PC汇编语言

Paul A. Carter

翻译:伍星

2007 年 3 月 4 日

目录

前言

目的

这本书的目的是为了让读者更好地理解计算机在相比于编程语言如Pascal的更底层如何工作。通过更深刻地了解计算机如何工作，读者通常可以更有能力用高级语言如C和C++来开发软件。学习用汇编语言来编程是达到这个目的的一个极好的方法。其它的PC汇编程序的书仍然在讲授着如何在1981年使用在初始的PC机上的8086处理器上进行编程!那时的8086处理器只支持实模式。在这种模式下，任何程序都可以寻址任意内存或访问计算机里的任意设备。这种模式不适合于安全，多任务操作系统。这本书改为叙述在80386和后来的处理器如何在*保护*模式(也就是Windows和Linux运行的模式)下进行编程。这种模式支持现在操作系统所期望的特征，比如：虚拟内存和内存保护。使用保护模式有以下几个原因：

1. 在保护模式下编程比在其它书使用的8086实模式下要容易。
2. 所有的现代的PC操作系统都运行在保护模式下。
3. 可以获得运行在此模式下的免费软件。

关于保护模式下的PC汇编编程的书籍的缺乏是作者书写这本书的主要原因。

就像上面所提到的，这本书使用了免费/开源的软件：也就是，NASM汇编器和DJGPP C/C++编译器。它们都可以在因特网上下载到。本书同样讨论如何在Linux操作系统下和在Windows下的Borland和Microsoft的C/C++编译器中如何使用NASM汇编代码。所有这些平台上的例子都可以在我的网页上找到：`http://www.drpaulcarter.com/pcasm`. 如何你想汇编和运行这本教程上的例子，你*必须*下载这些样例代码。

注意这本书并不打算涵盖汇编编程的各个方面。作者尽可能地涉及了*所有*程序员都应该熟悉的最重要的方面。

致谢

作者要感谢世界上许多为免费/开源运动作出贡献的程序员。这本书

的程序甚至是这本书本身都是使用免费软件来书写的。作者特别要感谢John S. Fine，Simon Tatham，Julian Hall和其它开发NASM汇编器的人，这本书的所有例子都基于这个汇编器；开发DJGPP C/C++DJ编译器的Delorie；众多对DJGPP所基于的GNU gcc编译器有贡献的人们；Donald Knuth和其它开发TEX 和LATEX 2ε 排版语言的人，也正是用于书写这本书的语言；Richard Stallman (免费软件基金会的创立者)，Linus Torvalds (Linux内核的创建者)和其它作者用来书写这本书的底层软件的创建者。

谢谢以下的人提出的修改意见：

- John S. Fine
- Marcelo Henrique Pinto de Almeida
- Sam Hopkins
- Nick D'Imperio
- Jeremiah Lawrence
- Ed Beroset
- Jerry Gembarowski
- Ziqiang Peng
- Eno Compton
- Josh I Cates
- Mik Mifflin
- Luke Wallis
- Gaku Ueda
- Brian Heward
- Chad Gorshing
- F. Gotti
- Bob Wilkinson
- Markus Koegel
- Louis Taber
- Dave Kiddell
- Eduardo Horowitz

- Sébastien Le Ray
- Nehal Mistry
- Jianyue Wang
- Jeremias Kleer
- Marc Janicki

因特网上的资源

作者的网页	http://www.drpaulcarter.com/
NASM源代码的网页	http://sourceforge.net/projects/nasm/
DJGPP	http://www.delorie.com/djgpp
Linux汇编器	http://www.linuxassembly.org/
汇编的艺术(The Art of Assembly)	http://webster.cs.ucr.edu/
USENET	comp.lang.asm.x86
Intel文件	http://developer.intel.com/design /Pentium4/documentation.htm

反馈

作者欢迎任意关于这本书的反馈信息。.

E-mail: pacman128@gmail.com
WWW: http://www.drpaulcarter.com/pcasm

第1章

简介

1.1 数制

计算机里的内存由数字组成。计算机内存并没有以十进制(基数为10)来储存这些数字。因为计算机以二进制(基数为2)格式来储存所有信息能极大地简化硬件。首先让我们来回顾一下十进制数制。

1.1.1 十进制

基数为10的数制由10个数码(0-9)组成。一个数的每一位有基于它在数中的位置相关联的10的乘方值。例如：

$$234 = 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

1.1.2 二进制

基数为2的数制由2个数码(0和1)组成。一个数的每一位有基于它在数中的位置相关联的2的乘方值。(一个二进制数位被称为一个比特位。)例如：

$$\begin{aligned} 11001_2 &= 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 16 + 8 + 1 \\ &= 25 \end{aligned}$$

这些演示了二进制如何转换成十进制。表 1.1展示了开始的几个十进制数如何以二进制替代。

图 1.1 演示单个的二进制数字(*也就是*: 位)相加。下面是一个例子：

$$\begin{array}{r} 11011_2 \\ +10001_2 \\ \hline 101100_2 \end{array}$$

如果有人考虑了下面的十进制的除法：

$$1234 \div 10 = 123\ r\ 4$$

十进制	二进制		十进制	二进制
0	0000		8	1000
1	0001		9	1001
2	0010		10	1010
3	0011		11	1011
4	0100		12	1100
5	0101		13	1101
6	0110		14	1110
7	0111		15	1111

表 1.1: 在二进制中十进制0到15的表示

以前无进位				以前有进位			
0	0	1	1	0	0	1	1
+0	+1	+0	+1	+0	+1	+0	+1
0	1	1	0	1	0	0	1
			c		c	c	c

图 1.1: 二进制加法(c代表*进位*)

他可以看到这个除法除去了这个数的最右边的十进制数而且将其它的十进制数向右移动了一位。除以2也是执行同样的操作，除了是为了得到一个数的二进制位外。考虑下面二进制数的除法[1]：

$$1101_2 \div 10_2 = 110_2\ r\ 1$$

这个现象可以用来将一个十进制转换成它的等价的二进制表示形式，像图 1.2展示的一样。这种方法首先找到最右边的数位，这个数位被称为*最低的有效位* (lsb)。最左边的数位称为*最高的有效位* (msb)。内存的基本单元由8位组成，称它为一个*字节*。

1.1.3 十六进制

十六进制数使用的基数为16。十六进制(或者简短称为*hex*)可以用作二进制数的速记形式。十六进制需要16个数码。这就产生了一个问题，因为没有符号可以用来表示在9之后的额外的数字。通过协定，字母被用来表示这些额外的数字。这16个十六进制数字是0-9，然后A，B，C，D，E和F。数A等价于十进制的10，B是11，等等。一个十六进制的每一位有基于它在数中的位置相关联的16的乘方值。例如：

$$2\mathrm{BD}_{16} = 2 \times 16^2 + 11 \times 16^1 + 13 \times 16^0$$

[1]2这个下标是用来表明这个数字是以二进制表示，而不是十进制

Decimal	Binary
$25 \div 2 = 12\ r\ 1$	$11001 \div 10 = 1100\ r\ 1$
$12 \div 2 = 6\ r\ 0$	$1100 \div 10 = 110\ r\ 0$
$6 \div 2 = 3\ r\ 0$	$110 \div 10 = 11\ r\ 0$
$3 \div 2 = 1\ r\ 1$	$11 \div 10 = 1\ r\ 1$
$1 \div 2 = 0\ r\ 1$	$1 \div 10 = 0\ r\ 1$

因此$25_{10} = 11001_2$

图 1.2: 十进制转换

$$\begin{aligned} 589 \div 16 &= 36\ r\ 13 \\ 36 \div 16 &= 2\ r\ 4 \\ 2 \div 16 &= 0\ r\ 2 \end{aligned}$$

因此$589 = 24\mathrm{D}_{16}$

图 1.3:

$$\begin{aligned} &= 512 + 176 + 13 \\ &= 701 \end{aligned}$$

将十进制转换成十六进制，可以使用和二进制转换同样的方法，除了除以16外。看例子图 1.3。

十六进制非常有用的原因是因为十六进制和二进制之间转换有一个非常简单的方法。二进制数非常大而且非常繁锁。十六进制提供一个比较舒服的方法来表示二进制数。

将一个十六进制数转换成二进制数，只需要简单地将每一位十六进制数转换成4位二进制数。例如：$24\mathrm{D}_{16}$转换成$0010\ 0100\ 1101_2$。注意在这些4位二进制数中领头的0非常重要！如果$24\mathrm{D}_{16}$中间的那位的4位二进制数的领头的0没用使用的话，那么结果就是错的。从二进制转换成十六进制同样简单。只需反过来做刚才那个处理，将二进制每4位一段转换成十六进制。从二进制数的最右端开始，而不是最左端。这样就能保证处理过程使用了正确的4位段[2]. 例如:

[2]如果不明白起点为什么是那样，那么换过来，试着将这个例子从左边开始转换。

word(字)	2个字节
double word(双字)	4个字节
quad word(四字)	8个字节
paragraph(一节)	16个字节

表 1.2: 内存单元

$$\begin{array}{cccccc} 110 & 0000 & 0101 & 1010 & 0111 & 1110_2 \\ 6 & 0 & 5 & A & 7 & E_{16} \end{array}$$

一个四位的数被称为*半字节*。因此每一位十六进制相当于一个半字节。两个半字节为一个字节，所以一个字节可以用两位十六进制数来表示。一个字节值的范围以二进制表示为0到11111111，以十六进制表示为0到FF，以十进制表示为0到255。

1.2 计算机结构

1.2.1 内存

内存以千字节(2^{10} = 1024字节)，兆字节(2^{20} = 1048576 字节)和十亿位元组(2^{30} = 1073741824 字节)来测量。

内存的基本单元是一个字节。一台有32兆内存的电脑大概能容纳3200万字节的信息。在内存里的每一个字节通过一个唯一的数字来标识作为它的地址像图 1.4展示的一样。

Address	0	1	2	3	4	5	6	7
Memory	2A	45	B8	20	8F	CD	12	2E

图 1.4: 内存地址

通常内存都是大块大块地使用而不是单个字节。在PC机结构中，命名了这些内存大块像表 1.2展示的一样。

在内存里的数据都是数字的。字符通过用数字来表示字符的*字符编码*来储存。其中一个最普遍的字符编码称为*ASCII*(美国信息互换标准编码)。一个新的，更完全的，正在替代ASCII的编码是Unicode。在这两种编码中最主要的区别是ASCII使用一个字节来编码一个字符，但是Unicode每个字符使用两个字节(或一个字)。例如：ASCII使用41_{16} (65_{10})来表示字符大写A；Unicode使用0041_{16}来表示。因为ASCII使用一个字节，所以它仅能表示256种不同的字符[3]。Unicode将ASCII的值扩展成一个字，允许表示更多的字符。这对于表示全世界所有的语言非常重要。

[3]事实上，ASCII仅仅使用低7位，所以只有128种不同的值可以使用。

1.2.2 CPU

中央处理器(CPU)是执行指令的物理设备。CPU 执行的指令通常非常简单。指令可能要求它们使用的数据存储在一个CPU称为*寄存器*的特殊的储存位置中。CPU可以比访问内存更快地访问寄存器里的数据。然而，在CPU里的寄存器是有限的，所以程序员必须注意只保存现在使用的数据到寄存器中。

各类CPU执行的指令组成了该CPU的*机器语言*。机器语言拥有比高级语言更基本的结构。机器语言指令被编码成未加工的数字，而不是友好的文本格式。为了更有效的运行，CPU必须能很快地解译一个指令的目的。机器语言就是为了这个目的设计的，而不是让人们更容易理解而设计。一个其它语言写的程序必须转换成CPU的本地机器语言，才能在电脑上运行。*编译器* 是一个将用程序语言写的程序翻译成特殊结构的电脑的机器语言的程序。通常，每一种类型的CPU都有它自己唯一的机器语言。这是为什么为Mac 写的程序不能在IBM类型PC机运行的一个原因。

电脑通过使用*时钟* 来同步指令的执行。时钟脉冲在一个固定的频率(称为*时钟频率*)。当你买了一台1.5 GHz 的电脑，1.5 GHz 就是时钟频率[4]。时钟并不记录分和秒。它以不变的速率简单跳动。电子计算机通过使用这个跳动来正确执行它们的操作，就像节拍器的跳动如何来帮助你以正确的节奏播放音乐。一个指令需要跳动的次数(或就像他们经常说的执行*周期*)依赖CPU的产生和模仿。周期的次数取决于在它之前的指令和其它因素。

GHz*代表十万万赫或是每秒十亿次循环。1.5 GHz的CPU每秒有15亿的时钟脉冲。*

1.2.3 CPU 80x86系列

IBM型号的PC机包含了一个来自Intel 80x86家族(或它的克隆)的CPU。在这个家族的所有CPU都有一些普遍的特征，包括有一种基本的机器语言。无论如何，最近的成员极大地加强了这个特征。

8088，8086: 这些CPU从编程的观点来看是完全相同的。它们是用在早期PC机上的CPU。它们提供一些16位的寄存器：AX，BX，CX，DX，SI，DI，BP，SP，CS，DS，SS，ES，IP，FLAGS。它们仅仅支持1M字节的内存，而且只能工作在*实模式*下。在这种模式下，一个程序可以访问任何内存地址，甚至其它程序的内存！这会使排除故障和保证安全变得非常困难！而且，程序的内存需要分成*段*。每段不能大于64K。

80286: 这种CPU使用在AT系列的PC机中。它在8088/86的基本机器语言中加入了一些新的指令。然而，它主要的新的特征是*16位保护模式*。在这种模式下，它可以访问16M字节的内存和通过阻止访问其它程序的内存来保护程序。可是，程序依然是分成不能大于64K的段。

[4]实际上，时钟脉冲使用在许多不同的CPU组件中。其它组件通常使用与CPU不同的时钟频率。

图 1.5: AX寄存器

80386: 这种CPU极大地增强了80286的性能。首先，它扩展了许多寄存器来容纳32位数据(EAX，EBX，ECX，EDX，ESI，EDI，EBP，ESP，EIP)而且增加了两个新的16位寄存器（FS，GS）。它同样增加了一个新的*32位保护模式*。在这种模式下，它可以访问4G字节。程序同样分成段，但是现在每段大小同样可以到4G。

80486/Pentium/Pentium Pro: 这些80x86家族的成员增加了不多的新的特征。它们主要是提高了指令执行的速度。

Pentium MMX: 这些处理器在Pentium基础上增加了MMX指令(多媒体扩展)。这些指令可以提高普通的图像操作的速率。

Pentium II: 它是拥有MMX 指令的Pentium处理器。(Pentium III 本质上就是一个更快的Pentium II。)

1.2.4 8086 16位寄存器

最初的8086CPU提供4个16位通用寄存器：AX，BX，CX 和DX。这些寄存器都可以分解成两个8位寄存器。例如：AX寄存器可以分解成AH和AL寄存器，像图 1.5展示的一样。AH寄存器包含AX的上(或高)8位，而AL包含AX的低8位。通常AH和AL都当做独立的一个字节的寄存器来用；但是，清楚它们不能独立于AX是非常重要的。改变AX的值将会改变AH和AL的值反之亦然。通用寄存器多数使用在数据移动和算术指令中。

这有两个16位指针寄存器：SI 和DI 。通常它们都是当作指针来使用，但是在许多情况下也可以像通用寄存器一样使用。但是，它们不可以分解成8位寄存器。

16位BP和SP 寄存器用来指向机器语言堆栈里的数据，被各自称为基址寄存器 和堆栈指针寄存器。这些将在以后讨论。

16位CS，DS，SS 和ES 寄存器是*段寄存器*。 它们指出程序不同部分所使用的内存。CS代表代码段，DS 代表数据段，SS 代表堆栈段和ES代表附加段。ES当作一个暂时段寄存器来使用。这些寄存器的细节描述在小节 1.2.6 和1.2.7中。

指令指针寄存器(IP) 与CS寄存器一起使用来跟踪CPU下一条执行指令的地址。通常，当一条指令执行时，IP提前指向内存里的下一条指令。

FLAGS寄存器储存了前面指令执行结果的重要信息。这些结果在寄存器里以单个的位储存。例如：如果前面指令执行结果是0，Z位为1，反之为0。并不是所有指令都修改FLAGS里的位，查看附录里的表看单个指令是如何影响FLAGS寄存器的。

1.2.5 80386 32位寄存器

80386及以后的处理器扩展了寄存器。例如：16位AX寄存器扩展成了32位。为了向后兼容，AX依然表示16位寄存器而EAX 用来表示扩展的32位寄存器。AX是EAX 的低16位就像AL是AX(EAX)的低8位一样。但是没有直接访问EAX 高16位的方法。其它的扩展寄存器是EBX，ECX，EDX，ESI 和EDI 。

许多其它类型的寄存器同样也扩展了。BP变成了EBP；SP 变成了ESP；FLAGS变成了EFLAGSEFLAGS 而IP变成了EIP。但是，不同于指针寄存器和通用寄存器，在32位保护模式下(下面将讨论的)只有这此寄存器的扩展形式被使用。

在80386里，段寄存器依然是16位的。这儿有两个新的段寄存器：FS和GS。它们名字并不代表什么。它们是附加段寄存器(像ES一样)。

术语中字的一个定义为CPU数据寄存器的大小。对于80x86家族，这个术语现在有一点混乱了。在表 1.2里，可以看到字被定义成两个字节。它是当8086第一次发行时被定义成这样的。当80386开发出来后，它被决定依旧保持这个字定义不改变，即使寄存器的大小已经改变了。

1.2.6 实模式

在实模式下，内存被限制为仅有1M字节(2^{20} 字节)。有效的地址从00000到FFFFF (十六进制)。 这些地址需要用20位的数来表示。显然，一个20位的数不适合任何一个8086的16位寄存器。Intel通过利用两个16位数值来决定一个地址的方法来解决这个问题。开始的16位值称为*段地址*(selector)。段地址的值必须存储在段寄存器中。第二个16位值称为*偏移地址*(offset)。用32位*段地址：偏移地址*表示的物理地址可以由下面的公式计算：

那么，无耻的DOS640K限制来自哪里？BIOS为它的代码和硬件设备如显示器要求了1M内存中的一些内存。

$$16 * \text{selector} + \text{offset}$$

在十六进制中乘以16是非常容易的，只需要在数的右边加0。例如：表示为047C:0048的物理地址通过这样得到：

$$\begin{array}{r} 047C0 \\ +0048 \\ \hline 04808 \end{array}$$

实际上，段地址的值是一节的首地址(看表 1.2)。

真实的段地址有以下的缺点：

- 一个段地址只能指向64K内存(16位偏移的上限)。如果一个程序拥有大于64K的代码那又怎么办呢？在CS里的一个单一的值不能满足整个程序执行的需要。程序必须分成小于64K的段(*segment*)。当执行从一段移到另一段时，CS里的值必须改变。同样的问题发生在大量的数据和DS 寄存器之间。这样使用是非常不方便的！

- 每个字节在内存里并不只有唯一的段地址。物理地址04808可以表示为：047C:0048，047D:0038，047E:0028 或047B:0058。 这将使段地址的比较变得复杂。

1.2.7 16位保护模式

在80286的16位保护模式下，段地址的值与实模式相比解释得完全不同。在实模式下，一个段地址的值是物理内存里的一节的首地址。在保护模式下，一个段地址的值是一个指向*描述符表*的*指针*。两种模式下，程序都是被分成段。在实模式下，这些段在物理内存的固定位置而且段地址的值表示段开始处所在节的首地址。在保护模式下，这些段不是在物理内存的固定的地址。事实上，它们根本不一定需要在内存中。

保护模式使用了一种叫做*虚拟内存* 的技术。虚拟内存的基本思想是仅仅保存程序现在正在使用的代码和数据到内存中。其它数据和代码暂时储存在硬盘中直到它们再次需要时。当一段从硬盘重新回到内存中，它很有可能放在不同于它移动到硬盘之前时的位置的内存中。所有这些都由操作系统透明地执行。程序并不需要因为要让虚拟内存工作而使用不同的书写方法。

在保护模式下，每一段都分配了一条描述符表里的条目。这个条目拥有系统想知道的关于这段的所有信息。这些信息包括：现在是否在内存中；如果在内存中，在哪；访问权限(*例如：* 只读)。段的条目的指针是储存在段寄存器里的段地址值。

一个非常著名的PC专家称286CPU为“死了的大脑”

16位保护模式的一个大的缺点是偏移地址依然是16位数。这个的后果就是段的大小依然限制为最大64K。这会导致使用大的数组时会有问题。

1.2.8 32位保护模式

80386引入了32位保护模式。386 32位保护模式和286 16位保护模式之间最主要的区别是：

1. 偏移地址扩展成了32位。这就允许偏移地址范围升至4G。因此，段的大小也升至4G。

2. 段可以分成较小的4K大小的单元，称为*内存页*。虚拟内存系统工作在页的方式下，代替了段方式。这就意味着一段在任何一个时刻只有部分可能在内存中。在28616位保护模式下，要么整个段在内存中，要么整个不在。这样在32位模式下允许的大的段的情况下很不实用。

在Windows 3.x系统中，*标准模式*为286 16位保护模式而*增强模式*为32位保护模式。Windows 9X，Windows NT/2000/XP，OS/2和Linux都运行在分页管理的32位保护模式下。

1.2.9 中断

有时候普通的程序流必须可以被要求快速反应的处理事件中断。电脑提供了一个称为*中断*的结构来处理这些事件。例如：当一个鼠标移动了，硬件鼠标中断现在的程序来处理鼠标移动(移动鼠标，等等)。中断导致控制权转移到一个*中断处理程序*。中断处理程序是处理中断的程序。每种类型的中断都分配了一个中断号。在物理内存的开始处，存在一张包含中断处理程序段地址的*中断向量* 表。中断号是这张表中最基本的指针。

外部中断由CPU的外部引起。(鼠标就是这一类型的例子。)许多I/O设备引起中断(*例如*：键盘，时钟，硬盘驱动器，CD-ROM和声卡)。内部中断由CPU的内部引起，要么是由一个错误引起，要么由中断指令引起。错误中断称为*陷阱*。由中断指令引起的中断称为*软中断*。DOS使用这些类型的中断来实现它的API(应用程序接口)。许多现代的操作系统(如：Windows和UNIX)使用一个基于C的接口。[5]

许多中断处理程序当它执行完成时，将控制权返回给被中断的程序。它们恢复寄存器，里面的值与中断发生之前的值相同。因此，被中断的程序就像没有任何事情发生一样运行(除了它失去了一些CPU周期)。陷阱通常不返回。通常它们中止程序。

1.3 汇编语言

1.3.1 机器语言

每种类型的CPU都能理解它们自己的机器语言。机器语言里的指令是以字节形式在内存中储存的数字。每条指令有它唯一的数字码称为*操作代码*，或简称为*操作码* 。80x86处理器的指令大小不同。操作码通常是在指令的开始处。许多指令还包含指令使用的数据(*例如*：常量或地址)。

机器语言很难直接进行编程。解译这些数字代码指令的意思对人类来说是沉闷的。例如：执行将EAX 和EBX 寄存器相加然后将结果送回到EAX的指令以十六进制码编译如下：

```
03 C3
```

这个很难理解。幸运的是，一个叫做*汇编的程序* 可以为程序员做这个沉闷的工作。

1.3.2 汇编语言

一个汇编语言程序以文本格式储存(正如一个高级语言程序)。每条汇编指令代表确切的一条机器指令。例如：上面描述的加法指令在汇编语言中将描述成：

```
add eax, ebx
```

[5] 然而，它们在内核级可能会使用一个更低等级的接口。

这里指令的意思比在机器代码表示得更清楚。代码add是加法指令的*助记符*。一条汇编指令的通常格式为：

mnemonic(助记符) operand(s)(操作数)

它花费了电脑科学家几年的时间来揣测如何编写一个编译器!

汇编程序 是一个读包含汇编指令的文本文件和将汇编语言转换成机器代码的程序。*编译器* 是为高级编程语言做同样转换的程序。一个汇编程序比一个编译器要简单。每条汇编语句接代表一个唯一的机器指令。高级语言更复杂而且可能要求更多的机器指令。

汇编和高级语言之间另一个更重要的区别是因为每种不同类型的CPU有它自己的机器代码，所以它同样有它自己的汇编语言。在不同的电脑构造中移植汇编语言比高级语言要困难得多。

这本书使用了Netwide Assembler，或简称为NASM 。它在Internet上是免费提供的(要得到URL，请看前言)。更普遍的汇编程序是Microsoft Assembler(MASM) 或Borland Assembler (TASM)。MASM/TASM和NASM之间有一些汇编语法区别。

1.3.3 指令操作数

机器代码指令拥有个数和类型不同的操作数；然而，通常每个指令有几个固定的操作数(0到3个)。操作数可以有下面的类型：

寄存器: 这些操作数直接指向CPU寄存器里的内容。

内存: 这些操作数指向内存里的数据。数据的地址可能是硬编码到指令里的常量或可能直接使用寄存器的值计算得到。距离段的起始地址的偏移值即为此地址。

立即数: 这些操作数是指令本身列出的固定的值。它们储存在指令本身(在代码段)，而不在数据段。

暗指的操作数: 这些操作数没有明确显示。例如：往寄存器或内存增加1的加法指令。1是暗指的。

1.3.4 基本指令

最基本指令是MOV 指令。它将数据从一个地方移到另一个地方(像高级语言里面的赋值操作一样)。它携带两个操作数：

mov *dest (目的操作数), src(源操作数)*

*src*指定的数据拷贝到了*dest*。一个指令的两个操作数不能同时是内存操作数。这就指出了一个汇编古怪的地方。通常，对于各种各样指令的使用都有某些强制性的规定。操作数必须是同样的大小。AX里的值就不能储存到BL 里去。

这儿有一个例子(分号表示注释的开始)：

```
mov     eax, 3    ; 将3存入 EAX 寄存器(3是一个立即数)。
mov     bx, ax    ; 将AX的值存入到BX寄存器。
```

ADD 指令用来进行整形数据的相加。

```
add     eax, 4    ; eax = eax + 4
add     al, ah    ; al = al + ah
```

SUB 指令用来进行整形数据的相减。

```
sub     bx, 10    ; bx = bx - 10
sub     ebx, edi  ; ebx = ebx - edi
```

INC 和DEC 指令将值加1或减1。因为1是一个暗指的操作数，INC 和DEC的机器代码比等价的ADD和SUB指令要少。

```
inc     ecx       ; ecx++
dec     dl        ; dl--
```

1.3.5 指示符

*指示符*是由汇编程序产生的而不是由CPU产生。它们通常用来要么指示汇编程序做什么要么提示汇编程序什么。它们并不翻译成机器代码。指示符普遍的应用有：

- 定义常量
- 定义用来储存数据的内存
- 将内存组合成段
- 有条件地包含源代码
- 包含其它文件

NASM代码像C一样要通过一个预处理程序。它拥有许多和C一样的预处理程序。但是，NASM 的预处理的指示符以%开头而不是像C一样以#开头。

equ 指示符

equ指示符可以用来定义一个*符号*。符号被命名为可以在汇编程序里使用的常量。格式是：

symbol equ *value*

符号的值以后不可以再定义。

单位	字母
字节	B
字	W
双字	D
四字	Q
十个字节	T

表 1.3: RESX和DX指示符的字母

%define 指示符

这个指示符和C中的#define非常相似。它通常用来定义一个宏常量，像在C里面一样。

```
%define SIZE 100
      mov     eax, SIZE
```

上面的代码定义了一个称为SIZE的宏通过使用一个MOV指令。宏在两个方面比符号要灵活。宏可以被再次定义而且可以定义比简单的常量数值更大的值。

数据指示符

数据指示符使用在数据段中用来定义内存空间。保留内存有两种方法。第一种方法仅仅为数据定义空间；第二种方法在定义数据空间的同时给与一个初始值。第一种方法使用RES*X*指示符中的一个。*X*可由字母替代，字母由需要储存的对象的大小来决定。表 1.3给出了可能的值。

第二种方法(同时定义一个初始值)使用D*X*指示符中的一个。*X*可以由字母替代，字母的值与RES*X*里的值一样。

使用变量 来标记内存位置是非常普遍的。变量使得在代码中指向内存位置变得容易。下面是几个例子：

```
L1     db      0          ;字节变量L1，初始值为0
L2     dw      1000       ;字变量L2，初始值为1000
L3     db      110101b    ;字节变量初始化成110101(十进制为53)
L4     db      12h        ;字节变量初始化成十六进制12(在十进制中为18)
L5     db      17o        ;字节变量初始化成八进制17(在十进制中为15)
L6     dd      1A92h      ;双字变量初始化成十六进制1A92
L7     resb    1          ;1个未初始化的字节
L8     db      "A"        ;字节变量初始化成ASCII值A(65)
```

双引号和单引号被同等对待。连续定义的数据储存在连续的内存中。也就是说，字L2就储存在L1的后面。内存的顺序可以同样被定义。

```
L9    db     0, 1, 2, 3              ; 定义4个字节
L10   db     "w", "o", "r", 'd', 0   ; 定义一个等于"word"的C字符串
L11   db     'word', 0               ; 等同于L10
```

指示符DD可以用来定义整形和单精度的浮点数常量[6]。但是，DQ指示符仅仅可以用来定义双精度的数常量。

对于大的序列，NASM 的TIMES 指示符常常非常有用。这个指示符每次都重复它的操作对象一个指定的次数。例如：

```
L12   times 100 db 0                 ; 等价于100个值为0的字节
L13   resw   100                     ; 储存空间为100个字
```

记住变量可以用来表示代码中的数据。变量的使用方法有两种。如果一个平常的变量被使用了，它被解释为数据的地址(或偏移)。如果变量被放置在方括号([])中，它就被解释为在这个地址中的数据。换句话说，你必须把变量当作一个指向数据的*指针*而方括号引用这个指针就像*号在C中一样。(MASM/TASM使用的是另外一个惯例。)在32位模式下，地址是32位。这儿有几个例子：

```
1       mov    al, [L1]        ; 复制L1里的字节数据到AL
2       mov    eax, L1         ; EAX = 字节变量L1代表的地址
3       mov    [L1], ah        ; 把AH拷贝到字节变量L1
4       mov    eax, [L6]       ; 复制L6里的双字数据到 EAX
5       add    eax, [L6]       ; EAX = EAX + L6里的双字数据
6       add    [L6], eax       ; L6 = L6里的双字数据 + EAX
7       mov    al, [L6]        ; 拷贝L6里的数据的第一个字节到AL
```

例子的第7行展示了NASM 一个重要性能。汇编程序并不 保持跟踪变量的数据类型。它由程序员来决定来保证他(或她)正确使用了一个变量。随后它一般将数据的地址储存到寄存器中，然后像在C中一样把寄存器当一个指针变量来使用。同样，没有检查使得指针能正确使用。以这种方式，汇编程序跟C相比有更易出错的倾向。

考虑下面的指令：

```
mov    [L6], 1              ; 储存1到L6中
```

这条语句产生一个operation size not specified(操作大小没有指定)的错误。为什么？因为汇编程序不知道是把1当作一个字节，还是字，或是双字来储存。为了修正这个，加一个大小指定：

```
mov    dword [L6], 1        ; 储存1到L6中
```

这个告诉汇编程序把1储存在从L6开始的双字中。另一些大小指定为：BYTE(字节)，WORD(字)，QWORD(四字)和TWORD(十字节)。[7]

[6] 单精度浮点数等价于C里的的float变量

[7] TWORD定义了十个字节大小的内存。浮点数协处理器使用了这种类型的数据

print_int	在屏幕上显示出储存在EAX 中的整形值
print_char	在屏幕上显示出以ASCII形式储存在AL中的字符
print_string	在屏幕上显示储存在EAX 里的*地址*指向的字符串的内容。这个字符串必须是C类型的字符串，(*也就是*:以null结束的字符串)。
print_nl	在屏幕上显示换行。
read_int	从键盘上读入一整形数据然而把它储存到EAX 寄存器。
read_char	从键盘读入一单个字符然而把它以ASCII形式储存到EAX 寄存器。

表 1.4: 汇编的I/O程序

1.3.6 输入和输出

输入和输出是真正系统依赖的活力。它牵涉到系统硬件的接口问题。高级语言，像C，提供了标准的，简单的，统一的程序I/O接口的程序库。汇编语言不提供标准库。它们必须要么直接访问硬件(在保护模式下为特权级操作)或使用任何操作系统提供的底层的程序。

汇编程序与C交互使用是非常普遍的。这样做的一个优点是汇编代码可以使用标准C I/O程序库。但是，你必须清楚C使用的程序之间传递信息的规则。这些规则放在这会非常麻烦。(它们将在以后提到！)为了简单化I/O，作者已经开发出了隐藏在复杂C规则里的自己的程序，而且提供了一个更简单的接口。表 1.4描述了提供的程序。所有这些程序保留了所有寄存器的值，除了读的程序外。这些程序确实修改了EAX 的值。为了使用这些程序，你必须包含一个汇编程序需要用到的信息的文件。为了在NASM中包含一个文件，你可以使用`%include`预处理指示符。下面几行包含了有作者的I/O程序的文件[8]:

```
%include "asm_io.inc"
```

为了使用一个打印程序，你必须加载正确的值到EAX 中，然后用`CALL`指令调用它。`CALL`指令等价于在高级语言里的函数call。它跳转到代码的另一段去执行，然后等程序执行完成后又回到原始的地方。下面的程序例子展示了调用这些I/O程序的几个样例。

1.3.7 调试

作者的库同样包含一些有用的调试程序。这些调试程序显示关于系统状态的信息但不改变它们。这些程序是一些保存CPU的当前状态后执行一个子程序调用的宏。这些宏定义在上面提到的`asm_io.inc`文件中。宏可以像普通的指令一样使用。宏的操作数由逗号隔开。

[8] `asm_io.inc` (和`asm_io.inc`需要的`asm_io`目标文件)在例子代码中，可以从这个指南的网页中下载到：`http://www.drpaulcarter.com/pcasm`

这儿有四个调试程序称为dump_regs，dump_mem，dump_stack和dump_math；它们分别显示寄存器，内存，堆栈和数字协处理器的值。

dump_regs 这个宏显示系统的寄存器里的值(十六进制)到stdout(*也就是：* 显示器)。它同时显示在FLAGS[9]寄存器里的位。例如，如果零标志位是1，*ZF*是显示的。如果是0，它就不被显示。它携带一个整形参数，这个参数同样被显示出来。这就可以用来区别不同dump_regs命令的输出。

dump_mem 这个宏同样以ASCII字符的形式显示内存区域的值(十六进制)。它带有三个用逗号分开的参数。第一个参数是一个用来标示输出的整形变量(就像dump_regs参数一样)。第二个参数需要显示的内存的地址。(它可以是个标号。)最后一个参数是在此地址后需要显示的16字节的节数。内存显示将从要求的地址之前的第一节的边界开始。

dump_stack 这个宏显示CPU堆栈的值。(这个堆栈将在第4章中提到。)这个堆栈由双字组成，所以这个程序也以这种格式显示它们。它带有三个用逗号隔开的参数。第一个参数是一个整形变量（像dump_regs一样）第二个参数是在EBP寄存器里的地址*下面*需要显示的双字的数目而第三个参数是在EBP寄存器里的地址*上面*需要显示的的数目。

dump_math 这个宏显示数字协处理器寄存器里的值。它只带有一个整形参数，这个参数用来标示输出就像参数dump_regs做的一样。

1.4 创建一个程序

现今，完全用汇编语言写的独立的程序是不经常的。汇编一般用在某些至关重要的程序。为什么？用高级语言来编程比用汇编要简单得多。同样，使用汇编将使得程序移植到另一个平台非常困难。事实上，根本很少使用汇编程序。

那么，为什么任何人都需要学习汇编程序呢？

1. 有时，用编程写的代码比起编译器产生的代码要少而且运行得更快。
2. 汇编允许直接访问系统硬件信息，而这个在高级语言中很难或根本不可能实现。
3. 学习汇编编程将帮助一个人深刻地理解系统如何运行。
4. 学习汇编编程将帮助一个人更好地理解编译器和高级语言像C如何工作。

