目录

第 4 章 寻址方式

通常，一条带有操作数的指令要指明两个问题，一是进行什么操作，二是用什么方式寻找操作数的存放地址。寻找操作数存放地址的方式被称为寻址方式，它在指令中所指出了计算操作数地址的方法。熟悉并灵活地应用机器所采用的各种寻址方式，对汇编语言程序设计是至关重要的。本章主要介绍 x86-32 指令系统中的寻址方式。在学习寻址方式时，应与 C 语言中的简单类型的变量、数组、指针、结构变量的访问方式进行类比，理解它们之间的共同点和差异点，以便在编写程序时更灵活地对变量所在空间的进行访问。

4．1 寻址方式概述

对一条指令，关注的焦点主要有以下几个方面。

(1) 执行什么操作？

由指令助记符回答这一问题。如 mov 表示数据传送(将 mov 称为数据拷贝更恰当，源操作数，即被传送的数保持不变)，add 表示加法，mul 表示乘法等等。

(2) 操作数在哪？

操作数的存放位置
- CPU 内的寄存器中，用寄存器名是一个符号地址
- 内存中，内存存储单元的地址表达
- IO 设备的端口中

当操作数在内存时，就要关注内存单元的物理地址是如何形成的，即使用的段寄存器是什么，在段内的偏移地址（亦称为有效地址）是多少。

(3) 操作数的类型是什么？

当操作数在 CPU 的寄存器中时，寄存器的名字就决定了其类型。例如 eax 是 32 位的寄存器，是双字类型；ax 是 16 位的寄存器，是字类型；ah 是 8 位的寄存器，是字节类型。

当操作数在内存中时，给定一个存储单元的地址后，还要进一步明确从该地址开始，是取一个字节数据、一个字数据还是一个双字数据。对于含有变量的地址表达式，其类型为变量的类型。使用 db、dw、dd 定义的变量的类型分别为字节、字、双字。地址类型决定了从同一地址开始所访问的数据的长度（字节数）。

对于数值常量，如数值 0，本身是没有类型的，因为一个字节的 0、两个字节的 0,四个字节的 0 的写法没有任何差别。

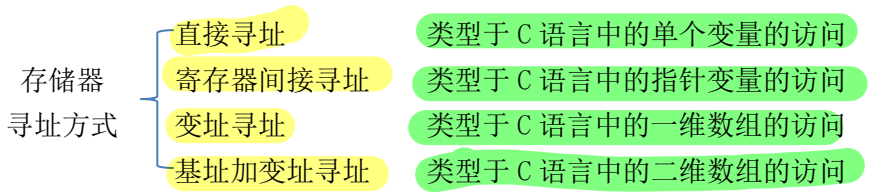
(4) 操作数寻址方式有哪几种？

操作数寻址方式共有六种，也可以分成三类。

① 立即寻址，操作数是一个常数，该数就编码在指令中；

② 寄存器寻址，操作数在 CPU 的一个寄存器中，该寄存器的编码出现在指令中；

③ 存储器寻址方式，操作数在内存的一个单元中，它有四种寻址方式。



（5）双操作数的指令格式及一般规则

当一条指令带有两个操作数时，称其为双操作数指令。其格式为：

操作符　opd，ops

其中 opd 表示目的操作数的寻址方式，ops 表示源操作数的寻址方式。目的操作数的地址一般也用来存放结果。

例如,add eax，edx 其功能为(eax)+(edx)->eax ,即寄存器 eax 和 edx 中的值相加，加的结果存放在 eax 中。

在双操作数的指令中，一般有如下规则：

① 源操作数和目的操作数不能同时使用存储器寻址方式；

② 立即寻址不能作为目的操作数的寻址方式，即 opd 不能是一个常数值；

③ 若 opd、ops 的数据类型都明确，则两者必须相同；

当 opd 或 ops 为寄存器时，其类型是明确的；当 opd 或 ops 中含变量时，其类型就是变量定义时的类型。

④ opd、ops 的数据类型不能都不明确；

若两个都不明确，可以使用 byte ptr、word ptr、dword ptr 等使其明确。这些类型运算符也可以用于强制类型转换，即将一个明确的地址类型转换成指定的类型。

当 opd、ops 中只有一个操作数的类型是明确的，则另一个操作数自动的按明确的类型转换。

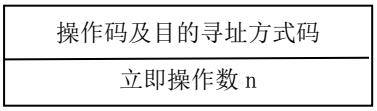
下面将详细介绍六种寻址方式。

4．2 立即寻址

立即寻址方式所提供的操作数是紧跟在指令操作码后面的一个可用 8 位、16 位或 32 位二进制补码表示的有符号数，构成了指令的一部分，位于代码段中。也就是说，操作数的存放地址就是指令操作码的下一单元，计算 PA 时使用的段寄存器是 CS，而 EA 来自指令指示器 IP/EIP 中的内容。

在指令中的使用格式：n

功能：指令最后一部分单元中的内容为操作数，即



其中，n 也称为立即操作数，可以占用 1 个、2 个、4 个字节的存储单元，占用的字节数由指令指定的操作数类型确定。在指令语句中，立即数 n 只能是常数或结果为确定值的表达式，且只能作源操作数。

【例 4.1】　mov　eax，12H

其中，目的操作数使用的是寄存器寻址方式，执行后(eax)=00000012H。该指令机器代码为 B8 12 00 00 00。

【例 4.2】 mov eax，-12H

执行后(eax)中存放的内容是 FFFFFFEE。为了表明这是一个十六进制数，在编写指令和表示某一单元中的内容时，以字母 A-F 开头的十六进制数前面加一个 0，以便区分它不是一个标识符（即程序中定义的变量、标号名、子程序名等）。同时，在数结束处加 H 表示十六进制。

根据上述约定，(eax)=0FFFFFFEEH。这是-12H 的 32 位的补码表示。该指令机器代码为 B8 EE FF FF FF。

mov eax,-12H 的等价语句是 mov eax, 0FFFFFFEEH。两者的机器码完全相同。

【例 4.3】 mov ax，-12H

该指令的机器码是：66 B8 EE FF ，执行后(ax)=0FFEEH。

【例 4.4】 mov al，-12H

该指令的机器码是：B0 EE ，执行后(al)=0EEH。

对比例 4.2-4.4 的指令的机器码，可以看到，对于立即数-12H，在机器码中出现的编码并不相同。-12H 是一个无类型的数，本身并未指令该数占几个字节。在这三条指令中，目的操作数地址为 eax、ax、al，它们的类型是明确的，分别是 32 位、16 位和 8 位。因此对-12H 按对应的长度转换为其补码表示。

注意：

① 在编写汇编源程序时，可以用数值表达式。例如： mov eax, 3*4+5*6 。不要以为这条指令中的源操作数含有乘法、加法运算，在编译后对执行程序再反汇编，看到的语句是 mov eax,2AH，机器码为 B8 2A 00 00 00。换句话说，在编译的时候已将数值表达式转换成了一个值，在机器指令中只有一个数值。

② 可以用一个字符或者字符串来作为源操作数。

mov al, '1' 等价于 mov al, 31H

mov ax,'12' 等价于 mov ax, 3132H

mov eax,'1234'等价于 mov eax, 31323334H

采用一个字符或者一个字符串的表达方法，比写成一个数值更直观。例如，设(al)中是一个大写字母的 ASCII， 直接使用指令 sub al, 'A',就可以知道该字母是字母表中的第（al）个字母。' A'是字母表中的第 0 个字母， 'Z'是第 25 个字母。

