第2章 IA-32处理器基本功能

2.1 IA-32处理器简介

2.2 通用寄存器及使用

2.3 标志寄存器及使用

2.4 段寄存器

2.5 寻址方式

2.6 指令指针寄存器和简单控制转移

2.7 堆栈和堆栈操作

2.4 段寄存器

2.4.1 存储器分段

2.4.2 逻辑地址

2.4.3 段寄存器

2.4.1 存储器分段

➢物理地址空间

✓**CPU**能够通过其总线直接寻址访问的存储器被称为**内存**

✓每一个字节存储单元有一个唯一的地址，称之为**物理地址**

✓**CPU**的地址线数量决定了可产生的最大物理地址

n根地址线，可形成的最大物理地址是2^n-1

✓所有可形成的物理地址的集合被称为**物理地址空间**

Intel8086有20根地址线，物理地址的范围是0到FFFFF

Intel80386有32根地址线，物理地址的范围是0到FFFFFFFF

物理地址空间大小不等于实际安装的物理内存大小

2.4.1 存储器分段

➢存储器分段

✓ 为了有效地管理存储器，常常把地址空间划分为若干逻辑段。对应存储空间被划分为若干存储段。逻辑段和存储段是一致的。

✓一般说来，运行着的程序在存储器中映像有三部分组成：

其一是代码，代码是要执行的指令序列

其二是数据，数据是要处理加工的内容

其三是堆栈，堆栈是按“先进后出”规则存取的区域

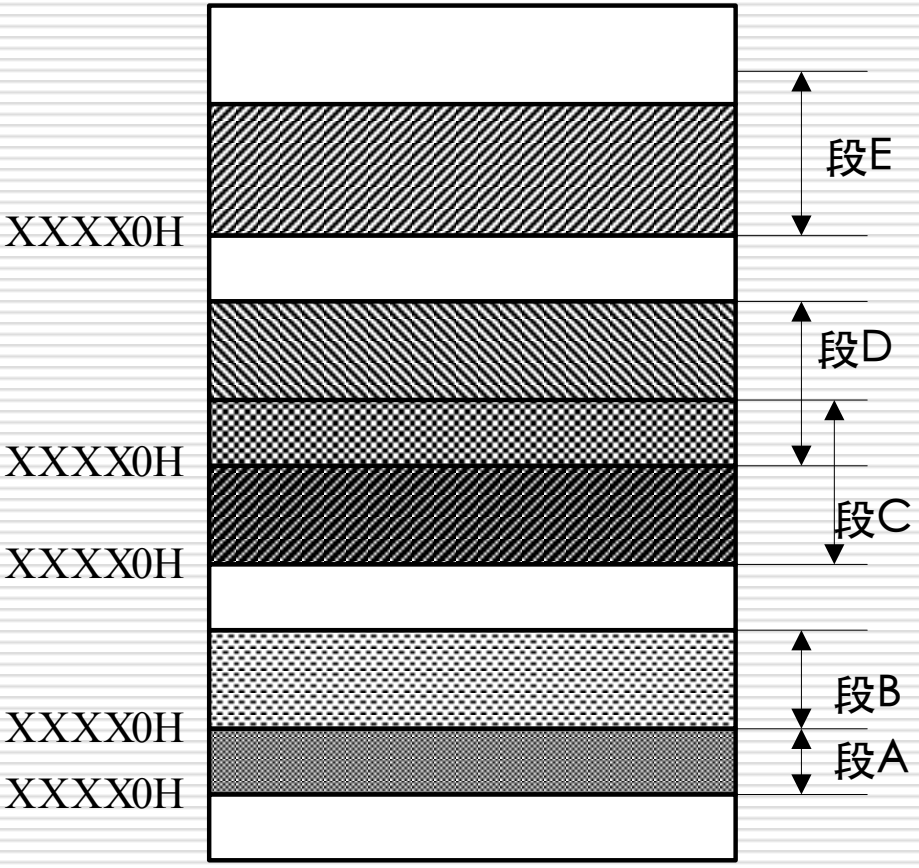
✓ 通常，代码、数据和堆栈分别占用不同的存储器段，相应的段也就被称为**代码段、数据段和堆栈段**。

