢？

换言之，规定的合理性体现在何处？

4.8 寻址方式综合举例

2. 操作数的类型

- 寄存器寻址方式中，操作数的类型由谁决定？

  寄存器

- 立即数没有类型

- 不含变量的存储器方式所表示的操作数类型

  未知

- 含变量的寻址方式对应的操作数类型 ？

  变量的类型

4.8 寻址方式综合举例

3. 双操作数的类型规定

- 双操作数中至少应有一个的类型是明确的；
- 若两个操作数的类型都明确，则两个的类型应相同。

(1) MOV BX, AX

(2) MOV BX, AL ✗ 两个操作数的类型不匹配

(3) MOV [EBX], 0 ✗ 两个操作数的类型不明确

属性定义算符 PTR

```
MOV BYTE PTR [EBX], 0
MOV WORD PTR [EBX], 0
MOV DWORD PTR[EBX], 0
```

4.8 寻址方式综合举例

思考题

从机器语言的角度，有六种寻址方式。

C语言程序是要编译成机器语言程序的。

C程序中存储单元的各种访问方式，

分别会翻译成怎样的寻址方式呢?

4.8 寻址方式综合举例


```
.DATA
X   DW   1122H

; 直接寻址方式
MOV   AX,  X

; 寄存器间接寻址方式
MOV   EBX,  OFFSET X
MOV   AX,     [EBX]

; 变址寻址方式
MOV   EBX,  0
MOV   AX,     X[EBX]

; 基址加变址寻址方式
MOV   EBX,  0
MOV   ESI,   0
MOV   AX,     X[EBX+ESI]
```

练习：

用四种寻址方式，

将 X 中的内容读入 到 AX 中

4.8 寻址方式综合举例

```
.DATA
BUF1  DB  20, 39, 50, -20, 0, -12
BUF2  DB  6  DUP(0)
.CODE
      ……
      MOV   CX,  6
      
LP:   
      DEC    CX
      JNZ    LP
      …..
```

练习：

将BUF1中的6个字节依次拷贝到以BUF2为首址的字节单元中。

算法思想？

采用什么寻址方式？

空中填什么语句？

4.8 寻址方式综合举例


```
.DATA
BUF1  DB  20, 39, 50, -20, 0, -12
BUF2  DB  6  DUP(0)
.CODE
      ……
      MOV   CX,  6
      MOV  EBX，  0
LP:   MOV  AL，BUF1 [EBX]
      MOV  BUF2[EBX],  AL
      INC    EBX
      DEC    CX
      JNZ    LP
      …..
```

采用变址寻址方式

用 EBX指明访问第几个元素

4.8 寻址方式综合举例

```
.DATA
BUF1  DB  20, 39, 50, -20, 0, -12
BUF2  DB  6  DUP(0)
.CODE
      ……
      MOV   CX,  6
      MOV  ESI，OFFSET BUF1
      MOV  EDI，OFFSET BUF2
LP:   MOV  AL，  [ESI]
      MOV [EDI],  AL
      INC    ESI
      INC    EDI
      DEC    CX
      JNZ    LP
      …..
```

采用寄存器间接寻址

ESI 指向源串

EDI 指向目的串

4.8 寻址方式综合举例


```
.DATA
BUF1  DB  20, 39, 50, -20, 0, -12
BUF2  DB  6  DUP(0)
.CODE
      ……
      MOV   CX,  6
      MOV  EAX，DWORD PTR  BUF1
      MOV  DWORD PTR  BUF2,  EAX
      MOV  AX，    WORD PTR  BUF1 +4
      MOV  WORD PTR  BUF2 +4,   AX
      DEC   CX
      JNZ   LP
      ……
```

采用直接寻址方式

先拷贝 4个字节

再拷贝剩下的2个字节

4.9 x86机器指令编码规则

从机器语言的角度，看待指令的组成部分；

判断写的汇编语句是否正确，可以看它能否编译成机器指令。

即各部分能否转换成指令中组成的要素。

- 计算机 是 0-1世界
- 编码与解码　　汇编与反汇编
- 指令组成的核心

  操作与操作数地址

4.9 x86机器指令编码规则

指令前缀	操作码	内存/寄存器操作数	索引寻址描述	地址偏移量	立即数

① 指令前缀(prefix，非必需，0个或多个字节)

② 操作码(opcode，必须，1字节～3字节)

③ 内存/寄存器操作数(ModR/M，非必需)

指明寻址方式

④ 索引寻址描述 (SIB，非必需)

指明：基址寄存器、变址寄存器、比例因子

⑤ 地址偏移量(Displacement，非必需)

⑥ 立即数(Immediate，非必需)

4.9 x86机器指令编码规则

指令前缀

种类	名称	二进制码	说明
LOCK	LOCK	F0H	让指令在执行时候先禁用数据线的复用特性，用在多核的处理器上，一般很少需要手动指定
REP	REPNE/REPNZ	F2H	用 CX(16 位下)或 ECX(32 位下)或 RCX(64 位下)作为指令是否重复执行的依据
	REP/REPE/REPZ	F3H	同上
Segment Override	CS	2EH	段重载(默认数据使用 DS 段)
	SS	36H	同上
	DS	3EH	同上
	ES	26H	同上
	FS	64H	同上
	GS	65H	同上
REX	64 位	40H-4FH	x86-64 位的指令前缀，见第 18 中的介绍
Operand size Override	Operand size Override	66H	用该前缀来区分：访问 32 位或 16 位操作数；也用来区分 128 位和 64 位操作数
Address Override	Address Override	67H	64 位下指定用 64 位还是 32 位寄存器作为索引

