PC組合語言

Paul A. Carter

翻譯：伍星

二〇〇七年三月五日

目录

前言

目的

這本書的目的是為了讓讀者更好地理解電腦在相比於編程語言如Pascal的更底層如何工作。通過更深刻地瞭解電腦如何工作，讀者通常可以更有能力用高階語言如C和C++來開發軟體。學習用組合語言來編程是達到這個目的的一個極好的方法。其他的PC組合語言程式的書仍然在講授著如何在1981年使用在初始的PC機上的8086處理器上進行編程!那時的8086處理器只支援實模式。在這種模式下，任何程式都可以定址任意記憶體或訪問電腦裏的任意設備。這種模式不適合於安全，多工作業系統。這本書改為敍述在80386和後來的處理器如何在保護模式(也就是Windows和Linux運行的模式)下進行編程。這種模式支援現在作業系統所期望的特徵，比如：虛擬記憶體和記憶體保護。使用保護模式有以下幾個原因：

1. 在保護模式下編程比在其他書使用的8086實模式下要容易。
2. 所有的現代的PC作業系統都運行在保護模式下。
3. 可以獲得運行在此模式下的免費軟體。

關於保護模式下的PC彙編編程的書籍的缺乏是作者書寫這本書的主要原因。

就像上面所提到的，這本書使用了免費/開源的軟體：也就是，NASM彙編器和DJGPP C/C++編譯器。它們都可以在網際網路上下載到。本書同樣討論如何在Linux作業系統下和在Windows下的Borland和Microsoft的C/C++編譯器中如何使用NASM彙編代碼。所有這些平臺上的例子都可以在我的網頁上找到： `http://www.drpaulcarter.com/pcasm`. 如何你想彙編和運行這本教程上的例子，你必須下載這些樣例代碼。

注意這本書並不打算涵蓋彙編編程的各個方面。作者盡可能地涉及了所有程式師都應該熟悉的最重要的方面。

致謝

作者要感謝世界上許多為免費/開源運動作出貢獻的程式師。這本書的程式甚至是這本書本身都是使用免費軟體來書寫的。作者特別要感謝John S. Fine， Simon Tatham，Julian Hall和其他開發NASM彙編器的人，這本書的所有例子都基於這個彙編器；開發DJGPP C/C++DJ編譯器的Delorie；眾多對DJGPP所基於的GNU gcc編譯器有貢獻的人們；Donald Knuth和其他開發TEX 和 LATEX 2ε 排版語言的人，也正是用於書寫這本書的語言；Richard Stallman (免費軟體基金會的創立者)，Linus Torvalds (Linux內核的創建者)和其他作者用來書寫這本書的底層軟體的創建者。

謝謝以下的人提出的修改意見：

- 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

網際網路上的資源

作者的網頁	`http://www.drpaulcarter.com/`
NASM源代碼的網頁	`http://sourceforge.net/projects/nasm/`
DJGPP	`http://www.delorie.com/djgpp`
Linux彙編器	`http://www.linuxassembly.org/`
彙編的藝術(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`

回饋

作者歡迎任意關於這本書的回饋資訊。.

E-mail: pacman128@gmail.com

WWW: http://www.drpaulcarter.com/pcasm

第一章　簡介

第一节　數制

電腦裏的記憶體由數位組成。電腦記憶體並沒有以十進位(基數為10)來儲存這些數位。因為電腦以二進位(基數為2)格式來儲存所有資訊能極大地簡化硬體。首先讓我們來回顧一下十進位數字制。

1.1.1　十進位

基數為10的數制由10個數碼(0-9)組成。一個數的每一位有基於它在數中的位置相關聯的10的乘方值。例如：

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

1.1.2　二進位

基數為2的數制由2個數碼(0和1)組成。一個數的每一位有基於它在數中的位置相關聯的2的乘方值。(一個二進位數字位被稱為一個比特位。)例如：

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

這些演示了二進位如何轉換成十進位。表 1.1展示了開始的幾個十進位數字如何以二進位替代。

圖 1.1 演示單個的二進位數字字(也就是: 位)相加。下面是一個例子：

十進位	二進位		十進位	二進位
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: 在二進位中十進位0到15的表示

以前無進位				以前有進位			
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: 二進位加法(c代表進位)

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

如果有人考慮了下面的十進位的除法：

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

他可以看到這個除法除去了這個數的最右邊的十進位數字而且將其他的十進位數字向右移動了一位。除以2也是執行同樣的操作，除了是為了得到一個數的二進位位元外。考慮下面二進位數字的除法[1]：

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

這個現象可以用來將一個十進位轉換成它的等價的二進位表示形式，

[1]2這個下標是用來表明這個數位是以二進位表示，而不是十進位

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: 十進位轉換

像圖 1.2展示的一樣。這種方法首先找到最右邊的數位，這個數位被稱為最低的有效位 (lsb)。最左邊的數位稱為最高的有效位 (msb)。記憶體的基本單元由8位元組成，稱它為一個位元組。

1.1.3 十六進位

十六進位數使用的基數為16。十六進位(或者簡短稱為hex)可以用作二進位數字的速記形式。十六進位需要16個數碼。這就產生了一個問題，因為沒有符號可以用來表示在9之後的額外的數字。通過協定，字母被用來表示這些額外的數位。這16個十六進位數字是0-9，然後A， B， C， D，E和 F。數A等價於十進位的10，B是11，等等。一個十六進位的每一位有基於它在數中的位置相關聯的16的乘方值。例如：

$$\begin{aligned} 2\text{BD}_{16} &= 2 \times 16^2 + 11 \times 16^1 + 13 \times 16^0 \\ &= 512 + 176 + 13 \\ &= 701 \end{aligned}$$

將十進位轉換成十六進位，可以使用和二進位轉換同樣的方法，除了除以16外。看例子圖 1.3。

十六進位非常有用的原因是因為十六進位和二進位之間轉換有一個非

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

因此 $589 = 24\mathrm{D}_{16}$

图 1.3:

常簡單的方法。二進位數字非常大而且非常繁鎖。十六進位提供一個比較舒服的方法來表示二進位數字。

將一個十六進位數轉換成二進位數字，只需要簡單地將每一位十六進位數轉換成4位二進位數字。例如：$24\mathrm{D}_{16}$轉換成$0010\ 0100\ 1101_2$。注意在這些4位二進位數字中領頭的0非常重要！如果$24\mathrm{D}_{16}$中間的那位的4位二進位數字的領頭的0沒用使用的話，那麼結果就是錯的。從二進位轉換成十六進位同樣簡單。只需反過來做剛才那個處理，將二進位每4位元一段轉換成十六進位。從二進位數字的最右端開始，而不是最左端。這樣就能保證處理過程使用了正確的4位段 [2]. 例如:

110	0000	0101	1010	0111	1110_2
6	0	5	A	7	E_{16}

一個四位元的數被稱為半位元組 。因此每一位元十六進位相當於一個半位元組。兩個半位元組為一個位元組，所以一個位元組可以用兩位元十六進位數來表示。一個位元組值的範圍以二進位表示為0到11111111，以十六進位表示為0到FF，以十進位表示為0到255。