2.4.1 存储器分段

➢存储器分段

✓可以按需要进行段的划分。逻辑段与逻辑段可以相连，也可以不相连，还可以部分重叠。

✓代码、数据和堆栈可以在同一个逻辑段内，占用不同区域。



2.4.2 逻辑地址

➢逻辑地址

✓在分段之后，程序中使用的某个存储单元总是属于某个段。所以，可以采用某某段某某单元的方式来表示存储单元。

✓在程序中用于表示存储单元的地址被称为**逻辑地址**。

✓由于采用分段存储管理方式，程序中使用的逻辑地址是二维的，第一维给出某某段，第二维给出段内的某某单元。

如何表示某某段？如何表示段内的某某单元？

2.4.2 逻辑地址

➢逻辑地址

✓二维的逻辑地址可以表示为：

段号:段内地址

✓ 存储单元的物理地址与所在段的起始地址的差值被称为**段内偏移**，简称为**偏移**。段内地址就是段内偏移，也就是偏移。于是，二维的逻辑地址可以表示为：

段号:偏移

2.4.2 逻辑地址

➢逻辑地址

✓二维的逻辑地址：

段号:偏移

✓ 在实方式和保护方式下，都通过偏移指定段内的某某单元。在实方式下，段号是段值；在保护方式下，段号则是段选择子。

在第6章介绍段值
在第9章介绍段选择子

2.4.2 逻辑地址

➢转换成物理地址

✓获得物理地址

物理地址 = 段起始地址 + 偏移

✓ 在实方式下，由段值可以得到段起始地址；在保护方式下，根据选择子可以得到段起始地址。总之，由段号可以得到段起始地址。二维的逻辑地址可以转换成一维的物理地址。逻辑地址转换为物理地址的过程可归纳为：由段号得到段起始地址，再加上偏移。

保护方式下，物理地址是**32**位，段起始地址是**32**位，偏移是**32**位；
在实方式下，物理地址是**20**位，段起始地址是**20**位，偏移是**16**位。

2.4.2 逻辑地址

➢转换成物理地址

✓获得物理地址

物理地址 = 段起始地址 + 偏移

✓ 如果整个程序只有一个段，则二维的逻辑地址退化成一维。由于段起始地址完全相同，偏移就决定一切。

VC2010环境中就是这样。
某种意义上，嵌入汇编只考虑偏移。

2.4.3 段寄存器

➢段寄存器

✓在一个已确定的段内，只需通过偏移便可指定要访问的存储单元。程序中绝大部分涉及存储器访问的指令都只给出偏移。

✓逻辑地址中的段号（段值或者段选择子）存放在哪里呢？答案是，当前使用段的段号存放在段寄存器（**Segment Registers**）中。

✓段寄存器是**16**位的。在实方式下，用于存放**16**位的段值；在保护方式下，用于存放**16**位的段选择子。

2.4.3 段寄存器

➢段寄存器

✓Intel 8086处理器有四个段寄存器

CS: 代码段(Code Segment)寄存器

SS: 堆栈段(Stack Segment)寄存器

DS: 数据段(Data Segment)寄存器

ES: 附加段(Extra Segment)寄存器

✓从80386处理器开始，增加了两个段寄存器

FS: 附加段寄存器

GS: 附加段寄存器

✓ **CS**指定当前代码段，**SS**指定当前堆栈段

✓ 一般情况下，**DS**指定当前数据段

✓ 附加段寄存器**ES**、**FS**、**GS**也可用于指定数据段

2.4.3 段寄存器

➢段寄存器

✓在访问存储单元时，**CPU**先根据对应的段寄存器得到段起始地址，再加上相应的偏移，形成存储单元的物理地址。

✓如果程序的代码段、数据段、堆栈段占用同一个存储段，那么代码段寄存器**CS**、数据段寄存器**DS**和堆栈段寄存器**SS**等指定同一个存储段，给出相同的段起始地址。

✓如果由段寄存器给出的段起始地址是**0**，那么偏移就相当于物理地址。

2.5 寻址方式

✓把表示指令中操作数所在的方法称为**寻址方式**

✓CPU常用的寻址方式可分为三大类：立即寻址、寄存器寻址和存储器寻址，此外还有固定寻址和I/O端口寻址等

2.5 寻址方式

2.5.1 立即寻址方式和寄存器寻址方式

2.5.2 32位的存储器寻址方式

2.5.3 取有效地址指令

2.5.1 立即寻址方式和寄存器寻址方式

➢立即寻址方式

✓操作数本身就包含在指令中，直接作为指令的一部分给出。把这种寻址方式称为**立即寻址方式**。

✓把这样的操作数称为**立即数**。

```
MOV   EAX, 12345678H      ;给EAX寄存器赋初值
ADD   BX, 1234H           ;给BX寄存器加上值1234H
SUB   CL, 2               ;从CL寄存器减去值2

MOV   EDX, 1              ;源操作数是32位
MOV   DX, 1               ;源操作数是16位
MOV   DL, 1               ;源操作数是8位
```

2.5.1 立即寻址方式和寄存器寻址方式

➢立即寻址方式

✓立即数作为指令的一部分，跟在操作码后存放在代码段。

✓如果立即数由多个字节构成，那么在作为指令的一部分存储时，也采用“高高低低”规则。

✓只有源操作数才可采用立即寻址方式，目的操作数不能采用立即寻址方式。

✓由于立即寻址方式的操作数是立即数，包含在指令中，所以执行指令时，不需要再到存储器中去取该操作数了。

2.5.1 立即寻址方式和寄存器寻址方式

➢寄存器寻址方式

✓操作数在**CPU**内部的寄存器中，指令中指定寄存器。把这种寻址方式称为**寄存器寻址方式**。

✓可以是**8**个**32**位的通用寄存器

✓可以是**8**个**16**位的通用寄存器

✓可以是**8**个**8**位的通用寄存器

```
MOV   EBP, ESP      ;把ESP之值送到EBP
ADD   EAX, EDX      ;把EAX之值与EDX之值相加，结果送到EAX
SUB   DI, BX        ;把DI之值减去BX之值，结果送到DI
XCHG  AH, DH        ;交换AH与DH之值
```