这两点表明学习汇编是非常有用的，即使在以后的日子里不在编程里用到它。事实上，作者很少用汇编编程，但是他每天都使用来自它的想法。

[9] 第2章将讨论这个寄存器

```
1 int main()
2 {
3   int ret_status ;
4   ret_status = asm_main();
5   return ret_status ;
6 }
```

图 1.6: driver.c代码

1.4.1 第一个程序

在这一节里的初期的程序将全部从图 1.6里的简单C驱动程序开始。它简单地调用另一个称为asm_main的函数。这个是真正意义将用汇编编写的程序。使用C驱动程序有几个优点。首先，这样使C系统正确配置程序在保护模式下运行。所有的段和它们相关的段寄存器将由C初始化。汇编代码不需要为这个担心。其次，C库同样提供给汇编代码使用。作者的I/O程序利用了这个优点。他们使用了C的I/O函数(printf，等)。下面显示了一个简单的汇编程序。

first.asm

```
1  ; 文件: first.asm
2  ; 第一个汇编程序。这个程序总共需要两个整形变量作为输入然后输出它们的和。
3  ;
4  ;
5  ; 利用 djgpp 创建执行文件:
6  ; nasm -f coff first.asm
7  ; gcc -o first first.o driver.c asm_io.o
8
9  %include "asm_io.inc"
10 ;
11 ; 初始化放入到数据段里的数据
12 ;
13 segment .data
14 ;
15 ; 这些变量指向用来输出的字符串
16 ;
17 prompt1 db    "Enter a number: ", 0       ; 不要忘记空结束符
18 prompt2 db    "Enter another number: ", 0
19 outmsg1 db    "You entered ", 0
20 outmsg2 db    " and ", 0
21 outmsg3 db    ", the sum of these is ", 0
22
23 ;
```

```
24  ; 初始化放入到.bss段里的数据
25  ;
26  segment .bss
27  ;
28  ; 这个变量指向用来储存输入的双字
29  ;
30  input1  resd 1
31  input2  resd 1
32
33  ;
34  ; 代码放入到.text段
35  ;
36  segment .text
37          global  _asm_main
38  _asm_main:
39          enter   0,0               ; 开始运行
40          pusha
41
42          mov     eax, prompt1      ; 输出提示
43          call    print_string
44
45          call    read_int          ; 读整形变量储存到input1
46          mov     [input1], eax     ;
47
48          mov     eax, prompt2      ; 输出提示
49          call    print_string
50
51          call    read_int          ; 读整形变量储存到input2
52          mov     [input2], eax     ;
53
54          mov     eax, [input1]     ; eax = 在input1里的双字
55          add     eax, [input2]     ; eax = eax + 在input2里的双字
56          mov     ebx, eax          ; ebx = eax
57
58          dump_regs 1                ; 输出寄存器值
59          dump_mem  2, outmsg1, 1    ; 输出内存
60  ;
61  ; 下面分几步输出结果信息
62  ;
63          mov     eax, outmsg1
64          call    print_string      ; 输出第一条信息
65          mov     eax, [input1]
```

```
66          call    print_int           ; 输出input1
67          mov     eax, outmsg2
68          call    print_string        ; 输出第二条信息
69          mov     eax, [input2]
70          call    print_int           ; 输出input2
71          mov     eax, outmsg3
72          call    print_string        ; 输出第三条信息
73          mov     eax, ebx
74          call    print_int           ; 输出总数(ebx)
75          call    print_nl            ; 换行
76
77          popa
78          mov     eax, 0              ; 回到C中
79          leave
80          ret
```

first.asm

这个程序的第13行定义了指定储存数据的内存段的部分代码(名称为.data)。只有是初始化的数据才需定义在这个段中。行17到20，声明了几个字符串。它们将通过C库输出，所以必须以*null*字符(ASCII值为0)结束。记住0和'0'有很大的区别。

不初始化的数据需声明在bss 段(名为.bss，在26行)。这个段的名字来自于早期的基于UNIX汇编运算符，意思是"由符号开始的块。"这同样会有一个堆栈段。它将在以后讨论。

代码段根据惯例被命名为.text。它是放置指令的地方。注意主程序(38行)的代码标号有一个下划线前缀。这个是*在C中称为约定*的一部分。这个约定指定了编译代码时C使用的规则。C和汇编交互使用时，知道这个约定是非常重要的。以后将会将全部约定呈现；但是，现在你只需要知道所有的C编译器里的C符号(*也就是：* 函数和全局变量)有一个附加的下划线前缀。(这个规定是为DOS/Windows指定的，在linux下C 编译器并不为C符号名上加任何东西。)

在37行的全局变量(global)指示符告诉汇编定义_asm_main为全局变量。与C不同的是，变量在缺省情况下只能使用在*内部范围*中。这就意味着只有在同一模块的代码才能使用这个变量。global指示符使指定的变量可以使用在*外部范围*中。这种类型的变量可以被程序里的任意模块访问。asm_io模块声明了全局变量print_int和*et.al.*。这就是为什么在first.asm模块里能使用它们的缘故。

1.4.2 编译器依赖

上面的汇编代码指定为基于GNU[10]的DJGPP C/C++编译器。[11] 这个编译器可以从Internet上免费下载。它要求一个基于386或更好的PC而且需

[10] GNU是一个以免费软件为基础的计划(http://www.fsf.org)

[11] http://www.delorie.com/djgpp

在DOS，Windows 95/98 或NT下运行。这个编译器使用COFF (Common Object File Format，普通目标文件格式)格式的目标文件。为了符合这个格式，`nasm`命令使用`-f coff`选项(就像上面代码注释展示的一样)。最终目标文件的扩展名为`o`。

Linux C编译器同样是一个GNU编译器。 为了转变上面的代码使它能在Linux下运行，只需简单将37到38行里的下划线前缀移除。Linux使用ELF(Executable and Linkable Format，可执行和可连接格式)格式的目标文件。Linux下使用`-f elf`选项。它同样产生一个扩展名为`o`的目标文件。

指定编译器的例子文件可以从作者的网址上得到，它已经修改成能在恰当的编译器上运行了。

Borland C/C++ 是另一个流行的编译器。它使用微软OMF 格式的目标文件。Borland编译器使用`-f obj`选项。目标文件的扩展名将会是`obj`。OMF与其它目标文件格式相比使用了不同的段指示符。数据段（13行）必须改成：

```
segment _DATA public align=4 class=DATA use32
```

bss 段（26）必须改成：

```
segment _BSS public align=4 class=BSS use32
```

text 段（36）必须改成：

```
segment _TEXT public align=1 class=CODE use32
```

必须在36行之前加上一新行：

```
group DGROUP _BSS _DATA
```

微软C/C++编译器可以使用OMF 或Win32格式的目标文件。(如果给出的是OMF格式，它将从内部把信息转变成Win32格式。)Win32允许像DJGPP和Linux一样来定义段。在这个模式下使用`-f win32`选项来输出。目标文件的扩展名将会是`obj`。

1.4.3 汇编代码

第一步是汇编代码。在命令行，键入：

```
nasm -f object-format first.asm
```

*object-format*要么是*coff*，*elf*，*obj*，要么是*win32*，它由使用的C编译器决定。(记住在Linux和Borland下，资源文件同样必须改变。)

1.4.4 编译C代码

使用C编译器编译`driver.c`文件。对于DJGPP ，使用：

```
gcc -c driver.c
```

`-c`选项意味着编译，而不是试图现在连接。同样的选项能使用在Linux，Borland和Microsoft编译器上。

1.4.5 连接目标文件

连接是一个将在目标文件和库文件里的机器代码和数据结合到一起产生一个可执行文件的过程。就像下面将展示的，这个过程是非常复杂的。

C代码要求运行标准C库和特殊的*启动代码*。与直接调用连接程序相比，C编译器更容易调用带几个正确的参数的连接程序。例如：使用DJGPP 来连接第一个程序的代码，使用:

```
gcc -o first driver.o first.o asm_io.o
```

这样产生一个`first.exe`(或在Linux下只是`first`)可执行文件。

对于Borland，你需要使用：

```
bcc32 first.obj driver.obj asm_io.obj
```

Borland使用列出的第一个文件名来确定可执行文件名。所以在上面的例子里，程序将被命名为`first.exe`。

将编译和连接两个步骤结合起来是可能的。例如：

```
gcc -o first driver.c first.o asm_io.o
```

现在gcc将编译`driver.c`然后连接。

1.4.6 理解一个汇编列表文件

`-l` *`listing-file`*选项可以用来告诉`nasm`创建一个指定名字的列表文件。这个文件将显示代码如何被汇编。这儿显示了17和18行(在数据段)在列表文件中如何显示。(行号显示在列表文件中；但是注意在源代码文件中显示的行号可能不同于在列表文件中显示的行号。)

```
48 00000000 456E7465722061206E-     prompt1 db    "Enter a number: ", 0
49 00000009 756D6265723A2000
50 00000011 456E74657220616E6F-     prompt2 db    "Enter another number: ", 0
51 0000001A 74686572206E756D62-
52 00000023 65723A2000
```

每一行的头一列是行号，第二列是数据在段里的偏移地址(十六进制显示)。第三列显示将要储存的十六进制值。这种情况下，十六进制数据符合ASCII编码。最终，显示来自资源文件的正文。列在第二行的偏移地址非常可能不是数据存放在完成后的程序中的真实偏移地址。每个模块可能在数据段(或其它段)定义它自己的变量。在连接这一步时(小节 1.4.5)，所有这些数据段的变量定义结合形成一个数据段。最终的偏移由连接程序计算得到。

这儿有一小部分text段代码(资源文件中54到56行)在列表文件中如何显示：

```
94 0000002C A1[00000000]          mov     eax, [input1]
95 00000031 0305[04000000]        add     eax, [input2]
96 00000037 89C3                  mov     ebx, eax
```

第三列显示了由汇编程序产生的机器代码。通常一个指令的完整代码不能完全计算出来。例如：在94行，input1的偏移(地址)要直到代码连接后才能知道。汇编程序可以算出mov 指令(在列表中为A1)的操作码，但是它把偏移写在方括号中，因为准确的值还不能算出来。这种情况下，0作为一个暂时偏移被使用，因为input1在这个文件中，被定义在bss段的开始。记住这不意味着它会在程序的最终bss 段的开始。当代码连接后，连接程序将在位置上插入正确的偏移。其它指令，如96行，并不涉及任何变量。这儿汇编程序可以算出完整的机器代码。

Big和Little Endian 表示法

如果有人仔细看过95行，将会发现机器代码中的方括号里的偏移地址非常奇怪。input2变量的偏移地址为4(像文件定义的一样)；但是显示在内存里偏移不是00000004，而是04000000。为什么？不同的处理器在内存里以不同的顺序储存多字节整形：*big endian*和*little endian*。Big endian是一种看起来更自然的方法。最大(*也就是：* 最高有效位)的字节首先被储存，然后才是第二大的，*依此类推*。例如：双字00000004将被储存为四个字节00 00 00 04。IBM主机，许多RISC 处理器和Motorola处理器都使用这种big endian方法。然而，基于Intel的处理器使用little endian方法！首先被储存是最小的有效字节。所以00000004在内存中储存为04 00 00 00。这种格式强制连入CPU而且不可能更改。通常情况下，程序员并不需要担心使用的是哪种格式。但是，在下面的情况下，它们是非常重要的。

*Endian的发音和*indian一样。

1. 当二进制数据在不同的电脑上传输时(不管来自文件还是网络)。
2. 当二进制数据作为一个多字节整形写入到内存中然后当作单个单个字节读出，*反之亦然*。

Endian格式并不应用于数组的排序。数组的第一个元素通常在最低的地址里。这个应用在字符串里(字符数组)。Endian格式依然用在数组的单个元素中。

1.5 骨架文件

图 1.7显示了一个可以用来书写汇编程序的开始部分的骨架文件。

skel.asm

```
%include "asm_io.inc"
segment .data
;
; 初始化数据放入到这儿的数据段中
;

segment .bss
;
; 未初始化的数据放入到 bss 段中
;

segment .text
        global  _asm_main
_asm_main:
        enter   0,0               ; 开始运行
        pusha

;
; 代码放在text段。不要改变在这个注释之前或之后的代码。
;
;

        popa
        mov     eax, 0            ; 返回到以中
        leave
        ret
```

skel.asm

图 1.7: 骨架程序

第2章

基本汇编语言

2.1 整形工作方式

2.1.1 整形表示法

整形有两种类型:有符号和无符号。无符号整形(即此类型没有负数)以一种非常直接的二进制方式来表示。数字200作为一个无符号整形数将被表示为11001000(或十六进制C8)。

有符号整形（即此类型可能为正数也可能为负数）以一种更复杂的方式来表示。例如，考虑−56。+56当作一个字节来考虑时将被表示为00111000。在纸上，你可以将−56表示为−111000，但是在电脑内存中如何以一个字节来表示，如何储存这个负号呢？

有三种普遍的技术被用来在电脑内存中表示有符号整形。所有的方法都把整形的最大有效位当作一个*符号位*来使用。如果数为正数，则这一位为0；为负数，这一位就为1。

原码

第一种方法是最简单的，它被称为*原码*。它用两部分来表示一个整形。第一部分是符号位，第二部分是整形的原码。所以56表示成字节形式为0̲0111000 (符号位加了下划线)而−56将表示为1̲0111000。最大的一个字节的值将是0̲1111111或+127，而最小的一个字节的值将是1̲1111111或−127。要得到一个数的相反数，只需要将符号位变反。这个方法很简单直接，但是它有它的缺点。首先，0会有两个可能的值：+0 (0̲0000000) 和−0 (1̲0000000)。因为0不是正数，也不是负数，所以这些表示法都应该把它表示成一样。这样会把CPU的运算逻辑弄得很复杂。第二，普通的运算同样是麻烦的。如果10加−56，这个将改变为10减去56。同样，这将会复杂化CPU的逻辑。

反码

第二种方法称为*反码*表示法。一个数的反码可以通过将这个数的每一位求反得到。(另外一个得到的方法是：新的位值等于1−老的位值。) 例如：0̲0111000 (+56)的反码是1̲1000111。在反码表示法中，计算一个数的反码等价于求反。因此，−56就可以表示为1̲1000111。注意，符号位在反码中是自动改变的，你需要两次求反码来得到原始的数值。就像第一种方法一样，0有两种表示：0̲0000000 (+0)和1̲1111111 (−0)。用反码表示的数值的运算同样是麻烦的。

这有一个小诀窍来得到一个十六进制数值的反码，而不需要将它转换成二进制。这个诀窍就是用F（或十进制中的15）减去每一个十六进制位。这个方法假定数中的每一位的数值是由4位二进制组成的。这有一个例子：+56 用十六进制表示为38。要得到反码，用F减去每一位，得到C7。这个结果是与上面的结果吻合的。

补码

前面描述的两个方法用在早期的电脑中。现代的电脑使用第三种方法称为*补码*表示法。一个数的补码可以由下面两步得到：

1. 找到该数的反码
2. 将第一步的结果加1

这有一个使用0̲0111000 (56)的例子。首先，经计算得到反码：1̲1000111 。然后加1：

$$\begin{array}{rr} & \underline{1}1000111 \\ + & \qquad\quad 1 \\ \hline & \underline{1}1001000 \end{array}$$

在补码表示法中，计算一个补码等价于对一个数求反。因此，1̲1001000是−56的补码。要得到原始数值需两次求反。令人惊讶的是补码不符合这个规定。通过对1̲1001000的反码加1得到补码。

$$\begin{array}{rr} & \underline{0}0110111 \\ + & \qquad\quad 1 \\ \hline & \underline{0}0111000 \end{array}$$

当在两个补码操作数间进行加法操作时，最左边的位相加可能会产生一个进位。这个进位是不被使用的。记住在电脑中的所有数据都是有固定大小的(根据位数多少)。两个字节相加通常得到一个字节的结果(就像两个字相加得到一个字，等。) 这个特性对于补码表示法来说是非常重要的。例如，把0作为一个字节来考虑它的补码形式(0̲0000000)。计算它的补码形式得到总数：

$$\begin{array}{rr} & \underline{1}1111111 \\ + & \qquad\quad 1 \\ \hline c & \underline{0}0000000 \end{array}$$

数值	十六进制表示
0	00
1	01
127	7F
-128	80
-127	81
-2	FE
-1	FF

表 2.1: 补码表示法

其中c代表一个进位。(稍后将展示如何侦查到这个进位，但是它在这的结果中不储存。)因此，在补码表示法中0只有一种表示。这就使得补码的运算比前面的方法简单。

使用补码表示法，一个有符号的字节可以用来代表从-128 到$+127$的数值。表 2.1 展示一些可选的值。如果使用了16位，那么可以表示从$-32,768$到$+32,767$的有符号数值。$+32,767$可以表示为7FFF，$-32,768$ 为8000, -128为FF80而-1为FFFF。32位的补码大约可以表示-20亿到$+20$亿的数值范围。

CPU对某一的字节(或字，双字)具体表示多少并不是很清楚。汇编语言并没有类型的概念，而高级语言有。数据解释成什么取决于使用在这个数据上的指令。到底十六进制数FF被看成一个有符号数-1 还是无符号数$+255$取决于程序员。C语言定义了有符号和无符号整形。这就使C编译器能决定使用正确的指令来使用数据。

2.1.2 正负号延伸

在汇编语言中，所有数据都有一个指定的大小。为了与其它数据一起使用而改变数据大小是不常用的。减小它的大小是最简单的。

减小数据的大小

要减小数据的大小，只需要简单地将多余的有效位移位即可。这是一个普通的例子：

```
mov     ax, 0034h       ; ax = 52 (以十六位储存)
mov     cl, al          ; cl = ax的低八位
```

当然，如果数字不能以更小的大小来正确描述，那么减小数据的大小将不能工作。例如，如果AX是0134h (或十进制的308) ，那么上面的代码仍然将CL置为34h。这种方法对于有符号和无符号数都能工作。考虑有符号数：如果AX是FFFFh (也就是-1)，那么CL 将会是FFh (一个字节表示的-1)。然而，注意如果在AX里的值是无符号的，这个就不正确了！

无符号数的规则是：为了能转换正确，所有需要移除的位都必须是0。有符号数的规则是：需要移除的位必须要么都是1,要么都是0。另外，没有移除的第一个比特位的值必须等于移除的位的第一位。这一位将会是变小的值的新的符号位。这一位与原始符号位相同是非常重要的!

增大数据的大小

增大数据的大小比减小数据的大小更复杂。考虑十六进制字节：FF。如果它扩展成一个字，那么这个字的值应该是多少呢？它取决于如何解释FF。如果FF是一个无符号字节(十进制中为)，那么这个字就应该是00FF；但是，如果它是一个有符号字节(十进制中为−1)，那么这个字就应该为FFFF。

一般说来，扩展一个无符号数，你需将所有的新位置为0.因此，FF就变成了00FF。但是，扩展一个有符号数，你必须*扩展*符号位。 这就意味着所有的新位通过复制符号位得到。因为FF的符号位为1，所以新的位必须全为1，从而得到FFFF。如果有符号数5A (十进制中为90)被扩展了，那么结果应该是005A。

80386提供了好几条指令用于数的扩展。谨记电脑是不清楚一个数是有符号的或是无符号的。这取决于程序员是否用了正确的指令。

对于无符号数，你可以使用MOV指令简单地将高位置0。例如，将一个在AL中的无符号字节扩展到AX中：

```
mov     ah, 0    ; 输出高8位为0
```

但是，使用MOV指令把一个在AX中的无符号字转换成在EAX中的无符号双字是不可能的。为什么不可以呢？因为在MOV指令中没有方法指定EAX的高16位。80386通过提供一个新的指令MOVZX来解决这个问题。 这个指令有两个操作数。目的操作数(第一个操作数)必须是一个16或32位的寄存器。源操作数(第二个操作数)可以是一个8或16位的寄存器或内存中的一个字。另一个限制是目的操作数必须大于源操作数。(许多指令要求源和目的操作数必须是一样的大小。) 这儿有几个例子：

```
movzx   eax, ax       ; 将ax扩展成eax
movzx   eax, al       ; 将al扩展成eax
movzx   ax, al        ; 将al扩展成ax
movzx   ebx, ax       ; 将ax扩展成ebx
```

对于有符号数，在任何情况下，没有一个简单的方法来使用MOV 指令。8086提供了几条用来扩展有符号数的指令。CBW (Convert Byte to Word(字节转换成字))指令将AL正负号扩展成AX。操作数是不显示的。CWD (Convert Word to Double word(字转换成双字))指令将AX正负号扩展成DX:AX。DX:AX表示法表示将DX和AX寄存器当作一个32位寄存器来看待，其中高16位在DX中，低16位在AX中。(记住8086没有32位寄存器！) 80386加了好几条新的指令。CWDE (Convert Word to Double word Extended(字转换成

```
1 unsigned char uchar = 0xFF;
2 signed char   schar = 0xFF;
3 int a = (int) uchar;      /* a = 255 (0x000000FF) */
4 int b = (int) schar;      /* b = -1 (0xFFFFFFFF) */
```

图 2.1:

```
char ch;
while( (ch = fgetc(fp)) != EOF ) {
  /* 对ch做一些事情 */
}
```

图 2.2:

扩展的双字))指令将AX正负号扩展成EAX。CDQ (Convert Double word to Quad word(双字扩展成四字))指令将EAX正负号扩展成EDX:EAX (64位!). 最后，MOVSX 指令像MOVZX指令一样工作，除了它使用有符号数的规则外。

C编程中的应用

无符号和有符号数的扩展同样发生在C中。C中的变量可以被声明成有符号的，或无符号的(int是有符号的)。考虑在图 2.1中的代码。在第3行中，变量a使用了无符号数的规则(使用MOVZX)进行了扩展，但是在第4行，变量b使用了有符号数的规则(使用MOVSX)进行了扩展。

ANSI C并没有定义char类型是有符号的，还是无符号的，它取决于各个编译器的决定。这就是为什么在图 2.1中明确指定类型的原因。

这有一个直接与这个主题相关的普遍的C编程中的一个bug。考虑在图 2.2中的代码。fgetc()的原型是:

```
int fgetc( FILE * );
```

一个可能的问题:为什么这个函数返回一个int类型，然后又被当作字符类型被读呢？原因是它一般确实是返回一个char 类型的值(使用0扩展成一个int类型的值)。但是，有一个值它可能不会以字符的形式返回：EOF。这是一个宏，通常被定义为−1。因此，fgetc()不是返回一个通过扩展成int类型得到的char类型的值(在十六进制中表示为000000*xx*)，就是EOF(在十六进制中表示为FFFFFFFF)。

图 2.2中的程序的基本的问题是fgetc()返回一个int类型，但是这个值以char的形式储存。C将会切去较高顺序的位来使int类型的值适合char类型。唯一的问题是数(十六进制) 000000FF和FFFFFFFF都会被切成字节FF。因此，while循环不能区别从文件中读到的字节FF和文件的结束。

实际上，在这种情况下代码会怎么做，取决于char是有符号的，还是无符号的。为什么？因为在第2行ch是与EOF进行比较。因为EOF是一个int类型的值[1]，ch将会扩展成一个int类型，以便于这两个值在相同大小下比

[1]这是一个普通的误解：在文件的最后有一个EOF字符。这是不 正确的!

较[2]。就像图 2.1展示的一样，变量是有符号的还是无符号的是非常重要的。

如果char是无符号的，那么FF就被扩展成000000FF。这个拿去与EOF (FFFFFFFF)比较，它们并不相等。因此，循环不会结束。

如果char是有符号的，FF就被扩展成FFFFFFFF。这就导致比较相等，循环结束。但是，因为字节FF可能是从文件中读到的，循环就可能过早地被结束了。

这个问题的解决办法是定义ch 变量为int类型，而不是char类型。当做了这个改变，在第2行就不会有切去和扩展操作执行了。在循环休内，对值进行切去操作是很安全的，因为ch在这儿*必须*实际上已经是一个简单的字节了。

2.1.3 补码运算

就像我们早些时候看到的，add指令执行加法操作，而sub指令的执行减法操作。在FLAGS寄存器中的两位能被这些指令设置，它们是：*overflow(溢出位)* 和*carry flag(进位标志位)*。如果操作的正确结果太大了以致于不匹配有符号数运算的目的操作数，溢出标志位将被置位。如果在加法中的最高有效位有一个进位或在减法中的最高有效位有一个借位，进位标志位将被置位。因此，它可以用来检查无符号数运算的溢出。进位标志位在有符号数运算中的使用将看起来非常简单。补码的一个最大的优点是加法和减法的规则实际上就与无符号整形的一样。因此，add和sub可以同时被用在有符号和无符号整形上。

$$\begin{array}{rr} 002\mathrm{C} & 44 \\ +\ \mathrm{FFFF} & +\ (-1) \\ \hline 002\mathrm{B} & 43 \end{array}$$

这儿有一个进位产生，但是它不是结果的一部分。

这有两个不同的乘法和除法指令。首先，使用MUL或IMUL 指令来进行乘法运算。MUL指令用于无符号数之间相乘，而IMUL指令用于有符号数之间相乘。为什么需要两个不同的指令呢？无符号数和有符号数补码的乘法规则是不同的。为什么会这样？考虑字节FF乘以它本身产生一个字的结果。使用无符号乘法这就是255乘上255，得65025 (或十六进制的FE01)。使用有符号数乘法这就是-1 乘上-1，得1 (或十六进制的0001)。

这儿有乘法指令的几种格式。最老的格式是像这样的：

```
mul   source
```

*source*要么是一个寄存器，要么是一个指定的内存。它不可以是一个立即数。实际上，乘法怎么执行取决于源操作数的大小。如果操作数大小是一个字节，它乘以在AL寄存器中的字节，而结果被储存到了16位寄存器AX中。如果源操作数是16位，它乘以在AX中的字，而32位的结果被储

[2]对于这个要求的原因将在下面介绍。

dest	source1	source2	操作
	reg/mem8		AX = AL*source1
	reg/mem16		DX:AX = AX*source1
	reg/mem32		EDX:EAX = EAX*source1
reg16	reg/mem16		dest *= source1
reg32	reg/mem32		dest *= source1
reg16	immed8		dest *= immed8
reg32	immed8		dest *= immed8
reg16	immed16		dest *= immed16
reg32	immed32		dest *= immed32
reg16	reg/mem16	immed8	dest = source1*source2
reg32	reg/mem32	immed8	dest = source1*source2
reg16	reg/mem16	immed16	dest = source1*source2
reg32	reg/mem32	immed32	dest = source1*source2

表 2.2: imul指令

存到了DX:AX。如果源操作数是32位的，它乘以在EAX中的数，而结果被储存到了EDX:EAX。

IMUL指令拥有与MUL指令相同的格式，但是同样增加了其它一些指令格式。这有两个和三个操作数的格式：

```
imul   dest (目的操作数), source1(源操作数1)
imul   dest (目的操作数), source1(源操作数1), source2(源操作数2)
```

表 2.2展示可能的组合。

两个除法运算符是DIV和IDIV。它们分别执行无符号整形和有符号整形的除法。普遍的格式是：

```
div   source
```

如果源操作数为8位，那么AX就除以这个操作数。商储存在AL中，而余数储存在AH中。如果源操作数为16位，那么DX:AX就除以这个操作数。商储存在AX中，而余数储存在DX中。如果源操作数为32位，那么EDX:EAX就除以这个操作数，同时商储存在EAX中，余数储存在EDX中。IDIV 指令以同样的方法进行工作。这没有像IMUL指令一样的特殊的IDIV指令。如果商太大了，以致于不匹配它的寄存器，或除数为0,那么这个程序将被中断和中止。一个普遍的错误是在进行除法之前忘记了初始化DX或EDX。

NEG 指令通过计算它的单一的操作数补码来得到这个操作数的相反数。它的操作数可以是任意的8位，16位或32位寄存器或内存区域。

2.1.4 程序例子

math.asm

```
1  %include "asm_io.inc"
2  segment .data          ; 输出字符串
3  prompt          db    "Enter a number: ", 0
4  square_msg      db    "Square of input is ", 0
5  cube_msg        db    "Cube of input is ", 0
6  cube25_msg      db    "Cube of input times 25 is ", 0
7  quot_msg        db    "Quotient of cube/100 is ", 0
8  rem_msg         db    "Remainder of cube/100 is ", 0
9  neg_msg         db    "The negation of the remainder is ", 0
10
11 segment .bss
12 input   resd 1
13
14 segment .text
15         global  _asm_main
16 _asm_main:
17         enter   0,0               ; 开始运行程序
18     pusha
19
20         mov     eax, prompt
21         call    print_string
22
23         call    read_int
24         mov     [input], eax
25
26         imul    eax               ; edx:eax = eax * eax
27         mov     ebx, eax          ; 保存结果到ebx中
28         mov     eax, square_msg
29         call    print_string
30         mov     eax, ebx
31         call    print_int
32         call    print_nl
33
34         mov     ebx, eax
35         imul    ebx, [input]      ; ebx *= [input]
36         mov     eax, cube_msg
37         call    print_string
38         mov     eax, ebx
39         call    print_int
40         call    print_nl
41
42         imul    ecx, ebx, 25      ; ecx = ebx*25
```

```
43          mov     eax, cube25_msg
44          call    print_string
45          mov     eax, ecx
46          call    print_int
47          call    print_nl
48
49          mov     eax, ebx
50          cdq                     ; 通过正负号延伸初始化edx
51          mov     ecx, 100        ; 不可以被立即数除
52          idiv    ecx             ; edx:eax / ecx
53          mov     ecx, eax        ; 保存商到ecx中
54          mov     eax, quot_msg
55          call    print_string
56          mov     eax, ecx
57          call    print_int
58          call    print_nl
59          mov     eax, rem_msg
60          call    print_string
61          mov     eax, edx
62          call    print_int
63          call    print_nl
64
65          neg     edx             ; 求这个余数的相反数
66          mov     eax, neg_msg
67          call    print_string
68          mov     eax, edx
69          call    print_int
70          call    print_nl
71
72          popa
73          mov     eax, 0          ; 返回到C中
74          leave
75          ret
```

math.asm

2.1.5 扩充精度运算

汇编语言同样提供允许你执行大于双字的数的加减法的指令。这些指令使用了进位标志位。就像上面规定的，ADD 和SUB 指令在进位或借位产生时分别修改了进位标志位。储存在进位标志位里的信息可以用来做大的数字的加减法，通过将这些操作数分成小的双字(或更小) 块。

ADC 和SBB 指令使用了进位标志位里的信息。ADC指令执行下面的操作：

operand1 = *operand1* + carry flag + *operand2*

SBB执行下面的操作：

operand1 = *operand1* - carry flag - *operand2*

这些如何使用？考虑在EDX:EAX和EBX:ECX中的64位整形的总数。下面的代码将总数储存到EDX:EAX中:

```
1  add    eax, ecx        ; 低32位相加
2  adc    edx, ebx        ; 高32位带以前总数的进位相加
```

减法也是一样的。下面的代码用EDX:EAX减去EBX:ECX:

```
1  sub    eax, ecx        ; 低32位相减
2  sbb    edx, ebx        ; 高32位带借位相减
```

对于*实际上*大的数字，可以使用一个循环(看小节 2.2)。对于一个求和的循环，对于每一次重复(替代所有的，除了第一次重复)使用ADC指令将会非常便利。通过在循环开始之前使用CLC (CLear Carry(清除进位))指令初始化进位标志位为0,可以使这个操作正确执行。如果进位标志位为0,那么ADD和ADC指令就没有区别了。这个在减法中也是一样的。

2.2 控制结构

高级语言提供高级的控制结构(*例如*，*if*和*while*语句)来控制执行的顺序。汇编语言并没有提供像这样的复杂控制结构。它使用声名狼藉的*goto*来替代，如果使用不恰当可能会导致非常复杂的代码。但是，它是能够写出结构化的汇编语言程序。基本的步骤是使用熟悉的高级语言控制结构来设计程序的逻辑，然后将这个设计翻译成恰当的汇编语言(就像一个编译器要做的一样)。

2.2.1 比较

控制结构决定做什么是基于数据的比较的。在汇编语言中，比较的结果储存在FLAGS寄存器中，以便以后使用。80x86提供CMP指令来执行比较操作。FLAGS寄存器根据CMP指令的两个操作数的不同来设置。具体的操作是相减，然后FLAGS根据结果来设置，但是结果是不在任何地方储存的。如果你需要结果，可以使用SUB来代替CMP指令。

对于无符号整形，有两个标志位(在FLAGS寄存器里的位) 是非常重要的：零标志位(zero flag(ZF)) 和进位标志位(carry flag(CF)) 。如果比较的结果是0的话，零标志位将置成(1) 。进位标志位在减法中当作一个借位来使用。考虑这个比较：

```
cmp    vleft, vright
```

vleft - vright的差别被计算出来，然后相应地设置标志位。如果CMP执行后得到差别为0，即vleft = vright那么ZF就被置位了(*也就是：* 1)，但是CF不被置位(*也就是：* 0)。如果vleft > vright，那么ZF就不被置位而且CF也不被置位(没有借位)。如果vleft < vright，那么ZF就不被置位，而CF就被置位了(有借位)。

对于有符号整形，有三个标志位非常重要：零标志位(zero flag (ZF))，溢出标志位(overflow flag(OF))和符号标志位(sign flag (SF))。如果一个操作的结果上溢(下溢)，那么溢出标志位将被置位。如果一个操作的结果为负数，那么符号标志位将被置位。如果vleft = vright，那么ZF将被置位(正好与无符号整形一样)。如果vleft > vright，那么ZF不被设置，而且SF = OF。如果vleft < vright，那么ZF不被设置而且SF $\neq$ OF。

如果vleft > vright，为什么SF = OF? 因为如果没有溢出，那么差别将是一个正确的值，而且肯定是非负的。因此，SF = OF = 0。但是，如果有溢出，那么差别将不是一个正确的值(而且事实上将会是个负数)。因此，SF = OF = 1。

不要忘记其它的指令同样会改变FLAGS寄存器，不仅仅CMP可以。

2.2.2 分支指令

分支指令可以将执行控制权转移到一个程序的任意一点上。换言之，它们像*goto*一样运作。有两种类型的分支：无条件的和有条件的。一个无条件的分支就跟goto一样，它总会产生分支。一个有条件分支可能也可能不产生分支，它取决于在FLAGS寄存器里的标志位。如果一个有条件分支没有产生分支，控制权将传递到下一指令。

JMP (*jump*的简称)指令产生无条件分支。它唯一的参数通常是一个指向分支指向的指令的*代码标号*。汇编器和连接器将用指令的正确地址来替代这个标号。这又是一个乏味的操作数，通过这个，汇编器使得程序员的日子不好过。能认识到在JMP指令后的指令不会被执行，除非另一条分支指令指向它，是非常重要的。

这儿有jump指令的几个变更形式：

SHORT 这个跳转类型局限在一小范围内。它仅仅可以在内存中向上或向下移动128字节。这个类型的好处是相对于其它的，它使用较少的内存。它使用一个有符号字节来储存跳转的*位移*。位移表示向前或向后移动的字节数(位移须加上EIP)。为了指定一个短跳转，需在JMP指令里的变量之前使用关键字SHORT。

NEAR 这个跳转类型是无条件和有条件分支的缺省类型，它可以用来跳到一段中的任意地方。事实上，80386支持两种类型的近跳转。其中一个的位移使用两个字节。它就允许你向上或向下移动32,000个字节。另一种类型的位移使用四个字节，当然它就允许你移动到代码段中的任意位置。四字节类型是386保护模式的缺省类型。两个字节类型可以通过在JMP指令里的变量之前放置关键字WORD来指定。

FAR 这个跳转类型允许控制转移到另一个代码段。在386保护模式下，这种事情是非常鲜见的。

JZ	如果ZF被置位了，就分支
JNZ	如果ZF没有被置位了，就分支
JO	如果OF被置位了，就分支
JNO	如果OF没有被置位了，就分支
JS	如果SF被置位了，就分支
JNS	如果SF没有被置位了，就分支
JC	如果CF被置位了，就分支
JNC	如果CF没有被置位了，就分支
JP	如果PF被置位了，就分支
JNP	如果PF没有被置位了，就分支

表 2.3: 简单条件分支

有效的代码标号遵守与数据变量一样的规则。代码标号通过在代码段里把它们放在它们标记的声明前面来定义它们。有一个冒号放在变量定义的地方的结尾处。这个冒号不是名字的一部分。