[2]如果不明白起點為什麼是那樣，那麼換過來，試著將這個例子從左邊開始轉換。

word(字)	2個位元組
double word(雙字)	4個位元組
quad word(四字)	8個位元組
paragraph(一節)	16個位元組

表 1.2: 記憶體單元

第二节 電腦結構

1.2.1 記憶體

記憶體的基本單元是一個位元組。 一台有32兆記憶體的電腦大概能容納3200萬位元組的資訊。在記憶體裏的每一個位元組通過一個唯一的數位來標識作為它的位址像圖 1.4展示的一樣。

記憶體以千位元組($2^{10} = 1024$位元組)，百萬位元組($2^{20} = 1048576$ 位元組)和十億位元元組($2^{30} = 1073741824$ 位元組)來測量。

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

图 1.4: 記憶體位址

通常記憶體都是大塊大塊地使用而不是單個位元組。在PC機結構中，命名了這些記憶體大塊像表 1.2展示的一樣。

在記憶體裏的資料都是數位的。字元通過用數位來表示字元的字元編碼來儲存。其中一個最普遍的字元編碼稱為 ASCII(美國資訊互換標準編碼)。一個新的，更完全的，正在替代ASCII的編碼是Unicode。在這兩種編碼中最主要的區別是ASCII使用一個位元組來編碼一個字元，但是Unicode每個字元使用兩個位元組(或一個 字)。例如：ASCII使用41_{16} (65_{10})來表示字元大寫A；Unicode使用0041_{16}來表示。因為ASCII使用一個位元組，所以它僅能表示256種不同的字元[3]。Unicode將ASCII的值擴展成一個字，允許表示更多的字元。這對於表示全世界所有的語言非常重要。

1.2.2 CPU

中央處理器(CPU)是執行指令的物理設備。 CPU 執行的指令通常非常

[3]事實上，ASCII僅僅使用低7位，所以只有128種不同的值可以使用。

簡單。指令可能要求它們使用的資料存儲在一個CPU稱為寄存器的特殊的儲存位置中。CPU可以比訪問記憶體更快地訪問寄存器裏的資料。然而，在CPU裏的寄存器是有限的，所以程式師必須注意只保存現在使用的資料到寄存器中。

各類CPU執行的指令組成了該CPU的機器語言。機器語言擁有比高階語言更基本的結構。機器語言指令被編碼成未加工的數位，而不是友好的文本格式。為了更有效的運行，CPU必須能很快地解譯一個指令的目的。機器語言就是為了這個目的設計的，而不是讓人們更容易理解而設計。一個其他語言寫的程式必須轉換成CPU的本地機器語言，才能在電腦上運行。編譯器 是一個將用程式語言寫的程式翻譯成特殊結構的電腦的機器語言的程式。通常，每一種類型的CPU都有它自己唯一的機器語言。這是為什麼為 Mac 寫的程式不能在IBM類型PC機運行的一個原因。

GHz代表十萬萬赫或是每秒十億次迴圈。1.5GHz 的CPU每秒有15億的時鐘脈衝。

電腦通過使用時鐘 來同步指令的執行。時鐘脈衝在一個固定的頻率(稱為時鐘頻率)。當你買了一台1.5 GHz 的電腦，1.5 GHz就是時鐘頻率[4]。時鐘並不記錄分和秒。它以不變的速率簡單跳動。電子電腦通過使用這個跳動來正確執行它們的操作，就像節拍器的跳動如何來幫助你以正確的節奏播放音樂。一個指令需要跳動的次數(或就像他們經常說的執行週期)依賴CPU的產生和模仿。週期的次數取決於在它之前的指令和其他因素。

1.2.3 CPU 80x86系列

IBM型號的PC機包含了一個來自Intel 80x86家族(或它的克隆)的CPU。在這個家族的所有CPU都有一些普遍的特徵，包括有一種基本的機器語言。無論如何，最近的成員極大地加強了這個特徵。

8088，8086: 這些CPU從編程的觀點來看是完全相同的。它們是用在早期PC機上的CPU。它們提供一些16位的寄存器：AX， BX，CX，DX，SI，DI，BP，SP，CS，DS，SS， ES，IP， FLAGS。它們僅僅支援1M位元組的記憶體，而且只能工作在實模式下。在這種模式下，一個程式可以訪問任何記憶體位址，甚至其他程式的記憶體！這會使排除故障和保證安全變得非常困難！而且，程式的記憶體需要分成段。每段不能大於64K。

[4]實際上，時鐘脈衝使用在許多不同的CPU元件中。其他元件通常使用與CPU不同的時鐘頻率。

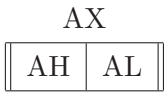

图 1.5: AX寄存器

80286: 這種CPU使用在AT系列的PC機中。它在8088/86的基本機器語言中加入了一些新的指令。然而，它主要的新的特徵是16位元保護模式。在這種模式下，它可以訪問16M位元組的記憶體和通過阻止訪問其他程式的記憶體來保護程式。可是，程式依然是分成不能大於64K的段。

80386: 這種CPU極大地增強了80286的性能。首先，它擴展了許多寄存器來容納32位元資料(EAX，EBX，ECX，EDX，ESI，EDI，EBP，ESP，EIP)而且增加了兩個新的16位寄存器（FS，GS）。它同樣增加了一個新的32位元保護模式。在這種模式下，它可以訪問4G位元組。程式同樣分成段，但是現在每段大小同樣可以到4G。

80486/Pentium/Pentium Pro: 這些80x86家族的成員增加了不多的新的特徵。它們主要是提高了指令執行的速度。

Pentium MMX: 這些處理器在Pentium基礎上增加了MMX指令 (多媒體擴展)。這些指令可以提高普通的圖像操作的速率。

Pentium II: 它是擁有 MMX 指令的Pentium處理器。(Pentium III 本質上就是一個更快的Pentium II。)

1.2.4 8086 16位 寄存器

最初的8086CPU提供4個16位通用寄存器：AX，BX， CX 和DX。這些寄存器都可以分解成兩個8位寄存器。例如：AX寄存器可以分解成AH和AL寄存器，像圖 1.5展示的一樣。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都當做獨立的一個位元組的寄存器來用；但是，清楚它們不能獨立於AX是非常重要的。改變AX的值將會改變AH和AL的值反之亦然。通用寄存器多數使用在資料移動和算術指令中。

這有兩個16位指針寄存器： SI 和 DI 。通常它們都是當作指標來使用，但是在許多情況下也可以像通用寄存器一樣使用。但是，它們不可以分解成8位寄存器。

16位BP和 SP 寄存器用來指向機器語言堆疊裏的資料，被各自稱為基址寄存器 和堆疊指標寄存器。這些將在以後討論。

16位CS，DS，SS 和ES 寄存器是段寄存器。 它們指出程式不同部分所使用的記憶體。CS代表代碼段， DS 代表資料段， SS 代表堆疊段和ES代表附加段。ES當作一個暫時段寄存器來使用。這些寄存器的細節描述在小節 1.2.6 和1.2.7中。

指令指標寄存器(IP) 與CS寄存器一起使用來跟蹤CPU下一條執行指令的位址。通常，當一條指令執行時，IP提前指向記憶體裏的下一條指令。

FLAGS寄存器儲存了前面指令執行結果的重要資訊。這些結果在寄存器裏以單個的位元儲存。例如：如果前面指令執行結果是0，Z位為1，反之為0。並不是所有指令都修改FLAGS裏的位，查看附錄裏的表看單個指令是如何影響FLAGS寄存器的。