③ 写汇编源程序时，要注意数值的大小是否在另一个操作数类型所限定的范围内。例如 mov al，1234H。Visual Studio 2019 中给出的出错信息是：error A2070: invalid instruction operands。注意，不同的编译器给出的错误提示是不同的，有些更直接的表述为 out of range。

在 Visual Studio 2019 中编写如下 C 语言语句：

char c = 0x1234;

编译程序虽然给出了警告信息： warning C4305: "初始化": 从"int"到"char"截断，但是依然可生成执行程序。反汇编后对应的语句是： mov byte ptr [c],34H。从机器语句的角度来看，立即数占的字节数一定与指令中指明的数据类型相同。

④ 立即寻址方式主要用来给寄存器或存储器赋初值，也可以与寄存器操作数或存储器操作数进行算术逻辑运算。立即寻址方式不仅能简化数据的存取，使指令的书写直观、清晰，而且由于它是随同指令码一起被预取到 CPU 内部的，不需要再单独进行存储器操作，因此执行速度快。

4．3 寄存器寻址

寄存器寻址方式用某一个寄存器作为操作数的存放地址，操作数在指令指明的这个寄存器中。寄存器的名字可以看成是 CPU 中的一个存储单元的名字，这个单元是有一个数值编号的（二进制编码），在机器指令中，出现的就是数字编号。但对于写程序或读程序的人而言，数字编号难记忆、易混淆，因而用一个符号名来表示，其本质就是一个符号地址。

在指令中的使用格式：　R

功能：寄存器 R 的内容即为操作数

R 是 CPU 内的某个寄存器的代表符号，它可以是 32 位的通用寄存器 eax、ebx、ecx、edx、esi、edi、ebp、esp；16 位的通用寄存器 ax、bx、cx、dx、si、di、bp、sp；8 位通用寄存器 ah、al、bh、bl、ch、cl、dh、dl；多媒体扩展寄存器 mm0、mm1、mm2、mm3、mm4、mm5、mm6、mm7，xmm0、xmm1、xmm2、xmm3、xmm4、xmm5、xmm6、xmm7，还可以是专用的段寄存器 cs、ds、ss、es 等，但是段寄存器的使用会有较多的限制，在一些特定情况下使用，一般在编程中不需要使用。寄存器寻址方式中只有单个寄存器，不能是含有寄存器的表达式。

【例 4.5】　inc　eax

inc 为加 1 指令的操作符，其操作数地址为寄存器 eax，即操作数在 eax 中。

假设执行前：（eax）=00000005H，也可简写成（eax）=5H

执行：　（eax）+1=5H+1=6H　→ eax

执行后：（eax）=00000006H。

【例 4.6】　add　eax，edx

这是一条双操作数指令，其中 add 为加法指令操作符，eax 为目的操作数地址，edx 为源操作数地址。

假设执行前：（eax）=12345678H，（edx）=0F0000000H

执行：　（eax）+（edx）=02345678H　→ eax

执行后：（eax）=02345678H，（edx）不变。

标志寄存器中的 CF=1，ZF=0，SF=0，OF=0。

若将(edx)中的数，看成一个有符号数的补码表示，则该数为 -10000000H。(eax)和(edx)相加，由于两个加数的最高二进制位不同，一定不会溢出，即 OF=0。

【例 4.7】　dec　cx

dec 为减 1 指令的操作符，寄存器 cx 为操作数地址。

假设执行前：（cx）= 80H ，也即 (cx)=0080H

执行：　（cx）- 1 = 80H - 1 = 7FH　→ cx

执行后：　（cx）= 7FH。

【例 4.8】　mov　ebx，offset x

假设 x 是数据段中定义的一个变量，执行该语句后，就会将变量 x 的地址送给 ebx。这相当于 C 语言语句：ebx=&x。

假设程序运行时，变量 x 的地址是 0BB4000H，则反汇编中看到的语句是：mov ebx,0BB4000h。

另外，在取变量的地址时，需要使用 32 位的寄存器。本书所有的示例程序，都是使用 32 位段，地址是 32 位的。若写指令 “mov bx，offset x”，则会报错：error A2022: instruction

operands must be the same size。

在寄存器寻址方式中，所用寄存器的位数决定了指令操作数的类型，例如：采用 32 位寄存器表示操作数是双字类型；采用 16 位寄存器表示操作数是字类型；采用 8 位寄存器表示操作数是字节类型，使用时应根据需要正确选择。由于寄存器是 CPU 的专用存储器，因此，对于那些经常存取的操作数采用寄存器寻址方式可以提高工作效率。

在汇编语句中，若两个操作数都是寄存器寻址方式，如例 4.6 中的“add eax，edx”，则要求两个寄存器的类型必须相同。例 4.6 中两个寄存器都是 32 位的，类型一致。但是，若写出的指令是 “add eax，dx”，则编译时会给出错误提示： error A2022: instruction operands must be the same size，两者的类型不同。此时，需要将加数(dx)扩展中一个 32 位数，然后再与(eax)相加。例如 “movzx edx，dx”；“movsx edx，dx”。前者将(dx)视为一个无符号数，高位以 0 扩展，执行后(edx)的高 16 位为 0，低 16 位为（dx)；后者将(dx)视为一个有符号数，进行的是符号扩展。如果(dx)的最高二进制位为 1，则执行后(edx)的高 16 位为 0FFFFH；如果(dx)的最高二进制位为 0，则执行后(edx)的高 16 位为 0000H。

4．4 直接寻址

4.4.1 直接寻址的基本概念

在直接寻址方式中，操作数存放在存储器中。操作数的偏移地址 EA 紧跟在指令操作码后面，构成了指令的一部分。

在指令中的使用格式：

① 变量　或者等价的写成［变量］

② 变量±一个数值表达式　或者［变量±一个数值表达式］

或者 变量[±一个数值表达式]　这些都是等价写法

③ 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。

要点：

① 正确理解“变量 ±一个数值表达式”的含义，它是用变量的偏移地址±一个数值表达式得到一个新地址，在编译的时候完成新地址的计算，在机器指令中只有一个新地址，不能像 C 语言那样，认为是变量单元中的内容±一个数值表达式的值。机器指令的编码形式为：

操作码及寻址方式编码
操作数的地址偏移量

② 正确理解汇编语言中的“变量[±一个数值表达式]”与 C 语言中的“变量[±一个数值表达式]”的差异。C 语言中的“变量[±一个数值表达式]”由数值表达式指明了要访问的数组的第几个元素，一个元素所占的字节数是由变量的类型所决定的。在汇编语言中，数值表达式确定的值是从变量的开始处再偏移的字节数。

③ 变量为全局变量（在 data 段定义的变量）时，单个变量或者“变量±一个数值表达式”是直接寻址方式；当变量为局部变量时（子程序或函数中定义的），该访问形式不是直接寻址而是变址寻址。在第 8 章子程序设计中再详细介绍。

④ 在 32 位段中，机器指令中存放 32 位的被访问单元的地址；

⑤ 变量有明确的数据类型；另一个操作数的类型应与其匹配，或者是无类型；

4.4.2 直接寻址的用法示例

【例 4.9】 写出实现指定功能的指令

设有如下数据段

```
.data
x   db   1, 2, 7, 8
```

① 将变量 x 的首字节内容，送入 al 中

```
mov   al, x
```
; 执行后 (al)=1

也可以写成 mov al, x[0] 或者 mov al, [x]。

假设变量 x 的地址是 00EE4000H，这三条语句在反汇编后对应的语句是相同的，皆为：

```
mov al, byte ptr ds:[00EE4000H]
```
; 机器码为 A0 00 40 EE 00。

在这三条语句中，目的操作数都是寄存器寻址方式，源操作数为直接寻址方式。

② 将变量 x 的第 1 个字节内容改为 32H

```
mov   x+1,  32H
```
或者 mov x[1], 32H, 或者 mov [x+1],32H