条件分支有许多不同的指令。它们都使用一个代码标号作为它们唯一的操作数。最简单的就是看FLAGS寄存器里的一个标志位来决定是否要分支。看表 2.3得到关于这些指令的列表。(PF是*奇偶标志位(parity flag)*，它表示结果中的低8位1的位数值为奇数个或偶数个。)

下面的伪码:

```
if ( EAX == 0 )
  EBX = 1;
else
  EBX = 2;
```

可以写成汇编形式，如：

```
1        cmp     eax, 0              ; 置标志位(如果eax - 0 = 0，ZF就被置位)
2        jz      thenblock           ; 如果ZF被置位了，就跳转到thenblock
3        mov     ebx, 2              ; IF结构的ELSE部分
4        jmp     next                ; 跳过IF结构中的THEN部分
5  thenblock:
6        mov     ebx, 1              ; IF结构的THEN部分
7  next:
```

其它比较使用在表 2.3里的条件分支并不是很容易。为了举例说明，考虑下面的伪码：

```
if ( EAX >= 5 )
  EBX = 1;
else
  EBX = 2;
```

有符号		无符号	
JE	如果vleft = vright，则分支	JE	如果vleft = vright，则分支
JNE	如果vleft ≠ vright，则分支	JNE	如果vleft ≠ vright，则分支
JL, JNGE	如果vleft < vright，则分支	JB, JNAE	如果vleft < vright，则分支
JLE, JNG	如果vleft ≤ vright，则分支	JBE, JNA	如果vleft ≤ vright，则分支
JG, JNLE	如果vleft > vright，则分支	JA, JNBE	如果vleft > vright，则分支
JGE, JNL	如果vleft ≥ vright，则分支	JAE, JNB	如果vleft ≥ vright，则分支

表 2.4: 有符号和无符号的比较指令

如果EAX大于或等于5，ZF可能被置位或不置位，而SF将等于OF。这是测试这些条件的汇编代码(假定EAX是有符号的):

```
        cmp     eax, 5
        js      signon              ; 如果SF = 1，就跳转到signon
        jo      elseblock           ; 如果OF = 1而且SF = 0，就跳转到elseblock
        jmp     thenblock           ; 如果SF = 0而且OF = 0，就跳转到thenblock
signon:
        jo      thenblock           ; 如果SF = 1而且OF = 1，就跳转到thenblock
elseblock:
        mov     ebx, 2
        jmp     next
thenblock:
        mov     ebx, 1
next:
```

上面的代码使用起来非常不便。幸运的是，80x86提供了额外的分支指令使这种类型的测试条件更容易些。每个版本都分为有符号和无符号两种。表 2.4展示了这些指令。等于或不等于分支(JE和JNE)对于有符号和无符号整形是相同的。(事实上，JE和JZ，JNE和JNZ基本上完全相同。) 每个其它的分支指令都有两个同义字。例如：看JL (jump less than)和JNGE (jump not greater than or equal to)。有相同的指令这是因为：

$$x < y \Longrightarrow \mathbf{not}(x \geq y)$$

无符号分支使用A代表大于而B代表小于，替换了L和G。

使用这些新的指令，上面的伪码可以更容易地翻译成汇编语言：

```
        cmp     eax, 5
        jge     thenblock
        mov     ebx, 2
        jmp     next
thenblock:
        mov     ebx, 1
next:
```

2.2.3 循环指令

80x86提供了几条专门为实现像*for*一样的循环而设计的指令。每一个这样的指令带有一个代码标号作为它们唯一的操作数。

LOOP ECX自减，如果ECX $\neq 0$，分支到代码标号指向的地址

LOOPE, LOOPZ ECX自减(FLAGS寄存器没有被修改)，如果ECX $\neq 0$ 而且ZF $= 1$，则分支

LOOPNE, LOOPNZ ECX自减(FLAGS没有改变)，如果ECX $\neq 0$ 而且ZF $= 0$，则分支

最后两个循环指令对于连续的查找循环是非常有用的。下面的伪码：

```
sum = 0;
for( i=10; i >0; i-- )
  sum += i;
```

可以翻译在汇编语言，如:

```
1        mov    eax, 0          ; eax是总数(sum)
2        mov    ecx, 10         ; ecx是i
3  loop_start:
4        add    eax, ecx
5        loop   loop_start
```

2.3 翻译标准的控制结构

这一小节讲述在高级语言里的标准控制结构如何应用到汇编语言中。

2.3.1 If语句

下面的伪码:

```
if ( 条件 )
  then_block;
else
  else_block ;
```

可以像这样被应用:

```
1        ; 设置FLAGS的代码
2        jxx    else_block     ; 选择xx，如果条件为假，则分支
3        ; then模块的代码
4        jmp    endif
5  else_block:
6        ; else模块的代码
7  endif:
```

如果没有else语句的话，那么else_block分支可以用endif分支取代。

```
        ; 设置FLAGS的代码
        jxx     endif           ; 选择xx，如果条件为假，则分支
        ; then模块的代码
endif:
```

2.3.2 While循环

*while*循环是一个顶端测试循环：

while(条件) {
  循环体;
}

这个可以翻译成：

```
while:
        ; 基于条件的设置FLAGS的代码
        jxx     endwhile        ; 选择xx，如果条件为假，则分支
        ; 循环体
        jmp     while
endwhile:
```

2.3.3 Do while循环

*do while*循环是一个末端测试循环：

do {
  循环体;
} **while**(条件);

这个可以翻译成：

```
do:
        ; 循环体
        ; 基于条件的设置FLAGS的代码
        jxx     do              ; 选择xx，如果条件为假，则分支
```

2.4 例子:查找素数

这一小节是一个查找素数的程序。根据回忆，素数是一个只能被1和它本身整除的数。没有公式来做这件事情。这个程序使用的基本方法是在一

```
unsigned guess;    /* 当前对素数的猜测  */
unsigned factor;   /* 猜测数的可能的因子  */
unsigned limit;    /* 查找这个值以下的素数  */

printf("Find primes up to: ");
scanf("%u", &limit);
printf("2\n");     /* 把头两个素数当特殊的事件处理  */
printf("3\n");
guess = 5;         /* 初始化猜测数 */
while ( guess <= limit ) {
  /* 查找一个猜测数的因子 */
  factor = 3;
  while ( factor*factor < guess &&
          guess % factor != 0 )
    factor += 2;
  if ( guess % factor != 0 )
    printf("%d\n", guess);
  guess += 2;    /* 只考虑奇数 */
}
```

图 2.3:

个给定的范围内查找所有奇数的因子[3]。如果一个奇数没有找到一个因子，那么它就是一个素数。图 2.3 展示了用C写的基本的算法。

这是汇编语言版:

prime.asm

```
%include "asm_io.inc"
segment .data
Message         db      "Find primes up to: ", 0

segment .bss
Limit           resd    1               ; 查找这个值以下的素数
Guess           resd    1               ; 当前对素数的猜测

segment .text
        global  _asm_main
_asm_main:
        enter   0,0                 ; 程序开始运行
        pusha

```

[3]2是唯一的偶数素数。

```
15          mov     eax, Message
16          call    print_string
17          call    read_int            ; scanf("%u", & limit );
18          mov     [Limit], eax
19
20          mov     eax, 2              ; printf("2\n");
21          call    print_int
22          call    print_nl
23          mov     eax, 3              ; printf("3\n");
24          call    print_int
25          call    print_nl
26
27          mov     dword [Guess], 5    ; Guess = 5;
28  while_limit:                        ; while ( Guess <= Limit )
29          mov     eax,[Guess]
30          cmp     eax, [Limit]
31          jnbe    end_while_limit     ; 因为数字为无符号数，所以使用jnbe
32
33          mov     ebx, 3              ; ebx等于factor = 3;
34  while_factor:
35          mov     eax,ebx
36          mul     eax                 ; edx:eax = eax*eax
37          jo      end_while_factor    ; 如果结果不匹配eax
38          cmp     eax, [Guess]
39          jnb     end_while_factor    ; if !(factor*factor < guess)
40          mov     eax,[Guess]
41          mov     edx,0
42          div     ebx                 ; edx = edx:eax % ebx
43          cmp     edx, 0
44          je      end_while_factor    ; if !(guess % factor != 0)
45
46          add     ebx,2               ; factor += 2;
47          jmp     while_factor
48  end_while_factor:
49          je      end_if              ; if !(guess % factor != 0)
50          mov     eax,[Guess]         ; printf("%u\n")
51          call    print_int
52          call    print_nl
53  end_if:
54          add     dword [Guess], 2    ; guess += 2
55          jmp     while_limit
56  end_while_limit:
```

```
57
58          popa
59          mov     eax, 0              ; 返回到C中
60          leave
61          ret
```

prime.asm

第3章

位操作

3.1 移位操作

汇编语言允许程序员对数据的单个比特位进行操作。一个最常见的的位操作称为*移位*。移位操作移动某一个数据的比特位的位置。移位可以是向左移(*也就是*：向最高有效位移动)，也可以向右移(最低有效位)。

3.1.1 逻辑移位

逻辑移位是移位中最简单的类型。它以一种最直接的方式进行移位操作。图 3.1展示了一个字节的移位操作的例子。

原始数据	1	1	1	0	1	0	1	0
向左移位	1	1	0	1	0	1	0	0
向右移位	0	1	1	1	0	1	0	1

图 3.1: 逻辑移位

注意：新进来的比特位总是为0。SHL 和SHR 指令分别用来执行逻辑左移和逻辑右移。这些指令允许你移动任意的位数。位数可以是一个常量，也可以是储存在CL寄存器的值。最后从数据中移出的比特位储存在进位标志位中。这有一些代码例子：

```
1  mov   ax, 0C123H
2  shl   ax, 1          ; 向左移动一个比特位，   ax = 8246H, CF = 1
3  shr   ax, 1          ; 向右移动一个比特位，   ax = 4123H, CF = 0
4  shr   ax, 1          ; 向右移动一个比特位，   ax = 2091H, CF = 1
5  mov   ax, 0C123H
6  shl   ax, 2          ; 向左移动两个比特位，   ax = 048CH, CF = 1
7  mov   cl, 3
8  shr   ax, cl         ; 向右移动三个比特位，   ax = 0091H, CF = 1
```

3.1.2 移位的应用

快速的乘法和除法是移位操作最普遍的应用。回忆在十进制系统中，乘以和除以10的几次方是非常简单的，只是移动位而已。在二进制中，乘以和除以2的几次方也是一样的。例如：要得到二进制数1011_2(或十进制11)的两倍，只需向左移动一位，得到10110_2 (或22)。一个除以2的几次方的除法的商相当于一个右移操作的结果。仅仅是除以2，向右移动一位；除以4(2^2),向右移动2位；除以8(2^3，向右移动3位，等等。 移位指令非常基础而且比功能相同的MUL 和DIV 指令执行要快得多!

实际上，逻辑移位只可以用于无符号数的乘法和除法。一般它们不能应用于有符号数。考虑两个字节的数值FFFF(有符号时为-1)。如果它向右逻辑移动一位，结果是7FFF，也就是$+32,767$!另一种类型的移位操作可以用在有符号数上。

3.1.3 算术移位

这些移位操作是为允许有符号数能快速地执行乘以和除以2的几次方的操作而设计的。它们保证符号位能被正确对待。

SAL 算术左移(Shift Arithmetic Left) - 这条指令只是SHL的同义词。它实际上被编译成与SHL一样的机器代码。只要符号位没有因移位而改变，结果就将是正确的。

SAR 算术右移(Shift Arithmetic Right) - 这是一条新的指令，它不会移动操作数的符号位(*也就是：* 最高有效位)。其它位被正常移动，除了从左边新进来的位通过复制符号位(也就是说，如果符号位为1,那么新的位值也同样为1)得到外。因此，如果一个字节使用这条指令来移位，只有低7位会被移动。就像其它移位指令一样，最后移出的位储存在进位标志位中。

```
1        mov    ax, 0C123H
2        sal    ax, 1           ; ax = 8246H, CF = 1
3        sal    ax, 1           ; ax = 048CH, CF = 1
4        sar    ax, 2           ; ax = 0123H, CF = 0
```

3.1.4 循环移位

循环移位指令像逻辑指令一样运作，除了把从数据的一端移出的比特位又移入到另一端外。因此，数据好像被当作一个循环结构体一样对待。ROL 和ROR 是两个最简单的循环移位指令，它们分别执行左移和右移操作。就像其它移位指令一样，这些移位指令把循环移出的最后一个比特位复制到进位标志位中。

```
1        mov    ax, 0C123H
2        rol    ax, 1           ; ax = 8247H, CF = 1
3        rol    ax, 1           ; ax = 048FH, CF = 1
4        rol    ax, 1           ; ax = 091EH, CF = 0
5        ror    ax, 2           ; ax = 8247H, CF = 1
6        ror    ax, 1           ; ax = C123H, CF = 1
```

有两个额外的循环指令用来在数据和进位标志位之间移动比特位，它们称为RCL 和RCR。例如，如果AX寄存器用这些指令来移位，那么有17位用来得到AX，进位标志位也包括在循环中。

```
1        mov    ax, 0C123H
2        clc                    ; 进位标志位清零(CF = 0)
3        rcl    ax, 1           ; ax = 8246H, CF = 1
4        rcl    ax, 1           ; ax = 048DH, CF = 1
5        rcl    ax, 1           ; ax = 091BH, CF = 0
6        rcr    ax, 2           ; ax = 8246H, CF = 1
7        rcr    ax, 1           ; ax = C123H, CF = 0
```

3.1.5 简单应用

这有一个代码小片断，它用来计算在EAX寄存器里“on”(*也就是*： 1)的比特位有多少个。

```
1        mov    bl, 0           ; bl将储存ON的比特位的总数
2        mov    ecx, 32         ; ecx是循环计数器
3  count_loop:
4        shl    eax, 1          ; 把比特位移入到进位标志位中
5        jnc    skip_inc        ; 如果CF == 0，那么跳转到skip_inc处执行
6        inc    bl
7  skip_inc:
8        loop   count_loop
```

上面的代码毁掉了在EAX中的初值(循环之后，EAX的值为0)。如果你想保留EAX中的值，那么用rol eax, 1替换第四行即可。

3.2 布尔型按位运算

有四个普遍的布尔型运算符，它们是：*AND*，*OR*，*XOR*和*NOT*。*真值表*展示了每一个可能的操作数得到的运算结果。

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

表 3.1: AND运算符

	1	0	1	0	1	0	1	0
AND	1	1	0	0	1	0	0	1
	1	0	0	0	1	0	0	0

图 3.2: 一个字节的AND运算

3.2.1 *AND*运算符

两个比特位的*AND*运算结果只有当这两位都是1时才为1，否则结果就为0，就像真值表 3.1展示的一样。

处理器支持这些运算指令对数据的所有位平等地进行独立的运算。例如：如果对AL和BL里的内容进行*AND*运算，那么基本的*AND*运算将应用于在这两个寄存器里的8对比特位中的每一对，像图 3.2 展示的一样。下面是一个代码例子：

```
1        mov     ax, 0C123H
2        and     ax, 82F6H           ; ax = 8022H
```

3.2.2 *OR*运算符

两个比特位的包含*OR*运算结果只有当这两位都是0时才为0，否则结果就为1，就像真值表 3.2展示的一样。下面是一个代码例子：

```
1        mov     ax, 0C123H
2        or      ax, 0E831H          ; ax = E933H
```

3.2.3 *XOR*运算

两个比特位的互斥*XOR*运算结果只有当这两位相等时为0，否则结果就为1，就像真值表 3.3展示的一样。下面是一个代码例子：

```
1        mov     ax, 0C123H
2        xor     ax, 0E831H          ; ax = 2912H
```

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

表 3.2: OR运算

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

表 3.3: XOR运算

3.2.4 *NOT*运算

*NOT*运算符是一元运算符(*也就是说*，它只对一个操作数进行运算，而不像二元运算符，如：*AND*)。一个比特位的*NOT*运算结果是这个位值的相反数，像真值表 3.4展示的一样。下面是一个代码例子：

```
1  mov    ax, 0C123H
2  not    ax                  ; ax = 3EDCH
```

注意，*NOT*能得到一个数的补码。与其它按位运算不同的是，NOT指令并不修改在FLAGS寄存器里的任何一位。

3.2.5 TEST指令

TEST指令执行一次*AND*运算，但是并不储存结果。它会基于可能的结果对FLAGS寄存器进行设置(非常像CMP指令执行了一次减法操作但是只是设置了FLAGS)。例如：如果结果是0，那么ZF就被置位了。

3.2.6 位操作的应用

位操作对于操纵数据单个位而不修改其它位来说是非常有用的。表 3.5展示了这些操作的三个普遍的应用。下面是一些实现了这些想法的代码例子。

```
1  mov    ax, 0C123H
2  or     ax, 8           ; 开启位3,    ax = C12BH
3  and    ax, 0FFDFH      ; 关闭位5,    ax = C10BH
4  xor    ax, 8000H       ; 求反位31,     ax = 410BH
```

X	NOT X
0	1
1	0

表 3.4: NOT运算

开启位i	将需修改的数与2^i(这是一个只有位i为on的二进制数)进行OR运算
关闭位i	将需修改的数与只有位i为off的二进制数进行AND运算。这个操作数通常称为*掩码*
求反位i	将需修改的数与2^i进行XOR运算

表 3.5: 布尔运算的使用

```
5       or      ax, 0F00H       ; 开启半个字节，   ax = 4F0BH
6       and     ax, 0FFF0H      ; 关闭半个字节，   ax = 4F00H
7       xor     ax, 0F00FH      ; 求反半个字节，   ax = BF0FH
8       xor     ax, 0FFFFH      ; 补码，    ax = 40F0H
```

AND运算还可以用来得到除以2的几次方之后的余数。要得到除以数2^i之后的余数，只需对这个数和等于$2^i - 1$的掩码进行AND运算。这个掩码从位0到位$2^i - 1$都为1，也就是这些位包含了余数。AND运算的结果将保留这些位，而将其它位输出为0。下面是一个得到100除以16的商和余数的代码小片断。

```
1       mov     eax, 100        ; 100 = 64H
2       mov     ebx, 0000000FH  ; 掩码 = 16 - 1 = 15 或 F
3       and     ebx, eax        ; ebx = 余数 = 4
```

使用CL寄存器，就使得修改数据中的任何一位变得有可能。下面是一个对EAX任意比特位置位(开启)的例子。需要置位的比特位储存在BH中。

```
1       mov     cl, bh          ; 首先需得到参加OR运算的数值
2       mov     ebx, 1
3       shl     ebx, cl         ; 向左移cl次
4       or      eax, ebx        ; 开启位
```

关闭一位稍微有点麻烦。

```
1       mov     cl, bh          ; 首先需得到参加AND运算的数值
2       mov     ebx, 1
3       shl     ebx, cl         ; 向左移cl次
4       not     ebx             ; 求反所有位
5       and     eax, ebx        ; 关闭位
```

```
1       mov     bl, 0           ; bl将储存值为ON的位数
2       mov     ecx, 32         ; ecx是循环计数器
3 count_loop:
4       shl     eax, 1          ; 移动一位到进位标志位中
5       adc     bl, 0           ; bl加上进位标志位
6       loop    count_loop
```

图 3.3: 用ADC计算位数

求反任意一个比特位的代码留给了读者，做为一个习题。

在一个80x86程序中看到这个莫名其妙的指令是很平常的：

```
xor     eax, eax          ; eax = 0
```

一个数字与自己进行*XOR*运算，其结果总是0。这条指令被使用是因为它产生的机器代码比功能相同的MOV指令要少。

3.3 避免使用条件分支

现代处理器使用了非常尖端的技术来尽可能快地执行代码。一个普遍技术称为*预测执行*。这种技术使用CPU的并行处理能力来同时执行多条指令。条件分支与这项技术有冲突。一般说来，处理器是不知道分支是否会执行。如果执行了，跟没有执行相比执行的是一组不同的指令。处理器试着预测分支是否执行。如果预测错误，处理器就浪费了它的时间去执行了一些错误的代码。

一个避免这个问题的办法就是如果可能尽量避免使用条件分支。在3.1.5的样例代码中就提供了一个可以这样做的简单的例子。在以前的例子中，EAX寄存器中的值为“on”的位被计算出来。它使用了一个分支跳转到INC指令。图 3.3展示了如何使用ADC指令直接加上进位标志位来替代这个分支。

SET*xx*指令提供了在一定情况下替换分支的方法。基于FLAGS寄存器的状态，这些指令将一个字节的寄存器或内存空间中的值置为0或1。在SET后的字符与条件分支使用的是一样的。如果SET*xx*条件为真，那么储存的结果就为1,如果为假，则储存的结果就为0。例如：

```
setz    al          ; AL = 如果ZF标志位置位了则为1，否则为0。
```

使用这些指令，你可以开发出一些进行数值运算的精巧的技术，而不需要使用分支。

例如，考虑查找两个数的最大数的问题。这个问题的标准解决方法是使用一条CMP指令再使用条件分支对最大值进行操作。下面这个例子展示了不使用分支如何找到最大值。

```
; file: max.asm
%include "asm_io.inc"
segment .data

message1 db "Enter a number: ",0
message2 db "Enter another number: ", 0
message3 db "The larger number is: ", 0

segment .bss

input1  resd    1        ; 键入的第一个数值

segment .text
        global  _asm_main
_asm_main:
        enter   0,0              ; 程序开始运行
        pusha

        mov     eax, message1    ; 显示第一条消息
        call    print_string
        call    read_int         ; 输入第一个数值
        mov     [input1], eax

        mov     eax, message2    ; 显示第二条消息
        call    print_string
        call    read_int         ; 输入第二个数值(在eax中)

        xor     ebx, ebx         ; ebx = 0
        cmp     eax, [input1]    ; 比较第一和第二个数值
        setg    bl               ; ebx = (input2 > input1) ?          1 : 0
        neg     ebx              ; ebx = (input2 > input1) ? 0xFFFFFFFF : 0
        mov     ecx, ebx         ; ecx = (input2 > input1) ? 0xFFFFFFFF : 0
        and     ecx, eax         ; ecx = (input2 > input1) ?     input2 : 0
        not     ebx              ; ebx = (input2 > input1) ?      0 : 0xFFFFFFFF
        and     ebx, [input1]    ; ebx = (input2 > input1) ?          0 : input1
        or      ecx, ebx         ; ecx = (input2 > input1) ?     input2 : input1

        mov     eax, message3    ; 显示结果
        call    print_string
        mov     eax, ecx
        call    print_int
```

```
42          call    print_nl
43
44          popa
45          mov     eax, 0              ; 返回到C中
46          leave
47          ret
```

这个诀窍是产生一个可以用来选择出正确的最大值的俺码。在30行的SETG指令如果第二个输入值是最大值就将BL置为1,否则就置为0。这不是真正需要的掩码。为了得到真正需要的掩码，31行对整个EBX寄存器使用了NEG指令。(注意：EBX在前面已经置为0了。)如果EBX等于0,那么这条指令就不做任何事情；但是如果EBX为1,结果就是-1的补码表示或0xFFFFFFFF。这正好是需要的位掩码。剩下的代码就是使用这个位掩码来选择出正确的输入的最大值。

另外一个可供选择的诀窍是使用DEC语句。在上面的代码中，如果用DEC代替NEG，那么结果同样会是0或0xFFFFFFFF。但是，与使用NEG相比，得到值将是反过来的。

3.4 在C中进行位操作

3.4.1 C中的按位运算

不同于某些高级语言的是，C提供了按位操作的运算符。*AND*运算符用二元运算符&来描述。[1]。*OR*运算符用二元运算符|来描述。而*NOT*运算符是用一元运算符~ 来描述。

C中的二元运算符<<和>>执行移位操作。运算符<<执行左移操作而运算符>>执行右移操作。这些运算符有两个操作数。左边的操作数是需要移位的数值，右边的操作数是需要移的位数。如果需要移位的数值是无符号类型，那么就执行了一次逻辑移位。如果需要移位的数值是有符号类型(比如：int)，那么就执行了一次算术移位。下面是一些使用了这些运算符的C代码例子：

```
1  short  int  s;          /* 假定short int类型为16位 */
2  short  unsigned u;
3  s = -1;                 /* s = 0xFFFF (补码) */
4  u = 100;                /* u = 0x0064 */
5  u = u | 0x0100;         /* u = 0x0164 */
6  s = s & 0xFFF0;         /* s = 0xFFF0 */
7  s = s ^ u;              /* s = 0xFE94 */
8  u = u << 3;             /* u = 0x0B20 (逻辑移位) */
9  s = s >> 2;             /* s = 0xFFA5 (算术移位) */
```

[1]这个运算符不同于二元运算符&&和一元运算符&!

宏	含意
S_IRUSR	用户可读
S_IWUSR	用户可写
S_IXUSR	用户可执行
S_IRGRP	组用户可读
S_IWGRP	组用户可写
S_IXGRP	组用户可执行
S_IROTH	其它用户可读
S_IWOTH	其它用户可写
S_IXOTH	其它用户可执行

表 3.6: POSIX文件权限宏

3.4.2 在C中使用按位运算

在C中使用按位运算的目的与在汇编语言中使用按位运算的目的是一样的。它们可以允许你操作数据的单个比特位，而且可以用在快速乘除法中。事实上，一个好的C编译器应该可以自动用移位来进行乘法运算如：`x *= 2`。

许多操作系统的API[2](例如：*POSIX*[3]和Win32)包含了一些函数，这些函数使用的操作数含有按位编码的数据。例如：POSIX系统就为三种不同类型的用户保留了文件的权限：*user* (用户，*owner*可能是一个更好的名字)，*group*(组用户)和*others*(其它用户)。每一种类型的用户可以被授予进行读，写和/或执行一个文件的权限。要改变一个文件的权限，要求C程序员进行单个的位操作。POSIX定义了几个宏来做这件事(看表 3.6)。`chmod`函数可以用来设置文件的权限。这个函数有两个参数，一个是表示需设置的文件文件名的字符串，另外一个是为需要的权限设置了正确位的整形[4] 。例如，下面的代码设置了这样的权限：允许文件的owner用户对文件可读可写，在group中的用户权限为可读而others用户没有权限访问。

```
chmod("foo", S_IRUSR | S_IWUSR | S_IRGRP );
```

POSIX中`stat`函数可以用来得到文件的当前权限位。与`chmod`函数一起使用，它可以用来改变某些权限而不影响到其它权限。下面是一个移除文件的others用户的写权限和增加owner用户的读权限的例子。同时，其它权限没有被改变。

```
1 struct stat file_stats ; /* stat()使用的结构体 */
2 stat("foo", & file_stats ); /* 读文件信息file_stats.st_mode中有权限位 */
3 chmod("foo", ( file_stats .st_mode & ~S_IWOTH) | S_IRUSR);
```

[2]Application Programming Interface，应用程序接口
[3]代表计算机环境的可移植操作系统接口。IEEE在UNIX上标准开发出来的。
[4]实际上就是一个`mode_t`类型的参数，`mode_t`是一个整形类型的类型定义。

```
unsigned short word = 0x1234;  /* 假定sizeof(short) == 2 */
unsigned char * p = (unsigned char *) &word;

if ( p[0] == 0x12 )
  printf ("Big Endian Machine\n");
else
  printf (" Little  Endian Machine\n");
```

图 3.4: 如何确定Endian格式

3.5 Big和Little Endian表示法

第一章介绍了多字节数据的big和little endian表示法的概念。但是，作者发现这个主题弄得很多人非常迷惑。这一节将详细介绍这一主题。

读者可能会回忆起endian表示法指的是一个多字节数据的单个字节元素储存在内存中的顺序。Big endian是最直接的方法。它首先储存的是最高有效字节，然后是第二个有效字节，以些类推。换句话说就是，大的位被首先储存。Little endian以一个相反的顺序来储存字节(最小的有效字节最先被储存)。x86家族的处理器使用的就是little endian表示法。

看一个例子：考虑双字12345678_{16}的表示。如果是big endian表示法，这些字节将像这样储存：12 34 56 78。如果是little endian表示法，这些字节就像这样储存：78 56 34 12。

现在读者可能会这样问自己：一个理智的芯片设计者怎么会使用little endian表示法？在Intel公司里的工程师是不是虐待狂？因为他们使广大的程序员承受了这种混乱的表示法。像这样，CPU看起来会因为在内存中向后储存字节而做额外的工作(而且从内存中读出时又要颠倒它们)。答案是CPU使用little endian格式读写内存是不需要做额外的工作的。你必须认识到CPU是由许多电子电路组成，简单地工作在位值上。位(和字节)在CPU中不需要有任何的顺序的。

考虑2个字节的寄存器AX。这可以分解成单个字节的寄存器：AH和AL。在CPU中的电路保留了AH 和AL的值。在一个CPU中，电路是没有任何顺序的。也就是说，储存AH的电路可以在储存AL的电路前面或后面。mov指令复制AX的值到内存中，首先复制AL的值，接着是AH。CPU做这件事一点也没有比先储存AH难。

同样的讨论还可以用到一个字节的单个比特位上。它们在CPU电路(或就此而言，内存)里并不真的有一定顺序。但是，因为在CPU或内存中单个的比特位并没有编址，所以没有办法知道(或关心)CPU在内部对它们是如何排序的。

在图 3.4中的C代码展示了如何确定CPU的Endian格式。p指针把word变量当作两个元素的字符数组来看待。因此，word在内存里的第一个字节赋值给了p[0]，而这取决于CPU的Endian格式。

```
1  unsigned invert_endian ( unsigned x )
2  {
3    unsigned invert;
4    const unsigned char * xp = (const unsigned char *) &x;
5    unsigned char * ip = (unsigned char *) & invert;
6
7    ip[0] = xp[3];    /* 逐个字节地进行交换 */
8    ip[1] = xp[2];
9    ip[2] = xp[1];
10   ip[3] = xp[0];
11
12   return invert;    /* 返回顺序相反的字节 */
13 }
```

图 3.5: invert_endian Function

3.5.1 什么时候需要在乎Little和Big Endian

当多字节字符来到时，比如：UNICODE，Endian格式甚至对于文本数据都是非常重要的。UNICODE支持任意一种Endian格式而且有一个能指出使用了哪一种Endian格式从而来描述数据的技巧。

对于典型的编程，CPU的Endian格式并不是很重要。它很重要的大多数时刻是在不同的计算机系统间传输二进制数据时。此时使用的要么是某种类型的物理数据媒介(例如一块硬盘)要么是网络。因为ASCII数据是单个字节的，Endian格式对它来说是没有问题的。

所有的内部的TCP/IP消息头都以big endian的格式来储存整形。(称为*网络字节续*). TCP/IP 库提供了可移植处理Endian格式问题的方法的C函数。例如：htonl() 函数把一个双字(或长整形)从*主机*格式转换成了*网络*格式。ntohl()函数执行一个相反的交换。[5]对于一个big endian系统，这两个函数仅仅是无修改地返回它们的输入。这就允许你写出的网络程序可以在任何的Endian格式系统上成功编译和运行。要得到关于Endian格式和网络编程更多的信息，请看W. Richard Steven写的优秀的书籍：*UNIX Network Programming*。

图 3.5展示了一个转换双字Endian格式的C函数。486处理器提供了一个名为BSWAP的新的指令来交换任意32位寄存器中的字节。例如：

```
bswap   edx             ; 交换edx中的字节
```

这条指令不可以使用在16位的寄存器上。但是XCHG 指令可以用来交换可以分解成8位寄存器的16寄存器中的字节。例如：

```
xchg    ah,al           ; 交换ax中的字节
```

[5]实际上，转换一个整形的Endian格式就是简单地颠倒一下字节；因此从big转换成little和从little转换成big是同样的操作。所以所有这些函数都在做同样的事。

```
1  int count_bits( unsigned int data )
2  {
3    int cnt = 0;
4
5    while( data != 0 ) {
6      data = data & (data - 1);
7      cnt++;
8    }
9    return cnt;
10 }
```

图 3.6: 计算位数:方法一

3.6 计算位数

前面介绍了一个简单的技术来计算一个双字中“on”的位数有多少。这一节来看看其它不是很直接来做这件事的方法，就当作使用在这一章中讨论的位操作的练习。

3.6.1 方法一

第一个方法很简单，但不是很明显。图 3.6展示了代码。

这个方法为什么会起作用？在循环中的每一次重复，data中就会有一个比特位被关闭了。当所有的位都关闭了(*也就是说*， 当data 等于0了)，循环就结束了。使data等于0需要重复的次数就等于原始值data中比特位为1的位数。

第6行表示data中的一个比特位被关闭了。这个是如何起作用的？考虑data二进制表示法最普遍的格式和在这种表示法中最右边的1。根据上面的定义，在这个1后面的所有位都为0。现在，data - 1的二进制表示是什么样的？最右边的1的所有左边的位与data是一样的，但是在最右边的1这一点的位将会是data原始位的反码。例如：

```
data       =  xxxxx10000
data - 1   =  xxxxx01111
```

x表示对于在这个位上这两个数的值是相等的。当data和data - 1进行*AND*运算后，在data中的最右边的1这一位的结果就会为0，而其它比特位没有被改变。

3.6.2 方法二

查找表同样可以用来计算出任意双字的位数。这个直接方法首先要算出每个双字的位数，还要把位数储存到一个数组中。但是，有两个与这个方法相关的问题。双字的值大约有*40亿*。这就意味着数组将会非常大而且会

```
1  static unsigned char byte_bit_count [256];   /* 查找表 */
2
3  void   initialize_count_bits  ()
4  {
5    int cnt, i, data;
6
7    for( i = 0; i < 256; i++ ) {
8      cnt = 0;
9      data = i;
10     while( data != 0 ) {     /* 方法一 */
11       data = data & (data - 1);
12       cnt++;
13     }
14     byte_bit_count [i] = cnt;
15   }
16 }
17
18 int  count_bits( unsigned int data )
19 {
20   const unsigned char * byte = ( unsigned char *) & data;
21
22   return  byte_bit_count [byte [0]] + byte_bit_count [byte [1]] +
23           byte_bit_count [byte [2]] + byte_bit_count [byte [3]];
24 }
```

图 3.7: 方法二

浪费很多时间在初始化这个数组上。(事实上，除非你确实打算使用一个超过40亿的数组，否则花在初始化这个数组的时间将远远大于用第一种方法计算位数的时间。

一个更现实的方法是提前算出所有可能的字节的位数，并把它们储存到一个数组中。然后，双字就可以分成四个字节来求。这四个字节的位数通过查找数组得到，然后将它们相加就得到原始双字的位数。图 3.7展示了如何用代码实现这个方法。

`initialize_count_bits`函数必须在第一次调用`count_bits`函数之前被调用。这个函数初始化了`byte_bit_count`全局数组。`count_bits`函数并不是以一个双字来看对待`data`变量，而是以把它看成四个字节的数组。`byte`指针作为一个指向这个四个字节数组的指针。因此，`byte[0]`是`data`中的一个字节(是最低有效字节还是最高有效字节取决于硬件是使用little还是big endian。)。当然，你可以像这样使用一条指令：

```
(data >> 24) & 0x000000FF
```

```
1  int count_bits(unsigned int x)
2  {
3    static unsigned int mask[] = { 0x55555555,
4                                   0x33333333,
5                                   0x0F0F0F0F,
6                                   0x00FF00FF,
7                                   0x0000FFFF };
8    int i;
9    int shift;    /* 向右移动的位数 */
10
11   for( i=0, shift=1; i < 5; i++, shift *= 2 )
12     x = (x & mask[i]) + ( (x >> shift) & mask[i] );
13   return x;
14 }
```

图 3.8: 方法三

来得到最高有效字节值，可以用同样的方法得到其它字节；但是这些指令会比引用一个数组要慢。

最后一点，使用`for`循环来计算在22和23行的总数是简单的。但是，`for`循环就会包含初始化一个循环变量，在每一次重复后比较这个变量和增加这个变量的时间开支。通过清楚的四个值来计算总数会快一些。事实上，一个好的编译器会将`for`循环形式转换成清楚的求和。这个简化和消除循环重复的处理是一个称为*循环展开*的编译器优化技术。