1.2.5 80386 32位 寄存器

80386及以後的處理器擴展了寄存器。例如：16位AX寄存器擴展成了32位。為了向後相容，AX依然表示16位寄存器而 EAX 用來表示擴展的32位寄存器。AX是 EAX 的低16位就像AL是AX(EAX)的低8位一樣。但是沒有直接訪問 EAX 高16位的方法。其他的擴展寄存器是 EBX，ECX，EDX，ESI 和 EDI 。

許多其他類型的寄存器同樣也擴展了。BP變成了EBP；SP 變成了ESP；FLAGS變成了EFLAGSEFLAGS 而IP變成了EIP。但是，不同於指標寄存器和通用寄存器，在32位元保護模式下(下面將討論的)只有這此寄存器的擴展形式被使用。

在80386裏，段寄存器依然是16位的。這兒有兩個新的段寄存器：FS和GS。它們名字並不代表什麼。它們是附加段寄存器(像ES一樣)。

術語中字的一個定義為CPU資料寄存器的大小。對於80x86家族，這個術語現在有一點混亂了。在表 1.2裏，可以看到字被定義成兩個位元組。它是當8086第一次發行時被定義成這樣的。當80386開發出來後，它被決定依舊保持這個字定義不改變，即使寄存器的大小已經改變了。

1.2.6 實模式

在實模式下，記憶體被限制為僅有1M位元組(2^{20}位元組)。有效的位址從00000
到 FFFFF(十六進位)。 這些位址需要用20位元的數來表示。顯然，一個20位的數不適合任何一個8086的16位寄存器。Intel通過利用兩個16位數值來決定一個位址的方法來解決這個問題。開始的16位值稱為段位址(selector)。段位址的值必須存儲在段寄存器中。第二個16位值稱為偏移位址(offset)。用32位段地址：偏移位址表示的物理位址可以由下面的公式計算：

那麼，無恥的DOS640K限制來自哪里？BIOS為它的代碼和硬體設備如顯示器要求了1M記憶體中的一些記憶體。

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

在十六進位中乘以16是非常容易的，只需要在數的右邊加0。例如：表示為047C:0048的物理位址通過這樣得到：

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

實際上，段地址的值是一節的首地址(看表 1.2)。

真實的段位址有以下的缺點：

- 一個段位址只能指向64K記憶體(16位元偏移的上限)。如果一個程式擁有大於64K的代碼那又怎麼辦呢？在CS裏的一個單一的值不能滿足整個程式執行的需要。程式必須分成小於64K的段(segment)。當執行從一段移到另一段時，CS裏的值必須改變。同樣的問題發生在大量的資料和 DS 寄存器之間。這樣使用是非常不方便的！
- 每個位元組在記憶體裏並不只有唯一的段位址。物理位址04808可以表示為：047C:0048， 047D:0038， 047E:0028 或047B:0058。 這將使段地址的比較變得複雜。

1.2.7 16位元保護模式

在80286的16位元保護模式下，段位址的值與實模式相比解釋得完全不同。在實模式下，一個段位址的值是實體記憶體裏的一節的首地址。在保護模式下，一個段位址的值是一個指向描述符表的指標。兩種模式下，程

式都是被分成段。在實模式下，這些段在實體記憶體的固定位置而且段位址的值表示段開始處所在節的首位址。在保護模式下，這些段不是在實體記憶體的固定的地址。事實上，它們根本不一定需要在記憶體中。

保護模式使用了一種叫做虛擬記憶體 的技術。虛擬記憶體的基本思想是僅僅保存程式現在正在使用的代碼和資料到記憶體中。其他資料和代碼暫時儲存在硬碟中直到它們再次需要時。當一段從硬碟重新回到記憶體中，它很有可能放在不同於它移動到硬碟之前時的位置的記憶體中。所有這些都由作業系統透明地執行。程式並不需要因為要讓虛擬記憶體工作而使用不同的書寫方法。

在保護模式下，每一段都分配了一條描述符表裏的條目。這個條目擁有系統想知道的關於這段的所有資訊。這些資訊包括：現在是否在記憶體中；如果在記憶體中，在哪；訪問許可權(例如： 唯讀)。段的條目的指標是儲存在段寄存器裏的段位址值。

一個非常著名的PC專家稱286CPU為“死了的大腦”

16位元保護模式的一個大的缺點是偏移位址依然是16位數。這個的後果就是段的大小依然限制為最大64K。這會導致使用大的陣列時會有問題。

1.2.8 32位元保護模式

80386引入了32位元保護模式。386 32位元保護模式和286 16位元保護模式之間最主要的區別是：

1. 偏移位址擴展成了32位。這就允許偏移位址範圍升至4G。因此，段的大小也升至4G。

2. 段可以分成較小的4K大小的單元，稱為記憶體頁。虛擬記憶體系統工作在頁的方式下，代替了段方式。這就意味著一段在任何一個時刻只有部分可能在記憶體中。在28616位元保護模式下，要麼整個段在記憶體中，要麼整個不在。這樣在32位元模式下允許的大的段的情況下很不實用。

在Windows 3.x系統中，標準模式為286 16位元保護模式而增強模式為32位元保護模式。Windows 9X，Windows NT/2000/XP，OS/2和Linux都運行在分頁管理的32位元保護模式下。

1.2.9 中斷

有時候普通的程式流必須可以被要求快速反應的處理事件中斷。電腦提供了一個稱為中斷的結構來處理這些事件。例如：當一個滑鼠移動了，硬體滑鼠中斷現在的程式來處理滑鼠移動(移動滑鼠， 等等)。中斷導致控制權轉移到一個中斷處理程式。中斷處理程式是處理中斷的程式。每種類型的中斷都分配了一個中斷號。在實體記憶體的開始處，存在一張包含中斷處理程式段位址的中斷向量 表。中斷號是這張表中最基本的指標。

外部中斷由CPU的外部引起。(滑鼠就是這一類型的例子。)許多I/O設備引起中斷(例如：鍵盤，時鐘，硬碟驅動器，CD-ROM和音效卡)。內部中斷由CPU的內部引起，要麼是由一個錯誤引起，要麼由中斷指令引起。錯誤中斷稱為陷阱。由中斷指令引起的中斷稱為軟中斷。DOS使用這些類型的中斷來實現它的API(應用程式介面)。許多現代的作業系統(如：Windows和UNIX)使用一個基於C的介面。 [5]

許多中斷處理程式當它執行完成時，將控制權返回給被中斷的程式。它們恢復寄存器，裏面的值與中斷發生之前的值相同。因此，被中斷的程式就像沒有任何事情發生一樣運行(除了它失去了一些CPU週期)。陷阱通常不返回。通常它們中止程式。

第三节 組合語言

1.3.1 機器語言

每種類型的CPU都能理解它們自己的機器語言。機器語言裏的指令是以位元組形式在記憶體中儲存的數位。每條指令有它唯一的數位碼稱為操作代碼，或簡稱為操作碼 。80x86處理器的指令大小不同。操作碼通常是在指令的開始處。許多指令還包含指令使用的資料(例如：常量或地址)。

機器語言很難直接進行編程。解譯這些數位代碼指令的意思對人類來說是沉悶的。例如：執行將 EAX 和 EBX 寄存器相加然後將結果送回到 EAX 的指令以十六進位碼編譯如下：

03 C3

[5]然而，它們在內核級可能會使用一個更低等級的介面。

這個很難理解。幸運的是，一個叫做彙編的程式 可以為程式師做這個沉悶的工作。

1.3.2 組合語言

一個組合語言程式以文本格式儲存(正如一個高階語言程式)。每條彙編指令代表確切的一條機器指令。例如：上面描述的加法指令在組合語言中將描述成：

```
add eax, ebx
```

這裏指令的意思比在機器代碼表示得更清楚。代碼add是加法指令的助記符。一條彙編指令的通常格式為：

```
mnemonic(助記符) operand(s)(運算元)
```

組合語言程式 是一個讀包含彙編指令的文字檔案和將組合語言轉換成機器代碼的程式。編譯器 是為高級編程語言做同樣轉換的程式。一個組合語言程式比一個編譯器要簡單。 每條彙編語句接代表一個唯一的機器指令。高階語言更複雜而且可能要求更多的機器指令。

它花費了電腦科學家幾年的時間來揣測如何編寫一個編譯器!