这三条语句在反汇编后对应的语句是相同的，皆为 mov byte ptr ds:[00EE4001H],32H,

它们的机器码都是：C6 05 01 40 EE 00 32。

注意： x[1] 是 x 的第 1 个字节，执行上述指令后， (x[1])=32H。

③ 将变量 x 的第 2 字节内容送给 AH

```
mov ah, x+2
```
或者 mov ah, x[2] 或者 mov ah, [x+2],执行后 (ah)=7

它们的机器码是：8A 25 02 40 EE 00，反汇编语句为 mov ah, byte ptr ds:[00EE4002H]。

请读者特别注意 mov ah,x+2 的语义。它与 C 语言中的 ah=x+2 是完全不同的,但是与 ah=x[2] 相同。首先，从其对应的机器码可以看到，源操作数是给出的一个直接的地址，它是变量 x 的地址再加上 2 后得到的新地址，而不是先把 x 单元中的内容取出来后加上 2。

换个角度看，“ah=x+2”,涉及到两个操作，一是 x+2，这是一个加法操作；二是结果传送，即运算后的结果送 ah 中。在指令中不会同时出现 mov 操作符的编码和“+”运算符的编码，一条指令只完成一个操作。

如果要完成 C 语句 (ah)=(x)+2 的功能，即将 x 单元中的内容加 2 后送给 ah，写成汇编指令为：

```
mov   ah,   x       ; (x)->ah,    (ah)=1
add   ah,   2       ; (ah)+2->ah , 即 (x)+2 -> ah     (ah)=3
```

此外，在编译程序生产机器指令时，编译器无法事先确定 x 单元中的内容，因为变量 x 中的内容是可以在程序运行中改变的；但是 x 单元的地址可以确定。

④ 将变量 x 的后 2 字节内容（即第 2,3 两个字节中的内容）送给 ax

```
mov   ax, word ptr x[2]
```
； 执行后 (ax)=0807H

注意，在指令中必须加 word ptr。若不加 word ptr，则编译时会报错。因为目的操作数是寄存器寻址方式，用的是 ax，是字类型。源操作数是直接寻址方式，变量 x 是用 db 定义的，是字节类型。源操作和目的操作数的类型不一致。

⑤ 将变量 x 的首字节内容加 1

add x，1　或者　inc x，　完成的功能是 (x)+1->x

注意，在指令中，并不能改变变量的地址，它是在编译时就确定了的。inc x 不是将 x 的地址加 1，而是先从 x 处取一个数加 1 后，再将结果送回 x 处。

等价的语句是：add byte ptr ds:[00EE4000H]，1　；x 的地址为 00EE4000H

inc byte ptr ds:[00EE4000H]

【例 4.10】 写出实现指定功能的指令

设有如下数据段：

```
.data
y   dw   10H, 20H, 70H, 80H
```

① 将变量 y 的第一字中的内容即 20H，送给 ax

```
mov   ax，y+2   ；或者 mov ax，y[2] 或者 mov ax，[y+2]
```

② 将变量 y 的第二字中的内容即 70H，改为 50H

```
mov   y+4,50H   ；或者 mov y[4],50H 或者 mov [y+4],50H
```

③ 与 C 语言语句的比较

```
y   dw   10H, 20H, 70H, 80H   等价于 C 语言语句：
unsigned short y[]={0x10,0x20,0x70,0x80};
```

实现① 的功能，C 语句为 ax=y[1];

实现② 的功能，C 语句为 y[2]=0x50;

对比两种表达方法，可以看出汇编语句与 C 语句的不同。在汇编语言（机器语言）中，是以字节为单位来计算地址的，因为一个 dw 定义一个字类型的数据，每个数要占 2 个字节，因此，y 中的第 1 个字、第 2 个字与 y 开头（第 0 个字的开头）分别相距 2 个和 4 个字节。在 C 语言中，y[1],y[2]分别表示数组的第 1 个和第 2 个元素。但是在对 C 语言语句编译后，就会发现，编译器会根据定义变量的类型，自动的将其转换为以字节为单位的地址。

C 语句 y[2]=0x50; 反汇编后对应的语句片段是：

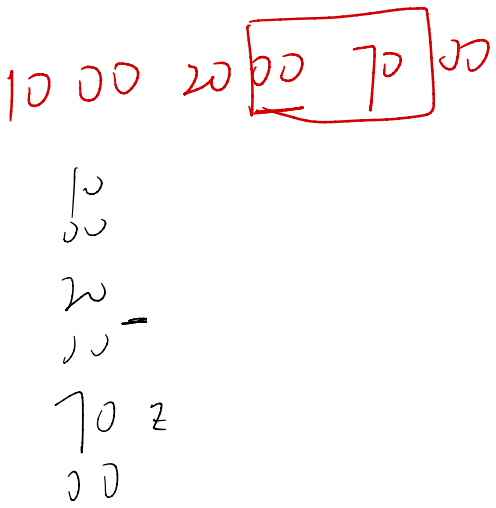
```
mov eax，2    ;(eax)=2
shl eax，1    ;逻辑左移 1 位，即 (eax)*2 = 4 ->eax
mov word ptr [eax+********]，50H; 其中********是 y 的起始地址。
```

注意，不同的编译器或者不同的优化程度，生成的机器语言程序是有差异的。另外， y 是全局变量和局部变量的编译结果也是不同的。

【例 4.11】 指出给定语句实现的功能

设有如下数据段：

```
.data
x   dw   10H
y   dw   20H
z   dw   70H
①   mov   ax, x+2    ; 执行后 (ax)=20H
②   mov   ax, y      ; 执行后 (ax)=20H
③   mov   ax, z-2    ; 执行后 (ax)=20H
④   mov   ax, x+3    ; 执行后 (ax)=7000H
```



⑤ mov ax, z-1 ; 执行后 (ax)=7000H

⑥ mov ax, y-x ; 执行后 (ax)=2H

该语句的反汇编结果是 mov ax, 2。两个变量相减，实际上是两个变量的地址相减。

如果写出语句 “mov ax, y+x”，则编译时报错。编译器有此规定也是非常合情合理的。正如两个日期可以相减，得到的不再是日期类型，而是一个时间间隔类型。一个日期加或减一个时间间隔，可以得到一个新的日期，但两个日期相加就没有任何意义。

4．5 寄存器间接寻址

4.5.1 寄存器间接寻址的基本用法

在寄存器间接寻址方式中，操作数存放在存储器中，而操作数的有效地址（Effective Address，EA）在指令指明的寄存器中，即寄存器的内容为操作数的有效地址 EA。

在指令中的使用格式：[R]

功能：寄存器 R 的内容为操作数的偏移地址 EA。

要点：

① 在 x86-32 中，本书采用的是 32 位段扁平内存管理模式，因此 R 是 8 个 32 位通用寄存器（eax、ebx、ecx、edx、edi、esi、ebp、esp）中的任何一个；

② 语法描述的是单个孤零零的寄存器出现在方括号中；

③ 操作数是在内存中，注意与寄存器寻址方法的差异；

④ 在写程序时，先要将待访问单元的地址送入某个 32 位通用寄存器中；

⑤ 寄存器间接寻址类似于 C 语言中的指针；

⑥ 单个 32 位的寄存器没有出现在方括号中时，是寄存器寻址方式。

【例 4.12】 用寄存器间接寻址方式，写出完成指定功能的程序段。

设有如下数据段：

```
.data
x  db  1, 2, 7, 8
```

① 将变量 x 的首字节内容，送入 al 中

```
mov ebx, offset x  ; 将变量 x 的地址送给 ebx
mov al, [ebx]
```