2.5.1 立即寻址方式和寄存器寻址方式

➢寄存器寻址方式

✓由于操作数在寄存器中，不需要通过访问存储器来取得操作数，所以采用寄存器寻址方式的指令执行速度较快

2.5.2 32位的存储器寻址方式

➢32位的存储器寻址方式

✓当指令的操作数在存储单元时，指定存储单元就指定了操作数。

✓在某个段内，通过偏移就能够指定存储单元。一般情况下访问存储单元的指令只需要给出存储单元的偏移。

✓**存储器寻址方式**指，给出存储单元偏移的方式。

✓采用**32**位的存储器寻址方式，能够给出**32**位的偏移。

✓常常把要访问的存储单元的段内偏移称为**有效地址EA(Effective Address)**。在**32**位存储器寻址方式下，存储单元的有效地址可达**32**位。

2.5.2 32位的存储器寻址方式

➢32位的存储器寻址方式

✓为了灵活方便地访问存储器，**IA-32**系列**CPU**提供了多种表示存储单元偏移的方式。换句话说，有多种存储器寻址方式。

- 直接寻址
- 寄存器间接
- 寄存器相对
- 基址加变址
- 通用

2.5.2 32位的存储器寻址方式

➢直接寻址方式

✓操作数在存储器中，指令直接包含操作数所在存储单元的有效地址。把这种寻址方式称为**直接寻址方式**。

```
MOV   ECX, [95480H]      ;源操作数采用直接寻址
MOV   [9547CH], DX       ;目的操作数采用直接寻址
ADD   BL, [95478H]       ;源操作数采用直接寻址
```