3.6.3 方法三

现在还有一个更聪明的方法来计算在数据里的位数。这个方法逐位地相加数据的0位和1位。得到的总数就等于在这个数据中1的位数。例如，考虑计算储存在`data`变量中的一个字节中为1的位数。第一步是执行下面这个操作：

```
data = (data & 0x55) + ((data >> 1) & 0x55);
```

这个做了些什么？十六进制常量`0x55`的二进制表示为01010101。在这个加法的第一个操作数中，`data`与这个常量进行了*AND*运算，奇数的位就被拿出来了。第二操作数`((data >> 1) & 0x55)`，首先移动所有的偶数位到奇数位上，然后使用相同的掩码得到这些相同的位。现在，第一个操作数含有`data`的奇数位而第二个操作数含有偶数位。把这两个操作数相加就相当于把`data`的奇数位和偶数位相加。例如，如果`data`等于10110011_2，那么：

$$\begin{array}{r} \texttt{data \& } 01010101_2 \\ +\ \texttt{(data >> 1) \& } 01010101_2 \end{array} \quad \text{or} \quad \begin{array}{r|cccc} & 00 & 01 & 00 & 01 \\ + & 01 & 01 & 00 & 01 \\ \hline & 01 & 10 & 00 & 10 \end{array}$$

显示在右边的加法展示了实际的位相加。这个字节的位被分成了四个2位的字段来展示实际上执行了四个独立的加法。因为所有这些字段的最大总数为2，总数超过它自身的字段且影响到其它字段的总数是不可能的。

当然，总的位数还没被计算出来。但是，可以使用跟上面一样的技术，经过同样的步骤来计算总数。下一步应该是：

```
data = (data & 0x33) + ((data >> 2) & 0x33);
```

继续上面的例子(谨记：`data`现在等于01100010_2):

`data & 00110011`$_2$			
`+ (data >> 2) & 00110011`$_2$	or	+	0010 0010
			0001 0000
			0011 0010

现在有两个4位的字段被单独地相加。

下一步是通过把这两位的总数相加来得到最终的结果：

```
data = (data & 0x0F) + ((data >> 4) & 0x0F);
```

使用上面的例子(`data`等于00110010_2):

`data & 00001111`$_2$			
`+ (data >> 4) & 00001111`$_2$	or	+	00000010
			00000011
			00000101

现在`data`等于5，这正好是正确的结果。图 3.8实现了用这个方法来计算一个双字的位数。它使用了一个`for`循环来计算总数。把循环展开可能会更快一点，但是使用循环能清晰地看到这个方法产生的不同大小的数据。

第4章

子程序

本章主要着眼于使用子程序来构成模块化程序和得到与高级语言(比如说C)的接口。函数和进程是高级语言中子程序的例子。

调用了一个子程序的代码和这个子程序必须协商它们之间的数据如何传输。数据如何传输的这些规则称为*调用约定*。这一章的很大一部分都是在讨论使用在汇编子程序和C程序接口上的标准C调用约定。这个约定(和其它约定)通常都是通过传递数据的地址(*也就是*指针)来允许子程序访问内存中的数据。

4.1 间接寻址

间接寻址允许寄存器像指针变量一样运作。要指出寄存器像一个指针一样被间接使用，需要用方括号(`[]`)将它括起来。例如：

```
1  mov    ax, [Data]      ; 一个字的标准的直接内存地址
2  mov    ebx, Data       ; ebx = & Data
3  mov    ax, [ebx]       ; ax = *ebx
```

因为AX可以容纳一个字，所以第三行代码从EBX储存的地址开始读取一个字。如果用AL替换AX，那么只有一个字节会被读取。认识到寄存器不像在C中的变量一样有类型是非常重要的。到底EBX具体指向什么完全取决于使用了什么指令。而且，甚至EBX是一个指针这个事实都完全取决于使用的指令。如果EBX错误地使用了，通常不会有编译错误；但是，程序将不会正确运行。这就是为什么相比于高级语言汇编程序较容易犯错的原因之一。

所有的32位通用寄存器(EAX，EBX，ECX，EDX)和指针寄存器(ESI, EDI)都可以用来间接寻址。一般来说，16位或8位的寄存器是不可以的。

4.2 子程序的简单例子

子程序是代码中的一个的独立的单元，它可以使用在程序的不同的地

方。换句话说，一个子程序就像一个C中的函数。可以使用跳转来调用子程序，但是返回会是一个问题。如果子程序要求能使用在程序中的任何地方，它必须要返回到调用它的代码段处。因此，子程序的跳转返回最好不要硬编码为标号。下面的代码展示了如何使用JMP指令的间接方式来做这件事。此指令方式使用一个寄存器的值来决定跳转到哪(因此，这个寄存器与C中的*函数指针*非常相似。) 下面使用子程序的方法来重写第一章中的第一个程序。

sub1.asm

```
; file: sub1.asm
; 子程序的简单例子
%include "asm_io.inc"

segment .data
prompt1 db    "Enter a number: ", 0       ; 不要忘记空结束符
prompt2 db    "Enter another number: ", 0
outmsg1 db    "You entered ", 0
outmsg2 db    " and ", 0
outmsg3 db    ", the sum of these is ", 0

segment .bss
input1  resd 1
input2  resd 1

segment .text
        global  _asm_main
_asm_main:
        enter   0,0               ; 程序开始运行
        pusha

        mov     eax, prompt1      ; 显示提示信息
        call    print_string

        mov     ebx, input1       ; 储存input1的地址到ebx中
        mov     ecx, ret1         ; 储存返回地址到ecx中
        jmp     short get_int     ; 读整形
ret1:
        mov     eax, prompt2      ; 输出提示信息
        call    print_string

        mov     ebx, input2
        mov     ecx, $ + 7        ; ecx = 当前地址 + 7
        jmp     short get_int
```

```
35
36          mov     eax, [input1]     ; eax = 在input1中的双字
37          add     eax, [input2]     ; eax += 在input2中的双字
38          mov     ebx, eax          ; ebx = eax
39
40          mov     eax, outmsg1
41          call    print_string      ; 输出第一条信息
42          mov     eax, [input1]
43          call    print_int         ; 输出input1
44          mov     eax, outmsg2
45          call    print_string      ; 输出第二条信息
46          mov     eax, [input2]
47          call    print_int         ; 输出input2
48          mov     eax, outmsg3
49          call    print_string      ; 输出第三条信息
50          mov     eax, ebx
51          call    print_int         ; 输出总数(ebx)
52          call    print_nl          ; 换行
53
54          popa
55          mov     eax, 0            ; 返回到C中
56          leave
57          ret
58  ; 子程序 get_int
59  ; 参数:
60  ;   ebx - 储存整形双字的地址
61  ;   ecx - 返回指令的地址
62  ; 注意:
63  ;   eax的值被已经被破坏掉了
64  get_int:
65          call    read_int
66          mov     [ebx], eax          ; 储存输入到内存中
67          jmp     ecx                 ; 返回到调用处
```

sub1.asm

子程序get_int使用了一个简单，基于寄存器的调用约定。它认为EBX寄存器中存的是输入双字的储存地址而ECX寄存器中存的是跳转返回指令的地址。25行到28行，使用了ret1标号来计算返回地址。在32行到34行，使用了$运算子来计算返回的地址。$运算子返回出现$这一行的当前地址。$ + 7表达式计算在36行的MOV指令的地址。

这两种计算返回地址的方法都是不方便的。第一种方法要求为每一次子程序调用定义一个标号。第二种方法不需要标号，但是需要仔细的思量。如果使用了近跳转来替代短跳转，那么与$相加的数就不会是7！幸运

的是，有一个更简单的方法来调用子程序。这种方法使用*堆栈*。

4.3 堆栈

许多CPU都支持内置堆栈。堆栈是一个先进后出(*LIFO*)的队列。它是以这种方式组织的一块内存区域。PUSH指令添加一个数据到堆栈中而POP指令从堆栈中移除数据。移除的数据就是最后入栈的数据(这就是称为先进后出队列的缘故)。

SS段寄存器指定包含堆栈的段(通常它与储存数据的段是一样)。ESP寄存器包含将要移除出栈数据的地址。这个数据也被称为栈顶。数据只能以双字的形式入栈。也就是说，你不可以将一个字节推入栈中。

PUSH指令通过把ESP减4来向堆栈中插入一个双字[1]，然后把双字储存到[ESP]中。POP指令从[ESP]中读取双字，然后再把ESP加4.下面的代码演示了这些指令如何工作，假定在ESP初始值为1000H。

```
1  push   dword 1     ; 1储存到0FFCh中，ESP = 0FFCh
2  push   dword 2     ; 2储存到0FF8h中，  ESP = 0FF8h
3  push   dword 3     ; 3储存到0FF4h中，  ESP = 0FF4h
4  pop    eax         ; EAX = 3，  ESP = 0FF8h
5  pop    ebx         ; EBX = 2，  ESP = 0FFCh
6  pop    ecx         ; ECX = 1，  ESP = 1000h
```

堆栈可以方便地用来临时储存数据。它同样可以用来形成子程序调用和传递参数和局部变量。

80x86同样提供一条PUSHA指令来把EAX，EBX，ECX，EDX，ESI，EDI和EBP寄存器的值推入栈中(不是以这个顺序)。POPA指令可以用来将它们移除出栈。

4.4 CALL和RET指令

80x86提供了两条使用堆栈的指令来使子程序调用变得快速而简单。CALL指令执行一个跳到子程序的无条件跳转，同时将下一条指令的地址*推入*栈中。RET指令*弹出*一个地址并跳转到这个地址去执行。使用这些指令的时候，正确处理堆栈以便RET指令能弹出正确的数值是非常重要的!

前面的例子可以使用这些新的指令来重写。把25行到34行改成：

```
mov    ebx, input1
call   get_int
```

[1]实际上，多字入栈也是可以的，但是在32位保护模式下，在堆栈上最好只操作单个双字。

```
      mov    ebx, input2
      call   get_int
```

同时把子程序get_int改成:

```
get_int:
      call   read_int
      mov    [ebx], eax
      ret
```

CALL和RET指令有几个优点：

- 它们很简单!
- 它们使子程序嵌套变得简单。注意：子程序get_int调用了read_int。这个调用将另一个地址压入到堆栈中了。在read_int代码的末尾是一条弹出返回地址的RET指令，通过执行指令重新回到get_int代码中去执行。然后，当get_int的RET指令被执行时，它弹出跳回到asm_main的返回地址。这个之所以能正确运行，是因此堆栈的LIFO特性。

记住弹出压入到堆栈的所有数据是非常重要的。例如，考虑下面的代码：

```
1 get_int:
2       call   read_int
3       mov    [ebx], eax
4       push   eax
5       ret                    ; 弹出EAX的值，没有返回地址！！
```

这个代码将不会正确返回!

4.5 调用约定

当调用了一个子程序，调用的代码和子程序(*被调用的代码*)必须协商好在它们之间如何传递数据。高级语言有标准传递数据的方法称为*调用约定*。要让高级语言接口于汇编语言，汇编语言代码就一定要使用与高级语言一样的约定。不同的编译器有不同的调用约定或者说不同的约定可能取决于代码如何被编译。(*例如*：是否进行了优化)。一个广泛的约定是：使用一条CALL指令来调用代码再通过RET指令返回。

所有PC的C编译器支持的调用约定将在本章的后续部分阶段进行描述。这些约定允许你创建*可重入的*子程序。一个可重入的子程序可以在程序中的任意一点被安全调用(甚至在子程序内部)。

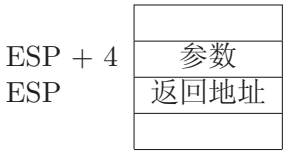

图 4.1:

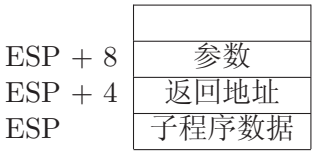

图 4.2:

4.5.1 在堆栈上传递参数

给子程序的参数需要在堆栈中传递。它们在CALL指令之前就已经被压入栈中了。和在C中是一样的，如果参数被子程序改变了，那么需要传递的是数据的*地址*，而不是*值*。如果参数的大小小于双字，它就需要在压入栈之前转换成双字。

在堆栈里的参数并没有由子程序弹出，取而代之的是：它们自己从堆栈中访问本身。为什么？

- 因为它们在CALL指令之前被压入栈中，所以返回时首先弹出的是返回地址(然后修改堆栈指针使其指向参数入栈以前的值)。
- 参数往往将会使用在子程序中几个的地方。通常在整个程序中，它们不可以保存在一个寄存器中，而应该储存在内存中。把它们留在堆栈里就相当于把数据复制到了内存中，这样就可以在子程序的任意一点访问数据。

当使用了间接寻址，80x86通过看间接寻址表达式里使用了什么寄存器来决定访问哪不同的段。ESP(和EBP)使用堆栈段，而EAX，EBX，ECX和EDX使用数据段。但是，这个对于保护模式通常是不重要的，因为对于保护模式数据段和堆栈段是同一段。

考虑 通过堆栈传递了一个参数的子程序。当子程序被调用了，堆栈状态如图 4.1。这个参数可以通过间接寻址访问到。(`[ESP+4]` [2])。

如果在子程序内部使用了堆栈储存数据，那么与ESP相加的数将要改变。例如：图 4.2展示了如果一个双字压入栈中后堆栈的状态。现在参数在`ESP + 8`中，而不在`ESP + 4`中。因此，引用参数时若使用ESP就很容易犯错了。为了解决这个问题，80386提供使用另外一个寄存器：EBP。这个寄存器的唯一目的就是引用堆栈中的数据。C调用约定要求子程序首先把EBP的值保存到堆栈中，然后再使EBP的值等于ESP。当数据压入或弹出堆栈时，这就允许ESP值被改变的同时EBP不会被改变。在子程序的结束处，EBP的原始值必须恢复出来(这就是为什么它在子程序的开始处被保存的缘故。)图 4.3展示了遵循这些约定的子程序的一般格式。

[2]使用间接寻址时，寄存器加上一个常量是合法的。许多更复杂的表达式也是合法的。这个话题将在下一章中介绍。

```
1 subprogram_label:
2       push    ebp             ; 把EBP的原始值保存在堆栈中
3       mov     ebp, esp        ; 新EBP的值 = ESP
4 ; subprogram code
5       pop     ebp             ; 恢复EBP的原始值
6       ret
```

图 4.3: 子程序的一般格式

ESP + 8	EBP + 8	参数
ESP + 4	EBP + 4	返回地址
ESP	EBP	保存的EBP值

图 4.4:

图 4.3中的第2行和第3行组成了一个子程序的大体上的*开始部分*。第5行和第6行组成了*结束部分*。图 4.4展示了刚执行完开始部分之后堆栈的状态。现在参数可以在子程序中的任何地方通过`[EBP + 8]`来访问，而不用担心在子程序中有什么数据压入到堆栈中了。

执行完子程序之后，压入栈中的参数必须移除掉。C调用约定规定调用的代码必须做这件事。其它约定可能不同。例如：Pascal 调用约定规定子程序必须移除参数。(RET指令的另一种格式可以很容易做这件事。)一些C编译器同样支持这种约定。关键字`pascal`用在函数的原型和定义中来告诉编译器使用这种约定。事实上，MS Windows API的C函数使用的`stdcall`调用约定同样以这种方式运作。这种方式有什么优点？它比C调用约定更有效一点。那为什么所有的C函数都使用C调用约定呢？一般说来，C允许一个函数的参数为变化的个数(*例如*：`printf`和`scanf`函数)。对于这种类型的函数，将参数移除出栈的操作在这次函数调用中和下次函数调用中是不同的。C调用约定能使指令简单地执行这种不同的操作。Pascal和stdcall调用约定执行这种操作是非常困难的。因此，Pascal调用约定(和Pascal语言一样)不允许有这种类型的函数。MS Windows只有当它的API函数不可以携带变化个数的参数时才可以使用这种约定。

图 4.5展示了一个将被调用的子程序如何使用C调用约定。第3行通过直接操作堆栈指针将参数移除出栈。同样可以使用`POP`指令来做这件事，但是常常使用在要求将无用的结果储存到一个寄存器的情况下。实际上，对于这种情况，许多编译器常常使用一条`POP ECX`来移除参数。编译器会使用`POP`指令来代替`ADD`指令，因为`ADD`指令需要更多的字节。但是，`POP`会改变ECX的值。下面是一个有两个子程序的例子，它们使用了上面讨论的C调用约定。54行(和其它行)展示了多个数据和文本段可以在同一个源文件中声明。进行连接处理时，它们将会组合成单一的数据段和文本段。把数据和

```
1       push    dword 1         ; 传递参数1
2       call    fun
3       add     esp, 4          ; 将参数移除出栈
```

图 4.5: 子程序调用示例

文本段分成单独的几段就允许数据定义在子程序代码附近，这也是子程序经常做的。

sub3.asm

```
1  %include "asm_io.inc"
2
3  segment .data
4  sum     dd   0
5
6  segment .bss
7  input   resd 1
8
9  ;
10 ; 伪码算法
11 ; i = 1;
12 ; sum = 0;
13 ; while( get_int(i, &input), input != 0 ) {
14 ;   sum += input;
15 ;   i++;
16 ; }
17 ; print_sum(num);
18 segment .text
19         global  _asm_main
20 _asm_main:
21         enter   0,0               ; 程序开始运行
22         pusha
23
24         mov     edx, 1            ; edx就是伪码里的i
25 while_loop:
26         push    edx               ; 保存i到堆栈中
27         push    dword input       ; 把input的地址压入堆栈
28         call    get_int
29         add     esp, 8            ; 将i和&input移除出栈
30
31         mov     eax, [input]
```

```
32          cmp     eax, 0
33          je      end_while
34
35          add     [sum], eax          ; sum += input
36
37          inc     edx
38          jmp     short while_loop
39
40  end_while:
41          push    dword [sum]         ; 将总数压入堆栈
42          call    print_sum
43          pop     ecx                 ; 将[sum]移除出栈
44
45          popa
46          leave
47          ret
48
49  ; 子程序get_int
50  ; 参数(顺序压入栈中)
51  ;   输入的个数(储存在[ebp + 12]中)
52  ;   储存输入字的地址(储存在[ebp + 8]中)
53  ; 注意:
54  ;   eax和ebx的值被毁掉了
55  segment .data
56  prompt  db      ") Enter an integer number (0 to quit): ", 0
57
58  segment .text
59  get_int:
60          push    ebp
61          mov     ebp, esp
62
63          mov     eax, [ebp + 12]
64          call    print_int
65
66          mov     eax, prompt
67          call    print_string
68
69          call    read_int
70          mov     ebx, [ebp + 8]
71          mov     [ebx], eax          ; 将输入储存到内存中
72
73          pop     ebp
```

```
74          ret                         ; 返回到调用代码
75
76  ; 子程序print_sum
77  ; 输出总数
78  ; 参数:
79  ;   需要输出的总数(储存在[ebp+8]中)
80  ; 注意: eax的值被毁掉了
81  ;
82  segment .data
83  result  db      "The sum is ", 0
84
85  segment .text
86  print_sum:
87          push    ebp
88          mov     ebp, esp
89
90          mov     eax, result
91          call    print_string
92
93          mov     eax, [ebp+8]
94          call    print_int
95          call    print_nl
96
97          pop     ebp
98          ret
```
sub3.asm

4.5.2 堆栈上的局部变量

堆栈可以方便地用来储存局部变量。这实际上也是C储存普通变量(或C lingo中的*自动变量*)的地方。如果你希望子程序是可重入的，那么使用堆栈存储变量是非常重要的。一个可重入的子程序不管在任何地方被调用都能正常运行，包括子程序本身。换句话说，可重入子程序可以*嵌套*调用。储存变量的堆栈同样在内存中。不储存在堆栈里的数据从程序开始到程序结束都使用内存(C称这种类型的变量为*全局变量*或*静态变量*)。储存在堆栈里的数据只有当定义它们的子程序是活动的时候才使用内存。

在堆栈中，局部变量恰好储存在保存的EBP值之后。它们通过在子程序的开始部分用ESP减去一定的字节数来分配存储空间。图 4.6展示了子程序新的骨架。EBP用来访问局部变量。考虑图 4.7中的C函数。图 4.8 展示如何用汇编语言编写等价的子程序。

图 4.9展示了执行完图 4.8中程序的开始部分后的堆栈状态。这一节的堆栈包含了参数，返回信息和局部变量，这样堆栈称为一个*堆栈帧*。C函数的每一次调用都会在堆栈上创建一个新的堆栈帧。

尽管ENTER和LEAVE指令事实上简化了开始部分和结束部分，但是它们并没有经常被使用。这是为什么呢？因为与等价的简单的指令相比，它们执行较慢！当你并不认为执行一队指令序列比执行一条复合的指令要快的时候，这就是一个榜样。

```
subprogram_label:
      push    ebp                 ; 保存原始EBP值到堆栈中
      mov     ebp, esp            ; 新EBP的值 = ESP
      sub     esp, LOCAL_BYTES    ; = #局部变量需要的字节数
; subprogram code
      mov     esp, ebp            ; 释放局部变量
      pop     ebp                 ; 恢复原始EBP值
      ret
```

图 4.6: 带有局部变量的子程序的一般格式

```
void calc_sum( int n, int * sump )
{
  int i, sum = 0;

  for( i=1; i <= n; i++ )
    sum += i;
  *sump = sum;
}
```

图 4.7: 求总数的C语言版

可以使用两条专门的指令来简化一个子程序的开始部分和结束部分，它们是为这个目的而专门设计的。ENTER指令执行开始部分的代码，而LEAVE指令执行结束部分。ENTER指令携带两个立即数。对于C调用约定，第二个操作数总是为0。第一个操作数是局部变量所需要的字节数。LEAVE指令没有操作数。图 4.10展示了如何使用这些指令。注意程序skeleton(图 1.7)同样使用了ENTER和LEAVE指令。

4.6 多模块程序

*多模块程序*是由不止一个目标文件组成的程序。这里出现的所有程序都是多模块程序。它们由C驱动目标文件和汇编目标文件(加上C库目标文件)组成。回忆一下连接程序将目标文件组合成一个可执行程序。连接程序必须把在一个模块(*也就是* 目标文件)中引用的每个变量匹配到定义该变量的模块。为了让模块A能使用定义在模块B里的变量，就必须使用extern(外部)指示符。在extern 指示符后面是用逗号隔开的变量列表。这个指示符告诉编译器把这些变量视为是模块*外部的*。也就是说，这些变量可以在这个模块中使用，但是却定义在另一模块中。asm_io.inc文件中就将read_int等程序定义为外部的。

```
1  cal_sum:
2        push    ebp
3        mov     ebp, esp
4        sub     esp, 4                  ; 为局部变量sum分配空间
5
6        mov     dword [ebp - 4], 0      ; sum = 0
7        mov     ebx, 1                  ; ebx (i) = 1
8  for_loop:
9        cmp     ebx, [ebp+8]            ; is i <= n?
10       jnle    end_for
11
12       add     [ebp-4], ebx            ; sum += i
13       inc     ebx
14       jmp     short for_loop
15
16 end_for:
17       mov     ebx, [ebp+12]           ; ebx = sump
18       mov     eax, [ebp-4]            ; eax = sum
19       mov     [ebx], eax              ; *sump = sum;
20
21       mov     esp, ebp
22       pop     ebp
23       ret
```

图 4.8: 求总数的汇编语言版

在编译语言中，缺省情况下变量不可以由外部程序访问。如果一个变量可以被一个模块访问，而这个模块又不是定义它的,那么在定义它的模块中，它一定被声明为*global(全局的)*。`global` 指示符就可以用来做这件事情。图 1.7的程序skeleton中的第13行定义了一个全局变量`_asm_main`。若没有这个声明，就可能会出错。为什么？因为C代码将会找不到*内部的*`_asm_main`变量。

下面是用两个模块重写的以前例子的代码。子程序(`get_int`和`print_sum`)在不同的源文件中，而不是在`_asm_main`程序中。

main4.asm

```
1  %include "asm_io.inc"
2
3  segment .data
4  sum     dd   0
5
```

ESP + 16	EBP + 12	sump
ESP + 12	EBP + 8	n
ESP + 8	EBP + 4	Return address
ESP + 4	EBP	saved EBP
ESP	EBP - 4	sum

图 4.9:

```
subprogram_label:
      enter  LOCAL_BYTES, 0     ; = #局部变量需要的字节数
; subprogram code
      leave
      ret
```

图 4.10: 通过使用ENTER和LEAVE指令，带有局部变量的子程序的一般格式

```
segment .bss
input   resd 1

segment .text
        global  _asm_main
        extern  get_int, print_sum
_asm_main:
        enter   0,0               ; 程序开始运行
        pusha

        mov     edx, 1            ; edx就是伪码中的i
while_loop:
        push    edx               ; 保存i到堆栈中
        push    dword input       ; 把input的地址压入堆栈
        call    get_int
        add     esp, 8            ; 将i和&input移除出栈

        mov     eax, [input]
        cmp     eax, 0
        je      end_while

        add     [sum], eax        ; sum += input

        inc     edx
        jmp     short while_loop
```

```
31
32 end_while:
33         push    dword [sum]        ; 将总数压入堆栈
34         call    print_sum
35         pop     ecx                ; 将总数[sum]移除出栈
36
37         popa
38         leave
39         ret
```

main4.asm

sub4.asm

```
1  %include "asm_io.inc"
2
3  segment .data
4  prompt  db      ") Enter an integer number (0 to quit): ", 0
5
6  segment .text
7          global  get_int, print_sum
8  get_int:
9          enter   0,0
10
11         mov     eax, [ebp + 12]
12         call    print_int
13
14         mov     eax, prompt
15         call    print_string
16
17         call    read_int
18         mov     ebx, [ebp + 8]
19         mov     [ebx], eax         ; 将输入储存到内存中
20
21         leave
22         ret                        ; 返回到调用代码
23
24 segment .data
25 result  db      "The sum is ", 0
26
27 segment .text
28 print_sum:
29         enter   0,0
30
31         mov     eax, result
32         call    print_string
```

```
33
34          mov     eax, [ebp+8]
35          call    print_int
36          call    print_nl
37
38          leave
39          ret
```
sub4.asm

上面的例子只有全局的代码变量；同样，全局数据变量也可以使用一模一样的方法。

4.7 C与汇编的接口技术

现今，完全用汇编书写的程序是非常少的。编译器能很好地将高级语言转换成有效的机器代码。因为用高级语言书写代码非常容易，所以高级语言变得很流行。此外，高级语言比汇编语言更容易移植！

当使用汇编语言时，我们经常将它使用在代码中的一小部分上。有两种使用汇编语言的方法：在C中调用汇编子程序或内嵌汇编。内嵌汇编允许程序员把汇编语句直接放入到C代码中。这样是非常方便的；但是，内嵌汇编同样存在缺点。汇编语言的书写格式必须是编译器使用的格式。目前没有一个编译器支持NASM格式。不同的编译器要求使用不同的格式。Borland和Microsoft要求使用MASM格式。DJGPP和Linux中gcc要求使用GAS[3] 格式。在PC机上，调用汇编子程序是更标准的技术。

在C中使用汇编程序通常是因为以下几个原因：

- 需要直接访问计算机的硬件特性，而用C语言很难或不可能做到。
- 程序执行必须尽可能地快，而且相比于编译器，程序员手动优化的代码更好。

最后一个原因不像它以前一样有根据。因为这些年编译器技术提高了，而且编译器通常可以产生非常有效的代码(特别是当开启编译器优化的时候)。调用汇编程序的缺点：可移植性和可读性减弱了。

绝大部分的C调用约定已经确定了。但是，还需要描述一些额外的特征。

4.7.1 保存寄存器

首先，C假定子程序保存了下面这几个寄存器的值：EBX，ESI，EDI，EBP，CS，DS，SS，ES。这并不意味着不能在子程序内部修改它们。相反，它表示如果子程序改变了它们的值，那么在子程序返回之前必须恢复

关键字`register`*可以使用在一个C变量声明中来暗示编译器：这个变量使用一个寄存器而不是内存空间。这些变量称为寄存器变量。现在的编译器会自动做这件事而不需要给它暗示。*

[3]GAS是所有的GNU编译器使用的汇编程序。它使用AT&T语法，这是一种完全不同于MASM，TASM 和NASM的语法。

```
1  segment .data
2  x             dd      0
3  format        db      "x = %d\n", 0
4
5  segment .text
6  ...
7        push    dword [x]       ; 将x的值压入栈中
8        push    dword format    ; 将format字符串的地址压入栈中
9        call    _printf         ; 注意下划线!
10       add     esp, 8          ; 从堆栈中移除参数
```

图 4.11: 调用printf

EBP + 12	x的值
EBP + 8	format字符串的地址
EBP + 4	返回地址
EBP	保存的EBP值

图 4.12: printf的堆栈结构

它们的原始值。EBX，ESI和EDI的值不能被改变，因为C将这些寄存器用于*寄存器变量*。通常都是使用堆栈来保存这些寄存器的原始值。

4.7.2 函数名

大多数C编译器都在函数名和全局或静态变量前附加一个下划线字符。例如，函数名f将指定为_f。因此，如果这是一个汇编程序，那么它必须标记为_f，而不是f。Linux gcc编译器并不附加任何字符。在可执行的Linux ELF下，对于C函数f，你只需要简单使用函数名f即可。但是，DJGPP的gcc却附加了一个下划线。注意，在汇编程序skeleton中(图 1.7)，主程序函数名是_asm_main。

4.7.3 传递参数

按照C调用约定，一个函数的参数将以一定顺序压入栈中，这个顺序与它们出现在函数调用里的顺序*相反*。

考虑这条C语句：printf("x = %d\n",x); 图 4.11展示了如何编译这条语句(用等价的NASM格式)。图 4.12展示了执行完printf函数的开始部分后，堆栈的状态。printf函数一个可以携带任意个参数的C语言库函数。C调用约定的规则就是专门为允许这些类型的函数而规定的。因为format字符串的地址最后压入堆栈，所以不管有多少参数传递到函数，

没有必要使用汇编语言来处理C中的可变参数函数。可以通过移植stdarg.h头文件中定义的宏来处理它们。看一本好的C语言的书来得到更详细的信息。

它在堆栈里的位置将总是EBP + 8。然后printf代码就可以通过看format字符串的位置来决定需要传递多少参数和在堆栈上如何找到它们。

当然，如果有错误发生，printf("x = %d\n")，printf代码仍然会将[EBP + 12]中的双字值输出，而这并不是x的值！

4.7.4 计算局部变量的地址

找到定义在data或bss段的变量的地址是非常容易的。基本上，连接程序做的就是这件事情。但是，要计算出在堆栈上的一个局部变量(或参数)的地址就不简单了。可是，当调用子程序的时候，这种需求是非常普通的。考虑传递一个变量(让我们称它为x)的地址到一个函数(让我们称它为foo)的情况。如果x处在堆栈的EBP − 8的位置，你不可以这样使用：

```
mov    eax, ebp - 8
```

为什么？因为指令MOV储存到EAX里的值必须能由汇编器计算出来(也就是说，它最后必须是一个常量)。但是，有一条指令能做这种需求的计算。它就是 LEA (即*Load Effective Address，载入有效地址*)。下面的代码就能计算出x的地址并将它储存到EAX中：

```
lea    eax, [ebp - 8]
```

现在EAX中存有了x的地址，而且当调用函数foo的时候，就可以将其压入到栈中。不要搞混了，这条指令看起来是从[EBP−8]中读数据；然而，这并不正确。LEA指令*永远不会*从内存中读数据。它仅仅计算出一个将会被其它指令使用到的地址，然后将这个地址储存到它的第一个操作数里。因为它并没有实际读内存，所以不指定内存大小(*例如：* dword)是必须的或说是允许的。

4.7.5 返回值

返回值不为空的C函数执行完后会返回一个值。C调用约定规定了这个要如何去做。返回值需通过寄存器传递。所有的整形类型(char，int，enum，等)通过EAX寄存器返回。如果它们小于32位，那么储存到EAX的时候，它们将被扩展成32位。(它们如何扩展取决于是有符号类型还是无符号类型。) 64位的值通过EDX:EAX寄存器对返回。浮点数储存在数学协处理器中的ST0寄存器中。(这个寄存器将在浮点数这一章来讨论。)

4.7.6 其它调用约定

所有的80x86 C编译器中都支持上面描述的标准C调用约定的规则。通常编译器也支持其它调用约定。当与汇编语言进行接口时，知道编译器调用你的函数时使用的是什么调用约定是*非常*重要的。通常，缺省时，使用的

是标准的调用约定；但是，并不总是这一种情况[4]。使用多种约定的编译器通常都拥有可以用来改变缺省约定的命令行开关。它们同样提供扩展的C语法来为单个函数指定调用约定。但是，各个编译器的这些扩展标准可以是不一样的。

GCC编译器允许不同的调用约定。一个函数的调用约定可以通过扩展语法__attribute__ 明确指定。例如，要声明一个返回值为空的函数f，它带有一个int参数，使用标准调用约定，需使用下面的语法来声明它的原型：

```
void f( int ) __attribute__ ((cdecl));
```

GCC同样支持*标准call* 调用约定。通过把cdecl替换成stdcall，上面的函数可以指定为使用这种约定。stdcall约定和cdecl约定的不同点是stdcall要求子程序将参数移除出栈(和Pascal调用约定一样)。因此，stdcall调用约定只能使用在带有固定参数的函数上(*也就是说*，不可以是函数printf和scanf)。

GCC同样支持称为regparm 的约定，这种约定告诉编译器前3个整形参数通过寄存器传递给函数，而不是通过堆栈。这是许多编译器支持的一个共同的优化模式。

Borland和Microsoft使用一样语法来声明调用约定。它们在C代码中加上关键字__cdecl和__stdcall。这些关键字用来修饰函数。在原型声明中，它们出现在函数名的前面例如，上面的函数f用Borland和Microsoft定义如下：

```
void __cdecl f( int );
```

每种调用约定都有各自的优缺点。cdecl调用约定的主要优点是它非常简单而且非常灵活。它可以用于任何类型的C函数和C编译器。使用其它约定会限制子程序的可移植性。它的主要缺点是与其它约定相比它执行较慢而且使用更多的内存(因为函数的每次调用都需要用代码将参数移除出栈。)。

stdcall调用约定的主要优点是相比于cdecl它使用较少的内存。在CALL指令之后，不需要清理堆栈。它的主要缺点是它不能使用于可变参数的函数。

使用寄存器传递参数的调用约定的优点是速度非常快。主要缺点是这种约定太复杂。有些参数可能在寄存器中，而另一些可能在堆栈中。

4.7.7 样例

下面是一个展示汇编程序如何与C程序接口的例子。(注意：这个程序并没有使用skeleton汇编程序(图 1.7)或driver.c模块。)

main5.c

[4]Watcom C编译器就是一个在缺省情况下*不*使用标准调用的例子。看Watcom的样例源文件代码来得到更详细的信息

```c
#include <stdio.h>
/* 汇编程序的原型声明 */
void calc_sum( int , int * ) __attribute__ ((cdecl));

int main( void )
{
  int n, sum;

  printf("Sum integers up to: ");
  scanf("%d", &n);
  calc_sum(n, &sum);
  printf("Sum is %d\n", sum);
  return 0;
}
```

main5.c

sub5.asm

```
; 子程序 _calc_sum
; 求整形1到n的和
; 参数:
;   n - 从1加到多少(储存在[ebp + 8])
;   sump - 指向总数储存地址的整形指针(储存在[ebp + 12])
; C伪码:
; void calc_sum( int n, int * sump )
; {
;   int i, sum = 0;
;   for( i=1;i <= n; i++ )
;     sum += i;
;   *sump = sum;
; }

segment .text
        global  _calc_sum
;
; 局部变量:
;   储存在[ebp-4]里的sum值
_calc_sum:
        enter   4,0               ; 在堆栈上为sum分配空间
        push    ebx               ; 重要!

        mov     dword [ebp-4],0   ; sum = 0
        dump_stack 1, 2, 4        ; 输出堆栈中从ebp-8到ebp+16的值
```

```
Sum integers up to: 10
Stack Dump # 1
EBP = BFFFFB70 ESP = BFFFFB68
 +16  BFFFFB80  080499EC
 +12  BFFFFB7C  BFFFFB80
  +8  BFFFFB78  0000000A
  +4  BFFFFB74  08048501
  +0  BFFFFB70  BFFFFB88
  -4  BFFFFB6C  00000000
  -8  BFFFFB68  4010648C
Sum is 55
```