设 x 的地址是 00EE4000H，可以用图 4.1 来表明寄存器与存储单元之间的关系。

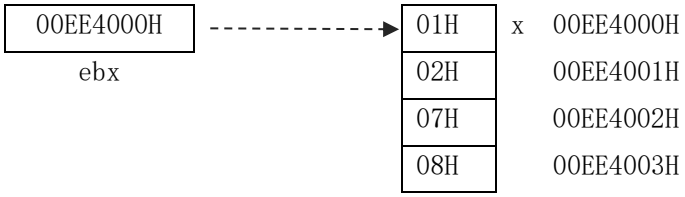


图 4.1 寄存器间接寻址

对于此例，若执行 mov eax,[ebx]，执行后 (eax)=08070201H，源操作数为寄存器间接寻址。若执行 mov eax, ebx，执行后 (eax)=00EE4000H，源操作数是寄存器寻址。

另外， mov al,[ebx] 语法正确；而 mov al,ebx 语法错误，目的操作数和源操作数的类型

都明确，但不匹配。

注意寄存器上有无方括号差别是很大的，但是对于一个变量，有无方括号是等价的。

② 将变量 x 的第一字节内容改为 32H

```
mov ebx, offset x +1
mov byte ptr [ebx], 32H
```

注意，offset x 得到的是一个表示 x 地址的数值常量，设 x 的地址是 00EE4000H，表达式就是 00EE4000H +1，编译器会完成计算工作，得到 00EE4001H。因此，源操作数是立即寻址方式。反汇编语句是 mov ebx, 00EE4001H。

当然，实现本功能也可以用如下语句片段：

```
mov  ebx, offset x
inc  ebx  或者 add ebx,1
mov  byte ptr [ebx], 32H
```

注意：“mov byte ptr [ebx],32H”中的 byte ptr 是不能省略的。若不出现 byte ptr，单纯的[ebx]只指明了一个地址而没有类型；32H 也没有类型，因而两个操作数的类型都不明确。

③ 将变量 x 的第 2 字节内容送给 ah

```
mov esi, offset x+2
mov ah, [esi]
```

此处，不用 esi 而用其他的 32 位通用寄存器也是可以的。

④ 将变量 x 的后 2 字节内容（即第 2,3 两个字节中的内容）送给 ax

```
mov esi, offset x+2
mov ax, [esi]
```

注意，目的操作数是寄存器寻址方式，用的是 ax，即字类型。源操作数是寄存器间接寻址方式，其数据类型是不明确的，编译器会自动的将不明确的类型按另一个操作数指明的类型来处理。

4.5.2 寄存器间接寻址与 C 语言指针的比较

例 4.12 中的功能②“将变量 x 的第一字节内容改为 32H”，可以用如下 C 语言语句完成。

定义全局变量：

```
unsigned char x[]={1,2,7,8};
char  *p;
```

在主程序中有如下语句：

```
p =x;  // p = &(x[0]);
p=p+1;
*p=0x32;
```

如果调试该程序，在反汇编窗口可看到如下语句（注意：指令的地址、变量的地址在不同的机器上或者同一机器上不同的运行时刻，都可发生变化）

```
p = x;
008A1908 C7 05 E8 A5 8A 00 00 A0 8A 00 mov dword ptr [p (08AA5E8h)],offset x (08AA000h)
p = &(x[0]);
008A1912  B8 01 00 00 00        mov     eax,1
```

```
008A1917  6B C8 00            imul    ecx,eax,0
008A191A  81 C1 00 A0 8A 00   add     ecx,offset x (08AA000h)
008A1920  89 0D E8 A5 8A 00   mov     dword ptr [p (08AA5E8h)],ecx
p = p + 1;
008A1926  A1 E8 A5 8A 00      mov     eax,dword ptr [p (08AA5E8h)]
008A192B  83 C0 01            add     eax,1
008A192E  A3 E8 A5 8A 00      mov     dword ptr [p (08AA5E8h)],eax
*p = 0x32;
008A1933  A1 E8 A5 8A 00      mov     eax,dword ptr [p (08AA5E8h)]
008A1938  C6 00 32            mov     byte ptr [eax],32h
```

从该例的反汇编结果中可以看出，“*p=0x32;”是通过寄存器间接寻址的方式来实现相应功能的。它首先将变量 p 中的内容（即 p 指向的单元的地址）送给了寄存器 eax,然后将 32h 传送到以(eax)为地址的单元中。

【例 4.13】 分析 C 语言程序片段的反汇编代码，理解指针和寄存器间接寻址的对应关系。

```
int  *q;  ……
q=q+1;
*q=50;
```

在反汇编窗口，可以看到如下代码（在反汇编窗口“查看选项”中勾选显示符号地址）：

```
q = q + 1;
008A18F6  8B 45 DC      mov   eax,dword ptr [q]
008A18F9  83 C0 04      add   eax,4
008A18FC  89 45 DC      mov   dword ptr [q],eax
*q = 50;
008A18FF  8B 45 DC      mov   eax,dword ptr [q]
008A1902  C7 00 32 00 00 00     mov  dword ptr [eax],32h
```

从反汇编代码中可以看到 q=q+1，实际上是将 q 中的内容加了 4。一个 int 类型数据占 4 个字节，要让 q 指向下一个 int 类型的数时，地址要增加 4 个字节。

4．6 变址寻址

变址寻址方式的操作数存放在存储器中，其偏移地址 EA 是指令中指定寄存器的内容乘以比例因子后与给出的位移量之和。

在指令中的使用格式：[R*F+V]　或：[R*F]+V　　或：V [R*F]

功能：寄存器 R 的内容乘以比例因子 F 后加上给定的位移量 V 作为操作数的地址。

要点：

① 在 x86-32 中，本书采用的是 32 位段扁平内存管理模式，因此 R 是 8 个 32 位通用寄存器（eax、ebx、ecx、edx、edi、esi、ebp、esp）中的任何一个；

② F 只能是 1、2、4 或 8；

③ 当 R 为 esp 时，F 只能为 1；

④ 若 V 是变量，则是取 V 的地址参与运算；

⑤ 若 V 是变量，则该操作数的类型是明确的，为定义 V 的类型；若 V 是一个常量，则类型是不明确的；

⑥ 变址寻址类似于 C 语言中的一维数组访问。

【例 4.14】 用变址寻址方式访问操作数，完成指定的功能

设数据段中定义变量 y。

```
y    dd   10,20,30,40
```

① 将 y 中的第一个双字数据（即 20），送给 eax

```
mov   ebx，1
mov   eax，y[ebx*4]
```

等价的 C 语言语句为 ebx=1； eax = y[ebx]。编译器完成数组元素下标乘 4 的工作。

② 将 100 送到 y 的第二个双字单元中（即 30 所在的位置）

方法 1：

```
mov   ebx，2
mov   y[ebx*4],100
```

注意，目的操作数是变址寻址，其中有变量 V，类型是明确的。

方法 2：

```
mov   ebx，8
mov   y[ebx],100
```

方法 3：

```
mov   ebx，offset y
mov   dword ptr [ebx+8],100
```

这三种方法目的操作数都是变址寻址方式。推荐使用第 1 种方法，它与 C 语言中的数组访问非常相似，用 R 来存储是数组的第几个元素，比例因子为每个元数的长度。

【例 4.15】 C 语言程序中一维数组的访问与变址寻址的对应关系

设有如下 C 语言程序段：

```
int y[4] = { 10, 20, 30, 40 };
int i=2;
y[i]=100;
```

“y[i]=100; ”的反汇编代码如下：

```
009118F0  8B 45 A0     mov  eax, dword ptr [i]
009118F3  C7 44 85 E8 64 00 00 00  mov  dword ptr y[eax*4],64h
```