立即寻址和直接寻址有本质区别！

直接寻址的地址要放在方括号中，在源程序中，往往用变量名表示。

2.5.2 32位的存储器寻址方式

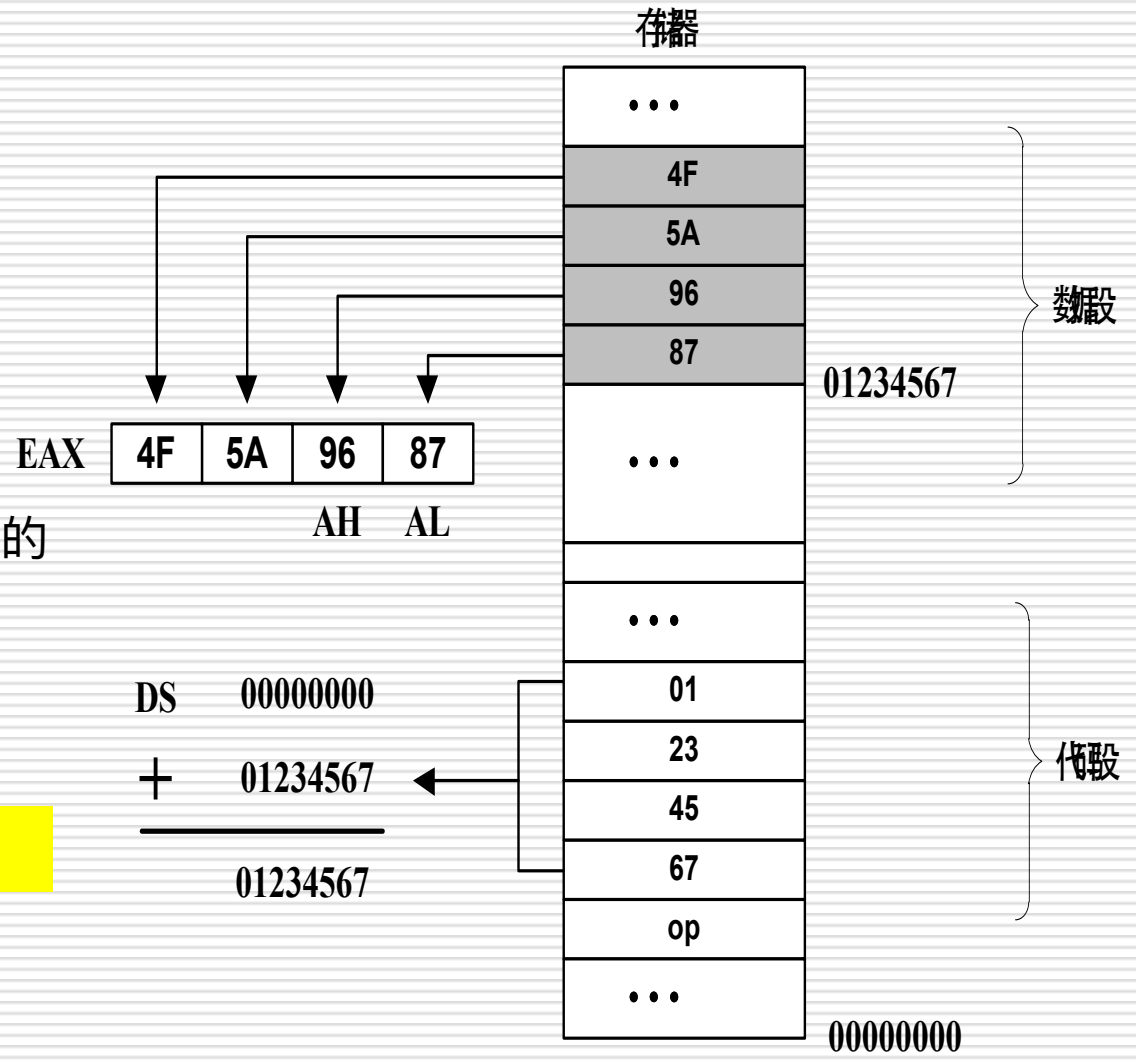
➢直接寻址方式

✓示例

假设数据段和代码段重叠，段起始地址都是**0**。

有效地址为**01234567H**的双字存储单元中内容是**4F5A9687H**。

MOV EAX，[01234567H]



2.5.2 32位的存储器寻址方式

➢寄存器间接寻址方式

✓操作数在存储器中，由八个**32**位的通用寄存器之一给出操作数所在存储单元的有效地址。把这种通过寄存器间接给出存储单元有效地址的方式称为**寄存器间接寻址方式**。

```
MOV   EAX, [ESI]    ;源操作数寄存器间接寻址，ESI给出有效地址
MOV   [EDI], CL     ;目的操作数寄存器间接寻址，EDI给出有效地址
SUB   DX, [EBX]     ;源操作数寄存器间接寻址，EBX给出有效地址
```

2.5.2 32位的存储器寻址方式

➢寄存器间接寻址方式

寄存器间接寻址与寄存器寻址有本质区别!

寄存器间接寻址的寄存器出现在方括号中。

```
MOV   [ESI], EAX      ;目的操作数采用寄存器间接寻址方式
MOV   ESI, EAX        ;目的操作数采用寄存器寻址方式
```

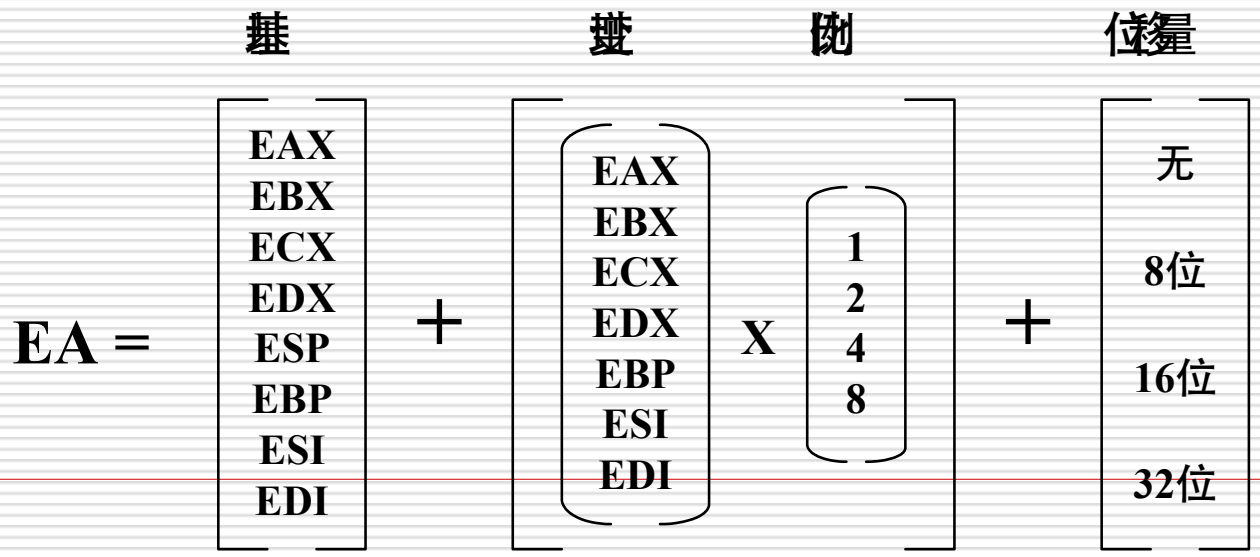
寄存器间接寻址方式中，给出操作数所在存储单元有效地址的寄存器，相当于**C**语言中的指针变量，它含有要访问存储单元的地址。