彙編和高階語言之間另一個更重要的區別是因為每種不同類型的CPU有它自己的機器代碼，所以它同樣有它自己的組合語言。在不同的電腦構造中移植組合語言比高階語言要困難得多。

這本書使用了 Netwide Assembler，或簡稱為 NASM 。它在Internet上是免費提供的(要得到URL，請看前言)。更普遍的組合語言程式是Microsoft Assembler(MASM) 或Borland Assembler (TASM)。MASM/TASM和 NASM之間有一些彙編語法區別。

1.3.3 指令運算元

機器代碼指令擁有個數和類型不同的運算元；然而，通常每個指令有幾個固定的運算元(0到3個)。運算元可以有下面的類型：

寄存器: 這些運算元直接指向CPU寄存器裏的內容。

記憶體: 這些運算元指向記憶體裏的資料。資料的位址可能是硬編碼到指令裏的常量或可能直接使用寄存器的值計算得到。距離段的起始位址的偏移值即為此位址。

立即數: 這些運算元是指令本身列出的固定的值。它們儲存在指令本身(在代碼段)，而不在資料段。

暗指的運算元: 這些運算元沒有明確顯示。例如：往寄存器或記憶體增加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。它將資料從一個地方移到另一個地方(像高階語言裏面的賦值操作一樣)。它攜帶兩個運算元：

```
mov dest (目的運算元), src(源運算元)
```

src指定的資料拷貝到了dest。一個指令的兩個運算元不能同時是記憶體運算元。這就指出了一個彙編古怪的地方。通常，對於各種各樣指令的使用都有某些強制性的規定。運算元必須是同樣的大小。AX裏的值就不能儲存到 BL 裏去。

這兒有一個例子(分號表示注釋的開始)：

```
mov     eax, 3    ; 將3存入 EAX 寄存器(3是一個立即數)。
mov     bx, ax    ; 將AX的值存入到BX寄存器。
```

ADD 指令用來進行整形資料的相加。

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

SUB 指令用來進行整形資料的相減。

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

INC 和DEC 指令將值加1或減1。因為1是一個暗指的運算元，INC和DEC的機器代碼比等價的ADD和SUB指令要少。

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

1.3.5 指示符

指示符是由組合語言程式產生的而不是由CPU產生。它們通常用來要麼指示組合語言程式做什麼要麼提示組合語言程式什麼。它們並不翻譯成機器代碼。指示符普遍的應用有：

- 定義常量
- 定義用來儲存資料的記憶體
- 將記憶體組合成段
- 有條件地包含源代碼
- 包含其他檔案

NASM代碼像C一樣要通過一個預處理程式。它擁有許多和C一樣的預處理程式。但是， NASM 的預處理的指示符以%開頭而不是像C一樣以#開頭。

1.3.5.1 equ 指示符

equ指示符可以用來定義一個符號。符號被命名為可以在組合語言程式裏使用的常量。格式是：

```
symbol equ value
```

符號的值以後不可以再定義。

1.3.5.2 %define 指示符

這個指示符和C中的#define非常相似。它通常用來定義一個宏常量，像在C裏面一樣。

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

上面的代碼定義了一個稱為 SIZE的宏通過使用一個MOV指令。巨集在兩個方面比符號要靈活。宏可以被再次定義而且可以定義比簡單的常量數值更大的值。

單位	字母
位元組	B
字	W
雙字	D
四字	Q
十個位元組	T

表 1.3: RESX和DX指示符的字母

1.3.5.3 數據指示符

資料指示符使用在資料段中用來定義記憶體空間。保留記憶體有兩種方法。第一種方法僅僅為資料定義空間；第二種方法在定義資料空間的同時給與一個初始值。第一種方法使用RESX指示符中的一個。X可由字母替代，字母由需要儲存的物件的大小來決定。表 1.3給出了可能的值。

第二種方法(同時定義一個初始值)使用DX指示符中的一個。X可以由字母替代，字母的值與RESX裏的值一樣。

使用變數 來標記記憶體位置是非常普遍的。變數使得在代碼中指向記憶體位置變得容易。下面是幾個例子：

```
L1    db    0          ;位元組變數L1，初始值為0
L2    dw    1000       ;字變數L2，初始值為1000
L3    db    110101b    ;位元組變數初始化成110101(十進位為53)
L4    db    12h        ;位元組變數初始化成十六進位12(在十進位中為18)
L5    db    17o        ;位元組變數初始化成八進制17(在十進位中為15)
L6    dd    1A92h      ;雙字變數初始化成十六進位1A92
L7    resb  1          ;1個未初始化的位元組
L8    db    "A"        ;位元組變數初始化成ASCII值A(65)
```

雙引號和單引號被同等對待。連續定義的資料儲存在連續的記憶體中。也就是說，字L2就儲存在L1的後面。記憶體的順序可以同樣被定義。

```
L9    db    0, 1, 2, 3              ; 定義4個位元組
L10   db    "w", "o", "r", 'd', 0   ; 定義一個等於"word"的C字串
L11   db    'word', 0               ; 等同於L10
```

指示符DD可以用來定義整形和單精確度的浮點數常量[6]。但是，DQ指示符僅僅可以用來定義雙精度的數常量。

對於大的序列， NASM 的TIMES 指示符常常非常有用。這個指示符每次都重複它的操作物件一個指定的次數。例如：

```
L12   times 100 db 0                    ; 等價於100個值為0的位元組
L13   resw   100                        ; 儲存空間為100個字
```

記住變數可以用來表示代碼中的資料。變數的使用方法有兩種。如果一個平常的變數被使用了，它被解釋為資料的位址(或偏移)。如果變數被放置在方括號([])中，它就被解釋為在這個位址中的資料。換句話說，你必須把變數當作一個指向資料的指標而方括號引用這個指標就像*號在C中一樣。(MASM/TASM使用的是另外一個慣例。)在32位元模式下，位址是32位。這兒有幾個例子：

```
1         mov    al, [L1]       ; 複製L1裏的位元組資料到AL
2         mov    eax, L1        ; EAX = 位元組變數L1代表的位址
3         mov    [L1], ah       ; 把AH拷貝到位元組變數L1
4         mov    eax, [L6]      ; 複製L6裏的雙字資料到 EAX
5         add    eax, [L6]      ; EAX = EAX + L6裏的雙字數據
6         add    [L6], eax      ; L6 = L6裏的雙字數據 + EAX
7         mov    al, [L6]       ; 拷貝L6裏的資料的第一個位元組到AL
```