图 4.13: sub5程序的运行示例

```
26          mov     ecx, 1              ; ecx是伪码中的i
27  for_loop:
28          cmp     ecx, [ebp+8]        ; 比较i和n
29          jnle    end_for             ; 如果i > n,则退出循环
30
31          add     [ebp-4], ecx        ; sum += i
32          inc     ecx
33          jmp     short for_loop
34
35  end_for:
36          mov     ebx, [ebp+12]       ; ebx = sump
37          mov     eax, [ebp-4]        ; eax = sum
38          mov     [ebx], eax
39
40          pop     ebx                 ; 恢复ebx的值
41          leave
42          ret
```

sub5.asm

为什么程序sub5.asm中的第 22行非常重要？因为C调用约定要求EBX的值不能被调用的函数更改。如果没有做这个，程序很可能不会正确运行。

第 25行演示了宏dump_stack如何运作。它的第一个参数只是一个数字标号，第二个参数决定需显示EBP以下多少个双字而第三个参数决定需显示EBP以上多少个双字。图 4.13展示了这个程序的运行示例。对于这次转储，你可以看到储存总数的双字地址是FBFFFFB80 (储存在EBP + 12)；n值为0000000A (储存在EBP + 8)；程序的返回地址为08048501 (储存在EBP + 4)；保存的EBP的值为BFFFFB88 (储存在EBP)；局部变量的值为0(储存在EBP - 4)；最后保存的EBX的值为4010648C (储存在EBP -

8)。

calc_sum函数可以这样重写：把sum当作返回值返回，替代使用的指针参数。因为sum是一个整形值，所以应保存到EAX寄存器中。main5.c文件中的第11行应该改成：

```
sum = calc_sum(n);
```

同样，calc_sum的原型也需要改变。下面是修改后的汇编代码：

sub6.asm

```
; 子程序 _calc_sum
; 求整形1到n的和 ; 参数:
;   n - 从1加到多少(储存在[ebp + 8]
; 返回值:
;   sum的值
; C伪码:
; int calc_sum( int n )
; {
;   int i, sum = 0;
;   for( i=1; i <= n; i++)
;     sum += i;
;   return sum;
; }
segment .text
        global  _calc_sum
;
; 局部变量:
;   储存在[ebp-4]里的sum值
_calc_sum:
        enter   4,0               ; 在堆栈上为sum分配空间

        mov     dword [ebp-4],0   ; sum = 0
        mov     ecx, 1            ; ecx是伪码中的i
for_loop:
        cmp     ecx, [ebp+8]      ; 比较i和n
        jnle    end_for           ; 如果i > n,则退出循环

        add     [ebp-4], ecx      ; sum += i
        inc     ecx
        jmp     short for_loop

end_for:
        mov     eax, [ebp-4]      ; eax = sum

```

```
segment .data
format       db "%d", 0

segment .text
...
      lea    eax, [ebp-16]
      push   eax
      push   dword format
      call   _scanf
      add    esp, 8
...
```

图 4.14: 在汇编程序中调用scanf函数

```
        leave
        ret
```

sub6.asm

4.7.8 在汇编程序中调用C函数

C与汇编接口的一个主要优点是允许汇编代码访问大型C库和用户写的函数。例如，如果你想调用一下scanf函数来从键盘读一个整形，该怎么办？图 4.14展示了完成这件事的代码。需要记住的非常重要的一点就是scanf函数遵循字面意义的C调用标准。这就意味着它保存了EBX，ESI和EDI寄存器的值；但是，EAX，ECX和EDX寄存器的值可能会被修改。事实上，EAX肯定会被修改，因为它将保存scanf调用的返回值。至于与C接口的其它例子，可以看用来产生asm_io.obj的asm_io.asm文件中的代码。

4.8 可重入和递归子程序

一个可重入子程序必须满足下面几个性质：

- 它不能修改代码指令。在高级语言中，修改代码指令是非常难的；但是在汇编语言中，一个程序要修改自己的代码并不是一件很难的事。例如：

  ```
  mov    word [cs:$+7], 5        ; 将5复制到前面七个字节的字中
  add    ax, 2                   ; 前面的语句将2改成了5!
  ```

  这些代码在实模式下可以运行，但是在保护模式下的操作系统上不行，因为代码段被标识为只读。在这些操作系统上，当执行了上面的第一行代码，程序将被终止。这种类型的程序从各个方面来看都非常差。它很混乱，很难维护而且不允许代码共享(看下面)。

- 它不能修改全局变量(比如在data和bss段里的数据)。所有的变量应储存在堆栈里。

书写可重入性代码有几个好处。

- 一个可重入子程序可以递归调用。
- 一个可重入程序可以被多个进程共享。在许多多任务操作系统上，如果一个程序有许多实例正在运行，那么只有一份代码的拷贝在内存中。共享库和DLL(*Dynamic Link Libraries，动态链接库*)同样使用了这种技术。
- 可重入子程序可以运行在*多线程* [5] 程序中。Windows 9x/NT和大多数类UNIX操作系统(Solaris，Linux，等)都支持多线程程序。

4.8.1 递归子程序

这种类型的子程序调用它们自己。递归可以是*直接的*或是*间接的*。当一个名为foo的子程序在foo内部调用自己就产生直接递归。当一个子程序虽然自己没有直接调用自己，但是其它子程序调用了它，就产生间接递归。例如：子程序foo可以调用bar且bar也可以调用foo。

递归子程序必须有一个*终止条件*。当这个条件为真时，就不再进行递归调用了。如果一个子程序没有终止条件或条件永不为真，那么递归将不会结束(非常像一个无穷循环)。

图 4.15展示了一个递归求n!的函数。在C中它可以这样被调用：

```
x = fact(3);            /* find 3! */
```

图 4.16展示了上面的函数调用的最深点的堆栈状态。

图 4.17展示了另一个更复杂的递归样例的C语言版而4.18展示了它的汇编语言版。对于f(3)，输出是什么？注意：每一次递归调用，ENTER指令都会在堆栈上给新的i值分配空间。因此，f的每一次递归调用都有它自己独立的变量i。若是在data段定义i为一双字，结果就不一样了。

4.8.2 回顾一下C变量的储存类型

C提供了几种变量储存类型。

global，全局 这些变量定义在任何函数的外面，且储存在固定的内存空间中(在data或bss段)，而且从程序的开始一直到程序的结束都存在。缺省情况下，它们能被程序中的任何一个函数访问；但是，如果它们被声明为static，那么只有在同一模块中的函数才能访问它们(*也就是说，* 依照汇编的术语，这个变量是内部的，不是外部的)。

[5] 一个多线程程序同时有多条线程在执行。也就是说，程序本身是多任务的。

```
1  ; 求n!
2  segment .text
3        global _fact
4  _fact:
5        enter  0,0
6
7        mov    eax, [ebp+8]     ; eax = n
8        cmp    eax, 1
9        jbe    term_cond        ; 如果n <= 1，则终止
10       dec    eax
11       push   eax
12       call   _fact            ; eax = fact(n-1)
13       pop    ecx              ; 结果在eax中
14       mul    dword [ebp+8]    ; edx:eax = eax * [ebp+8]
15       jmp    short end_fact
16 term_cond:
17       mov    eax, 1
18 end_fact:
19       leave
20       ret
```

图 4.15: 求n!的递归函数

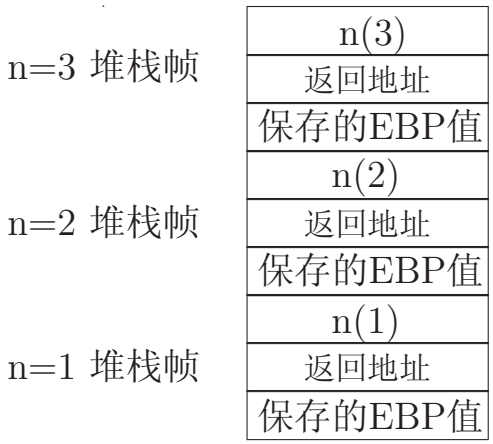

图 4.16: n!函数的堆栈帧

```
1 void f( int x )
2 {
3   int i;
4   for( i=0; i < x; i++ ) {
5     printf("%d\n", i);
6     f(i);
7   }
8 }
```

图 4.17: 另一个例子(C语言版)

static，静态 在一个函数中，它们是被声明为静态的*局部*变量。(不幸的是，C使用关键字`static`有两种目的！)这些变量同样储存在固定的内存空间中(在`data`或`bss`段)，但是只能被定义它的函数直接访问。

automatic，自动 它是定义在一个函数内的C变量的缺省类型。当定义它们的函数被调用了，这些变量就被分配在堆栈上，而当函数返回了又从堆栈中移除。因此，它们没有固定的内存空间。

register，寄存器 这个关键字要求编译器使用寄存器来储存这个变量的数据。这仅仅是一个*要求*。编译器并*不*一定要遵循。如果变量的地址使用在程序的任意的地方，那么就不会遵循(因为寄存器没有地址)。同样，只有简单的整形数据可以是寄存器变量。结构类型不可以；因为它们的大小不匹配寄存器！C编译器通常会自动将普通的自动变量转换成寄存器变量，而不需要程序员给予暗示。

volatile，不稳定 这个关键字告诉编译器这个变量值随时都会改变。这就意味着当变量被更改了，编译器不能做出任何推断。通常编译器会将一个变量的值暂时存在寄存器中，而且在出现这个变量的代码部分使用这个寄存器。但是，编译器不能对不稳定类型的变量做这种类型的优化。一个不稳定变量的最普遍的例子就是：它可以被多线程程序的两个线程修改。考虑下面的代码：

```
1 x = 10;
2 y = 20;
3 z = x;
```

如果`x`可以被另一个线程修改。那么其它线程可以会在第1行和第3行之间修改`x`的值，以致于`z`将不会等于10.但是，如果`x`没有被声明为不稳定类型，编译器就会推断`x`没有改变，然后再将`z`置为10。

不稳定类型的另一个使用就是避免编译器为一个变量使用一个寄存器。

```
%define i ebp-4
%define x ebp+8          ; useful macros
segment .data
format       db "%d", 10, 0     ; 10 = '\n'
segment .text
      global _f
      extern _printf
_f:
      enter  4,0           ; 在堆栈上为i分配空间

      mov    dword [i], 0  ; i = 0
lp:
      mov    eax, [i]      ; is i < x?
      cmp    eax, [x]
      jnl    quit

      push   eax           ; 调用printf
      push   format
      call   _printf
      add    esp, 8

      push   dword [i]     ; 调用f
      call   _f
      pop    eax

      inc    dword [i]     ; i++
      jmp    short lp
quit:
      leave
      ret
```

图 4.18: 另一个例子(汇编语言版)

第5章

数组

5.1 介绍

一组*数组*是内存中的一个连续数据列表块。在这个列表中的每个元素必须是同一种类型而且使用恰好同样大小的内存字节来储存。因为这些特性，数组允许通过数据在数组里的位置(或下标)来对它进行有效的访问。如果知道了下面三个细节，任何元素的地址都可以计算出来：

- 数组第一个元素的地址
- 每个元素的字节数
- 这个元素的下标

0(正如在C中)作为数组的第一个元素的下标是非常方便的。使用其它值作为第一个下标也是可能的，但是这将把计算弄得很复杂。

5.1.1 定义数组

在data和bss段中定义数组

在data段定义一个初始化了的数组，可以使用标准的db，dw，等等 指示符。NASM同样提供了一个有用的指示符，称为TIMES ，它可以用来反复重复一条语句，而不需要你手动来复制它。图 5.1展示关于这些的几个例子。

在bss段定义一个示初始化的数组，可以使用resb，resw，等等 指示符。记住，这些指示符包含一个指定保留多少个内存单元的操作数。图 5.1同样展示了关于这种类型定义的几个例子。

```
segment .data
; 定义10个双字的数组并初始化为1,2,..,10
a1          dd 1, 2, 3,4, 5, 6, 7, 8, 9, 10
; 定义10个字的数组并初始化为0
a2          dw 0, 0, 0, 0,0, 0, 0, 0, 0, 0
; 像以前一样使用TIMES
a3          times 10 dw 0
;定义一个字节数组,其中包含200个0和100个1
a4          times 200 db 0
            times 100 db 1

segment .bss
; 定义10个双字的数组，示未始化
a5          resd  10
;定义10了个字的数组，示未始化
a6          resw  100
```

图 5.1: 定义数组

以局部变量的方式在堆栈上定义数组

在堆栈上定义一个局部数组变量没有直接的方法。像以前一样，你可以首先计算出*所有*局部变量需要的全部字节，包括数组，然后再用ESP减去这个数值(或者直接使用ENTER指令)。例如，如果一个函数需要一个字符变量，两个双字整形和一个包含50个元素的字数组，你将需要$1+2\times4+50\times2=109$个字节。但是，为了保持ESP在双字的边界上，被ESP减的数值必须是4的倍数(这个例子中是112。)图 5.2展示了两种可能的方法。第一种排序的未使用部分用来保持双字在双字边界上，这样可以加速内存的访问。

5.1.2 访问数组中的元素

跟C不同的是，在汇编语言中没有[ ]运算符。要访问数组中的一个元素，必须将它的地址计算出来。考虑下面两个数组的定义：

```
array1      db      5, 4, 3, 2, 1       ; 字节数组
array2      dw      5, 4, 3, 2, 1       ; 字数组
```

下面是使用这些数组的例子：

```
        mov     al, [array1]            ; al = array1[0]
        mov     al, [array1 + 1]        ; al = array1[1]
        mov     [array1 + 3], al        ; array1[3] = al
        mov     ax, [array2]            ; ax = array2[0]
```

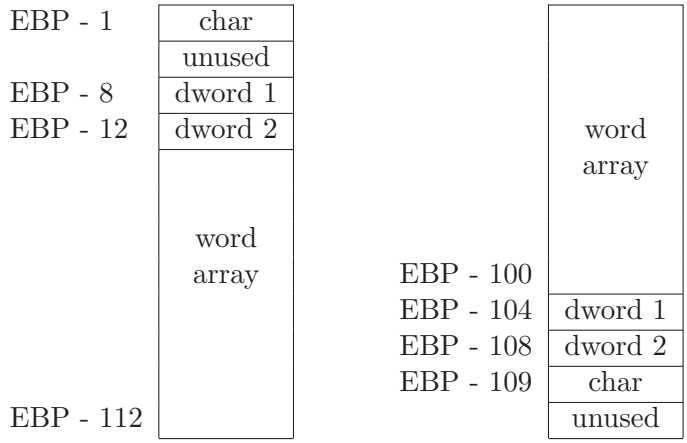

图 5.2: 堆栈上的数组排列

```
1       mov     ebx, array1             ; ebx = array1的地址
2       mov     dx, 0                   ; dx将储存总数
3       mov     ah, 0                   ; ?
4       mov     ecx, 5
5   lp:
6       mov     al, [ebx]               ; al = *ebx
7       add     dx, ax                  ; dx += ax (不是al!)
8       inc     ebx                     ; bx++
9       loop    lp
```

图 5.3: 对一组数组的元素求和(版本1)

```
5       mov     ax, [array2 + 2]        ; ax = array2[1] (不是array2[2]!)
6       mov     [array2 + 6], ax        ; array2[3] = ax
7       mov     ax, [array2 + 1]        ; ax = ??
```

在第5行，引用了字数组中的元素1,而不是元素2。为什么？因为字是两个字节的单元，所以移动到字数组的下一元素，你必须向前移动两个字节，而不是一个。第7行将从第一个元素中读取一个字节再从第二个元素中读取一个字节。在C中，编译器会根据一个指针的类型来决定使用指针运算的表达式需移动多少字节，而程序员就不需要管这些了。然而，在汇编语言中，当需要从一个元素移动到另一个元素时，它取决于程序员认为的数组元素的大小。

图 5.3展示了一个代码片段：对前面样例代码中的数组array1中的元素进行了求和。第 7行，AX与DX相加。为什么不是AL？首先，ADD指令的两个操作数必须为同样的大小。其次，这样做对于对字节求和后得到一个太大以致不能匹配一个字节的总数是非常容易的。通过使用DX，达

```
1       mov     ebx, array1             ; ebx = array1的地址
2       mov     dx, 0                   ; dx将储存总数
3       mov     ecx, 5
4   lp:
5       add     dl, [ebx]               ; dl += *ebx
6       jnc     next                    ; 若没有进位，则跳转到next
7       inc     dh                      ; inc dh
8   next:
9       inc     ebx                     ; bx++
10      loop    lp
```

图 5.4: 对一组数组的元素求和(版本2)

```
1       mov     ebx, array1             ; ebx = array1的地址
2       mov     dx, 0                   ; dx将储存总数
3       mov     ecx, 5
4   lp:
5       add     dl, [ebx]               ; dl += *ebx
6       adc     dh, 0                   ; dh += carry flag + 0
7       inc     ebx                     ; bx++
8       loop    lp
```

图 5.5: 对一组数组的元素求和(版本3)

到65,535的总数是允许。然而，认识到AH同样被相加了是非常重要的。这就是第3行为什么AH被置为0的缘故了。[1]

图 5.4和图5.5展示了两种可以替换的方法来计算总数。斜体字的行替换了图 5.3中的第6行和第7行。

5.1.3 更高级的间接寻址

不要惊讶，间接寻址经常与数组一起使用。最普遍的间接内存引用格式为：

```
[ base reg(基址寄存器) + factor(系数)*index reg(变址寄存器) +
                         constant(常量)]
```

其中：

[1] 置AH为0隐含地表示AL是一个无符号数值。如果它是有符号的，恰当的操作是在第6行和第7行之间插入一条CBW指令。

基址寄存器 可以是EAX，EBX，ECX，EDX，EBP，ESP，ESI或EDI寄存器。

系数 可以是1，2，4或8。(如果是1，系数是可以省略的。)

变址寄存器 可以是EAX，EBX，ECX，EDX，EBP，ESI或EDI寄存器。(注意ESP并不可以。)

常量 为一个32位的常量。这个常量可以是一个变量(或变量表达式)。

5.1.4 例子

这有一个使用数组并将它传递给函数的例子。它使用`array1c.c`程序(下面列出的)作为驱动程序，而不是`driver.c`程序。

array1.asm

```
%define ARRAY_SIZE 100
%define NEW_LINE 10

segment .data
FirstMsg        db   "First 10 elements of array", 0
Prompt          db   "Enter index of element to display: ", 0
SecondMsg       db   "Element %d is %d", NEW_LINE, 0
ThirdMsg        db   "Elements 20 through 29 of array", 0
InputFormat     db   "%d", 0

segment .bss
array           resd ARRAY_SIZE

segment .text
        extern  _puts, _printf, _scanf, _dump_line
        global  _asm_main
_asm_main:
        enter   4,0     ; 在EBP - 4处的局部双字变量
        push    ebx
        push    esi

; 将数组初始化为100，99，98，97，  ...

        mov     ecx, ARRAY_SIZE
        mov     ebx, array
init_loop:
        mov     [ebx], ecx
        add     ebx, 4
```

```
29          loop    init_loop
30
31          push    dword FirstMsg          ; 显示FirstMsg
32          call    _puts
33          pop     ecx
34
35          push    dword 10
36          push    dword array
37          call    _print_array            ; 显示数组的前10个元素
38          add     esp, 8
39
40  ; 提示用户输入元素下标
41  Prompt_loop:
42          push    dword Prompt
43          call    _printf
44          pop     ecx
45
46          lea     eax, [ebp-4]       ; eax = 局部双字变量的地址
47          push    eax
48          push    dword InputFormat
49          call    _scanf
50          add     esp, 8
51          cmp     eax, 1                  ; eax = scanf的返回值
52          je      InputOK
53
54          call    _dump_line  ; 转储当前行的剩余部分并重新开始
55          jmp     Prompt_loop             ; 如果输入无效
56
57  InputOK:
58          mov     esi, [ebp-4]
59          push    dword [array + 4*esi]
60          push    esi
61          push    dword SecondMsg         ; 显示元素的值
62          call    _printf
63          add     esp, 12
64
65          push    dword ThirdMsg          ; 显示元素20-29
66          call    _puts
67          pop     ecx
68
69          push    dword 10
70          push    dword array + 20*4      ; array[20]的地址
```

```
71          call    _print_array
72          add     esp, 8
73
74          pop     esi
75          pop     ebx
76          mov     eax, 0            ; 返回到C中
77          leave
78          ret
79
80  ;
81  ; 程序_print_array
82  ; 调用的C程序把双字数组的元素当作有符号整形来显示。
83  ; C函数原型:
84  ; void print_array( const int * a, int n);
85  ; 参数:
86  ;   a - 指向需要显示的数组的指针(堆栈上ebp+8处)
87  ;   n - 需要显示的整数个数(堆栈上ebp+12处)
88
89  segment .data
90  OutputFormat    db   "%-5d %5d", NEW_LINE, 0
91
92  segment .text
93          global  _print_array
94  _print_array:
95          enter   0,0
96          push    esi
97          push    ebx
98
99          xor     esi, esi                  ; esi = 0
100         mov     ecx, [ebp+12]             ; ecx = n
101         mov     ebx, [ebp+8]              ; ebx = 数组的地址
102 print_loop:
103         push    ecx                       ; printf将会改变ecx!
104
105         push    dword [ebx + 4*esi]       ; 将array[esi]压入堆栈
106         push    esi
107         push    dword OutputFormat
108         call    _printf
109         add     esp, 12                   ; 移除参数(留下ecx!)
110
111         inc     esi
112         pop     ecx
```

```
113         loop    print_loop
114
115         pop     ebx
116         pop     esi
117         leave
118         ret
```

array1.asm

array1c.c

```
1  #include <stdio.h>
2
3  int asm_main( void );
4  void dump_line( void );
5
6  int main()
7  {
8    int ret_status ;
9    ret_status = asm_main();
10   return ret_status ;
11 }
12
13 /* 函数dump_line* 转储输入缓冲区中当前行的所有字符*/
14 void dump_line()
15 {
16   int ch;
17
18   while( (ch = getchar()) != EOF && ch != '\n')
19     /*空程序体*/ ;
20 }
```

array1c.c

再看一下LEA指令

LEA指令不仅仅可以用来计算地址，也可以用作其它目的。一个相当普遍的目的是快速计算。考虑下面的代码：

```
lea     ebx, [4*eax + eax]
```

这条代码有效地将5 × EAX的值储存到EBX中。相比于使用MUL指令，使用LEA既简单又快捷。但是，你必须认识到在方括号里的表达式*必须*是一个合法的间接地址。因此，例如，这个指令就不可以用来快速乘6。

```
1   mov     eax, [ebp - 44]          ; ebp - 44是i的位置
2   sal     eax, 1                   ; i乘以2
3   add     eax, [ebp - 48]          ; 加上j
4   mov     eax, [ebp + 4*eax - 40]  ; ebp - 40是a[0][0]的地址
5   mov     [ebp - 52], eax          ; 将结果储存到x中(在ebp - 52中)
```

图 5.6: x = a[i][j]的汇编语言表示法

5.1.5 多维数组

多维数组和已经讨论的普遍的一维数组相比，差异并不是很大。事实上，在内存中，它们的描述方法和普遍的一维数组是一样的。

二维数组

不要感到意外，最简单的多维数组就是二维数组。一个二维数组通常以网格的形式来表示元素。每个元素通过两个下标来确定。按照惯例，第一个下标用来确定元素的行值，而第二下标用来确定元素的列值。

考虑一个三行二列的数组，像这样定义：

```
int a [3][2];
```

C编译器将为这个数组保留6($= 2 \times 3$)个整形数的空间，而且像下面一样来映射元素：

下标	0	1	2	3	4	5
元素	a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]

这张表试图展示的是a[0][0]引用的元素储存在6个元素的一维数组的开始。元素a[0][1]储存在下一个位置(下标 1)，以此类推。在内存中，二维数组的各个行都是连续储存的。一行的最后一个元素后面紧跟着下一行的第一个元素。这就是所谓的数组*依行*表示法，这也是C/C++编译器表示数组的表示方法。

在依行表示法中，编译器如何确定a[i][j]出现在哪？一个简单的公式将通过i和j来计算下标。在这个例子中，公式为$2i + j$。并不难看出如何得到这个公式。每一行有两个元素大小；所以行i的第一个元素的位置为$2i$。然后该行的j列的位置可以通过j和$2i$相加得到。这个分析同样展示了如何产生N列数组的公式：$N \times i + j$。注意，这个公式并不依赖于行的总数。

作为一个例子，我们来看看*gcc*如何编译下面的代码(使用上面定义的数组a)：

```
x = a[i][j];
```

图 5.6展示了这条语句翻译成的汇编语言。因此编译器实质上将代码转换成：

```
x = *(&a[0][0] + 2*i + j);
```

而且事实上，程序员可以以这种方法来书写，也可以得到同样的结果。

选择依行的数组表示法并没有什么魔力。依列的表示法同样可以工作：

下标	0	1	2	3	4	5
元素	a[0][0]	a[1][0]	a[2][0]	a[0][1]	a[1][1]	a[2][1]

在依列表示法中，各列被连续储存。元素`[i][j]`储存在$i+3j$位置中。其它语言(例如：FORTRAN)使用依列表示法。当与多种语言进行代码接口时，这是非常重要的。

大于二维

对于大于二维的数组，应用了同样原理的想法。考虑一个三维数组：

```
int b [4][3][2];
```

在内存中，这个数组将被当作四个大小为`[3][2]`的二维数组被连续储存。下面的表展示了它如何运作：

下标	0	1	2	3	4	5
元素	b[0][0][0]	b[0][0][1]	b[0][1][0]	b[0][1][1]	b[0][2][0]	b[0][2][1]
下标	6	7	8	9	10	11
元素	b[1][0][0]	b[1][0][1]	b[1][1][0]	b[1][1][1]	b[1][2][0]	b[1][2][1]

计算`b[i][j][k]`的位置的公式是$6i+2j+k$。其中6由`[3][2]`数组的大小决定。一般来说，对于维数为`a[L][M][N]`的数组，元素`a[i][j][k]`的位置将是$M\times N\times i+N\times j+k$。再次需要注意的是，第一个维的元素个数(`L`)并没有出现在公式中。

对于更高维的数组，可以通过推广来做同样的处理。对于一个从D_1到D_n的n数组，下标为i_1到i_n的元素的位置可以由下面这个公式得到：

$$D_2\times D_3\cdots\times D_n\times i_1+D_3\times D_4\cdots\times D_n\times i_2+\cdots+D_n\times i_{n-1}+i_n$$

或者对于超级的数学技客，它可以用更简洁地书写为：

$$\sum_{j=1}^{n}\left(\prod_{k=j+1}^{n}D_k\right)i_j$$

这里你可以断定作者是物理专业的。(或者说是引用了FORTRANO语言的一个免费样品？)

第一维D_1，并没有出现在公式中。

对于依列表示法，普遍的公式将是：

$$i_1+D_1\times i_2+\cdots+D_1\times D_2\times\cdots\times D_{n-2}\times i_{n-1}+D_1\times D_2\times\cdots\times D_{n-1}\times i_n$$

或表示为超级数学技客的书写方法：

$$\sum_{j=1}^{n}\left(\prod_{k=1}^{j-1}D_k\right)i_j$$

在这种情况下，是最后一维D_n，不出现在公式中。

在C语言中，传递多维数组参数

多维数组的依列表示法在C编程有一个直接的效果。对于一维的数组，当任何具体的元素被放置到内存中时，数组的大小并不需要计算出来。但这对于多维数组是不正确的。为了访问这些数组的元素，除了第一维的元素个数，编译器必须知道其它所有维数的元素个数。当一个函数的原型带有一个多维数组参数时，这就变得很明显了。下面的代码将不会被编译：

```
void f( int a[ ][ ] );  /* 没有维数信息 */
```

但是，下面的代码就会被编译：

```
void f( int a[ ][2] );
```

任何有两列的二维数组可以传递给这个函数。第一维的元素个数是不需要的[2]。

不要被这类函数的原型搞混了：

```
void f( int * a[ ] );
```

它定义了一个一维的整形指针数组。(它可以附带用来创建一个像二维数组一样运作的数组。)

对于更高维的数组，除了第一维的元素个数，数组参数的其它维数的必须指定。例如，一个四维的数组参数可以像这样被传递：

```
void f( int a[ ][4][3][2] );
```

5.2 数组/串处理指令

80x86家族的处理器提供了几条与数组一起使用的指令。这些指令称为*串处理指令*。它们使用变址寄存器(ESI和EDI)来执行一个操作，然后这两个寄存器自动地进行增1或减1操作。FLAGS寄存器里的*方向标志位*(DF) 决定了这些变址寄存器是增加还是减少。有两条指令用来修改方向标志位：

CLD 清方向标志位。这种情况下，变址寄存器是自动增加的。

STD 置方向标志位。这种情况下，变址寄存器是自动减少的。

80x86编程中的一个*非常*普遍的错误就是忘记了把方向标志位明确地设置为正确的状态。这就经常导致代码大部分情况下能正常工作(当方向标志位恰好就是所需要的状态时)，但并不能正常工作在*所有*情况下。

[2]它可以在这里指定，但是会被编译器忽略。

LODSB AL = [DS:ESI] ESI = ESI ± 1	STOSB [ES:EDI] = AL EDI = EDI ± 1
LODSW AX = [DS:ESI] ESI = ESI ± 2	STOSW [ES:EDI] = AX EDI = EDI ± 2
LODSD EAX = [DS:ESI] ESI = ESI ± 4	STOSD [ES:EDI] = EAX EDI = EDI ± 4

图 5.7: 从串取和存入串指令

```
1  segment .data
2  array1  dd  1, 2, 3, 4, 5, 6, 7, 8, 9, 10
3
4  segment .bss
5  array2  resd 10
6
7  segment .text
8        cld                    ; 别忘记这个!
9        mov    esi, array1
10       mov    edi, array2
11       mov    ecx, 10
12 lp:
13       lodsd
14       stosd
15       loop  lp
```

图 5.8: 从串取和存入串的例子

5.2.1 读写内存

最简单的串处理指令是读或写内存或同时读写内存。它们可以每次读或写一个字节，一个字或一个双字。图 5.7中的小段伪码展示了这些指令的作用。这有点需要注意的。首先，ESI是用来读的，而EDI是用来写的。如果你能记得SI代表*Source Index，源变址寄存器*和DI代表*Destination Index，目的变址寄存器*，那么这个就很容易记住了。 其次，注意包含数据的寄存器是固定的(AL，AX或EAX)。最后，注意存入串指令使用ES来决定需要写的段，而不是DS。在保护模式下，这通常不是问题，因为它只有一个数据段，而ES应自动地初始化为引用DS(和DS一样)。但是，在实模式下，将ES初始化为正确的段值对于程序员来说是非常重要的[3]。图 5.8展示了一

[3]另一复杂的地方是你不可以直接使用一条MOV指令来将DS寄存器中的值复制到ES寄存器中。取而代之的是，你需要使用两条MOV指令，先把DS寄存器的值复制到一个通用寄存器(比如：AX)中，然后再把这个通用寄存器的值复制到ES中。

```
MOVSB   byte [ES:EDI] = byte [DS:ESI]
        ESI = ESI ± 1
        EDI = EDI ± 1
MOVSW   word [ES:EDI] = word [DS:ESI]
        ESI = ESI ± 2
        EDI = EDI ± 2
MOVSD   dword [ES:EDI] = dword [DS:ESI]
        ESI = ESI ± 4
        EDI = EDI ± 4
```

图 5.9: 串传送指令

```
1  segment .bss
2  array  resd 10
3
4  segment .text
5        cld                    ; 别忘记这个!
6        mov    edi, array
7        mov    ecx, 10
8        xor    eax, eax
9        rep stosd
```

图 5.10: 零数组的例子

个使用这些指令将一个数组复制到另一数组的例子。

LODSx和STOSx指令的联合使用(如图 5.8中的13和14行)是非常普遍的。事实上，一条MOVSx串处理指令可以用来完成这个联合使用的功能。图 5.9描述了这些指令执行的操作。图 5.8的第13和14行可以用MOVSD指令来替代，能得到同样的效果。唯一的区别就是在循环时EAX寄存器根本就不会被使用。

5.2.2 REP前缀指令

80x86家族提供了一个特殊的前缀指令[4]，称为REP，它可以与上面的串处理指令一同使用。这个前缀告诉CPU重复执行下条串处理指令一个指定的次数。ECX寄存器用来计算重复的次数(和在LOOP指令中的使用是一样的)。使用REP前缀，在图 5.8中的12到15行的循环体可以替换成一行：

```
rep movsd
```

图 5.10展示了另一个例子：得到一个零数组。

[4]前缀指令并不是一个指令，它是放置在串处理指令前面的一个特殊的字节，用来修改指令的行为。其它前缀同样可以用来进行跨段内存访问

CMPSB	比较字节[DS:ESI]和[ES:EDI] ESI = ESI ± 1 EDI = EDI ± 1
CMPSW	比较字[DS:ESI]和[ES:EDI] ESI = ESI ± 2 EDI = EDI ± 2
CMPSD	比较双字[DS:ESI]和[ES:EDI] ESI = ESI ± 4 EDI = EDI ± 4
SCASB	比较AL和[ES:EDI] EDI ± 1
SCASW	比较AX和[ES:EDI] EDI ± 2
SCASD	比较EAX和[ES:EDI] EDI ± 4

图 5.11: 串比较指令

5.2.3 串比较指令

图 5.11展示了几个新的串处理指令：它们可以用来比较内存和内存或内存和寄存器。在比较和查找数组方面，它们是很有用的。它们会像CMP指令一样设置FLAGS寄存器。CMPSx 指令比较相应的内存空间，而SCASx 根据一指定的值扫描内存空间。

图 5.12展示了一个代码小片断：在一个双字数组中查找数字12。行 10里的SCASD指令总是对EDI进行加4操作，即使找到了需要的数值。因此，如果你想得到数组中数12的地址，就必须用EDI减去4(正如行 16所做的)。

5.2.4 REPx前缀指令

还有其它可以用在串比较指令中的，像REP一样的前缀指令。图 5.13展示了两个新的前缀并描述了它们的操作。REPE 和REPZ作用是一样的(REPNE和REPNZ也是一样)。如果重复的串比较指令因为比较的结果而终止了，变址寄存器同样会进行增量操作而ECX也会进行减1操作；但是FLAGS寄存器将仍然保持着重复终止时的状态。因此，使用ZF标志位来确定重复的比较是因为一次比较而结束，还是因为ECX等于0而结束是可能的。

在重复比较之后，为什么你不能单单看ECX是否等于0?