2.5.2 32位的存储器寻址方式

➢32位存储器寻址方式的通用表示

✓存储单元的有效地址可以由三部分内容相加构成：

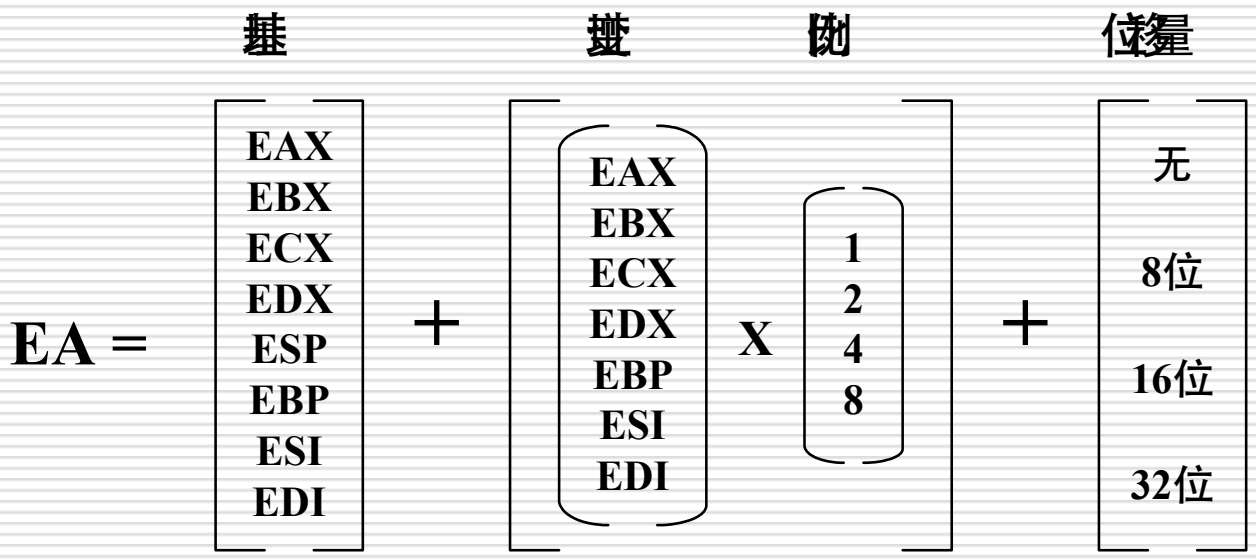
- 一个32位的基地址寄存器
- 一个可乘上比例因子1、2、4或8的32位变址寄存器
- 一个8位、16位或32位的位移量
- 这三部分可省去任意的两部分



2.5.2 32位的存储器寻址方式

➢32位存储器寻址方式的通用表示

✓支持灵活的32位有效地址的存储器寻址方式



补码表示，
如8位或16位，
计算时被扩展成32位

8个32位通用寄存器都可以作为基址寄存器；

除ESP寄存器外，其他7个通用寄存器都可以作为变址寄存器。

2.5.2 32位的存储器寻址方式

➢32位存储器寻址方式的通用表示

✓示例

```
MOV   EAX, [EBX+12H]      ;源操作数有效地址是EBX值加上12H
MOV   [ESI-4], AL         ;目的操作数有效地址是ESI值减去4
ADD   DX, [ECX+5328H]     ;源操作数有效地址是ECX值加上5328H
```

寄存器相对寻址方式

2.5.2 32位的存储器寻址方式

➢32位存储器寻址方式的通用表示

✓示例

```
MOV   EAX, [EBX+ESI]      ;源操作数有效地址是EBX值加上ESI值
SUB   [ECX+EDI], AL       ;目的操作数有效地址是ECX值加上EDI值
XCHG  [EBX+ESI], DX       ;目的操作数有效地址是EBX值加上ESI值
```

基址加变址寻址方式

2.5.2 32位的存储器寻址方式

➢32位存储器寻址方式的通用表示

✓示例

```
MOV   EAX, [ECX+EBX*4]      ;EBX作为变址寄存器，放大因子是4
MOV   [EAX+ECX*2], DL       ;ECX作为变址寄存器，放大因子是2
ADD   EAX, [EBX+ESI*8]      ;ESI作为变址寄存器，放大因子是8
SUB   ECX, [EDX+EAX-4]      ;EAX作为变址寄存器，放大因子是1
MOV   EBX, [EDI+EAX*4+300H] ;EAX作为变址寄存器，放大因子是4
```