例子的第7行展示了 NASM 一個重要性能。組合語言程式並不 保持跟蹤變數的資料類型。它由程式師來決定來保證他(或她)正確使用了一個變數。隨後它一般將資料的位址儲存到寄存器中，然後像在C中一樣把寄存器當一個指標變數來使用。同樣，沒有檢查使得指標能正確使用。以這種方式，組合語言程式跟C相比有更易出錯的傾向。

考慮下面的指令：

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

這條語句產生一個operation size not specified(操作大小沒有指定)的錯誤。為什麼？因為組合語言程式不知道是把1當作一個位元組，還是字，或是雙字來儲存。為了修正這個，加一個大小指定：

[6] 單精確度浮點數等價於C裏的的float變數

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

這個告訴組合語言程式把1儲存在從L6開始的雙字中。另一些大小指定為：BYTE(位元組)， WORD(字)， QWORD(四字)和TWORD(十位元組)。[7]

1.3.6 輸入和輸出

輸入和輸出是真正系統依賴的活力。它牽涉到系統硬體的介面問題。高階語言，像C，提供了標準的，簡單的，統一的程式I/O介面的程式庫。組合語言不提供標準庫。它們必須要麼直接訪問硬體(在保護模式下為特權級操作)或使用任何作業系統提供的底層的程式。

組合語言程式與C交互使用是非常普遍的。這樣做的一個優點是彙編代碼可以使用標準C I/O程式庫。但是，你必須清楚C使用的程式之間傳遞資訊的規則。這些規則放在這會非常麻煩。(它們將在以後提到！)為了簡單化I/O，作者已經開發出了隱藏在複雜C規則裏的自己的程式，而且提供了一個更簡單的介面。表 1.4描述了提供的程式。所有這些程式保留了所有寄存器的值，除了讀的程式外。這些程式確實修改了 EAX 的值。為了使用這些程式，你必須包含一個組合語言程式需要用到的資訊的檔。為了在 NASM 中包含一個檔，你可以使用%include預處理指示符。下面幾行包含了有作者的I/O程式的檔[8]:

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

為了使用一個列印程式，你必須載入正確的值到 EAX 中，然後用CALL指令調用它。CALL指令等價於在高階語言裏的函式call。它跳轉到代碼的另一段去執行，然後等程式執行完成後又回到原始的地方。下面的程式例子展示了調用這些I/O程式的幾個樣例。

1.3.7 調試

作者的庫同樣包含一些有用的調試程式。這些調試程式顯示關於系統狀態的資訊但不改變它們。這些程式是一些保存CPU的當前狀態後執行一

[7] TWORD定義了十個位元組大小的記憶體。浮點數輔助運算器使用了這種類型的資料

[8] asm_io.inc (和asm_io.inc需要的asm_io目標檔)在例子代碼中，可以從這個指南的網頁中下載到：http://www.drpaulcarter.com/pcasm

print_int	在螢幕上顯示出儲存在 EAX 中的整形值
print_char	在螢幕上顯示出以ASCII形式儲存在AL中的字元
print_string	在螢幕上顯示儲存在 EAX 裏的位址指向的字串的內容。這個字串必須是C類型的字串，(也就是:以null結束的字串)。
print_nl	在螢幕上顯示換行。
read_int	從鍵盤上讀入一整形資料然而把它儲存到 EAX 寄存器。
read_char	從鍵盤讀入一單個字元然而把它以ASCII形式儲存到 EAX 寄存器。

表 1.4: 彙編的I/O程式

個子程式調用的巨集。這些巨集定義在上面提到的`asm_io.inc`檔案中。巨集可以像普通的指令一樣使用。宏的運算元由逗號隔開。

這兒有四個調試程式稱為`dump_regs`，`dump_mem`，`dump_stack`和`dump_math`；它們分別顯示寄存器，記憶體，堆疊和數位輔助運算器的值。

dump_regs 這個巨集顯示系統的寄存器裏的值(十六進位)到`stdout`(也就是：顯示器)。它同時顯示在FLAGS[9]寄存器裏的位。例如，如果零標誌位元是1，ZF是顯示的。如果是0，它就不被顯示。它攜帶一個整形參數，這個參數同樣被顯示出來。這就可以用來區別不同`dump_regs`命令的輸出。

dump_mem 這個巨集同樣以ASCII字元的形式顯示記憶體區域的值(十六進位)。它帶有三個用逗號分開的參數。第一個參數是一個用來標示輸出的整形變數(就像`dump_regs`參數一樣)。第二個參數需要顯示的記憶體的位址。(它可以是個標號。)最後一個參數是在此位址後需要顯示的16位元組的節數。記憶體顯示將從要求的位址之前的第一節的邊界開始。

dump_stack 這個巨集顯示CPU堆疊的值。(這個堆疊將在第4章中提到。)這個堆疊由雙字組成，所以這個程式也以這種格式顯示它們。它帶有三個用逗號隔開的參數。第一個參數是一個整形變數

[9]第2章 將討論這個寄存器

（像dump_regs一樣）第二個參數是在EBP寄存器裏的位址下面需要顯示的雙字的數目而第三個參數是在EBP寄存器裏的位址上面需要顯示的的數目。

dump_math 這個巨集顯示數位輔助運算器寄存器裏的值。它只帶有一個整形參數，這個參數用來標示輸出就像參數dump_regs做的一樣。

第四节 創建一個程式

現今，完全用組合語言寫的獨立的程式是不經常的。彙編一般用在某些至關重要的程式。為什麼？用高階語言來編程比用彙編要簡單得多。同樣，使用彙編將使得程式移植到另一個平臺非常困難。事實上，根本很少使用組合語言程式。

那麼，為什麼任何人都需要學習組合語言程式呢？

1. 有時，用編程寫的代碼比起編譯器產生的代碼要少而且運行得更快。
2. 彙編允許直接訪問系統硬體資訊，而這個在高階語言中很難或根本不可能實現。
3. 學習彙編編程將幫助一個人深刻地理解系統如何運行。
4. 學習彙編編程將幫助一個人更好地理解編譯器和高階語言像C如何工作。

這兩點表明學習彙編是非常有用的，即使在以後的日子裏不在編程裏用到它。事實上，作者很少用彙編編程，但是他每天都使用來自它的想法。

1.4.1 第一個程式

在這一節裏的初期的程式將全部從圖 1.6裏的簡單C驅動程式開始。它簡單地調用另一個稱為asm_main的函式。這個是真正意義將用彙編編寫的程式。使用C驅動程式有幾個優點。首先，這樣使C系統正確配置程式在保護模式下運行。所有的段和它們相關的段寄存器將由C初始化。彙編代碼不需要為這個擔心。其次，C庫同樣提供給彙編代碼使用。作者的I/O程式利