图 5.14展示了一个样例子代码片断：确定两个内存块是否相等。例子中行 7中的JE指令是用来检查前面指令的结果。如果重复的比较是因为它找到了两个不相等的字节而终止，那么ZF标志位就为0,也就不会执行分支；但是，如果比较是因为ECX等于0而结束，那么ZF标志位就为1,而且代码将分支到equal标号处。

```
segment .bss
array        resd 100

segment .text
      cld
      mov    edi, array    ; 指向数组开始部分的指针
      mov    ecx, 100      ; 元素的个数
      mov    eax, 12       ; 需扫描的个数
lp:
      scasd
      je     found
      loop   lp
 ; 若没有找到，执行的代码
      jmp    onward
found:
      sub    edi, 4          ; edi现在指向数组中的12
 ; 若找到了，执行的代码
onward:
```

图 5.12: 查找的例子

REPE, REPZ	当ZF标志位为1或重复次数不超过ECX时，重复执行指令
REPNE, REPNZ	当ZF标志为0或重复次数不超过ECX时，重复执行指令

图 5.13: REPx前缀指令

5.2.5 样例

这一节包含了一个汇编源文件，源文件中包含了几个应用串处理指令进行数组操作的函数。其中大部分都是C语言库中的常见函数的复制品。

memory.asm

```
global _asm_copy, _asm_find, _asm_strlen, _asm_strcpy

segment .text
; 函数 _asm_copy
; 复制内存块
; C原型
; void asm_copy( void * dest, const void * src, unsigned sz);
; 参数:
;   dest - 指向复制操作的目的缓冲区的指针
;   src  - 指向复制操作的源缓冲区的指针
```

```
segment .text
      cld
      mov    esi, block1         ; block1的地址
      mov    edi, block2         ; block2的地址
      mov    ecx, size           ; block以字节表示的大小
      repe   cmpsb               ; 当ZF为1时，重复执行
      je     equal               ; 如果ZF为1,跳转到equal
   ; 如果block不相等，执行的代码
      jmp    onward
equal:
   ; 如果相等，执行的代码
onward:
```

图 5.14: 比较内存块

```
;   sz   - 需要复制的字节数

; 下面，定义了一些有用的变量

%define dest [ebp+8]
%define src  [ebp+12]
%define sz   [ebp+16]
_asm_copy:
        enter   0, 0
        push    esi
        push    edi

        mov     esi, src        ; esi = 复制操作的源缓冲区的地址
        mov     edi, dest       ; edi = 复制操作的目的缓冲区的地址
        mov     ecx, sz         ; ecx = 需要复制的字节数

        cld                     ; 清方向标志位
        rep     movsb           ; 执行movsb操作ECX次

        pop     edi
        pop     esi
        leave
        ret


; 函数 _asm_find
```

```
37  ; 根据一给定的字节值查找内存
38  ; void * asm_find( const void * src, char target, unsigned sz);
39  ; 参数:
40  ;   src    - 指向需要查找的缓冲区的指针
41  ;   target - 需要查找的字节值
42  ;   sz     - 在缓冲区中的字节总数
43  ; 返回值:
44  ;   如果找到了target，返回指向在缓冲区中第一次出现target的地方的指针
45  ;   否则
46  ;     返回NULL
47  ; 注意: target是一个字节值，但是被当作一个双字压入栈中。
48  ;       字节值储存在低8位上。
49  ;
50  %define src    [ebp+8]
51  %define target [ebp+12]
52  %define sz     [ebp+16]
53
54  _asm_find:
55          enter   0,0
56          push    edi
57
58          mov     eax, target     ; al包含需查找的值
59          mov     edi, src
60          mov     ecx, sz
61          cld
62
63          repne   scasb           ; 扫描直到ECX == 0或[ES:EDI] == AL才停止
64
65          je      found_it        ; 如果ZF为1,则找到了相应的值
66          mov     eax, 0          ; 如果没有找到，返回NULL指针
67          jmp     short quit
68  found_it:
69          mov     eax, edi
70          dec     eax              ; 如果找到了，则返回(DI - 1)
71  quit:
72          pop     edi
73          leave
74          ret
75
76
77  ; 函数 _asm_strlen
78  ; 返回字符串的大小
```

```
79  ; unsigned asm_strlen( const char * );
80  ; 参数:
81  ;   src - 指向字符串的指针
82  ; 返回值:
83  ;   字符串中的字符数(以0结束，若碰到0,则不再计数) (储存在EAX中)
84
85  %define src [ebp + 8]
86  _asm_strlen:
87          enter   0,0
88          push    edi
89
90          mov     edi, src        ; edi = 指向字符串的指针
91          mov     ecx, 0FFFFFFFFh ; 使用可能的ECX的最大值
92          xor     al,al           ; al = 0
93          cld
94
95          repnz   scasb           ; 扫描终止符0
96
97  ;
98  ; repnz将会多执行一步，所以ECX的大小是FFFFFFFE,
99  ; 而不是FFFFFFFF
100 ;
101         mov     eax,0FFFFFFFEh
102         sub     eax, ecx        ; 大小 = 0FFFFFFFEh - ecx
103
104         pop     edi
105         leave
106         ret
107
108 ; 函数 _asm_strcpy
109 ; 复制一个字符串
110 ; void asm_strcpy( char * dest, const char * src);
111 ; 参数:
112 ;   dest - 指向进行复制操作的目的字符串
113 ;   src  - 指向进行复制操作的源字符串
114 ;
115 %define dest [ebp + 8]
116 %define src  [ebp + 12]
117 _asm_strcpy:
118         enter   0,0
119         push    esi
120         push    edi
```

```
121
122         mov     edi, dest
123         mov     esi, src
124         cld
125 cpy_loop:
126         lodsb                   ; 载入AL & inc si
127         stosb                   ; 储存AL & inc di
128         or      al, al          ; 设置条件标志位
129         jnz     cpy_loop        ; 如果没到终止符0,则继续
130
131         pop     edi
132         pop     esi
133         leave
134         ret
```

memory.asm

memex.c

```
1  #include <stdio.h>
2
3  #define STR_SIZE 30
4  /* 原型 */
5
6  void asm_copy( void *, const void *, unsigned ) __attribute__ ((cdecl));
7  void * asm_find( const void *,
8                   char target, unsigned ) __attribute__ ((cdecl));
9  unsigned asm_strlen( const char * ) __attribute__ ((cdecl));
10 void asm_strcpy( char *, const char * ) __attribute__ ((cdecl));
11
12 int main()
13 {
14   char st1[STR_SIZE] = "test string";
15   char st2[STR_SIZE];
16   char * st;
17   char   ch;
18
19   asm_copy(st2, st1, STR_SIZE);  /* 复制源字符串的所有30个字符到目的
     字符串 */
20    printf("%s\n", st2);
21
22    printf("Enter a char: ");  /* 在字符串中查找一字节值 */
23   scanf("%c%*[^\n]", &ch);
24   st = asm_find(st2, ch, STR_SIZE);
```

```
25    if ( st )
26      printf("Found it: %s\n", st);
27    else
28      printf("Not found\n");
29
30    st1[0] = 0;
31    printf("Enter string :");
32    scanf("%s", st1);
33    printf("len = %u\n", asm_strlen(st1));
34
35    asm_strcpy( st2, st1);      /* 复制源字符串的有效字符到目的字符串 */
36    printf("%s\n", st2 );
37
38    return 0;
39  }
```

memex.c

第6章

浮点

6.1 浮点表示法

6.1.1 非整形的二进制数

在第一章讨论数制的时候，我们只讨论了整形。显然，和十进制一样，其它进制必须也能表示非整形数。在十进制中，在小数点右边的数字关联了10的负乘方值：

$$0.123 = 1 \times 10^{-1} + 2 \times 10^{-2} + 3 \times 10^{-3}$$

不必惊讶，二进制也是以同样的方法表示：

$$0.101_2 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0.625$$

这个办法与第一章中的整形办法相结合就可以用来转换一个一般数值：

$$110.011_2 = 4 + 2 + 0.25 + 0.125 = 6.375$$

将十进制转换成二进制也不是很难。一般来说，需将十进制数分成两块：整数部分和分数部分。使用第一章中的方法来将整数部分转换成二进制。分数部分的转换可以使用下面描述的方法。

考虑一个用$a, b, c, \ldots$标记比特位的二进制分数。这个数用二进制表示为：

$$0.abcdef\ldots$$

将此数乘2.新得到的数的二进制表示将是：

$$a.bcdef\ldots$$

注意，第一个比特位现在在权值为1的位置。用0替换a得到：

$$0.bcdef\ldots$$

$$
\begin{array}{rcl@{\qquad\qquad}l@{\quad}c@{\quad}l}
0.5625 \times 2 & = & 1.125 & \text{第一个比特位} & = & 1 \\
0.125 \times 2 & = & 0.25 & \text{第二个比特位} & = & 0 \\
0.25 \times 2 & = & 0.5 & \text{第三个比特位} & = & 0 \\
0.5 \times 2 & = & 1.0 & \text{第四个比特位} & = & 1
\end{array}
$$

图 6.1: 将0.5625转换成二进制

$$
\begin{array}{rcl}
0.85 \times 2 & = & 1.7 \\
0.7 \times 2 & = & 1.4 \\
0.4 \times 2 & = & 0.8 \\
0.8 \times 2 & = & 1.6 \\
0.6 \times 2 & = & 1.2 \\
0.2 \times 2 & = & 0.4 \\
0.4 \times 2 & = & 0.8 \\
0.8 \times 2 & = & 1.6
\end{array}
$$

图 6.2: 将0.85转换成二进制

再乘以2得到：

$$b.cdef\ldots$$

现在第二个比特位(b)在权值为1的位置。重复这个过程，直到得到了需要的尽可能多的比特位。图 6.1展示了一个实例：将0.5625转换成二进制。这种方法当分数部分为0了才停止。

另一个例子，将23.85转换成二进制。将整数部分($23 = 10111_2$)转换成二进制是容易的，但是分数部分呢？图 6.2展示了这个计算的开始部分。如果你仔细看了这个数值，就会发现一个无限循环。这就意味着0.85是一个无限循环的二进制数(与基数为10的无限循环十进制数相对应)[1]。这里显示了这个数的计算模式。在这个模式中，你可以看到$0.85 = 0.11\overline{0110}_2$。因此，$23.85 = 10111.11\overline{0110}_2$。

[1]不要大惊小怪，一个数值在一种数制下是一个无限循环，而在另一种数制下可能不是。考虑下$\frac{1}{3}$，以十进制表示，它是一个无穷数，但是以三进制(基数为3)表示，它就为0.1_3。

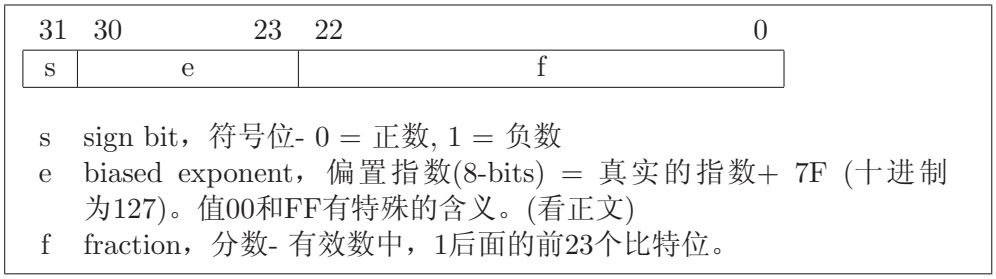

图 6.3: IEEE单精度

上面计算的一个重要结论是23.85不可以用有限的比特位来*精确*表示成二进制数。(就像$\frac{1}{3}$不能表示成有限的十进制数。)正如这一章展示的，C语言中的`float`和`double`变量是以二进制储存的。因此，类似23.85的数值不能精确地储存到这些变量中。只能储存23.85的近似值。

为了简化硬件，采用固定的格式来储存浮点数。这种格式采用科学计数法(但是是在二进制中，是2的乘方,不是10)。例如，23.85或$10111.11011001100110\ldots_2$将储存为：

$$1.011111011001100110\ldots \times 2^{100}$$

(其中指数(100)是二进制形式)。*规范*的浮点数有下面的形式：

$$1.sssssssssssssss \times 2^{eeeeeee}$$

其中1.*sssssssssssss*是*有效数*而*eeeeeeee*是*指数*。

6.1.2 IEEE浮点表示法

IEEE(Institute of Electrical and Electronic Engineers，电气与电子工程师学会)是一个国际组织，它已经设计了存储浮点数的特殊的二进制格式。这种格式应用在大多数(但不是全部)现在的电脑上。通常电脑本身的硬件就支持它。例如，Intel的数学协处理器(从Pentium开始，就嵌入到所有它的CPU中了)就使用它。IEEE为不同的精度定义了不同的格式：单或双精度。在C语言中，`float`变量使用单精度，而`double`变量使用双精度。

Intel数学协处理器使用第三种，更高的精度，称为*扩展精度*。事实上，在数学协处理器自身里的所有数据都是这种格式。当数据从协处理器储存到内存中时，将自动转换成单或双精度。[2]跟IEEE的浮点双精度格式相比，扩展精度使用了一种有细微差别的格式，所以将不在这讨论。

IEEE单精度

单精度浮点使用32个比特位来编码数字。通常它精确到小数点后七位。相比于整数，浮点数的储存格式更复杂。图 6.3展示了IEEE单精度数的基

[2] 有些编译器的(例如Borland) `long double`类型使用这种扩展精度。但是，其它的编译器的`double`和`long double`都使用双精度。(在ANSI C就允许这样做。)

$e=0$ and $f=0$	表示数0(它不可以被规范化)。注意这儿有+0和-0之分。
$e=0$ and $f\neq 0$	表示一个*非规范数*。它们将在下一节中讨论。
$e=\text{FF}$ and $f=0$	表示无穷大(∞)，包括正无穷大和负无穷大。
$e=\text{FF}$ and $f\neq 0$	表示一个不可以定义的结果，称为*NaN* (Not a Number，不是数)。

表 6.1: f和e的特殊值

本格式。这种格式有几个古怪的地方。负的浮点数并不使用补码表示法。它们使用符号量值表示法。如图显示，第31位决定数的符号。

二进制的指数并不会直接储存。取而代之的是将指数和7F的和储存到位23 30中。这个*偏置指数*总是非负的。

分数部分假定是一个规范的有效数(格式为1.*sssssssss*)。因为第一个比特位总是1,所以领头的1是*不储存的！*这就允许在后面储存一额外的比特位，稍微地扩展了精度。这个想法称为*隐藏一的表示法.*

你必须永远记住：这些字节41 BE CC CD可以用不同的方法解释，使用什么方法解释取决于程序如何使用它们。因为，当作为一个单精度浮点数时，它表示23.850000381，但是当它作为一个双字整形时，它表示1,103,023,309！CPU并不知道哪种才是正确的解释！

怎样储存23.85呢？首先，它是个正数，所以符号位为0。其次，真实的指数为4,所以偏置指数为7F + 4 = 83_{16}。最后，分数部分应表示为01111101100110011001100 (记住领头的1是隐藏的)。把这些放到一起得到(为了帮助澄清浮点格式的不同部分，符号位和分数部分都加了下划线，而且所有的比特位都分成了四个比特位一组。)：

$$\underline{0}\,100\ 0001\ 1\,\underline{011\ 1110\ 1100\ 1100\ 1100\ 1100}_2 = 41\text{BECCCC}_{16}$$

这不是准确的23.85(因为它是一个无限循环的二进制数)。如果你将上面的数值转换回十进制形式，你会发现它大约等于23.849998474。这个数与23.85非常接近，但是它并不准确。实际上，在C语言中，23.85的描述和上面的是一样的。因为该数的精确描述被截去后的最左边的位为1,所以最后一个比特位经四舍五入后为1。因此单精度数23.85将表示成十六进制41 BE CC CD。将这个转换成十进制得23.850000381，这个数就更接近23.85。

怎么描述-23.85呢？只需要改变符号位得：C1 BE CC CD。*不要使用补码！*

IEEE浮点格式中，e和f的某些组合有特殊的含义。表 6.1描述了这些特殊的值。溢出或除以0将产生一个无穷数。一个无效的操作将产生一个不确定的结果，例如：试图求一个负数的平方根，将两个无穷数相加，等等。

规范的单精度数的数量级范围为从1.0×2^{-126} ($\approx 1.1755\times 10^{-35}$) 到$1.11111\ldots\times 2^{127}$ ($\approx 3.4028\times 10^{35}$)。

非规范化数

非规范化数可以用来表示那些值太小了以致于不能以规范格式描述的数(*也就是*小于1.0×2^{-126})。例如：考虑下数$1.001_2\times 2^{-129}$ ($\approx 1.6530\times 10^{-39}$)。在约定的规范格式中，这个指数太小了。但是，它可以用非规范

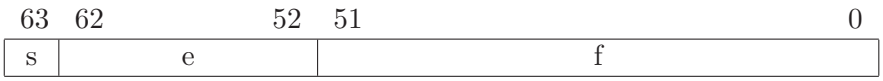

图 6.4: IEEE双精度

的格式来描述：$0.01001_2 \times 2^{-127}$。为了储存这个数，偏置指数被置为0(看表 6.1)，而且分数部分是以2^{-127}方式书写得到的所有有效数(*也就是说*储存了所有的比特位，包括小数点左边的1).1.001×2^{-129}将表示成：

0 000 0000 0 001 0010 0000 0000 0000 0000

IEEE双精度

IEEE双精度使用64位来表示数字，而且通常精确到小数点后15位。如图 6.4所示，基本的结构和单精度是非常相似的。只是相比于单精度，它使用了更多的位来描述偏置指数(11)和分数(52)。

更大范围的偏置指数会导致两个后果。一是计算的将是真实指数和3FF(1023)的和(而不是单精度中的7F)。二是，允许描述更大范围的真实的指数(因此也可以描述更大范围的数量级)。双精度的数量级范围大约为从10^{-308}到10^{308}。

双精度值中增加的有效位是增大分数字段的原因。

作为一个例子，再次考虑下。偏置指令用十六进制表示为$4 + 3FF = 403$。因此，该数用双精度表示为：

0 100 0000 0011 0111 1101 1001 1001 1001 1001 1001 1001 1001 1001 1001 1010

或在十六进制中为40 37 D9 99 99 99 99 9A。如果你将它转换回十进制，你将得到23.8500000000000014 (这有12个0！)，这个数就更接近23.85。

双精度和单精度一样有一些特殊的值[3]。非规范化数同样也是一样的。最主要的区别是双精度的非规范数使用2^{-1023}替换2^{-127}。

6.2 浮点运算

电子计算机里的浮点运算和持续精确的数学运算是不同的。数学中，所有的数都可以精确表示。但就如前面的章节所示，在电子计算机里，许多数不能用有限个比特位来描述。所有的计算都在一定的精度下执行。在这节的例子中，为了简单化，将使用8位的有效数。

[3]唯一的区别是：对于无穷数和不确定的值，偏置指数是7FF，而不是FF。

6.2.1 加法

要将两个浮点数相加，它们的指数必须是相等的。如果它们并不相等，那么通过移动较小指数的数的有效数来使它们相等。例如：考虑$10.375 + 6.34375 = 16.71875$或在十进制中：

$$\begin{array}{rr} & 1.0100110 \times 2^3 \\ + & 1.1001011 \times 2^2 \\ \hline \end{array}$$

这两个数字的指数不一样，所以通过移动有效数使指数相同，然后再相加：

$$\begin{array}{rr} & 1.0100110 \times 2^3 \\ + & 0.1100110 \times 2^3 \\ \hline & 10.0001100 \times 2^3 \end{array}$$

注意，移位丢掉了1.1001011×2^2中的末尾的1,经过四舍五入后得0.1100110×2^3。加法的结果，10.0001100×2^3 (or 1.00001100×2^4)等于10000.110_2或16.75。这个数并不等于准确的答案(16.71875)！它只是一个近似值，是在进行加法操作时四舍五入后的应有误差。

认识到在电子计算机(或计算器)里的浮点运算得到的结果经常是近似值是非常重要的。对于电子计算机里的浮点运算，算术法则不总是对的。算术中假定的无穷精度是任何电子计算机都无法做的。例如，算术法则告诉我们$(a + b) - b = a$；但是，在电子计算机里，并不能完全保证它正确。

6.2.2 减法

减法和加法一样运作，而且有和加法一样的问题。作为一个例子，考虑$16.75 - 15.9375 = 0.8125$：

$$\begin{array}{rr} & 1.0000110 \times 2^4 \\ - & 1.1111111 \times 2^3 \\ \hline \end{array}$$

移位1.1111111×2^3后得到(四舍五入) 1.0000000×2^4

$$\begin{array}{rr} & 1.0000110 \times 2^4 \\ - & 1.0000000 \times 2^4 \\ \hline & 0.0000110 \times 2^4 \end{array}$$

$0.0000110 \times 2^4 = 0.11_2 = 0.75$ 它并不完全正确的。

6.2.3 乘法和除法

对于乘法，有效数执行乘法操作而指数执行相加操作。考虑$10.375 \times 2.5 = 25.9375$：

$$\begin{array}{rr} & 1.0100110 \times 2^3 \\ \times & 1.0100000 \times 2^1 \\ \hline & 10100110 \\ + & 10100110 \\ \hline & 1.10011111000000 \times 2^4 \end{array}$$

当然，真正的结果需四舍五入成8位，得：

$$1.1010000 \times 2^4 = 11010.000_2 = 26$$

除法更复杂，但是也有同样的四舍五入的误差问题。

6.2.4 分支程序设计

这一节的重点是浮点运算的结果并不准确。程序员必须意识到这点。一个程序员经常犯的浮点运算错误就是在假定一个运算是精确的情况下，用它们去比较。例如，考虑一个执行复杂运算的f(x)函数和一个求这个函数的根的程序[4]。你可能会试图用下面的语句来检查x是不是一个根：

```
if ( f(x) == 0.0 )
```

但是，如果f(x)返回1×10^{-30}又该怎么办呢？这个数的最合适的含义是x是一个实根的*非常*好的近似值。可能没有一个IEEE浮点值x能恰好返回0，因为f(x)的四舍五入误差。

一个比较好的方法是使用：

```
if ( fabs(f(x)) < EPS )
```

其中的EPS是一个宏，定义为一个非常小的正数(比如说1×10^{-10})。当f(x)非常接近0时，它就为真。一般来说，一个浮点数(譬如x)和另一个浮点数(y)的比较，需使用：

```
if ( fabs(x - y)/fabs(y) < EPS )
```

6.3 数字协处理器

6.3.1 硬件

早期的Intel处理器并没有提供支持浮点操作的硬件。这并不意味着它们不可以执行浮点操作。它仅仅表示它们需要通过由许多非浮点指令组成的程序来执行这些操作。对于早期的系统，Intel提供了一片额外的称为数学

[4] 函数的根是满足$f(x) = 0$条件的x值

*协处理器*的芯片。相比于使用软件程序，数学协处理器拥有能快速执行许多浮点操作的机器指令(在早期的处理器上，至少快10倍！)。8086/8088的协处理器为8087。80286的协处理器为80287，80386的为80387。80486DX处理器将数学协处理器内置到80486中了。[5]从Pentium开始，所有生产的80x86处理器都内置数学协处理器；但是，它依然被规划成好像它是一个分离的单元。即使是早期没有协处理器的系统都可以安装一个模拟数学协处理器的软件。当一个程序执行了一条协处理器指令时，这个模拟软件包将自动激活并执行一个软件程序来得到与真实协处理器一样的结果(虽然，毫无疑问，会比较慢)。

数学协处理器有八个浮点数寄存器。每个寄存器储存着80位的数据。在这些寄存器中，浮点数总是储存成80位的扩展精度。这些寄存器称为ST0，ST1，ST2，... ST7。浮点寄存器与主CPU中的整形寄存器的使用方法是不同的。浮点寄存器被当作一个堆栈来管理。回想一下堆栈是一个*后进先出* (LIFO)队列。ST0总是指向栈顶的值。所有新的数都被加入到栈顶中。已经存在的数被压入到堆栈中，为了为新来的数提供空间。

在数学协处理器中同样有一个状态寄存器。它有几个标志位。只有4个用来比较的标志位将会提到：C_0, C_1, C_2 and C_3。这些位的使用将在以后讨论。

6.3.2 指令

为了很容易地将普通的CPU指令和协处理器指令区分开来，所有的协处理器助词符都是以F开头。

导入和储存

用来将数据导入到协处理器寄存器栈顶的指令有几条：

FLD *source*	从内存导入一个浮点数到栈顶。*source*可以是单，双或扩展精度数或是一个协处理器寄存器。
FILD *source*	从内存中读出一个*整形数*，将它转换成浮点数，再将结果储存到栈顶。*source*可以是字，双字或四字。
FLD1	将1储存到栈顶。
FLDZ	将0储存到栈顶。

将堆栈中的数据储存到内存的指令同样也有几条。其中有几条指令当它们储存好一个数后，会将这个数从栈中*弹出*(*也就是*删除)。

[5]但是，80486SX并*没有*内置数学协处理器。这些机器有分离的80487SX芯片。

FST *dest*	将栈顶的值(ST0)储存到内存中。*dest*可以是单，双精度数或是一个协处理器寄存器。
FSTP *dest*	像FST一样，将栈顶的值储存到内存中；但是，当储存完这个数后，它的值将被弹出栈。*dest*可以是单，双或扩展精度数或是一个协处理器寄存器。
FIST *dest*	将栈顶的值转换成整形后再储存到内存中。*dest*可以是字或双字。堆栈本身的值是不改变的。浮点数如何转换成整形取决于协处理器的*控制字*中的某些比特位。这是一个特殊的(非浮点)字寄存器，用来控制协处理器如何工作。缺省情况下，控制字会被初始化，以便于当需要转换成整形时，它会四舍五入成最接近的整形数。但是，FSTCW (Store Control Word，储存控制字)和FLDCW (Load Control Word，导入控制字)指令可以用来改变这种行为。
FISTP *dest*	它和FIST是一样，除了两件事：栈顶的值会被弹出，*dest*同样可以是四字的。

同样有两条其它的指令用来从堆栈自身中移动或删除数据。

FXCH ST*n*	将堆栈中的ST0的值和ST*n*的值相互交换(其中*n*是一个从1到7的寄存器号)。
FFREE ST*n*	通过标记寄存器为未被使用或为空来释放堆栈中的一个寄存器。

加法和减法

每一条加法指令都是计算ST0和另一个操作数的和。结果总是储存到一个协处理器寄存器中。

FADD *src*	ST0 += *src*。*src*可以是任何协处理器寄存器或内存中的单或双精度数。
FADD *dest*, ST0	*dest* += ST0。*dest*可以是任何协处理器寄存器。
FADDP *dest* or FADDP *dest*, ST0	*dest* += ST0然后再被弹出栈。*dest*可以是任何协处理器寄存器。
FIADD *src*	ST0 += (float) *src*。ST0和一个整形相加。*src*必须是内存中的字或双字。

减法指令是加法指令的两倍，因为在减法中，操作数的次序是重要的。(*也就是说*，$a+b=b+a$，但是，$a-b \neq b-a$！)。对于每一条指令，都有一条跟它次序相反的反向指令。这些反向指令要都是以R或RP结尾。图 6.5展示了一小段代码：对一个双字数组的元素求和。在第10和第13行中，你必须指定内存操作数的大小。否则汇编器将不会知道内存操作数是一个单精度浮点数(双字)还是双精度数(四字)。

```
1  segment .bss
2  array        resq SIZE
3  sum          resq 1
4
5  segment .text
6        mov    ecx, SIZE
7        mov    esi, array
8        fldz                  ; ST0 = 0
9  lp:
10       fadd   qword [esi]    ; ST0 += *(esi)
11       add    esi, 8         ; 移动到下个双字
12       loop   lp
13       fstp   qword sum      ; 将结果储存到sum中
```

图 6.5: 数组求和的例子

FSUB *src*	ST0 -= *src*。*src*可以是任何协处理器寄存器或内存中单，双精度数。
FSUBR *src*	ST0 = *src* - ST0。*src*可以是任何协处理器寄存器或内存中单，双精度数。
FSUB *dest*, ST0	*dest* -= ST0。*dest*可以是任何协处理器寄存器。
FSUBR *dest*, ST0	*dest* = ST0 - *dest*。*dest*可以是任何协处理器寄存器。
FSUBP *dest* or FSUBP *dest*, ST0	*dest* -= ST0然后被弹出栈。*dest*可以是任何协处理器寄存器。
FSUBRP *dest* or FSUBRP *dest*, ST0	*dest* = ST0 - *dest*然后被弹出栈。*dest*可以是任何协处理器寄存器。
FISUB *src*	ST0 -= (float) *src*。用ST0减去一个整数。*src*必须是内存中的一个字或双字。
FISUBR *src*	ST0 = (float) *src* - ST0。用一个整数减去ST0。*src*必须是内存中的一个字或双字。

乘法和除法

乘法指令和加法指令完全类似。

FMUL *src*	ST0 *= *src*。*src*可以是任何协处理器寄存器或内存中的单或双精度数。
FMUL *dest*, ST0	*dest* *= ST0。*dest*可以是任何协处理器寄存器。
FMULP *dest* or FMULP *dest*, ST0	*dest* *= ST0然后被弹出栈。*dest*可以是任何的协处理器寄存器。
FIMUL *src*	ST0 *= (float) *src*。ST0与一个整数相乘。*src*必须是内存中的一个字或双字。

不要惊讶，除法指令和减法指令非常类似。除以0结果将是一个无穷数。

FDIV *src*	ST0 /= *src*。*src*可以是任何协处理器寄存器或内存中的单或双精度数。
FDIVR *src*	ST0 = *src* / ST0。*src*可以是任何协处理器寄存器或内存中的单或双精度数。
FDIV *dest*, ST0	*dest* /= ST0。*dest*可以是任何协处理器寄存器。
FDIVR *dest*, ST0	*dest* = ST0 / *dest*。*dest*可以是任何协处理器寄存器。
FDIVP *dest* or FDIVP *dest*, ST0	*dest* /= ST0然后被弹出栈。*dest*可以是任何协处理器寄存器。
FDIVRP *dest* or FDIVRP *dest*, ST0	*dest* = ST0 / *dest*然后被弹出栈。*dest*可以是任何协处理器寄存器。
FIDIV *src*	ST0 /= (float) *src*。ST0除以一个整数。*src*必须是内存中的一个字或双字。
FIDIVR *src*	ST0 = (float) *src* / ST0。一个整数除以ST0。The *src*必须是内存中的一个字或双字。

比较

协处理器同样能执行浮点数的比较操作。FCOM家族的指令就是执行比较操作的。

FCOM *src*	比较ST0和*src*。*src*可以是协处理器寄存器或内存中的单或双精度数。
FCOMP *src*	比较ST0和*src*，然后再弹出堆栈。*src*可以是协处理器寄存器或内存中的单或双精度数。
FCOMPP	比较ST0和ST1，然后执行两次出栈操作。
FICOM *src*	比较ST0和(float) *src*。*src*可以是内存中的一个整形字或整形双字。
FICOMP *src*	比较ST0和(float)*src*，然后再弹出堆栈。*src*可以是内存中的一个整形字或整形双字。
FTST	比较ST0和0。

```
1  ;      if ( x > y )
2  ;
3        fld     qword [x]        ; ST0 = x
4        fcomp   qword [y]        ; 比较ST0和y
5        fstsw   ax               ; 将C状态标志位储存到FLAGS中
6        sahf
7        jna     else_part        ; 如果x不大于y，则跳转到else_part
8  then_part:
9        ; then部分的代码
10       jmp     end_if
11 else_part:
12       ; else部分的代码
13 end_if:
```

图 6.6: 比较指令的例子

这些指令会改变协处理器状态寄存器中的C_0，C_1，C_2和C_3比特位的值。不幸的是，CPU直接访问这些位是不可能的。条件分支指令使用FLAGS寄存器，而不是协处理器中的状态寄存器。但是，使用几条新的指令可以相当容易地将状态字的比特位传递到FLAGS寄存器上相同的比特位中。

FSTSW *dest*	存储协处理器状态字到内存的一个字或AX寄存器中。
SAHF	将AH寄存器中的值储存到FLAGS寄存器中。
LAHF	将FLAGS寄存器中的比特位导入到AH寄存器中。

图 6.6展示了一小段样例代码。第5行和第6行将C_0，C_1，C_2和C_3比特位传递到FLAGS寄存器相同的比特位中了。传递了这些比特位，所以它们就类似于两个无符号整形的比较结果。这也是为什么第7行使用JNA指令的缘故。

Pentium处理器(和它以后的处理器(Pentium II and III))支持两条新比较指令，用来直接改变CPU中FLAGS寄存器的值。

FCOMI *src*	比较ST0和*src*。*src*必须是一个协处理器寄存器。
FCOMIP *src*	比较ST0和*src*，然后再弹出堆栈。*src*必须是一个协处理器寄存器。

图 6.7展示了一个子程序例子：使用FCOMIP指令来找出两个双精度数的较大值。不要把这些指令和整形比较函数(FICOM 和FICOMP)混起来。

杂项指令

这一节包括了协处理器提供的其它杂项指令。

FCHS	ST0 = - ST0改变ST0的符号位
FABS	ST0 = |ST0| 求ST0的绝对值
FSQRT	ST0 = $\sqrt{\text{ST0}}$ 求ST0的平方根
FSCALE	ST0 = ST0 × $2^{\lfloor \text{ST1} \rfloor}$ 快速执行ST0乘以2的几次方的操作。ST1并不会从协处理器堆栈中移除。图 6.8展示了一个如何使用这些指令的例子。

6.3.3 样例

6.3.4 二次方程求根公式

第一个例子展示了如何用汇编语言编写二次方程求根公式。回忆一下如何用求根公式计算二次方程等式的根：

$$ax^2 + bx + c = 0$$

公式本身给出两个根x：x_1和x_2。

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

在平方根$(b^2 - 4ac)$里的表达式称为*判别式*。这个值在判定是下面三种可能的根的情况中的哪一种时非常有用。

1. 只有一个实根。当$b^2 - 4ac = 0$时
2. 有两个实根。当$b^2 - 4ac > 0$时
3. 有两个复根。当$b^2 - 4ac < 0$时

这是一个使用汇编子程序的小的C程序：

quadt.c

```
1  #include <stdio.h>
2
3  int quadratic( double, double, double, double *, double *);
4
5  int main()
6  {
7    double a,b,c, root1, root2;
8
9    printf("Enter a, b, c: ");
10   scanf("%lf %lf %lf", &a, &b, &c);
11   if (quadratic( a, b, c, &root1, &root2) )
12     printf("roots: %.10g %.10g\n", root1, root2);
```

```
13    else
14      printf("No real roots\n");
15    return 0;
16  }
```

quadt.c

汇编子程序如下：

quad.asm

```
1  ; 函数 quadratic
2  ; 求二次等式的根：
3  ;       a*x^2 + b*x + c = 0
4  ; C函数原型：
5  ;   int quadratic( double a, double b, double c,
6  ;                  double * root1, double *root2 )
7  ; 参数：
8  ;   a, b, c - 二次等式中各次方的系统(看上面)
9  ;   root1   - 指向存储第一个根的双精度变量的指针
10 ;   root2   - 指向存储第二个根的双精度变量的指针
11 ; 返回值：
12 ;   如果存在实根，则返回1,否则返回0
13
14 %define a               qword [ebp+8]
15 %define b               qword [ebp+16]
16 %define c               qword [ebp+24]
17 %define root1           dword [ebp+32]
18 %define root2           dword [ebp+36]
19 %define disc            qword [ebp-8]
20 %define one_over_2a     qword [ebp-16]
21
22 segment .data
23 MinusFour       dw      -4
24
25 segment .text
26         global  _quadratic
27 _quadratic:
28         push    ebp
29         mov     ebp, esp
30         sub     esp, 16         ; 分配两个双精度数大小的空间(disc & one_over_2a)
31         push    ebx             ; 必须保存原始的ebx值
32
33         fild    word [MinusFour]; stack -4
```

```
34          fld     a               ; stack: a, -4
35          fld     c               ; stack: c, a, -4
36          fmulp   st1             ; stack: a*c, -4
37          fmulp   st1             ; stack: -4*a*c
38          fld     b
39          fld     b               ; stack: b, b, -4*a*c
40          fmulp   st1             ; stack: b*b, -4*a*c
41          faddp   st1             ; stack: b*b - 4*a*c
42          ftst                    ; test with 0
43          fstsw   ax
44          sahf
45          jb      no_real_solutions ; 如果disc < 0，则没有实根
46          fsqrt                   ; stack: sqrt(b*b - 4*a*c)
47          fstp    disc            ; 储存然后再弹出堆栈
48          fld1                    ; stack: 1.0
49          fld     a               ; stack: a, 1.0
50          fscale                  ; stack: a * 2^(1.0) = 2*a, 1
51          fdivp   st1             ; stack: 1/(2*a)
52          fst     one_over_2a     ; stack: 1/(2*a)
53          fld     b               ; stack: b, 1/(2*a)
54          fld     disc            ; stack: disc, b, 1/(2*a)
55          fsubrp  st1             ; stack: disc - b, 1/(2*a)
56          fmulp   st1             ; stack: (-b + disc)/(2*a)
57          mov     ebx, root1
58          fstp    qword [ebx]     ; store in *root1
59          fld     b               ; stack: b
60          fld     disc            ; stack: disc, b
61          fchs                    ; stack: -disc, b
62          fsubrp  st1             ; stack: -disc - b
63          fmul    one_over_2a     ; stack: (-b - disc)/(2*a)
64          mov     ebx, root2
65          fstp    qword [ebx]     ; 储存到*root2中
66          mov     eax, 1          ; 返回值为1
67          jmp     short quit
68
69  no_real_solutions:
70          mov     eax, 0          ; 返回值为0
71
72  quit:
73          pop     ebx
74          mov     esp, ebp
75          pop     ebp
```

```
76          ret
```
quad.asm