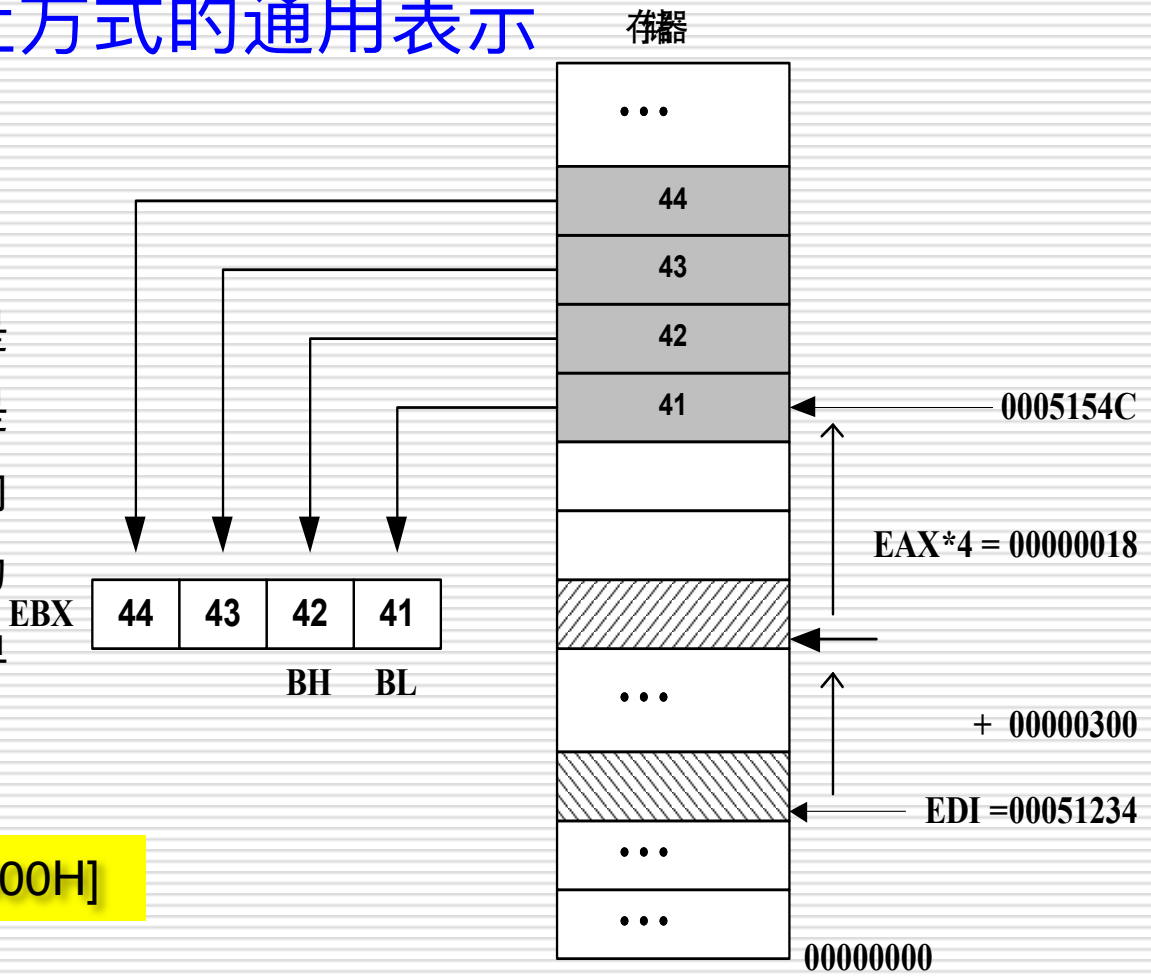
基址加带放大因子的变址寻址方式

2.5.2 32位的存储器寻址方式

➢32位存储器寻址方式的通用表示

✓示例

假设由**DS**得段起始地址是**0**，寄存器**EDI**的内容是**51234H**，寄存器**EAX**的内容是**6**，并且有效地址为**0005154CH**的双字存储单元的内容是**44434241H**。




```
MOV   EBX, [EDI+EAX*4+300H]
```

2.5.2 32位的存储器寻址方式

➢演示程序dp27

```
#include <stdio.h>
int   vari = 0x12345678;  //定义整型变量。设有效地址为x
char  buff[] = "ABCDE";   //定义字符数组。设首元素有效地址为y
int  main( )
{   int  dv1, dv2, dv3, dv4;    //定义4个整型变量
    _asm {    //嵌入汇编
    。。。。。。
    }
    printf("dv1=%08XH\n",dv1);   //显示为dv1=12345678H
    printf("dv2=%08XH\n",dv2);   //显示为dv2=41123456H
    printf("dv3=%08XH\n",dv3);   //显示为dv3=41121234H
    printf("dv4=%08XH\n",dv4);   //显示为dv4=41121244H
    return  0;
}
```