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

图 1.6: driver.c代碼

用了這個優點。他們使用了C的I/O函式(printf， 等)。下面顯示了一個簡單的組合語言程式。

first.asm

```
1  ; 檔案: first.asm
2  ; 第一個組合語言程式。這個程式總共需要兩個整形變數作為輸入然後輸出它們的和。
3  ;
4  ;
5  ; 利用 djgpp 創建執行檔:
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 ; 初始化放入到資料段裏的數據
12 ;
13 segment .data
14 ;
15 ; 這些變數指向用來輸出的字串
16 ;
17 prompt1 db    "Enter a number: ", 0       ; 不要忘記空結束符
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段裏的數據
25  ;
26  segment .bss
27  ;
28  ; 這個變數指向用來儲存輸入的雙字
29  ;
30  input1  resd 1
31  input2  resd 1
32
33  ;
34  ; 代碼放入到.text段
35  ;
36  segment .text
37          global  _asm_main
38  _asm_main:
39          enter   0,0               ; 開始運行
40          pusha
41
42          mov     eax, prompt1      ; 輸出提示
43          call    print_string
44
45          call    read_int          ; 讀整形變數儲存到input1
46          mov     [input1], eax     ;
47
48          mov     eax, prompt2      ; 輸出提示
49          call    print_string
50
51          call    read_int          ; 讀整形變數儲存到input2
52          mov     [input2], eax     ;
53
54          mov     eax, [input1]     ; eax = 在input1裏的雙字
```

```
55          add     eax, [input2]      ; eax = eax + 在input2裏的雙字
56          mov     ebx, eax           ; ebx = eax
57
58          dump_regs 1                 ; 輸出寄存器值
59          dump_mem  2, outmsg1, 1     ; 輸出記憶體
60  ;
61  ; 下面分幾步輸出結果資訊
62  ;
63          mov     eax, outmsg1
64          call    print_string       ; 輸出第一條資訊
65          mov     eax, [input1]
66          call    print_int          ; 輸出input1
67          mov     eax, outmsg2
68          call    print_string       ; 輸出第二條資訊
69          mov     eax, [input2]
70          call    print_int          ; 輸出input2
71          mov     eax, outmsg3
72          call    print_string       ; 輸出第三條資訊
73          mov     eax, ebx
74          call    print_int          ; 輸出總數(ebx)
75          call    print_nl           ; 換行
76
77          popa
78          mov     eax, 0             ; 回到C中
79          leave
80          ret
```

first.asm

這個程式的第13行定義了指定儲存資料的記憶體段的部分代碼(名稱為.data)。只有是初始化的資料才需定義在這個段中。行17到20，聲明了幾個字串。它們將通過C庫輸出，所以必須以null字元(ASCII值為0)結束。記住0和'0'有很大的區別。

不初始化的數據需聲明在 bss 段(名為.bss，在26行)。這個段的名字來自於早期的基於UNIX彙編運算符，意思是"由符號開始的塊。"這同樣會有

一個堆疊段。它將在以後討論。

代碼段根據慣例被命名為.text。它是放置指令的地方。注意主程序(38行)的代碼標號有一個下劃線首碼。這個是在C中稱為約定的一部分。這個約定指定了編譯代碼時C使用的規則。C和彙編交互使用時，知道這個約定是非常重要的。以後將會將全部約定呈現；但是，現在你只需要知道所有的C編譯器裏的C符號(也就是： 函式和總體變數)有一個附加的下劃線首碼。(這個規定是為DOS/Windows指定的，在linux下C 編譯器並不為C符號名上加任何東西。)

在37行的總體變數(global)指示符告訴彙編定義_asm_main為總體變數。與C不同的是，變數在缺省情況下只能使用在內部範圍中。這就意味著只有在同一模組的代碼才能使用這個變數。 global指示符使指定的變數可以使用在外部範圍中。這種類型的變數可以被程式裏的任意模組訪問。asm_io模組聲明了總體變數print_int和et.al.。這就是為什麼在first.asm模組裏能使用它們的緣故。

1.4.2 編譯器依賴

上面的彙編代碼指定為基於GNU[10]的 DJGPP C/C++編譯器。[11] 這個編譯器可以從Internet上免費下載。它要求一個基於386或更好的PC而且需在DOS， Windows 95/98 或NT下運行。這個編譯器使用 COFF (Common Object File Format，普通目標檔格式)格式的目標檔。為了符合這個格式，nasm命令使用-f coff選項(就像上面代碼注釋展示的一樣)。最終目標檔的副檔名為o。

Linux C編譯器同樣是一個GNU編譯器。 為了轉變上面的代碼使它能在Linux下運行，只需簡單將37到38行裏的下劃線首碼移除。Linux使用ELF(Executable and Linkable Format，可執行和可連接格式)格式的目標檔。Linux下使用-f elf選項。它同樣產生一個副檔名為o的目標檔。

Borland C/C++ 是另一個流行的編譯器。它使用微軟 OMF 格式的目標檔。Borland編譯器使用-f obj選項。目標檔的副檔名將會是obj。 OMF與其他目標檔格式相比使用了不同的段指示符。資料段（13行）必須改成：

指定編譯器的例子檔可以從作者的網址上得到，它已經修改成能在恰當的編譯器上運行了。

[10]GNU是一個以免費軟體為基礎的計畫(http://www.fsf.org)

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

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

bss 段（26）必須改成：

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

text 段（36）必須改成：

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

必須在36行之前加上一新行：

```
group DGROUP _BSS _DATA
```

微軟C/C++編譯器可以使用 OMF 或Win32格式的目標檔。(如果給出的是OMF格式，它將從內部把資訊轉變成Win32格式。)Win32允許像DJGPP和Linux一樣來定義段。在這個模式下使用`-f win32`選項來輸出。目標檔的副檔名將會是`obj`。

1.4.3 彙編代碼

第一步是彙編代碼。在命令行，鍵入：

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

object-format要麼是coff， elf， obj，要麼是win32，它由使用的C編譯器決定。(記住在Linux和Borland下，資源檔案同樣必須改變。)

1.4.4 編譯C代碼

使用C編譯器編譯`driver.c`檔案。對於 DJGPP ，使用：

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

`-c`選項意味著編譯，而不是試圖現在連接。同樣的選項能使用在Linux，Borland和Microsoft編譯器上。

1.4.5 連接目標檔

連接是一個將在目標檔和庫檔裏的機器代碼和資料結合到一起產生一個可執行檔的過程。就像下面將展示的，這個過程是非常複雜的。

C代碼要求運行標準C庫和特殊的啟動代碼 。與直接調用連接程式相比，C編譯器更容易調用帶幾個正確的參數的連接程式。例如：使用DJGPP 來連接第一個程式的代碼，使用:

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

這樣產生一個`first.exe`(或在Linux下只是`first`)可執行檔。

對於Borland，你需要使用：

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

Borland使用列出的第一個檔案名來確定可執行檔案名。所以在上面的例子裏，程式將被命名為`first.exe`。

將編譯和連接兩個步驟結合起來是可能的。例如：

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

現在gcc將編譯`driver.c`然後連接。

1.4.6 理解一個彙編列表檔

`-l listing-file`選項可以用來告訴nasm創建一個指定名字的列表檔。這個檔將顯示代碼如何被彙編。這兒顯示了17和18行(在資料段)在列表檔中如何顯示。(行號顯示在列表檔中；但是注意在源代碼檔中顯示的行號可能不同於在列表檔中顯示的行號。)

```
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
```

每一行的頭一列是行號，第二列是資料在段裏的偏移位址(十六進位顯示)。第三列顯示將要儲存的十六進位值。這種情況下，十六進位資料符

合ASCII編碼。最終，顯示來自資源檔案的正文。列在第二行的偏移位址非常可能不是資料存放在完成後的程式中的真實偏移位址。每個模組可能在資料段(或其他段)定義它自己的變數。在連接這一步時(小節 1.4.5)，所有這些資料段的變數定義結合形成一個資料段。最終的偏移由連接程式計算得到。