6.3.5 从文件中读数组

在这个例子中，有一个从文件中读取双精度数的汇编程序。这是一个简短的C测试程序：

readt.c

```
1  /*这个程序用来测试32位的read_doubles()汇编程序。它从stdin中读取双精
   度数。(使用重定向从文件中读取。)*/
2  #include <stdio.h>
3  extern int read_doubles( FILE *, double *, int );
4  #define MAX 100
5
6  int main()
7  {
8    int i,n;
9    double a[MAX];
10
11   n = read_doubles(stdin, a, MAX);
12
13   for( i=0; i < n; i++ )
14     printf("%3d %g\n", i, a[i]);
15   return 0;
16 }
```

readt.c

这是汇编程序：

read.asm

```
1  segment .data
2  format  db      "%lf", 0        ; format for fscanf()
3
4  segment .text
5          global  _read_doubles
6          extern  _fscanf
7
8  %define SIZEOF_DOUBLE   8
9  %define FP              dword [ebp + 8]
10 %define ARRAYP          dword [ebp + 12]
11 %define ARRAY_SIZE      dword [ebp + 16]
```

```
12  %define TEMP_DOUBLE     [ebp - 8]
13
14  ;
15  ; 函数 _read_doubles
16  ; C函数原型:
17  ;   int read_doubles( FILE * fp, double * arrayp, int array_size );
18  ; 这个函数从一个文本文件中读取双精度数，并将它们储存到一个数组里，直到遇到
19  ; EOF或数组满了。
20  ; 参数:
21  ;   fp         - 指向需要读取的文件的指针(必须允许输入)
22  ;   arrayp     - 指向写入的双精度数组的指针
23  ;   array_size - 数组的元素个数
24  ; 返回值:
25  ;   储存到数组中的双精度数的个数(保存在EAX中)
26
27  _read_doubles:
28          push    ebp
29          mov     ebp,esp
30          sub     esp, SIZEOF_DOUBLE      ; 在堆栈中定义一个双精度数
31
32          push    esi                     ; 保存esi
33          mov     esi, ARRAYP             ; esi = ARRAYP
34          xor     edx, edx                ; edx = 数组的下标(最开始为0)
35
36  while_loop:
37          cmp     edx, ARRAY_SIZE         ; edx < ARRAY_SIZE?
38          jnl     short quit              ; 如果不是，退出循环
39  ;
40  ; 调用fscanf()函数读一个双精度数到TEMP_DOUBLE中
41  ; fscanf()会改变edx，所以需要保存它
42  ;
43          push    edx                     ; 保存edx
44          lea     eax, TEMP_DOUBLE
45          push    eax                     ; 将&TEMP_DOUBLE压入栈中
46          push    dword format            ; 将&format压入栈中
47          push    FP                      ; 将文件指针压入栈中
48          call    _fscanf
49          add     esp, 12
50          pop     edx                     ; 恢复edx的值
51          cmp     eax, 1                  ; fscanf函数是否返回1?
52          jne     short quit              ; 如果不是，则退出循环
53
```

```
54  ;
55  ; 复制TEMP_DOUBLE到ARRAYP[edx]中
56  ; (8个字节的双精度数是通过分成两个4字节的数来完成复制的)
57  ;
58          mov     eax, [ebp - 8]
59          mov     [esi + 8*edx], eax      ; 首先复制低4字节
60          mov     eax, [ebp - 4]
61          mov     [esi + 8*edx + 4], eax  ; 接着复制高4字节
62
63          inc     edx
64          jmp     while_loop
65
66  quit:
67          pop     esi                     ; 恢复esi
68
69          mov     eax, edx                ; 将返回值储存到eax中
70
71          mov     esp, ebp
72          pop     ebp
73          ret
```

read.asm

6.3.6 查找素数

最后一个例子又是查找素数的例子。这次的实现方法比以前的方法更有效。它将找到的素数储存到一个数组中，而且在查找新的素数时，只除以它已经找到的素数，而不是去除以每一个奇数。

另一个区别是它会计算出猜想的下一个素数的平方根来决定查找因子时应停在哪一个数。它修改了协处理器的控制字，所以当它把平方根当作一个整数来储存时，是通过直接截去来得到整数，而不是四舍五入。这是由控制字中的第10位和第11位来控制的。这些位称为RC(Rounding Control，四舍五入控制)位。如果这两位都是0(缺省值)，则协处理器转换成整数时，采用四舍五入的方法。如果是1,则通过直接截去来得到整数。注意：程序必须小心保存好控制字的原始值，当它返回时须恢复它的值。

这是C驱动程序：

fprime.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  /* 函数find_primes* 查找给定范围的素数* 参数:* a - 保存素数的数组* n -
   找到的素数的个数*/
4  extern void find_primes( int * a, unsigned n );
```

```
5
6  int main()
7  {
8    int status;
9    unsigned i;
10   unsigned max;
11   int * a;
12
13   printf("How many primes do you wish to find? ");
14   scanf("%u", &max);
15
16   a = calloc( sizeof(int), max);
17
18   if ( a ) {
19
20     find_primes(a,max);
21
22     /* print out the last 20 primes found */
23     for(i= ( max > 20 ) ? max - 20 : 0; i < max; i++ )
24       printf("%3d %d\n", i+1, a[i]);
25
26     free(a);
27     status = 0;
28   }
29   else {
30     fprintf(stderr, "Can not create array of %u ints\n", max);
31     status = 1;
32   }
33
34   return status;
35 }
```

fprime.c

下面是汇编程序：

prime2.asm

```
1  segment .text
2          global  _find_primes
3  ;
4  ; 函数 find_primes
5  ; 查找给定范围的素数
6  ; 参数:
```

```
7  ;   array  - 保存素数的数组
8  ;   n_find - 找到的素数的个数
9  ; C函数原型:
10 ;extern void find_primes( int * array, unsigned n_find )
11 ;
12 %define array         ebp + 8
13 %define n_find        ebp + 12
14 %define n             ebp - 4          ; 目前为止找到的素数的个数
15 %define isqrt         ebp - 8          ; 猜想的下一个素数开平方后得到的整数
16 %define orig_cntl_wd  ebp - 10         ; 原始控制字
17 %define new_cntl_wd   ebp - 12         ; 新的控制字
18
19 _find_primes:
20         enter   12,0                   ; 为局部变量分配空间
21
22         push    ebx                    ; 保存可能的寄存器变量
23         push    esi
24
25         fstcw   word [orig_cntl_wd]    ; 得到当前控制字
26         mov     ax, [orig_cntl_wd]
27         or      ax, 0C00h              ; 设置RC位为11(截去)
28         mov     [new_cntl_wd], ax
29         fldcw   word [new_cntl_wd]
30
31         mov     esi, [array]           ; esi指向数组
32         mov     dword [esi], 2         ; array[0] = 2
33         mov     dword [esi + 4], 3     ; array[1] = 3
34         mov     ebx, 5                 ; ebx = guess = 5
35         mov     dword [n], 2           ; n = 2
36 ;
37 ; 这个外部的循环用来查找一个新的素数，新的素数将被加到
38 ; 数组的末尾。跟以前的查找素数程序不同的是，这个函数
39 ; 并不是通过除以所有的奇数来决定它是不是素数。它仅仅
40 ; 除以已经找到的素数。(这也是为什么它们
41 ; 被储存到数组中的缘故。)
42 ;
43 while_limit:
44         mov     eax, [n]
45         cmp     eax, [n_find]          ; while ( n < n_find )
46         jnb     short quit_limit
47
48         mov     ecx, 1                 ; ecx用来表示数组的下标
```

```
49          push    ebx                     ; 将猜想的素数储存到堆栈中
50          fild    dword [esp]             ; 将猜想的素数导入到协处理器堆栈中
51          pop     ebx                     ; 将猜想的素数移除出栈
52          fsqrt                           ; 求sqrt(guess)
53          fistp   dword [isqrt]           ; isqrt = floor(sqrt(quess))
54  ;
55  ; 这个内部的循环用猜想的素数(ebx)除以已经找到的素数,
56  ; 直到找到一个猜想的素数的因子(也就意味着这个猜想的素数不是素数),
57  ; 或直到猜想的素数除以的找到的素数大于floor(sqrt(guess))
58  ;
59  while_factor:
60          mov     eax, dword [esi + 4*ecx]        ; eax = array[ecx]
61          cmp     eax, [isqrt]                    ; while ( isqrt < array[ecx]
62          jnbe    short quit_factor_prime
63          mov     eax, ebx
64          xor     edx, edx
65          div     dword [esi + 4*ecx]
66          or      edx, edx                        ; && guess % array[ecx] != 0 )
67          jz      short quit_factor_not_prime
68          inc     ecx                             ; 试下一个素数
69          jmp     short while_factor
70
71  ;
72  ; found a new prime !
73  ;
74  quit_factor_prime:
75          mov     eax, [n]
76          mov     dword [esi + 4*eax], ebx        ; 将猜想的素数加到数组的末尾
77          inc     eax
78          mov     [n], eax                        ; inc n
79
80  quit_factor_not_prime:
81          add     ebx, 2                          ; 试下一个奇数
82          jmp     short while_limit
83
84  quit_limit:
85
86          fldcw   word [orig_cntl_wd]             ; 恢复控制字
87          pop     esi                             ; 恢复寄存器变量
88          pop     ebx
89
90          leave
```

prime2.asm

```
91          ret
```

```
1  global _dmax
2
3  segment .text
4  ; 函数 _dmax
5  ; 返回两个参数中的较大的一个
6  ; C函数原型
7  ; double dmax( double d1, double d2 )
8  ; 参数:
9  ;   d1   - 第一个双精度数
10 ;   d2   - 第二个双精度数
11 ; 返回值:
12 ;   d1和d2中较大的一个 (储存在ST0中)
13 %define d1   ebp+8
14 %define d2   ebp+16
15 _dmax:
16         enter   0, 0
17
18         fld     qword [d2]
19         fld     qword [d1]          ; ST0 = d1, ST1 = d2
20         fcomip  st1                 ; ST0 = d2
21         jna     short d2_bigger
22         fcomp   st0                 ; 从堆栈中弹出d2
23         fld     qword [d1]          ; ST0 = d1
24         jmp     short exit
25 d2_bigger:                      ; 如果d2不是较大的那个数，不做任何事
26 exit:
27         leave
28         ret
```

图 6.7: FCOMIP指令的例子

```
segment .data
x            dq  2.75          ; 转换成双精度格式
five         dw  5

segment .text
      fild   dword [five]      ; ST0 = 5
      fld    qword [x]         ; ST0 = 2.75, ST1 = 5
      fscale                   ; ST0 = 2.75 * 32, ST1 = 5
```

图 6.8: FSCALE指令的例子

第7章

结构体与C++

7.1 结构体

7.1.1 简介

在C语言中的结构体用来将相关的数据集合到一个组合变量中。这项技术有几个优点：

1. 通过展示定义在结构体内的数据是紧密相联的来使代码变得清晰明了。

2. 它使传递数据给函数变得简单。代替单独地传递多个变量，它通过传递一个单元来传递多个变量。

3. 它增加了代码的*局部性*[1] 。

从汇编语言的观点看，结构体可以认为是拥有*不同*大小的元素的数组。而真正的数组的元素的大小和类型总是一样的。如果你知道数组的起始地址，每个元素的大小和需要的元素的下标，有这个特性就能计算出这个元素的地址。

结构体中的元素的大小并不一定要是一样的(而且通常情况下是不一样的)。因为这个原因，结构体中的每个元素必须清楚地指定而且需要给每个元素一个*标号*(或者名称)，而不是给一个数字下标。

在汇编语言中，结构体中的元素可以通过和访问数组中的元素一样的方法来访问。为了访问一个元素，你必须知道结构体的起始地址和这个元素相对于结构体的*相对偏移地址*。但是，和数组不一样的是：不可以通过元素的下标来计算该偏移地址，结构体的元素的地址需要通过编译器来赋值。

例如，考虑下面的结构体：

[1] 可以看任何操作系统书中关于虚拟内存管理中的讨论这个术语的部分。

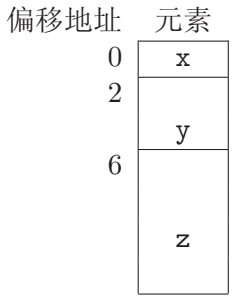

图 7.1: 结构体S

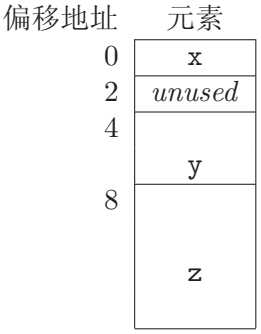

图 7.2: 结构体S

```
struct S {
  short int x;    /* 2个字节的整形 */
  int       y;    /* 4个字节的整形 */
  double    z;    /* 8个字节的浮点数 */
};
```

图 7.1展示了一个S结构体变量在电脑内存中是如何储存的。ANSI C标准规定结构体中的元素在内存中储存的顺序和在`struct`定义中的顺序是一样的。它同样规定第一个元素需恰好在结构体的起始地址中(*也就是说*偏移地址为0)。它同样在`stddef.h`头文件中定义了另一个有用的宏`offsetof()`。这个宏用来计算和返回结构体中任意元素的偏移地址。这个宏携带两个参数，第一个是结构体*类型*的变量名，第二个是需要得到偏移地址的元素名。因此，图 7.1中的，`offsetof(S, y)`的结果将是2。

7.1.2 内存地址对齐

回想一下一个地址如果除以了4以后，地址是处在双字界上的。

如果在*gcc*编译器中你使用`offsetof`宏来得到`y`的偏移地址，那么它们将找到并返回4，而不是2！为什么呢？ 因此*gcc*(和其它许多编译器)，在缺省情况下，变量是对齐在双字界上的。在32位保护模式下，如果数据是从双字界开始储存的，那么CPU能快速地读取内存。图 7.2展示了如果使用*gcc*，那么S结构体在内存中是如何储存的。编译器在结构体中插入了两

```
struct S {
  short int x;     /* 2个字节的整形 */
  int       y;     /* 4个字节的整形 */
  double    z;     /* 8个字节的浮点数 */
} __attribute__ ((packed));
```

图 7.3: 使用*gcc*的压缩结构体

个没有使用的字节，用来将y(和z)对齐在双字界上。这就表明了在C中定义的结构体，使用offsetof计算偏移来代替元素自己来计算自己的偏移为什么是一个好的想法。

当然，如果只是在汇编程序中使用结构体，程序员可以自己决定偏移地址。但是，如果你需要使用C和汇编的接口技术，那么在汇编代码和C代码中约定好如何计算结构体元素的偏移地址是非常重要的！一个麻烦的地方是不同的C编译器给出的元素的偏移地址是不同的。例如：就像我们已经知道的，*gcc*编译器创建结构体S如图 7.2；但是，Borland的编译器将创建结构体如图 7.1。C编译器提供了指定数据对齐的方法。但是，ANSI C标准并没有指定它们该如何完成，因此不同的编译器使用不同的方法来完成内存地址对齐。

*gcc*编译器有一个灵活但是复杂的方法来指定地址对齐。它允许你使用特殊的语法来指定任意类型的地址对齐。例如，下面一行：

```
typedef short int unaligned_int __attribute__ (( aligned (1)));
```

定义了一个名为unaligned_int的新类型，它采用的是字节界对齐方式。(是的，所以在__attribute__后面的括号都是需要的！)aligned的参数1可以用其它的2的乘方值来替代，用来表示采用的是其它对齐方式。(2为字边界，4表示双字界，等等。)如果结构体里的y元素改为unaligned_int类型，那么*gcc*给出的y的偏移地址为2.但是，z依然处在偏移地址8的位置，因为双精度类型的缺省对齐方式为双字对齐。要想z的偏移地址为6，那么还要改变它的类型定义。

*gcc*编译器同样允许你*压缩*一个结构体。它告诉编译器使用尽可能小的空间来储存这个结构体。图 7.3展示了S如何以这种方法来定义。这种形式下的S将使用可能的最少的字节数，14个字节。

Microsoft和Borland的编译器都支持使用#pragma指示符的方法来指定对齐方式。

```
#pragma pack(1)
```

上面的指示符告诉编译器采用字节界的对齐方式来压缩结构体中的元素。(*也就是说*，没有额外的填充空间)。其中的1可以用2，4，8或16代替，分别用来指定对齐方式为字边界，双字界，四字界和节边界。这个指示符在被另一个指示符置为无效之前保持有效。这就可能会导致一些问题，因为这些指示符通常使用在头文件中。如果这个头文件在包含结构体的其它

```
#pragma pack(push)  /* 保存对齐方式的状态值 */
#pragma pack(1)     /* 设置为字节界    */

struct S {
  short int x;     /* 2个字节的整形 */
  int       y;     /* 4个字节的整形 */
  double    z;     /* 8个字节的浮点数   */
};

#pragma pack(pop)   /* 恢复原始的对齐方式 */
```

图 7.4: 使用Microsoft或Borland的压缩结构体

```
struct S {
  unsigned f1 : 3;   /* 3个位域  */
  unsigned f2 : 10;  /* 10个位域 */
  unsigned f3 : 11;  /* 11个位域 */
  unsigned f4 : 8;   /* 8个位域  */
};
```

图 7.5: 关于位域的例子

头文件之前被包含到程序中，那么这些结构体的放置方式将和它们缺省的放置方式不同。这将导致非常严重的查找错误。程序中的不同模块将会将结构体元素放置在不同的地方。

有一个方法来避免这个问题。Microsoft和Borland都支持这个方法：保存当前对齐方式状态值和随后恢复它。图 7.4展示了如何使用这种方法。

7.1.3 位域s

位域允许你指定结构体中的成员的大小为只使用指定的比特位数。比特位数的大小并不一定要是8的倍数。一个位域成员的定义和unsigned int或int的成员定义是一样，只是在定义的后面增加了冒号和位数的大小。图 7.5展示了一个例子。它定义了一个32位的变量，它由下面的几部分组成：

8个比特	11个比特	10个比特	3个比特
f4	f3	f2	f1

第一个位域被指定到此双字的最低有效位处。[2]

但是，如果你看了这些比特位实际上在内存中是如何储存的，你就会发现格式并不是如此简单。难点发生在当位域跨越字节界时。因为在little

[2]实际上，ANSI/ISO C标准在实际上如何放置比特位方面给了编译器少许灵活性。但是，普遍的C编译器(*gcc*，*Microsoft*和*Borland*)都将像这样放置比特位域。

字节\ 位	7	6	5	4	3	2	1	0
0	操作码(08h)							
1	逻辑单元#			LBA的msb				
2	逻辑块地址的中间部分							
3	逻辑块地址的lsb							
4	传递的长度							
5	控制字							

图 7.6: SCSI读命令格式

endian处理器上的字节将以相反的顺序储存到内存中。例如，S结构体在内存中将如下所示：

5个比特	3个比特	3个比特	5个比特	8个比特	8个比特
f2l	f1	f3l	f2m	f3m	f4

*f2l*变量表示*f2*位域的末尾五个比特位(*也就是*，五个最低有效位)。*f2m*变量表示*f2*的五个最高有效位。双垂直线的地方表示字节界。如果你将所有的字节反向，*f2*和*f3*位域将重新结合到正确的位置。

物理内存的放置方式通常并不是很重要，除非有数据需要传送到程序中或从程序中传出(实际上这和位域是非常相同的)。硬件设备的接口使用奇数的比特位是非常普遍的，此时使用位域来描述是非常有用的。

SCSI[3]就是一个例子。SCSI设备的直接读命令被指定为传送一个六个字节的信息到设备，格式指定为图 7.6中的格式。使用位域来描述这个的难点是*逻辑区块地址(logical block address)*，它在此命令中跨越了三个不同的字节。从图 7.6中，你可以看到数据是以big endian的格式储存的。图 7.7展示了一个试图在所有编译器中工作的定义。前两行定义了一个宏，如何代码是由Microsoft或Borland编译器来编译时，则它就为真。可能比较混乱的部分是11行到14行。首先，你可能会想为什么lba_mid和 lba_lsb 位域要分开被定义，而不是定义成一个16位的域？原因是数据是以big endian顺序储存的。而编译器将把一个16位的域以little endian顺序来储存。其次，lba_msb和 logical_unit 位域看起来似乎方向反了；但是，情况并不是这样。它们必须得以这样的顺序来摆放。图 7.8展示了作为一个48位的实体，它的位域图是怎样的。(字节界同样是以双垂直线来表示。)当它在内存中是以little endian的格式来储存，那么比特位将以要求的格式来排列。(图 7.6).

考虑得复杂一点，我们知道SCSI_read_cmd的定义在Microsoft C编译器中不能完全正确工作。如果 sizeof (SCSI_read_cmd)表达式被赋值了，Microsoft C将返回8,而不是6！这是因为Microsoft编译器使用位域的类型来决定如何绘制比特图。因为所有的位域都被定义为unsigned类型，所以编译器在结构体的末尾加了两个字节使得它成为一个双字类型的整数。这个问题可以

[3]Small Computer Systems Interface，小型计算机系统接口，一个硬盘，等等的工业标准

```
#define MS_OR_BORLAND (defined(__BORLANDC__) \
                       || defined(_MSC_VER))

#if MS_OR_BORLAND
#  pragma pack(push)
#  pragma pack(1)
#endif

struct SCSI_read_cmd {
  unsigned opcode : 8;
  unsigned lba_msb : 5;
  unsigned logical_unit : 3;
  unsigned lba_mid : 8;     /* 中间的比特位 */
  unsigned lba_lsb : 8;
  unsigned transfer_length : 8;
  unsigned control : 8;
}
#if defined(__GNUC__)
    __attribute__ ((packed))
#endif
;

#if MS_OR_BORLAND
#  pragma pack(pop)
#endif
```

图 7.7: SCSI读命令格式的结构

通过用unsignedshort替代所有的位域定义类型来修正。现在，Microsoft编译器不需要增加任何的填充字节，因为六个字节是两个字节字类型的整数。[4]有了这个改变，其它的编译器也能正确工作。图 7.9展示了另外一种定义，能在所有的三种编译器上工作。它通过使用unsignedchar避免了除2位的域以外的所有位域的问题。

如果发现前面的讨论非常混乱的读者，请不要气馁。它本来就是混乱的！通过经常完全地避免使用位域而采用位操作来手动地检查和修改比特位，作者发现能避免一些混乱。

7.1.4 在汇编语言中使用结构体

就像上面讨论，在汇编语言中访问结构体就类似于访问数组。作为一

[4]混乱的不同类型的位域将导致非常混乱的行为！读者需要自己去实验。

8个比特	8个比特	8个比特	8个比特	3个比特	5个比特	8个比特
控制字	传输的长度	lba_lsb	lba_mid	logical_unit	lba_msb	opcode

图 7.8: SCSI_read_cmd的位域图

```
struct SCSI_read_cmd {
  unsigned char opcode;
  unsigned char lba_msb : 5;
  unsigned char  logical_unit  : 3;
  unsigned char lba_mid;     /* 中间的比特位 */
  unsigned char  lba_lsb ;
  unsigned char  transfer_length ;
  unsigned char  control ;
}
#if defined(__GNUC__)
    __attribute__ ((packed))
#endif
;
```

图 7.9: 另一种SCSI读命令格式的结构

个简单的例子，考虑一下你如何写这样一个汇编程序：将0写入到S结构体的y中。假定这个程序的原型是这样的：

```
void zero_y( S * s_p );
```

汇编程序如下：

```
%define      y_offset  4
_zero_y:
      enter  0,0
      mov    eax, [ebp + 8]       ; 从堆栈中得到s_p(结构体的指针)
      mov    dword [eax + y_offset], 0
      leave
      ret
```

C语言允许你把一个结构体当作数值传递给函数；但是，通常这都是一个坏主意。当以数值来传递时，在结构体中的所有数据都必须复制到堆栈中，然后在程序中再拿出来使用。用一个结构体指针来替代能有更高的效率。

C语言同样允许一个结构体类型作为一个函数的返回值。很明显，一个结构体不能通过储存到EAX寄存器中来返回。不同的编译器处理这种情况的方法也不同。一个编译器普遍使用的解决方法是在内部重写函数，让它携

```
#include <stdio.h>

void f( int x )
{
   printf("%d\n", x);
}

void f( double x )
{
   printf("%g\n", x);
}
```

图 7.10: 两个名为f()的函数

带一个结构体指针参数。这个指针用来将返回值放入到结构体中，这个结构体是在调用的程序外面定义的。

大多数汇编器(包括NASM)都有在你的汇编代码中定义结构体的内置支持。查阅你的资料来得到更详细的信息。

7.2 汇编语言和C++

C++编程语言是C语言的一种扩展形式。许多C语言和汇编语言接口的基本规则同样适用于C++。但是，有一些规则需要修正。同样，拥有一些汇编语言的知识，你能很容易理解C++中的一些扩展部分。这一节假定你已经有一定的C++基础知识。

7.2.1 重载函数和名字改编

C++允许不同的函数(和类成员函数)使用同样的函数名来定义。当不止一个函数共享同一个函数名时，这些函数就称为*重载函数*。在C语言中，如果定义的两个函数使用的函数名是一样，那么连接器将产生一个错误，因为在它连接的目标文件中，一个符号它将找到两个定义。例如，考虑图 7.10中的代码。等价的汇编代码将定义两个名为_f的标号，而这明显是错误的。

C++使用和C一样的连接过程，但是通过执行*名字改编*或修改用来标记函数的符号来避免这个错误。在某种程序上，C也早已经使用了名字改编。当创建函数的标号时，它在C函数名上增加了一条下划线。但是，C语言将以同样的方法来改编图 7.10中的两个函数名，那么将会产生一个错误。C++使用一个更高级的改编过程：为这些函数产生两个不同的标号。例如：图 7.10中的第一个函数将由DJGPP指定为标号_f__Fi，而第二个函数，指定为_f__Fd。这样就避免了任何的连接错误。

不幸的是，关于在C++中如何改编名字并没有一个标准，而且不同的编译器改编的名字也不一样。例如，Borland C++将使用标号`@f$qi`和`@f$qd`来表示图 7.10中的两个函数。但是，规则并不是完全任意的。改编后的名字编码成函数的签名。一个函数的签名是通过它携带的参数的顺序和类型来定义的。注意，，携带了一个`int`参数的函数在它的改编名字的末尾将有一个*i*(对于DJGPP 和Borland都是一样)，而携带了一个`double`参数的函数在它的改编名字的末尾将有一个*d*。如果有一个名为`f`的函数，它的原型如下：

```
void f( int x, int y, double z);
```

DJGPP将会把它的名字改编成`_f__Fiid`而Borland将会把它改编成`@f$qiid`。

函数的返回类型并*不*是函数签名的一部分，因此它也不会编码到它的改编名字中。这个事实解释了在C++中的一个重载规则。只有签名唯一的函数才能重载。就如你能看到的，如果在C++中定义了两个名字和签名都一样的函数，那么它们将得到同样的签名，而这将产生一个连接错误。缺省情况下，所有的C++函数都会进行名字改编，甚至是那些没有重载的函数。它编译一个文件时，编译器并没有方法知道一个特定的函数重载与否，所以它将所有的名字改编。事实上，和函数签名的方法一样，编译器同样通过编码变量的类型来改编全局变量的变量名。因此，如果你在一个文件中定义了一个全局变量为某一类型然后试图在另一个文件中用一个错误的类型来使用它，那么将产生一个连接错误。C++这个特性被称为*类型安全连接*。它同样暴露出另一种类型的错误：原型不一致。当在一个模块中函数的定义和在另一个模块使用的函数原型不一致时，就发生这种错误。在C中，这是一个非常难调试出来的问题。C并不能捕捉到这种错误。程序将被编译和连接，但是将会有未定义的操作发生，就像调用的代码会将和函数期望不一样的类型压入栈中一样。在C++中，它将产生一个连接错误。

当C++编译器语法分析一个函数调用时，它通过查看传递给函数的参数的类型来寻找匹配的函数[5]。如果它找到了一个匹配的函数，那么通过使用编译器的名字改编规则，它将创建一个`CALL`来调用正确的函数。

因为不同的编译器使用不同的名字改编规则，所以不同编译器编译的C++代码可能不可以连接到一起。当考虑使用一个预编译的C++库时，这个事实是非常重要的！如果有人想写出一个能在C++代码中使用的汇编程序，那么他必须知道要使用的C++编译器使用的名字改编规则(或使用下面将解释的技术)。

机敏的学生可能会询问在图 7.10中的代码到底能不能如预期般工作。因为C++改编了所有函数的函数名，那么`printf`将被改编，而编译器将不会产生一个到标号`_printf`处的`CALL`调用。这是一个非常正确的担忧！如果`printf`的原型被简单地放置在文件的开始部分，那么这就将发生。原型为：

[5]这个匹配并不一定要是精确匹配，编译器将通过强制转型参数来考虑匹配。这个过程的规则超出了本书的范围。查阅一本C++的书来得到更详细的信息。

```
int printf( const char *, ...);
```

DJGPP将会把它改编为_printf__FPCce。(F表示*function，函数*，P表示*pointer，指针*，C表示*const，常量*，c表示*char*而e表示省略号。)那么它将不会调用正规C库中的printf函数！当然，必须有一种方法让C++代码用来调用C代码。这是非常重要的，因为到处都有*许多*非常有用的旧的C代码。除了允许你调用遗留的C代码外，C++同样允许你调用使用了正规的C改编约定的汇编代码。

C++扩展了extern关键字，允许它用来指定它修饰的函数或全局变量使用的是正规C约定。在C++术语中，称这些函数或全局变量使用了*C链接*。例如，为了声明printf为C链接，需使用下面的原型：

```
extern "C" int printf( const char *, ... );
```

这就告诉编译器不要在这个函数上使用C++的名字改编规则，而使用C规则来替代。但是，如果这样做了，那么printf将不可以重载。这就提供了一个简易的方法用在C++和汇编程序接口上：使用C链接定义一个函数，然后再使用C调用约定。

为了方便，C++同样允许定义函数或全局变量块的C链接。通常函数或全局变量块用卷曲花括号表示。

```
extern "C" {
  /* C链接的全局变量和函数原型 */
}
```

如果你检查了当今的C/C++编译器中的ANSI C头文件，你会发现在每个头文件上面都有下面这个东西：

```
#ifdef __cplusplus
extern "C" {
#endif
```

而且在底部有一个包含闭卷曲花括号的同样的结构。C++编译器定义了宏__cplusplus(有*两*条领头的下划线)。上面的代码片断如果用C++来编译，那么整个头文件就被一个extern "C"块围起来了，但是如果使用C来编译，就不会执行任何操作(因为对于extern "C"，C编译器将产生一个语法错误)。程序员可以使用同样的技术用来在汇编程序中创建一个能被C或C++使用的头文件。

7.2.2 引用

*引用*是C++的另一个新特性。它允许你传递参数给函数，而不需要明确使用指针。例如，考虑图 7.11中的代码。事实上，引用参数是非常简单，实际上它们就是指针。只是编译器对程序员隐藏它而已(正如Pascal编译器把var参数当作指针来执行)。当编译器产生此函数调用的第7行代码的汇编

```
1  void f( int & x )      // &表示是一个引用参数 { x++; }
2
3  int main()
4  {
5    int y = 5;
6    f(y);                // 传递了引用y，注意这里没有&!
7    printf ("%d\n", y);  // 显示6!
8    return 0;
9  }
```

图 7.11: 引用的例子

语句时，它将y的*地址*传递给函数。如果有人是用汇编语言书写的f函数，那么他们操作的情况，就好像原型如下似的：[6]：

```
void f( int * xp);
```

引用是非常方便的，特别是对于运算符重载来说是非常有用的。运算符重载又是C++的另一个特性，它允许你在对结构体或类类型进行操作时赋予普通运算符另一种功能。例如，一个普遍的使用是赋予加号(+)运算符能将字符串对象连接起来的功能。因此，如果a和b是字符串，那么a + b将得到a和b连接后的字符串。实际上，C++可以调用一个函数来做这件事(事实上，上面的表达式可以用函数的表示法来重写为：operator +(a,b))。为了提高效率，有人可能会希望传递字符串的地址来代替传递他们的值。若没有引用，那么将需要这样做：operator +(&a,&b)，但是若要求你以运算符的语法来书写应为：&a + &b。这是非常笨拙而且混乱的。但是，通过使用引用，你可以像这样书写：a + b，这样就看起来非常自然。

7.2.3 内联函数

到目前为止，*内联函数*又是C++的另一个特性[7]。内联函数照道理应该可以取代容易犯错误的，携带参数的，基于预处理程序的宏。回想一下在C中，书写一个求数的平方的宏可以是这样的：

```
#define SQR(x) ((x)*(x))
```

因为预处理程序不能理解C而采用简单的替换操作，在大多数情况下，圆括号里要求是能正确计算出来的值。但是，即使是这个版本也不能给出SQR(x++)的正确答案。

宏之所以被使用是因为它除去了进行一个简单函数的函数调用的额外时间开支。就像子程序那一章描述的，执行一个函数调用包括好几步。

[6] 当然，他们可能想使用C链接来声明函数，用来避免名字改编，就像小节 7.2.1中讨论的

[7] C编译器通常支持这种特性，把它当作ANSI C的扩展。

```
1  inline int inline_f ( int x )
2  { return x*x; }
3
4  int f( int x )
5  { return x*x; }
6
7  int main()
8  {
9    int y, x = 5;
10   y = f(x);
11   y = inline_f (x);
12   return 0;
13 }
```

图 7.12: 内联函数的例子

对于一个非常简单的函数来说，用来进行函数调用的时间可能比实际上执行函数里的操作的时间还要多！内联函数是一个更为友好的用来书写代码的方法，让代码看起来象一个标准的函数，但是它并不是CALL指令能调用的普通代码块。出现内联函数的调用表达式的地方将被执行函数的代码替换。C++允许通过在函数定义前加上inline关键字来使函数成为内联函数。如果，考虑在图 7.12中声明的函数。第10行对f的调用将执行一个标准的函数调用(在汇编语言中，假定x的地址为ebp-8而y地址为ebp-4)：

```
1  push   dword [ebp-8]
2  call   _f
3  pop    ecx
4  mov    [ebp-4], eax
```

但是，第11行对inline_f的调用将得到如下结果：

```
1  mov    eax, [ebp-8]
2  imul   eax, eax
3  mov    [ebp-4], eax
```

这种情况下，使用内联函数有两个优点。首先，内联函数更快。没有参数需要压入栈中，也不需要创建和毁坏堆栈帧，也不需要进行分支。其次，内联函数调用使用的代码是非常少！后面一点对这个例子来说是正确的，但是并不是在所有情况下都是正确的。

内联函数的主要优点是内联代码不需要连接，所以对于使用内联函数的*所有*源文件来说，内联函数的代码都必须有效。前面的汇编代码的例子展

```
1  class Simple {
2  public:
3    Simple();                  // 缺省的构造函数
4    ~Simple();                 // 析构函数
5    int get_data() const;      // 函数成员
6    void set_data( int );
7  private:
8    int data;                  // 数据成员
9  };
10
11 Simple::Simple()
12 { data = 0; }
13
14 Simple::~Simple()
15 { /* 空程序体 */ }
16
17 int Simple::get_data() const
18 { 返回值; }
19
20 void Simple::set_data( int x )
21 { data = x; }
```

图 7.13: 一个简单的C++类

示了这一点。对于非内联函数的调用，只要求知道参数，返回值类型，调用约定和函数的函数名。所有的这些信息都可以从函数的原型中得到。但是，使用内联函数调用，就必须知道这个函数的所有代码。这就意味着如果改变了一个内联函数中的任何部分，那么*所有*使用了这个函数的源文件必须重新编译。回想一下对于非内联函数，如果函数原型没有改变，通常使用这个函数的源文件就不需要重新编译。由于所有的这些原因，内联函数的代码通常放置在头文件中。这样做违反了在C语言中标准的稳定和快速准则：执行的代码语句*决不能*放置在头文件中。

7.2.4 类

C++中的类描述了一个*对象类型*。一个对象包括数据成员(data member)和函数成员(function member)[8]。换句话说就是，它是由跟它相关联的数据和函数组成的一个`struct`结构体。考虑在图 7.13中定义的那个简单的类。一个`Simple`类型的变量非常类似于包含一个`int`成员的标准C`struct`结构体。这些函数并不会储存到指定结构体的内存中。但是，成员函数和其

事实上，C++使用`this`*关键字从成员函数内部来访问指向此函数能起作用的对象的指针。*

[8]在C++中，通常称之为*成员函数(member function)*或者更为普遍地称之为*方法(method)*。