从该例中可以清楚的看到一维数组的访问方法，即将数组元素的下标值送到一个 32 位的寄存器中，通过 V[R*F] 的形式来访问元素，其中比例因子 F 是一个元素的长度。

更严格的说，当数组变量是全局变量时，数组元素的访问对应的是变址寻址；当数组变量定义为局部变量时，它的访问对应的是基址加变址的寻址方式。

4．7 基址加变址寻址

基址加变址寻址方式的操作数存放在存储器中，其偏移地址 EA 是指令中指定的基址寄存器的内容、变址寄存器的内容乘以比例因子、位移量 V 三项相加之和。

在指令中的使用格式：[BR+IR*F+V]　或：V[BR][IR*F]　或：V[BR+IR*F]

功能：变址寄存器 IR 的内容乘以比例因子 F，与基址寄存器 BR 的内容和位移量 V 相加，作

为操作数的地址。

要点：

① 在 x86-32 中，本书采用的是 32 位段扁平内存管理模式，因此 BR 是 8 个 32 位通用寄存器（eax、ebx、ecx、edx、edi、esi、ebp、esp）中的任何一个；IR 是除 esp 之外的任一 32 位通用寄存器（BR 和 IR 可以相同）；

② F 只能是 1、2、4 或 8；

③ 若 V 是变量，则是取 V 的地址参与运算；

④ 若 V 是变量，则该操作数的类型是明确的，为定义 V 的类型；若 V 是一个常量，则类型是不明确的；

⑤ 基址加变址寻址方式与 C 语言中的二维数组访问类似。

【例 4.16】 写出实现指定功能的程序段，要求使用基址加变址寻址方式

设数据段中定义有如下变量

```
x   dd   10，  20，  30，  40，  50
    dd   60，  70，  80，  90，  100
    dd   110，120，130，140，150
```

这一排列形式，很像 C 语言中的二维数组，类似于 int x[3][5]；一行有 5 个元素（从第 0 列到第 4 列），共 3 行（从第 0 行到第 2 行）。从数据存储的角度来看，它与如下形式定义的变量是完全相同的。

```
x dd 10，20，30，40，50，60，70，80，90,100，110，120,130，140,150
```

这也就是 C 语言中常说的按行序优先方法存储的结果。

```
i   dd   1
j   dd   2
```

假设要实现将 x[i][j]送给 eax 的功能，程序段如下：

```
imul   ebx, I, 5*4
mov    esi, j
mov    eax, x[ebx][esi*4]
```

程序段分析：按照二维数组元素的存储顺序原则及每个数据的长度，可以计算出一行元素的长度，即每行 5 个元素，每个元素占 4 个字节，共计 5*4=20 字节。数组第 i 行的起始元素相对于整个数组开头的字节距离为 i*20，换言之，由数组的起始地址加上 i*20,可以得到第 i 行的起始元素的地址。对第 i 行第 j 列元素的地址，是在第 i 行的起始地址的基础上再加上 j*4 个字节。

【例 4.17】使用指定的寻址方式，将 x[i][j]->eax,其中 x,i,j 的定义与例 4.16 相同

① 用寄存器间接寻址方式访问该单元

```
mov   ebx，offset x
imul esi，i，4*5
add   ebx，esi
imul esi，j，4
add   ebx，esi
mov   eax,[ebx]
```

② 用变址寻址方式访问该单元

```
imul ebx, i, 5
add  ebx, j
mov eax, x[ebx*4]  ; (ebx)中的值是要访问的二维数组中的第几个元素
```

③ 用直接寻址方式

```
mov  eax, x+1*5*4+2*4
```

注意：绝对不能写成 mov eax, x[i][j], 也不能写成 mov eax,x[i*4*5+j*4]。一定要改变C 语言中的写法习惯。i、j、x 都是变量,在编译的时候,编译器只可能用到变量的地址，而不会用变量中的值，因为变量中的值是可变的。只有对常量运算的表达式，如 1*5*4+2*4，编译器是可以算出其值的，即 28，在加上 X 的地址，得到要访问单元的地址。

当然，单纯从完成一个给定的功能来看，是很容易将变址寻址变成基址加变址寻址方式的，只需要增加一个基址寄存器，并且让该寄存器中的内容为 0。

例如：mov eax, x[ebx*4] 完成的功能等价于如下两条指令运行后的功能。

```
mov esi, 0
mov eax, x[esi+ebx*4]
```

4．8 寻址方式综合举例

在六种寻址方式中，立即寻址、寄存器寻址相对简单。另外四种寻址方式，即直接寻址、寄存器间接寻址、变址寻址、基址加变址寻址，都是访问内存单元，使用比较灵活。本节给出一个例子，分别用四种寻址访问完成对存储单元的访问。

【例 4.18】求以变量 x 为起始地址的 4 个双字数据的和,并显示结果(显示的结果是 100)。

在下面的程序中给出了 4 种实现方法，分别用不同寻址方式访问存储单元。在程序中以注释的方式给出了寄存器的分配和算法思想。完整的程序如下。

```
.686P
.model flat, stdcall
  ExitProcess proto :dword
  printf      proto c :ptr sbyte, :vararg
  includelib  libcmt.lib
  includelib  legacy_stdio_definitions.lib
.data
  lpFmt db  "%d",0ah, 0dh, 0
  x     dd  10, 20, 30, 40
.stack 200
.code
main proc  c
   ;方法 1：  用寄存器间接寻址方式，访问数组 x 中的各个单元的内容
   ;  用 eax 来存放所求的 和
   ;  用 ebx 来指向要访问的数据，即 ebx 中的值为操作数的地址
   ;  用 ecx 来控制循环次数，每循环一次，ecx 减 1。当减到值为 0 时，循环结束
```

```
    ;  类似 C 语句 eax=0; ebx=&(x[0]); ecx=4;
    ;          do {eax+=*ebx; ebx+=1; ecx-=1}while(ecx!=0)
    mov  eax, 0
    mov  ebx, offset x
    mov  ecx, 4
lp1:
    add  eax, [ebx]
    add  ebx, 4       ; 以字节为单位，计算下一个数的地址
    dec  ecx          ; (ecx)-1->ecx, 当差为 0 时 ZF=1，否则 ZF=0
    jnz  lp1          ; 当 ZF=0（即结果 not zero）时，转移到 lp1 处执行
    invoke printf,offset lpFmt,eax
    ;方法 2：用变址寻址方式，访问数组 x 中的各个单元的内容
    ;  用 eax 来存放所求的 和
    ;  用 ebx 来表示待访问的数组元素的下标
    ;  类似 C 语句 eax=0;  for (ebx=0;ebx<4;ebx++) eax+=x[ebx]
    mov  eax, 0
    mov  ebx, 0
lp2:
    cmp  ebx, 4       ; 执行 (ebx)-4, 不保存差，但根据差设置标志位
    jge  exit_2       ; 与上一句联合的作用是 (ebx)>=4 转移到 exit_2
                      ; 否则不转移，继续执行下面的语句
    add  eax, x[ebx*4]
    inc  ebx          ; (ebx)+1->ebx
    jmp  lp2          ; 无条件跳转到 lp2 处执行
exit_2:
    invoke printf,offset lpFmt,eax
    ;方法 3：用基址加变址寻址方式，访问数组 x 中的各个单元的内容
    ;  用 eax 来存放所求的 和
    ;  用 ebx 来表示待访问的数组的起始地址；
    ;  用 esi 来表示待访问的数组元素的下标
    ;  类似 C 语句 eax=0;  for (esi=0;esi<4;esi++) eax+=[ebx][esi]
    mov  eax, 0
    mov  ebx, offset x
    mov  esi, 0
lp3:
    cmp  esi, 4
    jge  exit_3
    add  eax, [ebx][esi*4]
    inc  esi
    jmp  lp3
```

```
exit_3:
    invoke printf,offset lpFmt,eax
    ;方法 4：用直接寻址方式，访问数组 x 中的各个单元的内容
    ;  用 eax 来存放所求的 和
    mov  eax, x
    add  eax, x[4]
    add  eax, x+8
    add  eax, [x+12]
    invoke printf,offset lpFmt,eax
    invoke ExitProcess, 0
main  endp
end
```