4.9 x86机器指令编码规则

操作码

- 指明了要进行的操作
- 指明操作数的类型（一般看最后一个二进制）

  0：字节操作； 1：字操作（32位指令中为双字操作）；

  有指令前缀 66H时，对字操作。
- 指明源操作数是寄存器寻址，还是目的操作数是寄存器寻址 （操作码的倒数第二个二进制位）

  1：目的操作数是寄存器寻址，0：源操作数寄存器寻址

  与[寻址方式字节]配合使用
- 在有些指令中（如立即数传送给寄存器），操作码含有寄存器的编码。
- 一般在opcode的编码中体现了源操作数是否为立即数。

4.9 x86机器指令编码规则

内存/寄存器操作数(ModR/M)

Mod （6-7 位）	Reg/Opcode（3-5 位）	R/M （0-2 位）

- Mod由2个二进制位组成，取值是00、01、10、11。
- Mod与为R/M配合使用，明确一个操作的获取方法。
- Reg/Opcode 确定另外一个寄存器寻址的寄存器编码

4.9 x86机器指令编码规则

内存/寄存器操作数(ModR/M)

Mod （6-7 位）	Reg/Opcode(3-5 位)	R/M （0-2 位）

R/M	Mod
[EAX], [ECX], [EDX], [EBX], [--][--], disp32, [ESI], [EDI]	00
[EAX]+disp8, [ECX]+disp8, [EDX]+disp8, [EBX]+disp8, [--][--]+disp8, [EBP]+disp8, [ESI]+disp8, [EDI]+disp8	01
[EAX]+disp32, [ECX]+disp32, [EDX]+disp32, [EBX]+disp32, [--][--]+disp32, [EBP]+disp32, [ESI]+disp32, [EDI]+disp32	10
EAX/AX/AL/MM0/XMM0, ECX/CX/CL/MM1/XMM1, EDX/DX/DL/MM2/XMM2, EBX/BX/BL/MM3/XMM3, ESP/SP/AH/MM4/XMM4, EBP/BP/CH/MM5/XMM5, ESI/SI/DH/MM6/XMM6, EDI/DI/BH/MM7/XMM7	11

Mod=00, R/M =000，表示 用 [EAX] 寻址

Mod=00，R/M=100, 表示用[--][--]，无位移量的基址加变址

在 SIB 字节指明基址/变址寄存器的编码

4.9 x86机器指令编码规则

内存/寄存器操作数(ModR/M)

Mod （6-7 位）	Reg/Opcode（3-5 位）	R/M （0-2 位）

000	001	010	011	100	101	110	111
AL	CL	DL	BL	AH	CH	DH	BH
AX	CX	DX	BX	SP	BP	SI	DI
EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
MMO	MM1	MM2	MM3	MM4	MM5	MM6	MM7
XMMO	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7

Reg/Opcode 的编码

- 同一编码有多个寄存器
- 用哪一个寄存器，取决于指令前缀和操作码中的编码

mov eax, [ebx] ；机器码是：8B 03 =》 00 000 011 [EBX]

mov [ebx], eax ；机器码是：89 03 =》 1000 1001 VS 1000 1011

操作码的倒数第二位： 1：OPD是寄存器， 0：OPS是寄存器

4.9 x86机器指令编码规则

内存/寄存器操作数(ModR/M)

Mod （6-7 位）	Reg/Opcode(3-5 位)	R/M （0-2 位）

000	001	010	011	100	101	110	111
AL	CL	DL	BL	AH	CH	DH	BH
AX	CX	DX	BX	SP	BP	SI	DI
EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
MMO	MM1	MM2	MM3	MM4	MM5	MM6	MM7
XMMO	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7

mov eax, [ebx] ；机器码是：8B 03 =》 00 000 011 [EBX]

mov [ebx], eax ；机器码是：89 03 =》 1000 1001 VS 1000 1011

mov ax, [ebx] ；机器码是：66 8B 03 指令前缀，字类型操作

mov al, [ebx] ；机器码是：8A 03 => 1000 1010

32位指令 op码的最后一位为 0，字节操作；为1；双字操作

有前缀 66，为 1：字操作

4.9 x86机器指令编码规则

索引寻址描述 SIB

Scale （6-7 位）	Index(3-5 位)	Base （0-2 位）

000	001	010	011	100	101	110	111
AL	CL	DL	BL	AH	CH	DH	BH
AX	CX	DX	BX	SP	BP	SI	DI
EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7

- 与内存/寄存器操作数(ModR/M) 配合使用
- 在32位指令系统中，基址寄存器与变址寄存器都是 32 位的
- Base =000，表示基址寄存器为 EAX

```
mov   eax,[ebx+ecx*4]      ; 机器码是 8B 04 8B
 10 001 011 ：比例因子10（4），变址寄存器 001 ，即ECX
                基址寄存器 011，即 EBX
04 ： 00 000 100 =》  00 /100 对应的就是 [-][-]
```

4.9 x86机器指令编码规则

地址偏移量(Displacement)

- 由1、2 或 4 个字节组成，分别对应8位、16位或32位的偏移量;
- 数据按照小端顺序存放，即数据的低位存放在小地址单元中。

立即数(Immediate)

- 对应立即寻址方式
- 占1、2 或 4个字节, 按照小端顺序存放。

总结

6种寻址方式的使用格式、语法规定；

6种寻址方式地址表示的含义及应用；

注意：共同点与差异

机器指令中的地址表达

汇编语言源程序中的地址表达

C语言中的地址表达