```
void set_data( Simple * object, int x )
{
  object->data = x;
}
```

图 7.14: Simple::set_data()的C版本

```
1  _set_data__6Simplei:            ; 改编后的名字
2        push   ebp
3        mov    ebp, esp
4
5        mov    eax, [ebp + 8]    ; eax = 指向对象的指针(this)
6        mov    edx, [ebp + 12]   ; edx = 整形参数
7        mov    [eax], edx        ; data在偏移地址0处
8
9        leave
10       ret
```

图 7.15: 编译Simple::set_data(int)的输出

它函数是不一样的。它们传递了一个*隐藏*的参数。这个参数是一个指向成员函数能起作用的对象的指针。

例如，考虑图 7.13中的`Simple`类的成员函数`set_data`。如果用C语言来书写此函数，这个函数将像这样：明确传递一个指向成员函数能起作用的对象的指针，如图 7.14所示。使用*DJGPP*编译器加上`-S`选项(*gcc*和Borland编译器也是一样)来告诉编译器输出一个包含此代码产生的等价的汇编语言代码的源文件。对于*DJGPP*和*gcc*编译器，此汇编源文件是以`.s`扩展名结尾的，但是不幸的是使用的语法是AT&T汇编语言语法，这种语法和NASM和MASM语法区别非常大[9]。(Borland和MS编译器产生一个以`.asm`扩展名结尾的源文件，使用的是MASM语法。) 图 7.15展示了将*DJGPP*的输出转换成NASM语法后的代码，增加了阐明语句目的的注释。在第一行中，注意成员函数`set_data`的函数名被指定为一个改编后的标号，此标号是通过编码成员函数名，类名和参数后得到的。类名被编码进去的是因为其它类中可能也有名为`set_data`的成员函数，而这两个成员函数*必须*使用不同的标号。参数之所以被编码进去是为了类能通过携带其它参数来重载成员函数`set_data`，正如标准的C++函数。但是，和以前一样，不同的编译器在改编标号时编码信息的方式也不同。

[9] *gcc*编译系统包含了一个属于自己的称为*gas*的汇编器。*gas*汇编器使用AT&T语法，因此编译器以*gas*的格式来输出代码。网页中有好几页用来讨论INTEL和AT&T语法的区别。同时有一个名为`a2i`的免费程序(http://www.multimania.com/placr/a2i.html)，此程序将AT&T格式转换成了NASM格式。

下面的第2和第3行，出现了熟悉的函数的开始部分。在第5行，把堆栈中的第一个参数储存到EAX中了。这并不是参数x！替代它的是那个隐藏的参数[10]，它是指向此函数能起作用的对象的指针。第6行将参数x储存到EDX中了，而第7行又将EDX储存到了EAX指向的双字中。它是Simple对象中的data成员，也是这个类中的唯一的数据，它储存在Simple结构体中偏移地址为0的地方。

样例

这一节使用了这章中的思想创建了一个C++类：用来描述任意大小的无符号整形。因为要描述任意大小的整形，所以它需要储存到一个无符号整形的数组(双字的)中。可以使用动态分配来实现任意大小的整形。双字是以相反的方向储存的[11] (*也就是说*，双字的最低有效位的下标为0)。图 7.16展示了Big_int类的定义[12]。Big_int的大小是通过测量unsigned数组的大小得到的，用来储存它的数据。此类中的size_数据成员的偏移地址为0，而number_成员的偏移为4。

为了简单化这些例子，只有拥有大小相同的数组的对象实例才可以相互进行加减操作。

这个类有三个构造函数(constructor)：第一个构造函数(第9行)使用了一个正常的无符号整形来初始化类实例；第二个构造函数(第18行)使用了一个包含一个十六进制值的字符串来初始化类实例。第三个构造函数(第21行)是*拷贝构造函数(copy constructor)*。

因为这里使用的是汇编语言，所以讨论的焦点在于加法和减法运算符如何工作。图 7.17展示了与这些运算符相关的部分头文件。它们展示了如何创建运算符来调用汇编程序。因为不同的编译器使用完全不同的名字改编规则来改编运算符函数，所以创建了内联的运算符函数来调用C链接汇编程序。这就使得在不同编译器间的移植变得相对容易些，而且和直接调用速度一样快。这项技术同样免去了从汇编中抛出异常的必要！

为什么在这里使用的全部是汇编语言呢？回想一下，在执行多倍精度运算时，进位必须从一个双字移去与下一个有效的双字进行加法操作。C++(和C)并不允许程序员访问CPU的进位标志位。只有通过让C++独立地重新计算出进位标志位后有条件地与下一个双字进行加法操作，才能执行这个加法操作。使用汇编语言来书写代码会更有效，因为它可以访问进位标志位，可以使用ADC指令来自动将进位标志位加上，这样做更有道理。

为了简化，只有add_big_ints的汇编程序将在这讨论。下面是这个程序的代码(来自big_math.asm)：

big_math.asm

```
1 segment .text
2         global  add_big_ints, sub_big_ints
```

[10] 像平常一样，*没有东西*能隐藏在汇编代码中！

[11] 为什么呢？因为加法运算将从数组的开始处开始逐渐向前进行操作。

[12] 查阅样例源代码来得到这个例子的全部的代码。本文中将只引用部分代码。

```
class Big_int {
public:
   /*
   * Parameters:
   *    size              – 表示成正常无符号整数的整形大小
   *
   *    initial_value     – 将Big_int的值初始化为一个正常的无符号整形
   */
   explicit Big_int( size_t     size,
                     unsigned initial_value = 0);
   /*
   * Parameters:
   *    size              – 表示成正常无符号整数的整形大小
   *
   *    initial_value     – 将Big_int的值初始化为一个包含一个以十六进制表示的值的字符串
   *
   */
   Big_int( size_t          size,
            const char *  initial_value );

   Big_int( const Big_int & big_int_to_copy );
   ~Big_int ();

   // 返回Big_int的大小(以无符号整形的形式)
   size_t  size() const;

   const Big_int & operator = ( const Big_int & big_int_to_copy );
   friend Big_int operator + ( const Big_int & op1,
                               const Big_int & op2 );
   friend Big_int operator - ( const Big_int & op1,
                               const Big_int & op2);
   friend bool operator == ( const Big_int & op1,
                             const Big_int & op2 );
   friend bool operator < ( const Big_int & op1,
                            const Big_int & op2);
   friend ostream & operator << ( ostream &      os,
                                  const Big_int & op );
private:
   size_t        size_ ;     // 无符号数组的大小
   unsigned * number_;  // 指向拥有数值的无符号数组的指针
};
```

图 7.16: Big_int类的定义

```
1  // 汇编程序的原型
2  extern "C" {
3    int  add_big_ints ( Big_int  &        res ,
4                        const Big_int  & op1,
5                        const Big_int  & op2);
6    int  sub_big_ints ( Big_int  &        res ,
7                        const Big_int  & op1,
8                        const Big_int  & op2);
9  }
10
11 inline  Big_int  operator + ( const Big_int  & op1, const Big_int  & op2)
12 {
13   Big_int  result (op1. size ());
14   int  res  =  add_big_ints ( result ,  op1,  op2);
15   if  (res  == 1)
16     throw Big_int :: Overflow ();
17   if  (res  == 2)
18     throw Big_int :: Size_mismatch ();
19   return  result ;
20 }
21
22 inline  Big_int  operator - ( const Big_int  & op1, const Big_int  & op2)
23 {
24   Big_int  result (op1. size ());
25   int  res  =  sub_big_ints ( result ,  op1,  op2);
26   if  (res  == 1)
27     throw Big_int :: Overflow ();
28   if  (res  == 2)
29     throw Big_int :: Size_mismatch ();
30   return  result ;
31 }
```

图 7.17: Big_int类的算术代码

```
3  %define size_offset 0
4  %define number_offset 4
5
6  %define EXIT_OK 0
7  %define EXIT_OVERFLOW 1
8  %define EXIT_SIZE_MISMATCH 2
9
10 ; 加法和减法程序的参数
11 %define res ebp+8
12 %define op1 ebp+12
13 %define op2 ebp+16
14
15 add_big_ints:
16         push    ebp
17         mov     ebp, esp
18         push    ebx
19         push    esi
20         push    edi
21         ;
22         ; 首先设置：esi指向op1
23         ;           edi指向op2
24         ;           ebx指向res
25         mov     esi, [op1]
26         mov     edi, [op2]
27         mov     ebx, [res]
28         ;
29         ; 要保证所有3个Big_int类型数有同样的大小
30         ;
31         mov     eax, [esi + size_offset]
32         cmp     eax, [edi + size_offset]
33         jne     sizes_not_equal                 ; op1.size_ != op2.size_
34         cmp     eax, [ebx + size_offset]
35         jne     sizes_not_equal                 ; op1.size_ != res.size_
36
37         mov     ecx, eax                        ; ecx = Big_int的大小
38         ;
39         ; 现在，让寄存器指向它们各自的数组
40         ;       esi = op1.number_
41         ;       edi = op2.number_
42         ;       ebx = res.number_
43         ;
44         mov     ebx, [ebx + number_offset]
```

```
45          mov     esi, [esi + number_offset]
46          mov     edi, [edi + number_offset]
47
48          clc                                     ; 清进位标志位
49          xor     edx, edx                        ; edx = 0
50          ;
51          ; 加法循环
52  add_loop:
53          mov     eax, [edi+4*edx]
54          adc     eax, [esi+4*edx]
55          mov     [ebx + 4*edx], eax
56          inc     edx                             ; 不要改变进位标志位
57          loop    add_loop
58
59          jc      overflow
60  ok_done:
61          xor     eax, eax                        ; 返回值 = EXIT_OK
62          jmp     done
63  overflow:
64          mov     eax, EXIT_OVERFLOW
65          jmp     done
66  sizes_not_equal:
67          mov     eax, EXIT_SIZE_MISMATCH
68  done:
69          pop     edi
70          pop     esi
71          pop     ebx
72          leave
73          ret
```

big_math.asm

希望，到此刻为止读者能明白大部分这里的代码。第25行到27行将Big_int对象传递的指针储存到寄存器中。记住引用的仅仅是指针。第31行到35行检查保证三个对象数组的大小是一样的。(注意，size_的偏移被加到指针中了，为了访问数据成员。)第44行和第46行调整寄存器，让它们指向被各自对象使用的数组，用来替代使用对象本身。(同样，number_的偏移被加到对象指针中了。)

在第52行到57行的循环中，将储存在数组里的整形一起相加，首先加的是最低有效的双字，然后是下一最低有效的双字，等等。多倍精度运算必须以这样的顺序来完成(看小节 2.1.5)。第59行用来检查溢出，一旦溢出，进位标志位将由最后进行加法运算的最高有效位置位。因为数组里的双字是以little endian顺序储存的，所以循环从数组的开始处开始，依次向前直到结束。

```
#include "big_int.hpp"
#include <iostream>
using namespace std;

int main()
{
  try {
    Big_int b(5,"8000000000000a00b");
    Big_int a(5,"80000000000010230");
    Big_int c = a + b;
    cout << a << " + " << b << " = " << c << endl;
    for( int i=0; i < 2; i++ ) {
      c = c + a;
      cout << "c = " << c << endl;
    }
    cout << "c-1 = " << c - Big_int(5,1) << endl;
    Big_int d(5, "12345678");
    cout << "d = " << d << endl;
    cout << "c == d " << (c == d) << endl;
    cout << "c > d " << (c > d) << endl;
  }
  catch( const char * str ) {
    cerr << "Caught: " << str << endl;
  }
  catch( Big_int::Overflow ) {
    cerr << "Overflow" << endl;
  }
  catch( Big_int::Size_mismatch ) {
    cerr << "Size mismatch" << endl;
  }
  return 0;
}
```

图 7.18: `Big_int`的简单应用

图 7.18展示了Big_int的简单应用的简短的例子。注意，Big_int常量必须明确声明，如第16行。这有两个原因。首先，没有转换构造函数来将一个无符号整形转换成Big_int类型。其次，只有相同大小的Big_int数才能用来进行相加操作。这里进行类型转换是有问题的，因为要知道需转换的大小是非常困难的。此类的一个更高级的实现将允许任意大小的数之间的相加。作者不打算因为要实现任意大小的数的相加而把这个例子弄得过度复杂。(但是，鼓励读者来实现它。)

7.2.5 继承和多态

*继承(Inheritance)*允许一个类继承另一个类的数据和成员函数。例如，考虑图 7.19中的代码。它展示了两个类，A和B，其中类B是通过继承类A得到的。程序的输出如下：

```
Size of a: 4 Offset of ad: 0
Size of b: 8 Offset of ad: 0 Offset of bd: 4
A::m()
A::m()
```

注意，两个类的数据成员ad(B通过继承A得到的)在相同的偏移处。这是非常重要的，因为f函数将传递一个指针到一个A对象或任意一个由A派生(*也就是*，通过继承得到)的对象类型中。图 7.20展示了此函数的(编辑过的)汇编代码(*gcc*得到的)。

注意在输出中，a和b对象调用的都是A的成员函数m。从汇编程序中，我们可以看到对A::m()的调用被硬编码到函数中了。对于真正的面向对象编程，成员函数的调用取决于传递给函数的对象类型是什么。这就是所谓的*多态*。缺省情况下，C++关掉了这个特性。你可以使用*virtual* 关键字来激活它。图 7.21展示了如何修改这两个类。其它代码不需要修改。多态可以用许多方法来实现。不幸的是，当在以这种方法书写的时候，*gcc*的实现方法正处在改变中，而且与它最初的实现方法相比，明显变得更复杂了。为了简单化讨论的目的，作者只涉及基于Microsoft和Borland编译器Windows使用的多态的实现方法。这种实现方法很多年没有改变了，而且可能在未来几年也不会改变。

有了这些改变，程序的输出如下：

```
Size of a: 8 Offset of ad: 4
Size of b: 12 Offset of ad: 4 Offset of bd: 8
A::m()
B::m()
```

现在，对f的第二次调用调用了B::m()的成员函数，因为它传递了对象B。但是，这并不是唯一的修改的地方。A的大小现在为8(而B为12)。同样，ad的偏移为4,不是0。在偏移0处是的什么呢？这个问题的答案与如何实现多态相关。

```
#include <cstddef>
#include <iostream>
using namespace std;

class A {
public:
  void __cdecl m() { cout << "A::m()" << endl; }
  int ad;
};

class B : public A {
public:
  void __cdecl m() { cout << "B::m()" << endl; }
  int bd;
};

void f( A * p )
{
  p->ad = 5;
  p->m();
}

int main()
{
  A a;
  B b;
  cout << "Size of a: " << sizeof(a)
       << " Offset of ad: " << offsetof(A,ad) << endl;
  cout << "Size of b: " << sizeof(b)
       << " Offset of ad: " << offsetof(B,ad)
       << " Offset of bd: " << offsetof(B,bd) << endl;
  f(&a);
  f(&b);
  return 0;
}
```

图 7.19: 简单继承

```
1  _f__FP1A:                          ; 改编后的函数名
2        push   ebp
3        mov    ebp, esp
4        mov    eax, [ebp+8]          ; eax指向对象
5        mov    dword [eax], 5        ; ad的偏移为0
6        mov    eax, [ebp+8]          ; 将对象的地址传递给A::m()
7        push   eax
8        call   _m__1A                ; A::m()改编后的成员函数名
9        add    esp, 4
10       leave
11       ret
```

图 7.20: 简单继承的汇编代码

```
1  class A {
2  public:
3    virtual void __cdecl m() { cout << "A::m()" << endl; }
4    int ad;
5  };
6
7  class B : public A {
8  public:
9    virtual void __cdecl m() { cout << "B::m()" << endl; }
10   int bd;
11 };
```

图 7.21: 多态继承

```
1  ?f@@YAXPAVA@@@Z:
2        push   ebp
3        mov    ebp, esp
4
5        mov    eax, [ebp+8]
6        mov    dword [eax+4], 5  ; p->ad = 5;
7
8        mov    ecx, [ebp + 8]    ; ecx = p
9        mov    edx, [ecx]        ; edx = 指向vtable
10       mov    eax, [ebp + 8]    ; eax = p
11       push   eax               ; 将"this"指针压入栈中
12       call   dword [edx]       ; 调用在vtable里的第一个程序
13       add    esp, 4            ; 清理堆栈
14
15       pop    ebp
16       ret
```

图 7.22: f()函数的汇编代码

含有任意虚成员函数的C++类有一个额外的隐藏的域，它是一张指向成员函数指针数组的指针表。[13]。这个表通常称为*vtable*。对于A和B类，指针表储存在偏移地址0处。Windows编译器总是把此指针表放到继承树顶部的类的开始处。从拥有虚成员函数的程序版本(源自图 7.19)中的f函数产生的汇编代码(图 7.22)中，你可以看到对成员函数m的调用不是使用一个标号。第9行来查找对象的vtable的地址。对象的地址在第11行中被压入堆栈。第12行通过分支到vtable里的第一个地址处来调用虚成员函数。[14]。这次调用并不使用一个标号，它分支到EDX指向的代码地址处。这种类型的调用是一个*晚绑定(late binding)*的例子。晚绑定将调用哪个成员函数的判定延迟到代码运行时。这就允许代码为对象调用恰当的成员函数。标准的案例(图 7.20)硬编码某个成员函数的调用，也称为*早绑定(early binding)* (因为这儿成员函数被早绑定了，在编译的时候。)。

用心的读者将会觉得奇怪为什么在图 7.21中的类的成员函数通过使用__cdecl关键字来明确声明使用的是C调用约定。缺省情况下，Microsoft对于C++类成员函数使用的是不同的调用约定，而不是标准C调用约定。此调用约定将指向成员函数能起作用的对象的指针传递到ECX寄存器，而不是使用堆栈。成员函数的其它明确的参数仍然使用堆栈。修改为__cdecl告

[13]对于没有虚成员函数的类，C++编译器通过一个包含同样数据成员的标准C结构体来对这种类进行兼容。

[14]当然，这个值已经在ECX寄存器中了。它是在第8行放置到该寄存器的，并且可以移除第10行，再把下一行改变为push ECX。这些代码并不十分有效，因为它是在没有开启优化编译选项的情况下产生的。

```
1  class A {
2  public:
3    virtual void __cdecl m1() { cout << "A::m1()" << endl; }
4    virtual void __cdecl m2() { cout << "A::m2()" << endl; }
5    int ad;
6  };
7
8  class B : public A {     // B继承了A的m2()
9  public:
10   virtual void __cdecl m1() { cout << "B::m1()" << endl; }
11   int bd;
12 };
13 /* 显示给定的对象的vtable */
14 void print_vtable ( A * pa ) {
15   // p把pa看作是一个双字数组
16   unsigned * p = reinterpret_cast<unsigned *>(pa);
17   // vt把vtable看作是一个指针数组
18   void ** vt = reinterpret_cast<void **>(p[0]);
19   cout << hex << "vtable address = " << vt << endl;
20   for( int i=0; i < 2; i++ )
21     cout << "dword " << i << ": " << vt[i] << endl;
22
23   // 用极端的没有权限的方法来调用虚函数!
24   void (*m1func_pointer)(A *);    // 函数指针变量
25   m1func_pointer = reinterpret_cast<void (*)(A*)>(vt[0]);
26   m1func_pointer(pa);             // 通过函数指针调用成员函数m1
27
28   void (*m2func_pointer)(A *);    // 函数指针变量
29   m2func_pointer = reinterpret_cast<void (*)(A*)>(vt[1]);
30   m2func_pointer(pa);             // 通过函数指针调用成员函数m2
31 }
32
33 int main()
34 {
35   A a;    B b1;   B b2;
36   cout << "a: " << endl;   print_vtable (&a);
37   cout << "b1: " << endl;  print_vtable (&b);
38   cout << "b2: " << endl;  print_vtable (&b2);
39   return 0;
40 }
```

图 7.23: 更复杂的例子

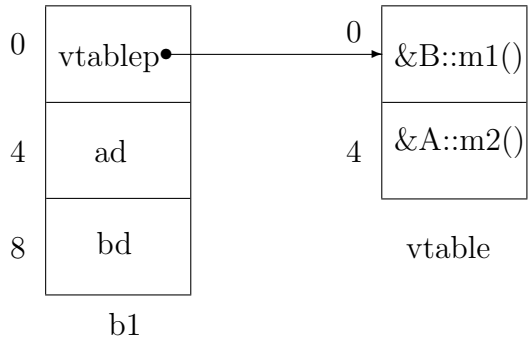

图 7.24: b1的内部表示

诉编译器使用标准C调用约定。Borland C++缺省情况下使用的是C调用约定。

下面我们再看一个稍微复杂一点的例子。(图 7.23)。在这个例子中，类A和B都有两个成员函数：m1和m2。记住因为类B并没有定义自己的成员函数m2，它继承了A类的成员函数。图 7.24展示了对象b在内存中如何储存。图 7.25展示了此程序的输出。首先，看看每个对象的vtable的地址。两个B对象的vtable地址是一样的，因此他们共享同样的vtable。一张vtable表是类的属性而不是一个对象(就如一个static数据成员)。其次，看看在vtable里的地址。从汇编程序的输出中，你可以确定成员函数m1指针在偏移地址 0处(或双字 0)而m2在偏移地址 4处(双字 1)。m2成员函数指针在类A和B的vtable中是一样的，因为类B从类A继承了成员函数m2。

第25行到32行展示了你可以通过从对象的vtable读地址的方法来调用一个虚函数[15]。成员函数地址通过一个清楚的*this*指针储存到了一个C类型函数指针中了。从图 7.25的输出中，你可以看到它确实可以运行。但是，请不要像这样写代码！这只是用来举例说明虚成员函数如何使用vtable。

从这里我们可以学到一些实践的教训。一个重要的事实是当你读或写类变量到一个二进制源文件中时，你必须非常小心。你不可以在整个对象中仅仅使用一个二进制读或写，因为可能会读或写源文件之外的vtable指针！这是一个指向留在程序内存中的vtable的指针，而且不同的程序将不同。同样的问题会发生在C语言的结构中，但是在C语言中，结构体只有当程序员明确将指针放到结构体中时，结构体内部才有指针。类A或类B中，并没有明显地定义过指针。

再次，认识到不同的编译器实现虚成员函数的方法是不一样的是非常重要的。在In Windows中，COM(组件对象模型，Component Object Model) 类对象使用vtable来实现COM接口[16]。只有像Microsoft一样用来实现虚成员函数的编译器才可以创建COM类。这也是为什么Borland采用和Microsoft一样的实现方法的原因，也是为什么不可以用*gcc*来创建COM类的原因之一。

[15] 记住这些代码只能在MS和Borland编译器下运行，*gcc*不行。

[16] COM类同样使用__stdcall 调用约定，而不是标准C调用约定。

```
a:
vtable address = 004120E8
dword 0:  00401320
dword 1:  00401350
A::m1()
A::m2()
b1:
vtable address = 004120F0
dword 0:  004013A0
dword 1:  00401350
B::m1()
A::m2()
b2:
vtable address = 004120F0
dword 0:  004013A0
dword 1:  00401350
B::m1()
A::m2()
```

图 7.25: 图 7.23中程序的输出

虚成员函数的代码和非常虚的成员函数的代码非常相像。只是调用它们的代码是不同的。如果汇编器能绝对保证调用哪个虚成员函数，那么它可以忽略vtable，直接调用成员函数。(*例如*，使用早绑定)。

7.2.6 C++的其它特性

C++其它特性的工作方式(*例如*，除了处理继承和多继承，还有运行时类型识别)不属于本书的范围。如果读者希望走得更远一些，一个好的起点是Ellis 和Stroustrup写的*The Annotated C++ Reference Manual*和Stroustrup写的*The Design and Evolution of C++*。

附录 A

80x86指令

A.1 非浮点指令

这一节列出和描述了Intel 80x86CPU家族的非浮点指令的行为和格式。这些格式使用下面的约定：

R	通用寄存器
R8	8位寄存器
R16	16位寄存器
R32	32位寄存器
SR	段寄存器
M	内存
M8	字节
M16	字
M32	双字
I	立即数

上面这些可以结合使用于多操作数指令。例如：格式*R, R*表示指令携带两个寄存器器操作数。许多双操作数指令允许同样类型的操作数。约定*O2*可以用来表示这些操作数：*R,R R,M R,I M,R M,I*。如果一个操作数可以是8位的寄存器或内存，则使用这样的约定：*R/M8*。

这个表同样展示了每一条指令如何影响FLAGS寄存器中的不同的位。如果列为空，则表示与它相应的位没有被影响。如果这些位总是被改为一特定的值，则在相应的列中显示一个1或0。如果位的改变的值依赖于指令的操作数，则在相应的列中显示为*C*。最后，如果位被某种未定义的形式修改，则在列中显示*?*。因为改变方向标志位的唯一指令是CLD和STD，所以在FLAGS列中，它们没有被列出来。

名称	描述	格式	标志位					
			O	S	Z	A	P	C
ADC	带进位相加	O2	C	C	C	C	C	C

名称	描述	格式	标志位 O	S	Z	A	P	C
ADD	整数相加	O2	C	C	C	C	C	C
AND	按位AND	O2	0	C	C	?	C	0
BSWAP	字节次序变反	R32						
CALL	调用程序	R M I						
CBW	将字节转换成字							
CDQ	将双字转换成四字							
CLC	进位标志位清0							0
CLD	方向标志位清0							
CMC	进位标志位变反							C
CMP	整数比较	O2	C	C	C	C	C	C
CMPSB	字节比较		C	C	C	C	C	C
CMPSW	字比较		C	C	C	C	C	C
CMPSD	双字比较		C	C	C	C	C	C
CWD	将字转换成双字，并储存到DX:AX中							
CWDE	将字转换成双字，并储存到EAX中							
DEC	整数减一	R M	C	C	C	C	C	
DIV	无符号数相除	R M	?	?	?	?	?	?
ENTER	建立堆栈帧	I,0						
IDIV	有符号数相除	R M	?	?	?	?	?	?
IMUL	有符号数相乘	R M R16,R/M16 R32,R/M32 R16,I R32,I R16,R/M16,I R32,R/M32,I	C	?	?	?	?	C
INC	整数加一	R M	C	C	C	C	C	
INT	产生中断	I						
JA	如果大于则跳转	I						
JAE	如果大于等于则跳转	I						
JB	如果小于则跳转	I						
JBE	如果小于等于则跳转	I						
JC	如果进位为1则跳转	I						
JCXZ	如果CX = 0则跳转	I						
JE	如果等于则跳转	I						
JG	如果大于则跳转	I						
JGE	如果大于等于则跳转	I						
JL	如果小于则跳转	I						

名称	描述	格式	标志位					
			O	**S**	**Z**	**A**	**P**	**C**
JLE	如果小于等于则跳转	I						
JMP	无条件跳转	R M I						
JNA	如果不大于则跳转	I						
JNAE	如果不大于行于则跳转	I						
JNB	如果不小于则跳转	I						
JNBE	如果不小于等于则跳转	I						
JNC	如果没有进位则跳转	I						
JNE	如果不等于则跳转	I						
JNG	如果不大于则跳转	I						
JNGE	如果不大于等于则跳转	I						
JNL	如果不小于则跳转	I						
JNLE	如果不小于等于则跳转	I						
JNO	如果不溢出则跳转	I						
JNS	如果SF=0则跳转	I						
JNZ	如果ZF=0则跳转	I						
JO	如果溢出则跳转	I						
JPE	如果PF=1则跳转	I						
JPO	如果PF=0则跳转	I						
JS	如果SF=1则跳转	I						
JZ	如果ZF=1则跳转	I						
LAHF	将FLAGS的低字节载入到AH中							
LEA	载入有效的地址	R32,M						
LEAVE	释放堆栈帧							
LODSB	载入字节							
LODSW	载入字							
LODSD	载入双字							
LOOP	循环	I						
LOOPE/LOOPZ	如果ZF=1则循环	I						
LOOPNE/LOOPNZ	如果ZF=0则循环	I						
MOV	移动数据	O2 SR,R/M16 R/M16,SR						
MOVSB	移动字节							
MOVSW	移动字							
MOVSD	移动双字							

名称	描述	格式	标志位 O	S	Z	A	P	C
MOVSX	符号扩展移动	R16,R/M8						
		R32,R/M8						
		R32,R/M16						
MOVZX	零扩展移动	R16,R/M8						
		R32,R/M8						
		R32,R/M16						
MUL	无符号数相乘	R M	C	?	?	?	?	C
NEG	求反	R M	C	C	C	C	C	C
NOP	无操作							
NOT	非运算	R M						
OR	按位OR	O2	0	C	C	?	C	0
POP	出栈	R/M16						
		R/M32						
POPA	全部出栈							
POPF	出栈送FLAGS		C	C	C	C	C	C
PUSH	进栈	R/M16						
		R/M32 I						
PUSHA	全部进栈							
PUSHF	FLAGS进栈							
RCL	带进位循环左移	R/M,I	C					C
		R/M,CL						
RCR	带进位循环右移	R/M,I	C					C
		R/M,CL						
REP	重复执行							
REPE/REPZ	如果ZF=1则重复执行							
REPNE/REPNZ	如果ZF=0则重复执行							
RET	返回							
ROL	循环左移	R/M,I	C					C
		R/M,CL						
ROR	循环右移	R/M,I	C					C
		R/M,CL						
SAHF	将AH复制到FLAGS中			C	C	C	C	C
SAL	算术左移	R/M,I						C
		R/M, CL						
SBB	带借位相减	O2	C	C	C	C	C	C
SCASB	扫描字节		C	C	C	C	C	C
SCASW	扫描字		C	C	C	C	C	C
SCASD	扫描双字		C	C	C	C	C	C
SETA	如果大于则目的字节置1	R/M8						

名称	描述	格式	标志位					
			O	**S**	**Z**	**A**	**P**	**C**
SETAE	如果大于等于则目的字节置1	R/M8						
SETB	如果小于则目的字节置1	R/M8						
SETBE	如果小于等于则目的字节置1	R/M8						
SETC	如果进位标志位为1则目的字节置1	R/M8						
SETE	如果等于则目的字节置1	R/M8						
SETG	如果大于则目的字节置1	R/M8						
SETGE	如果大于等于则目的字节置1	R/M8						
SETL	如果小于则目的字节置1	R/M8						
SETLE	如果小于等于则目的字节置1	R/M8						
SETNA	如果不大于则目的字节置1	R/M8						
SETNAE	如果不大于等于则目的字节置1	R/M8						
SETNB	如果不小于则目的字节置1	R/M8						
SETNBE	如果不小于等于则目的字节置1	R/M8						
SETNC	如果进位标志位为0则目的字节置1	R/M8						
SETNE	如果不等于则目的字节置1	R/M8						
SETNG	如果不大于则目的字节置1	R/M8						
SETNGE	如果不大于等于则目的字节置1	R/M8						
SETNL	如果不小于则目的字节置1	R/M8						
SETNLE	如果不小于等于则目的字节置1	R/M8						
SETNO	如果OF=0则目的字节置1	R/M8						

名称	描述	格式	标志位 O	S	Z	A	P	C
SETNS	如果SF=0则目的字节置1	R/M8						
SETNZ	如果ZF=0则目的字节置1	R/M8						
SETO	如果OF=1则目的字节置1	R/M8						
SETPE	如果PF=1则目的字节置1	R/M8						
SETPO	如果PF=0则目的字节置1	R/M8						
SETS	如果SF=1则目的字节置1	R/M8						
SETZ	如果ZF=1则目的字节置1	R/M8						
SAR	算术右移	R/M,I R/M, CL						C
SHR	逻辑右移	R/M,I R/M, CL						C
SHL	逻辑左移	R/M,I R/M, CL						C
STC	进位标志位置1							1
STD	方向标志位置1							
STOSB	储存字节							
STOSW	储存字							
STOSD	储存双字							
SUB	相减	O2	C	C	C	C	C	C
TEST	逻辑比较	R/M,R R/M,I	0	C	C	?	C	0
XCHG	交换	R/M,R R,R/M						
XOR	按位XOR	O2	0	C	C	?	C	0

A.2 浮点数指令

在这一节中，描述了许多80x86数字协处理器的指令。说明部分简要地描述了指令的操作。为了节省空间，关于指令是否出栈的信息并没有在描述中给出。

格式列展示了每个指令可以使用的操作数类型。使用的是下面的约定：

STn	一个协处理器寄存器
F	内存中的单精度数
D	内存中的双精度数
E	内存中的扩展精度数
I16	内存中的整形字
I32	内存中的整形双字
I64	内存中的整形四字

需要奔腾或更好的处理器的指令用星号标示出来了(*)。

指令	描述	格式
FABS	ST0 = |ST0|	
FADD *src*	ST0 += *src*	STn F D
FADD *dest*, ST0	*dest* += ST0	STn
FADDP *dest*[,ST0]	*dest* += ST0	STn
FCHS	ST0 = −ST0	
FCOM *src*	比较ST0和*src*	STn F D
FCOMP *src*	比较ST0和*src*	STn F D
FCOMPP *src*	比较ST0和ST1	
FCOMI* *src*	比较并设置FLAGS	STn
FCOMIP* *src*	比较并设置FLAGS	STn
FDIV *src*	ST0 /= *src*	STn F D
FDIV *dest*, ST0	*dest* /= ST0	STn
FDIVP *dest*[,ST0]	*dest* /= ST0	STn
FDIVR *src*	ST0 = *src*/ST0	STn F D
FDIVR *dest*, ST0	*dest* = ST0/*dest*	STn
FDIVRP *dest*[,ST0]	*dest* = ST0/*dest*	STn
FFREE *dest*	Marks as empty	STn
FIADD *src*	ST0 += *src*	I16 I32
FICOM *src*	比较ST0和*src*	I16 I32
FICOMP *src*	比较ST0和*src*	I16 I32
FIDIV *src*	ST0 /= *src*	I16 I32
FIDIVR *src*	ST0 = *src*/ST0	I16 I32
FILD *src*	将*src*压入栈中	I16 I32 I64
FIMUL *src*	ST0 *= *src*	I16 I32
FINIT	初始化协处理器	

指令	描述	格式
FIST *dest*	保存ST0	I16 I32
FISTP *dest*	保存ST0	I16 I32 I64
FISUB *src*	ST0 -= *src*	I16 I32
FISUBR *src*	ST0 = *src* - ST0	I16 I32
FLD *src*	将*src*压入栈中	STn F D E
FLD1	将1.0压入栈中	
FLDCW *src*	装载控制字寄存器	I16
FLDPI	将π压入栈中	
FLDZ	将0.0压入栈中	
FMUL *src*	ST0 *= *src*	STn F D
FMUL *dest*, ST0	*dest* *= ST0	STn
FMULP *dest*[,ST0]	*dest* *= ST0	STn
FRNDINT	ST0取整	
FSCALE	ST0 = ST0 $\times 2^{\lfloor \text{ST1} \rfloor}$	
FSQRT	ST0 = $\sqrt{\text{ST0}}$	
FST *dest*	储存ST0	STn F D
FSTP *dest*	储存ST0	STn F D E
FSTCW *dest*	储存控制字寄存器	I16
FSTSW *dest*	储存状态字寄存器	I16 AX
FSUB *src*	ST0 -= *src*	STn F D
FSUB *dest*, ST0	*dest* -= ST0	STn
FSUBP *dest*[,ST0]	*dest* -= ST0	STn
FSUBR *src*	ST0 = *src*-ST0	STn F D
FSUBR *dest*, ST0	*dest* = ST0-*dest*	STn
FSUBP *dest*[,ST0]	*dest* = ST0-*dest*	STn
FTST	比较ST0和0.0	
FXCH *dest*	将ST0和*dest*内容交换	STn

索引