说明：相同的寻址方式也可以用多种不同的形式来表达。本书给出的例子仅仅只是一个示例，完全可以给出不同的写法。

从上例中可以看到，写程序前应该明确寄存器的分配，即确定各个寄存器各起什么作用；明确所使用的算法，如程序中给出的几种循环控制方法；严格按照各种寻址方式的语法格式写出指令。此外，对双操作数指令，还要判断源、目的操作数是否同时用了存储器寻址方式；两者的类型是否明确和匹配；目的操作数是否使用了立即数。

当然，写指令需要一个熟练的过程，需要加实验和练习。同时打破用 C 语言写复杂表达式的习惯，将复杂表达式要完成的功能拆解成多个小的简单步骤。

4．9 x86 机器指令编码规则

本节介绍 x86-32 机器指令格式及编码解析，有兴趣的读者可以阅读本节，了解 Intel x86 是如何对机器指令进行编码的，探索 Intel 指令编码的设计奥秘。当然，理解机器编码的规定，对理解编译器对汇编语句加工后生成的目标，推想出汇编语句中应给出哪些组成要素等是有帮助的。在学习本节时，不需要记忆有关编码的细节，理解相关规则即可。

x86 机器指令编码的依次由以下部分组成：

① 指令前缀(prefix，非必需)
② 操作码(opcode，必须)
③ 内存/寄存器操作数(Mod R/M，非必需)
④ 索引寻址描述 (SIB，非必需)
⑤ 地址偏移量(Displacement，非必需)
⑥ 立即数(Immediate，非必需)

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

下面，逐一进行解释。

① 指令前缀(prefix)

指令前缀，可以有多个前缀，每一个前缀都用 1 个字节来表示。指令前缀的名称和编码如表 4.1 所示。

表 4.1 指令前缀编码

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
Oprand size Override	Oprandsize Override	66H	用该前缀来区分：访问 32 位或 16 位寄存器；也用来区分 128 位和 64 位寄存器
Address Override	Address Override	67H	64 位下指定用 64 位还是 32 位寄存器作为索引

REP 是串操作指令前缀，它又可以细分为 REP、REPZ、REPNZ，重复地执行操作指令，直到满足或者不满足某些条件。当然，这三个前缀不会同时使用，最多只使用其中的一个。

Segment Override 是跨段前缀。在以前的分段内存管理模式下，可以用跨段前缀来指明要访问的段。当然，在扁平内存管理模式下，依然可以在语句中用跨段前缀，但是指令的功能等同无前缀。

例如：mov ebx，offset x ；x 是 data 段中定义的一个变量

```
mov  eax, cs:[ebx]  ; 机器码是：2E 8B 03
mov  eax, [ebx]     ; 机器码是：8B 03
```

后两条语句执行的结果是相同的。

Oprand size Override，在不是使用扁平内存管理模式时，例如 .model small 下，可以用 16 位寄存器，也可以使用 32 位寄存器来访问内存单元，为了区分是用的哪一类型的寄存器，编译器会自动生成相应的前缀。例如：

```
mov ax,  [bx]   ; 机器码是：67 66 8B 07
mov ax, [ebx]   ; 机器码是：66 8B 03
```

在本书中的程序例子都是使用 .model flat，不能够使用 16 位寄存器来访问内存单元，mov ax, [bx]有语法错误。

② 操作码(opcode)

操作码是操作符的编码，指明了要进行什么操作。它可以是 1 个字节、2 个字节或者 3 个字节。大多数通用指令的操作码是单字节的，最多 2 个字节，但 FPU 指令、SSE 等指令可以有 3 个字节。在操作码的编码中，还指明了操作数的类型，即是字节、字还是双字数据。操作码中也含有指明操作对象的信息（寄存器、内存地址、立即数），甚至指明特定的操作对象。

③ 内存/寄存器操作数(Mod R/M)

如果需要有“内存/寄存器操作数”编码，它占 1 个字节，指明了操作数的寻址方式。在双

操作数指令中，有源操作数的寻址方法、目的操作数的寻址方法，但是又规定两个操作数不能同时为存储器寻址方式，因而只可能是下面的组合形式<目的操作数，源操作数>：

<寄存器，寄存器>、<寄存器,立即数>、<寄存器,存储器>、<存储器，寄存器>、<存储器，立即数>。

在双操作数指令中，有一个是寄存器或立即数，另一个是寄存器或存储器。它们的顺序上的不同，如<寄存器,存储器>和<存储器，寄存器>，在操作码的编码中指明。在“内存/寄存器操作数(Mod R/M) ”编码中，只需要指明有两种组成成份，“寄存器或立即数 Reg/Opcode”和“寄存器或存储器 R/M”，而不说明两者之间的前后关系。

“内存/寄存器操作数(Mod R/M) ”信息分为三部分，Mod（2 个二进制位）、Reg/Opcode（3 个二进制位）、R/M（3 个二进制位）。它们的摆放顺序如下。

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

I. Mod 的编码规则

Mod 由 2 个二进制位组成，取值只能是 00、01、10、11。它与为 R/M 配合使用，明确一个操作的获取方法。在 32 位段和 16 位段中，编码是不同的，如表 4.2 和表 4.3 所示。

表 4.2 32 位段中 Mod 的编码

有效地址 Effective Address	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/MMO/XMMO,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

表 4.3 16 位段中 Mod 的编码

有效地址 Effective Address	Mod
[BX+SI], [BX+DI], [BP+SI], [BP+DI], [SI], [DI], disp16, [BX]	00
[BX+SI]+disp8, [BX+DI]+disp8, [BP+SI]+disp8, [BP+DI]+disp8, [SI]+disp8, [DI]+disp8, [BP]+disp8, [BX]+disp8	01
[BX+SI]+disp16, [BX+DI]+disp16, [BP+SI]+disp16, [BP+DI]+disp16, [SI]+disp16, [DI]+disp16, [BP]+disp16, [BX]+disp16	10
EAX/AX/AL/MMO/XMMO, 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

从表 4.2-4.3 中可以看到，同一个 Mod 值，可以对应多种情况。如果仔细的研究一下，每一个 Mod 值，实际上是对应 8 种情况。Intel 采用 R/M 这三个二进制位来区分是 8 种情况中的哪一种。注意，在表 4.2 中，当 Mod=11 时，第一种情况是：EAX/AX/AL/MMO/XMMO，它们拥有相同的 R/M 编码，这 5 个寄存器对应的类型（长度）不同，在操作码中体现。

II. R/M 的编码规则

与 Mod 配合使用。例如，在表 4.2 中，当 Mod=00 时，有 8 种情况 [EAX]、[ECX]、[EDX]、[EBX]、 [--][--]、disp32、[ESI]、[EDI]，它们正好可以用 3 位二进制数进行编码，即 R/M 编码。这 8 种情况依次编码为 000-111，这与表 4.4 中寄存器的编码对应起来。从表 4.4 可知，当 R/M 编码为 000 时，所使用的是[EAX]；当 R/M 编码为 001 时，所使用的是[ECX]，依此类推[EDX]、[EBX]、[ESI]、[EDI]的编码。当 R/M 编码为 100 时，它指明了一种不带偏移量的基址加变址的寻址方式，其基址寄存器、变址寄存器、比例因子由“索引寻址描述 (SIB) ”来指明。