這兒有一小部分text段代碼(資源檔案中54到56行)在列表檔中如何顯示：

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

第三列顯示了由組合語言程式產生的機器代碼。通常一個指令的完整代碼不能完全計算出來。例如：在94行，input1的偏移(位址)要直到代碼連接後才能知道。組合語言程式可以算出mov 指令(在列表中為A1)的操作碼，但是它把偏移寫在方括號中，因為準確的值還不能算出來。這種情況下，0作為一個暫時偏移被使用，因為input1在這個檔中，被定義在bss段的開始。記住這不意味著它會在程式的最終 bss 段的開始。當代碼連接後，連接程式將在位置上插入正確的偏移。其他指令，如96行，並不涉及任何變數。這兒組合語言程式可以算出完整的機器代碼。

1.4.6.1 Big和Little Endian 表示法

如果有人仔細看過95行，將會發現機器代碼中的方括號裏的偏移地址非常奇怪。input2變數的偏移位址為4(像檔定義的一樣)；但是顯示在記憶體裏偏移不是00000004，而是04000000。為什麼？不同的處理器在記憶體裏以不同的順序儲存多位元組整形：big endian和little endian。Big endian是一種看起來更自然的方法。最大(也就是：最高有效位元)的位元組首先被儲存，然後才是第二大的，依此類推。例如：雙字00000004將被儲存為四個位元組00 00 00 04。IBM主機，許多RISC 處理器和Motorola處理器都使用這種big endian方法。然而，基於Intel的處理器使用little endian方法！首先被儲存是最小的有效位元組。所以00000004在記憶體中儲存為04 00 00 00。這種格式強制連入CPU而且不可能更改。通常情況下，程式師並不需要擔心使用的是哪種格式。但是，在下面的情況下，它們是非常重要的。

Endian的發音和indian一樣。

1. 當二進位資料在不同的電腦上傳輸時(不管來自檔還是網路)。
2. 當二進位資料作為一個多位元組整形寫入到記憶體中然後當作單個單個位元組讀出，反之亦然。

Endian格式並不應用於陣列的排序。陣列的第一個元素通常在最低的位址裏。這個應用在字串裏(字元陣列)。 Endian格式依然用在陣列的單個元素中。

第五节 骨架檔案

圖 1.7顯示了一個可以用來書寫組合語言程式的開始部分的骨架檔。

skel.asm

```
1  %include "asm_io.inc"
2  segment .data
3  ;
4  ; 初始化數據放入到這兒的資料段中
5  ;
6
7  segment .bss
8  ;
9  ; 未初始化的數據放入到 bss 段中
10 ;
11
12 segment .text
13         global  _asm_main
14 _asm_main:
15         enter   0,0               ; 開始運行
16         pusha
17
18 ;
19 ; 代碼放在text段。不要改變在這個注釋之前或之後的代碼。
20 ;
21 ;
22
23         popa
24         mov     eax, 0            ; 返回到以中
25         leave
26         ret
```

skel.asm

图 1.7: 骨架程式

第二章　基本組合語言

第一节　整形工作方式

2.1.1　整形表示法

整形有兩種類型:有符號和無符號。無符號整形(即此類型沒有負數)以一種非常直接的二進位方式來表示。數字200作為一個 無符號整形數將被表示為11001000(或十六進位C8)。

有符號整形（即此類型可能為正數也可能為負數）以一種更複雜的方式來表示。例如，考慮 −56。+56當作一個位元組來考慮時將被表示為00111000。在紙上，你可以將−56表示為−111000，但是在電腦記憶體中如何以一個位元組來表示，如何儲存這個負號呢？

有三種普遍的技術被用來在電腦記憶體中表示有符號整形。所有的方法都把整形的最大有效位元當作一個符號位元來使用。如果數為正數，則這一位為0；為負數，這一位就為1。

2.1.1.1　原碼

第一種方法是最簡單的，它被稱為原碼。它用兩部分來表示一個整形。第一部分是符號位元，第二部分是整形的原碼。所以56表示成位元組形式為00111000 (符號位元加了下劃線)而−56將表示為 10111000。最大的一個位元組的值將是 01111111或+127，而最小的一個位元組的值將是 11111111或−127。要得到一個數的相反數，只需要將符號位元變反。 這個方法很簡單直接，但是它有它的缺點。首先，0會有兩個可能的值：+0 (00000000) 和 −0 (10000000)。因為0不是正數，也不是負數，所以這些表示法都應該把它表示成一樣。這樣會把CPU的運算邏輯弄得很複雜 。第二，普通的運算同樣 是麻煩的。如果10加−56，這個將改變為10減去56。

同樣，這將會複雜化CPU的邏輯。

2.1.1.2 反碼

第二種方法稱為反碼表示法。一個數的反碼可以通過將這個數的每一位求反得到。(另外一個得到的方法是：新的位值等於1−老的位值。) 例如： 00111000 (+56)的反碼是11000111。在反碼表示法中，計算一個數的反碼等價於求反。因此， −56就可以表示為11000111。注意，符號位元在反碼中是自動改變的，你需要兩次求反碼來得到原始的數值。就像第一種方法一樣，0有兩種表示： 00000000 (+0)和11111111 (−0)。用反碼表示的數值的運算同樣是麻煩的。

這有一個小訣竅來得到一個十六進位數值的反碼，而不需要將它轉換成二進位。這個訣竅就是用F（或十進位中的15）減去每一個十六進位位。這個方法假定數中的每一位的數值是由4位元二進位組成的。這有一個例子：+56 用十六進位表示為38。要得到反碼，用F減去每一位，得到C7。這個結果是與上面的結果吻合的。

2.1.1.3 補數

前面描述的兩個方法用在早期的電腦中。現代的電腦使用第三種方法稱為補數表示法。一個數的補數可以由下面兩步得到：

1. 找到該數的反碼
2. 將第一步的結果加1

這有一個使用00111000 (56)的例子。首先，經計算得到反碼：11000111 。然後加1：

$$\begin{array}{rr} & 11000111 \\ + & \qquad 1 \\ \hline & 11001000 \end{array}$$

在補數表示法中，計算一個補數等價於對一個數求反。因此，11001000是−56的補數。要得到原始數值需兩次求反。令人驚訝的是補數不符合這個

數值	十六進位表示
0	00
1	01
127	7F
-128	80
-127	81
-2	FE
-1	FF

表 2.1: 補數表示法

規定。通過對11001000的反碼加1得到補數 。

$$\begin{array}{rr} & 00110111 \\ + & \qquad 1 \\ \hline & 00111000 \end{array}$$

當在兩個補數運算元間進行加法操作時，最左邊的位相加可能會產生一個進位。這個進位是不被使用的。記住在電腦中的所有資料都是有固定大小的(根據位元數多少)。兩個位元組相加通常得到一個位元組的結果 (就像兩個字相加得到一個字，等。) 這個特性對於補數表示法來說是非常重要的。例如，把0作為一個位元組來考慮它的補數形式(00000000)。計算它的補數形式得到總數：

$$\begin{array}{rr} & 11111111 \\ + & \qquad 1 \\ \hline c & 00000000 \end{array}$$