演示32位存储器寻址方式使用，
演示字节存储单元与双字存储单元关系

2.5.2 32位的存储器寻址方式

演示32位存储器寻址方式使用，
演示字节存储单元与双字存储单元关系

➢演示程序**dp27**

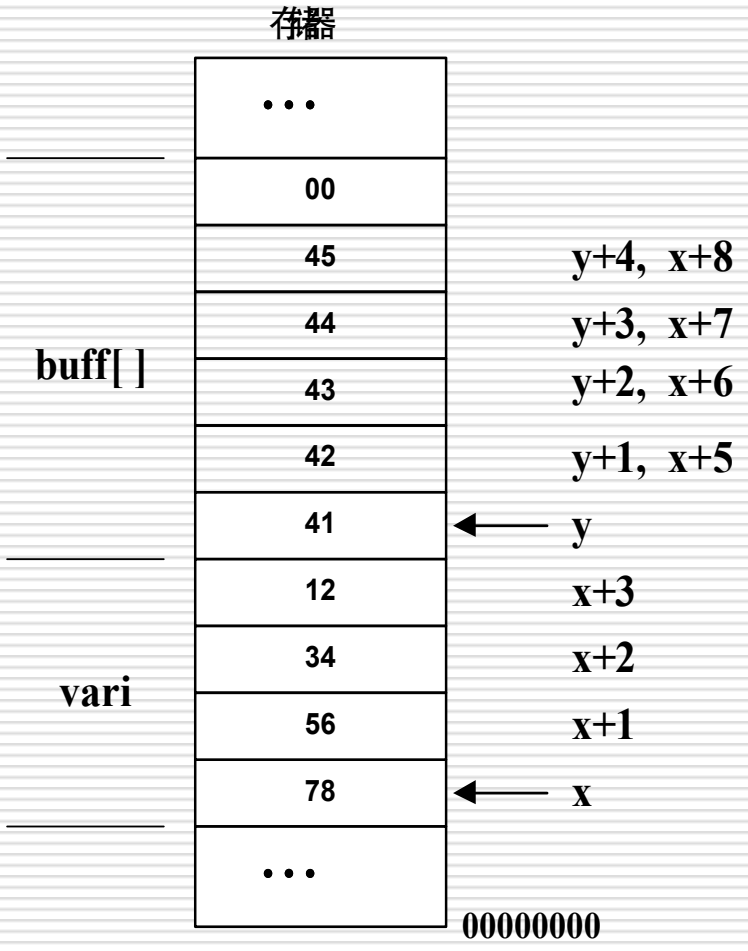
```
_asm {
   LEA   EBX, vari      //把变量vari的有效地址x送到EBX
   MOV   EAX, [EBX]     //把有效地址为x的双字(12345678H)送到EAX
   MOV   dv1, EAX
   MOV   EAX, [EBX+1]   //把有效地址x+1的双字(41123456H)送到EAX
   MOV   dv2, EAX
   ;
   MOV   ECX, 2
   MOV   AX, [EBX+ECX]  //把有效地址为x+2的字(1234H)送到AX
   MOV   dv3, EAX
   ;
   MOV   AL, [EBX+ECX*2+3] //把有效地址为x+7的字节(44H)送到AL
   MOV   dv4, EAX
}
```

ASM

2.5.2 32位的存储器寻址方式

➢演示程序dp27

```
_asm {
   LEA   EBX, vari
   MOV   EAX, [EBX]
   MOV   dv1, EAX
   MOV   EAX, [EBX+1]
   MOV   dv2, EAX
   ;
   MOV   ECX, 2
   MOV   AX, [EBX+ECX]
   MOV   dv3, EAX
   ;
   MOV   AL, [EBX+ECX*2+3]
   MOV   dv4, EAX
}
```



2.5.2 32位的存储器寻址方式

➢关于存储器寻址方式的说明

✓如果指令的操作数允许是存储器操作数，那么各种存储器寻址方式都适用。

✓存储器操作数的尺寸可以是字节、字或者双字。

✓在某条具体的指令中，如果有存储器操作数，那么其尺寸是确定的。在大多数情况下，存储器操作数的尺寸是一目了然的，因为通常要求一条指令中的多个操作数的尺寸一致，所以指令中的寄存器操作数的尺寸就决定了存储器操作数的尺寸。在少数情况下，需要显式地指定存储器操作数的尺寸。

2.5.2 32位的存储器寻址方式

➢演示程序**dp28**

演示显式地标明存储器操作数的尺寸

```
#include  <stdio.h>
int  var1 = 0x33333333, var2 = 0x44444444, var3 = 0x55555555;
int  bufi[3] = {0x66666666, 0x77777777, 0x88888888};
int  main()
{
    _asm {    //嵌入汇编代码
       MOV   var1, 9              //双字存储单元
       MOV   WORD  PTR var2, 9     //字存储单元
       MOV   BYTE  PTR var3, 9     //字节存储单元
    }
    printf("%08XH\n",var1);        //显示为00000009H
    printf("%08XH\n",var2);        //显示为44440009H
    printf("%08XH\n",var3);        //显示为55555509H
    ;
```

2.5.2 32位的存储器寻址方式

➢演示程序**dp28**

演示显式地标明存储器操作数的尺寸

```
    _asm {    //嵌入汇编代码
       LEA   EBX, bufi              //把bufi的有效地址送到EBX
       MOV   DWORD PTR [EBX], 5     //双字存储单元
       MOV   WORD  PTR [EBX+4], 5   //字存储单元
       MOV   BYTE  PTR [EBX+8], 5   //字节存储单元
    }
    printf("%08XH\n", bufi[0]);      //显示为00000005H
    printf("%08XH\n", bufi[1]);      //显示为77770005H
    printf("%08XH\n", bufi[2]);      //显示为88888805H
    return  0;
}
```

2.5.2 32位的存储器寻址方式

➢关于存储器寻址方式的说明

✓如果基址寄存器不是**EBP**或者**ESP**，那么缺省引用的段寄存器是**DS**；如果基址寄存器是**EBP**或者**ESP**，那么缺省引用的段寄存器是**SS**。当**EBP**作为变址寄存器使用（**ESP**不能作为变址寄存器使用）时，缺省引用的段寄存器仍然是**DS**。