III. Reg/Opcode 的编码规则

对于寄存器，其编码规定如下。

表 4.4 寄存器的编码

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

下面，给出一些简单的例子，来验证编码的规定。

```
mov  ebx,dword ptr [ecx]   ; 机器码是 8B 19
mov  dword ptr [ecx],ebx   ; 机器码是 89 19
```

这两条指令中，寻址方式编码都是 19H，即 00 011 001 B，Mod 为 00，表示有一个操作数可能用的是寄存器间接寻址，R/M 为 001，即 ECX/CX/CL/MM1/XMM1,结合 Mod=00，用的是 32 位寄存器，因此，可以确定一个操作数为 [ECX]；另一个操作数由 Reg/Opcode 给定，为 011，即 EBX/BX/BL/MM3/XMM3 组，但是在操作码中指明了操作数是双字类型，则可确定为是 EBX。但是从寻址方式编码中，只确定了这两个成份，但是并不能确定哪一个是目的操作数寻址，哪一个是源操作数寻址方式。该信息在操作码中反映，这两条指令的操作码是不同的。

另一个关于类型的比较例子如下，可以看出指令访问数据类型的不同，对指令编码的影响。

```
mov  ebx,dword ptr [ecx] ; 机器码是 8B 19
mov  bx,word ptr [ecx]   ; 机器码是 66 8B 19 ，前缀 66H,表示访问 16 位寄存器
mov  bl,byte ptr [ecx]   ; 机器码是 8A 19
```

通过编码，也可以了解为什么要求两个操作数的类型一致了，这样数据类型只编码一次。

④ 索引寻址描述 (SIB)

如果需要有“索引寻址描述”编码，它占 1 个字节。SIB 字节信息也分为三部分，Scale（2 个二进制位）、Index（3 个二进制位）、Base（3 个二进制位）。它们的摆放顺序如下。

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

在寻址方式编码中，可以看到，当 Mod=00 时，有一种方式是[--][--]；当 Mod=01 时，有一种方式是[--][--]+disp8；当 Mod=10 时，有一种方式是[--][--]+disp32。disp8、disp32 分别表示有 8 位和 32 的偏移量，而[--][--]是待定成份，就是在索引寻址描述中体现出来。

顾名思义，Scale 表示比例因子，从 00-11 分别对应 1,2,4,8。Base 表示基址寄存器的编码，

Index 表示变址寄存器的编码。寄存器的编码仍采用表 4.4 的规定。

下面给出几条指令的机器码，验证上述编码规则。

```
mov   eax,[ebx+ecx*4]     ; 机器码是 8B 04 8B
mov   eax,[ecx+ebx*4]     ; 机器码是 8B 04 99
mov   eax,[ebx+ecx*4]+5   ; 机器码是 8B 44 8B 05
```

对前两条指令，寻址方式编码都是 04，即 00 000 100。Mod=00，R/M 为 100，即为方式是[--][--]，另一个操作数在寄存器中，编号 000 对应 EAX。在它们之后，就有一个 SIB 字节。8BH 对应 10 001 011 中，即比例因子（10）对应为 4；基址寄存器（Base）对应为 011，即 EBX；基址寄存器（Index）对应为 001，即 ECX。99H 对应为 10 011 001，基址寄存器为 ECX，变址寄存器为 EBX。比例因子都是乘在变址寄存器上的。

对于第三条指令，44H 对应 01 000 100。同上分析 01 和 100 决定了一种寻址方式是[--][--]+disp8，另一个操作数在 EAX 中，SIB 是 8B，与第一条指令相同。之后，有一个字节的偏移量 05。

⑤ 地址偏移量(Displacement)

地址偏移量由 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。

⑥ 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。

有兴趣的读者，可以通过反汇编查看汇编语句对应的机器指令，分析指令的编码规律。

同样一个 0-1 子串，在不同的场景下表示的含义可以是不同的，不能够静态的、一成不变的看待一个 0-1 串。这就像一个学生，在父母眼中是听话孝顺的孩子，在老师眼中是一个可爱的学生，当然在朋友的眼中又有另一种解读。例如，如果将一个执行程序文件拷贝到另外一个地方，那么这个执行程序文件对拷贝程序而言，都是属于数据。如果是运行该程序，也即在 CPU 中执行指令，CPU 又会给出一种新的解释。总之，0-1 数据串的解析依赖于解析时的上下文环境，相同 0-1 串的解析是会动态的变化的。

在中央处理器 CPU 中，有一个非常重要的寄存器——指令指针（EIP），记录了要执行的指令的起始地址。从这一地址开始解析指令，就可以区分操作码和地址码，因为编码都是有一定规范的。在操作符编码之后，还有指令运算的数据类型编码（字节运算、字运算、双字运算等）、寻址方式编码。寻址方式编码指明了源操作、目的操作数各使用什么寻址方式，每一种寻址方式都有各自的组成成份。在寻址方式编码后，是对各种操作数地址表达式或者操作数的编码。正是因为有严格的编码规定，所以在写汇编语言指令时，就必须严格的遵循这些规定。因为汇编语言的编译器不像高级语言的编译器，它是将一条汇编语句翻译成一条机器指令，这就要求我们必须学会“拆解”，在一条语句只能完成简单的功能，而不能像 C 语言语句那样写出复杂的表达式或语句。

4．10 8086/80386 的寻址方式

8086/80386 CPU 的寻址方式也是 6 种：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。但是在内存分段管理的模式下，寄存器间接寻址、变址寻址、基址

加变址寻址中有关寄存器使用的规则比 32 位段扁平内存管理模式要复杂得多。读者可以跳过此部分的内容，以免影响 32 位段扁平内存管理模式下程序的阅读和编写。本小节仅仅为了和8086/80386 汇编语言程序设计的书有一个对照而介绍的，它们在 32 位段扁平内存管理模式下不再适用。

1、寄存器间接寻址

在 8086 CPU 中,没有 32 位的寄存器。此时的寄存器间接寻址是用 4 个 16 位通用寄存器(dx、di、si、bp）中的一个。那时的内存采用的是 16 位分段管理模式，数据段与堆栈段是两个独立的段。如果使用的寄存器是 bx、si、di，则系统默认操作数在数据段，等价于 ds:[R]。如果使用的寄存器是 bp，则系统默认操作数在堆栈段，操作数地址为“ss:[R]”。

在 80386 中，出现了 32 位的寄存器，但内存管理可以采用 32 位分段管理模式和 16 位分段管理模式。此时，寄存器间接寻址可以用 8 个 32 位的寄存器（eax、ebx、ecx、edx、edi、esi、ebp、esp）或 4 个 16 位寄存器(bx、di、si、bp）中的一个。当 R 是 bp、ebp、esp 时，系统默认操作数在堆栈段中，等价于 ss:[R]。其他的寄存器均默认操作数在 ds 所指示的段，操作数地址为“ds:[R]”。

2、变址寻址

在 8086 中，变址寻址的格式是 V[R]，没有比例因子，R 为 4 个 16 位通用寄存器（bx、di、si、bp）中的一个。如果 V 是变量，使用的段寄存器取决于该变量定义所在的段与哪一个段寄存器建立了联系；如果 V 是常量，如果使用的寄存器是 bx、si、di，则系统默认操作数在数据段，等价于 ds:[R+V]。如果使用的寄存器是 bp,则系统默认操作数在堆栈段,操作数地址为“ss:[R+V]”。

在 80386 中，在 32 位分段管理模式下，可以用 8 个 32 位的寄存器（eax、ebx、ecx、edx、edi、esi、ebp、esp）或 4 个 16 位寄存器(bx、di、si、bp）中的一个。若使用的是 16 位寄存器和 esp 时， F 只能为 1；用其他 32 位通用寄存器，F 可为 1、2、4 或 8。如果 V 是变量，则取 V 的地址参与运算，此时用的段依赖于定义 V 所在的段与哪一个段寄存器建立了联系。若 V 是一个常量，当 R 是 bp、ebp、esp 时，系统默认操作数在堆栈段中，等价于 ss:[R*F+V]。其他的寄存器均默认操作数在 DS 所指示的段，操作数地址为“ds:[R*F+V]”。

3、基址加变址寻址

在 8086 中，变址寻址的格式是 V[BR+IR]，没有比例因子，BR 为 bx、bp 中的一个，IR 为 si、di 中的一个。如果 V 是变量，使用的段寄存器取决于该变量定义所在的段与哪一个段寄存器建立了联系；如果 V 是常量，如果使用的基址寄存器是 bx，则系统默认操作数在数据段，等价于 ds:[BR+IR+V]；如果使用的寄存器是 bp,则系统默认操作数在堆栈段,操作数地址为“ss:[BR+IR+V]”。

在 80386 中，在 32 位分段管理模式下，BR 可以用 8 个 32 位的寄存器（eax、ebx、ecx、edx、edi、esi、ebp、esp）或者 16 位的寄存器 bx、bp。IR 是除 esp 之外的任一 32 位通用寄存器（BR 和 IR 可以相同）或者为 16 位的寄存器 si、di 中的一个。比例因子的规定、V 为变量时使用段寄存器的规定与变址寻址相同。若 V 是一个常量，当 BR 是 bp、ebp、esp 时，系统默认操作数在堆栈段中，等价于 ss:[BR+IR*F+V]。其他的寄存器均默认操作数在 ds 所指示的段，操作数地址为“ds:[BR+IR*F+V]”。

不难看出，在 4.5-4.7 节中介绍的寻址方式规则要简单多了。在 32 位段扁平内存管理模式下，只能使用 32 位的寄存器作为基址寄存器和变址寄存器。此外，(ds)与(ss)相同，它们是在同一个空间之下，不再区分数据段和堆栈段，不存在寄存器与段寄存器的对应关系。因此，编写程序时更简单。

习题 4

4.1　x86-32 中有哪六种寻址方式？各种寻址方式的语法符号是什么？

4.2 分别指出下列指令中源操作数和目的操作数各是什么寻址方式。

（1）mov　esi，10

（2）mov　x,　 10　;x 是双字类型的变量

（3）mov x[4], 10　;x 是双字类型的变量

（4）mov di，[eax]

（5）add eax，4[ebx]

（6）sub al，　[ebx +ecx*2+2]

（7）mov [edi*4+6]，ax

（8）mov eax，x+20　; x 是双字类型的变量

（9）mov x[ebx]，30 ; x 是双字类型的变量

4.3　判断下列指令是否有语法错误，若有错误请指出错误原因。

在指令中的 x 和 y 都是在数据段中定义的双字类型变量。

(1)　mov　eax, bx

(2)　mov　[ebx], 20

(3)　mov　x, y

(4)　mov　ebx，offset　x

　　mov　[ebx], y

(5) add　x+2, 20

(6)　　mov　ax, 20

　　add　x+2, ax

(7) mov　eax, 20

　add　x+2, eax

(8) cmp　10, eax

(9)　mov　eax, ebx +ecx

(10) mov　eax, ebx [10]

(11) mov　eax, [ebx *10]

(12) mov　eax, x+y

4.4 阅读下列程序，指出程序的功能，并指出访问存储单元时使用的寻址方式，程序中的寄存器各自的功能分配（作用）。

```
.386
.model flat, stdcall
 ExitProcess proto :dword
 includelib  kernel32.lib  ; ExitProcess 在 kernel32.lib 中实现
 printf      proto c :vararg
```

```
 includelib  msvcrt.lib  ; printf 在 msvcrt.lib 中实现
.data
 lpFmt  db  "%s",0ah, 0dh, 0
 buf1   db  '00123456789',0
 buf2   db  12 dup(0)   ; 12 个字节的空间，初值均为 0
.stack 200
.code
start:
   mov  ebx,0
L1:
   mov  al, buf1[ebx]
   mov  buf2[ebx],al
   inc  ebx
   cmp  ebx,12
   jnz  L1
   invoke printf,offset lpFmt,OFFSET buf1
   invoke printf,offset lpFmt,OFFSET buf2
   invoke ExitProcess, 0
end start
```

4.5 阅读下列程序，指出程序的功能，指出访问存储单元时使用的寻址方式，程序中的寄存器各自的功能分配（作用）。

```
……; code 段之上的内容同 4.4。
.code
 start:
   mov  esi, offset buf1
   mov  edi, offset buf2
   mov  ecx,0
L1:
   mov  eax, [esi]
   mov  [edi],eax
   add  esi, 4
   add  edi, 4
   add  ecx, 4
   cmp  ecx,12
   jnz  L1
   invoke printf,offset lpFmt,offset buf1
   invoke printf,offset lpFmt,offset buf2
   invoke ExitProcess, 0
end start
```

4.6 阅读下列程序，指出程序的功能，并指出访问存储单元时使用的寻址方式，程序中的寄存器各自的功能分配（作用）。

```
……; code 段之上的内容同 4.4。
.code
 start:
   mov  esi, offset buf1
   mov  edi, offset buf2
   mov  ecx,0
L1:
   mov  eax, [esi][ecx*4]
   mov  [edi][ecx*4],eax
   inc  ecx
   cmp  ecx,3
   jnz  L1
   invoke printf,offset lpFmt, offset buf1
   invoke printf,offset lpFmt, offset buf2
   invoke ExitProcess, 0
end start
```

4.7 阅读下列程序，并指出程序的功能，程序中的寄存器各自的功能分配（作用）。

```
……; code 段之上的内容同 4.4。
.code
 start:
   mov  ecx,0
L1:
   mov  eax, dword ptr buf1 [ecx*4]
   mov  dword ptr buf2 [ecx*4],eax
   inc  ecx
   cmp  ecx,3
   jnz  L1
   invoke printf,offset lpFmt, offset buf1
   invoke printf,offset lpFmt, offset buf2
   invoke ExitProcess, 0
end start
```

4.8 阅读下列程序，并指出程序显示的结果是什么。

```
……; code 段之上的内容同 4.4。
.code
 start:
```

```
    mov  ecx,0
L1:
    mov  al, buf1
    mov  buf2, AL
    inc  buf1
    inc  buf2
    inc  ecx
    cmp  ecx,3
    jnz  L1
    invoke printf,offset lpFmt, offset buf1
    invoke printf,offset lpFmt, offset buf2
    invoke ExitProcess, 0
end start
```

上机实践 4

1、编写如下 C 语言程序，用反汇编的方式观察对各个变量的访问形式。

```
int  x;
int  y[10];
int  z[10][5];
int  *p;
char c1;
char c2[10];
char *cp;
x=10;
p=&x;
*p=20;
y[5]=30;
*(y+6)=40;
z[5][3]=50;
c1 = 48;
c2[3]= 65;
cp = c2;
*(cp+4)=66;
```

注意，将这些变量定义成全局变量。若定义成局部变量，其对应的地址有所不同。在反汇编窗口，不要勾选显示符号地址。