✓无论存储器寻址方式简单或者复杂，如果由基址寄存器、带比例因子的变址寄存器和位移量这三部分相加所得超过**32**位，那么有效地址仅为低**32**位。

2.5.3 取有效地址指令

- 取有效地址指令
- 取有效地址指令的应用

2.5.3 取有效地址指令

➢取有效地址指令

✓取有效地址（Load Effective Address）指令的一般格式

```
LEA   REG，OPRD
```

- 该指令把操作数**OPRD**的有效地址传送到寄存器**REG**。

注意：
源操作数**OPRD**必须是一个存储器操作数，
目的操作数**REG**必须是一个**16**位或者**32**位的通用寄存器。

2.5.3 取有效地址指令

➢取有效地址指令

✓使用举例

```
MOV   EDI, 51234H              ;EDI=00051234H
MOV   EAX, 6                   ;EAX=00000006H
LEA   ESI, [EDI+EAX]           ;ESI=0005123AH
LEA   ECX, [EAX*4]             ;ECX=00000018H
LEA   EBX, [EDI+EAX*4+300H]    ;EBX=0005154CH
```

LEA指令与**MOV**指令有本质上的区别!

2.5.3 取有效地址指令

➢演示程序**dp29**

```
#include  <stdio.h>
char  chx, chy;              //全局字符变量
int  main( )
{
   char  *p1, *p2;           //两字符型指针变量
   //嵌入汇编代码之一
   _asm {
      LEA   EAX, chx         //取变量chx的存储单元有效地址
      MOV   p1, EAX          //送到指针变量p1
      LEA   EAX, chy         //取变量chy的存储单元有效地址
      MOV   p2, EAX          //送到指针变量p2
   }
```

高级语言中的指针本质

p1 = &chx;

p2 = &chy;

2.5.3 取有效地址指令

➢演示程序**dp29**

```
    printf("Input:");          //提示
    scanf("%c", p1);           //键盘输入一个字符
    //嵌入汇编代码之二
    _asm {
        MOV   ESI, p1          //取回变量chx的有效地址
        MOV   EDI, p2          //取回变量chy的有效地址
        MOV   AL, [ESI]        //取变量chx之值
        MOV   [EDI], AL        //送到变量chy中
    }
    printf("ASCII:%02XH\n", *p2);  //显示之
    return  0;
}
```

temp = *p1;
*p2 = temp;

寄存器作为指针

2.5.3 取有效地址指令

➢演示程序**dp210**

```
#include <stdio.h>
int     iarr[5] = {55, 87, -23, 89, 126};     //整型数组
double  darr[5] = {9.8, 2.77, 3.1415926, 1.414, 1.73278};
                                        //双精度浮点数组    int  main( )
{   int     ival;           //整型变量
    double  dval;           //双精度浮点
    //嵌入汇编
    _asm {
        。。。。。。
    }
    printf("iVAL=%d\n",ival);     //显示为iVAL=89
    printf("dVAL=%.8f\n",dval);   //显示为dVAL=3.14159260
    return  0;
}
```

演示**32**位寻址方式

演示取有效地址指令

2.5.3 取有效地址指令

➢演示程序**dp210**

演示32位寻址方式
演示取有效地址指令

```
_asm {
    LEA   EBX, iarr        //把整型数组首元素的有效地址送EBX
    MOV   ECX, 3
    MOV   EDX, [EBX+ECX*4]  //取出iarr的第4个元素
    MOV   ival, EDX
    ;
    LEA   ESI, darr        //把浮点数组首元素的有效地址送ESI
    LEA   EDI, dval        //把变量dval的有效地址送EDI
    MOV   ECX, 2
    MOV   EAX, [ESI+ECX*8]    //取darr的第3个元素的低双字
    MOV   EDX, [ESI+ECX*8+4]  //取darr的第3个元素的高双字
    MOV   [EDI], EAX          //保存低双字
    MOV   [EDI+4], EDX        //保存高双字
}
```

2.5.3 取有效地址指令

➢取有效地址指令的应用

通过高级语言源程序的目标代码来说明LEA指令的妙用。

```
int  _fastcall  cf211(int x, int y)      //由寄存器传参数
{
    return ( 2 * x + 5 * y + 100 );
}
```

编译器生成的目标代码（速度最大化）

```
    ;函数cf211目标代码（使速度最大化）
    lea   eax, DWORD PTR [edx+edx*4+100]  ;DWORD PTR表示双字存储单元
    lea   eax, DWORD PTR [eax+ecx*2]
    ret
```

ECX传递x
EDX传递y

2.5.3 取有效地址指令

➢取有效地址指令的应用

通过高级语言源程序的目标代码来说明LEA指令的妙用。

```
int  _fastcall  cf212(int x, int y)    //由寄存器传参数
{
    return ( 3 * x + 7 * y + 200 );
}
```

编译器生成的目标代码（速度最大化）

```
    ;函数cf212目标代码（使速度最大化）
    lea   eax, DWORD PTR [ecx+ecx*2]    ; eax=3*x
    lea   ecx, DWORD PTR [edx*8]        ; ecx=8*y
    sub   ecx, edx                      ; ecx=7*y
    lea   eax, DWORD PTR [eax+ecx+200]  ; eax=3*x+7*y+200
    ret
```

ECX传递x
EDX传递y