其中c代表一個進位。(稍後將展示如何偵查到這個進位，但是它在這的結果中不儲存。)因此，在補數表示法中0只有一種表示。這就使得補數的運算比前面的方法簡單。

使用補數表示法，一個有符號的位元組可以用來代表從-128 到$+127$的數值。表 2.1 展示一些可選的值。如果使用了16位，那麼可以表示從$-32,768$
到$+32,767$的有符號數值。$+32,767$可以表示為7FFF， $-32,768$ 為8000, -128為FF80而-1為FFFF。32位的補數大約可以表示-20億到$+20$億的數值範圍。

CPU對某一的位元組(或字，雙字)具體表示多少 並不是很清楚。組合語言並沒有類型的概念，而高階語言有。資料解釋成什麼取決於使用在這個資料上的指令。到底十六進位數FF被看成一個有符號數 −1 還是無符號數+255取決於程式師。C語言定義了有符號和無符號整形。這就使C編譯器能決定使用正確的指令來使用資料。

2.1.2 正負號延伸

在組合語言中，所有資料都有一個指定的大小。為了與其他資料一起使用而改變資料大小是不常用的。減小它的大小是最簡單的。

2.1.2.1 減小數據的大小

要減小資料的大小，只需要簡單地將多餘的有效位移位即可。這是一個普通的例子：

```
mov     ax, 0034h       ; ax = 52 (以十六位元儲存)
mov     cl, al          ; cl = ax的低八位
```

當然，如果數字不能以更小的大小來正確描述，那麼減小資料的大小將不能工作。例如，如果AX是 0134h (或十進位的308) ，那麼上面的代碼仍然將 CL置為34h。這種方法對於有符號和無符號數都能工作。考慮有符號數：如果AX是FFFFh (也就是−1)，那麼CL 將會是FFh (一個位元組表示的−1)。然而，注意如果在AX裏的值是無符號的，這個就不正確了！

無符號數的規則是：為了能轉換正確，所有需要移除的位元都必須是0。有符號數的規則是：需要移除的位元必須要麼都是1,要麼都是0。另外，沒有移除的第一個比特位的值必須等於移除的位的第一位。這一位將會是變小的值的新的符號位元。這一位元與原始符號位元相同是非常重要的！

2.1.2.2 增大數據的大小

增大資料的大小比減小資料的大小更複雜。考慮十六進位位元組：FF。如果它擴展成一個字，那麼這個字的值應該是多少呢？它取決於如何解釋FF。如果FF是一個無符號位元組(十進位中為)，那麼這個字

就應該是00FF；但是，如果它是一個有符號位元組(十進位中為−1)，那麼這個字就應該為 FFFF。

一般說來，擴展一個無符號數，你需將所有的新位置為0.因此，FF就變成了00FF。但是，擴展一個有符號數，你必須擴展符號位元。 這就意味著所有的新位元通過複製符號位元得到。因為FF的符號位元為1，所以新的位必須全為1，從而得到FFFF。如果有符號數5A (十進位中為90)被擴展了，那麼結果應該是005A。

80386提供了好幾條指令用於數的擴展。謹記電腦是不清楚一個數是有符號的或是無符號的。這取決於程式師是否用了正確的指令。

對於無符號數，你可以使用MOV指令簡單地將高位置0。例如，將一個在AL中的無符號位元組擴展到AX中：

```
mov     ah, 0    ; 輸出高8位為0
```

但是，使用MOV指令把一個在AX中的無符號字轉換成在EAX中的無符號雙字是不可能的。為什麼不可以呢？因為在MOV指令中沒有方法指定EAX的高16位。80386通過提供一個新的指令MOVZX來解決這個問題。 這個指令有兩個運算元。目的運算元 (第一個運算元)必須是一個16或32位的寄存器。源運算元(第二個運算元)可以是一個8或16位元的寄存器或記憶體中的一個字。另一個限制是目的運算元必須大於源運算元。(許多指令要求源和目的運算元必須是一樣的大小。) 這兒有幾個例子：

```
movzx   eax, ax       ; 將ax擴展成eax
movzx   eax, al       ; 將al擴展成eax
movzx   ax, al        ; 將al擴展成ax
movzx   ebx, ax       ; 將ax擴展成ebx
```

對於有符號數，在任何情況下，沒有一個簡單的方法來使用MOV 指令。8086提供了幾條用來擴展有符號數的指令。CBW (Convert Byte to Word(位元組轉換成字))指令將AL正負號擴展成AX。運算元是不顯示的。CWD (Convert Word to Double word(字轉換成雙字))指令將AX正負號擴展成DX:AX。

DX:AX表示法表示將DX和AX寄存器當作一個32位寄存器來看待，其中高16位在DX中，低16位在AX中。(記住8086沒有32位寄存器！) 80386加了好幾條新的指令。CWDE (Convert Word to Double word Extended(字轉換成

```
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 ) {
  /* 對ch做一些事情 */
}
```

图 2.2:

擴展的雙字))指令將AX正負號擴展成EAX。CDQ (Convert Double word to Quad word(雙字擴展成四字))指令將EAX正負號擴展成EDX:EAX (64位!). 最後， MOVSX 指令像MOVZX指令一樣工作，除了它使用有符號數的規則外。

2.1.2.3 C編程中的應用

ANSI C並沒有定義char類型是有符號的，還是無符號的，它取決於各個編譯器的決定。這就是為什麼在圖 2.1中明確指定類型的原因。

無符號和有符號數的擴展同樣發生在C中。 C中的變數可以被聲明成有符號的，或無符號的(int是有符號的)。考慮在圖 2.1中的代碼。在第3行中，變數a使用了無符號數的規則(使用MOVZX)進行了擴展，但是在第4行，變數b使用了有符號數的規則(使用 MOVSX)進行了擴展。

這有一個直接與這個主題相關的普遍的C編程中的一個bug。考慮在圖 2.2中的代碼。 fgetc()的原型是:

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

一個可能的問題:為什麼這個函式返回一個int類型，然後又被當作字元類型被讀呢？ 原因是它一般確實是返回一個char 類型的值(使用0擴展成一個int類型的值)。但是，有一個值它可能不會以字元的形式返回：EOF。這是一個宏，通常被定義為 −1。因此，fgetc()不是返回一個通過擴展成int類型得到的char類型的值 (在十六進位中表示為000000xx)，就是EOF (在十六進位中表示為FFFFFFFF)。

圖 2.2中的程式的基本的問題是 fgetc()返回一個int類型，但是這個值以char的形式儲存。C將會切去較高順序的位元來使int類型的值適合char類型。唯一的問題是數 (十六進位) 000000FF和FFFFFFFF都會被切成位元組FF。因此，while迴圈不能區別從檔中讀到的位元組FF和檔的結束。

實際上，在這種情況下代碼會怎麼做，取決於char是有符號的，還是無符號的。為什麼？因為在第2行ch是與 EOF進行比較。 因為EOF是一個int類型的值[1]，ch將會擴展成一個int類型，以便於這兩個值在相同大小下比較[2]。 就像圖 2.1展示的一樣，變數是有符號的還是無符號的是非常重要的。

如果char是無符號的，那麼FF就被擴展成 000000FF。這個拿去與EOF (FFFFFFFF)比較，它們並不相等。因此，迴圈不會結束。

如果char是有符號的，FF就被擴展成FFFFFFFF。這就導致比較相等，迴圈結束。但是，因為位元組FF可能是從檔中讀到的，迴圈就可能過早地被結束了。

這個問題的解決辦法是定義ch 變數為 int類型，而不是 char類型。當做了這個改變，在第2行就不會有切去和擴展操作執行了。在迴圈休內，對值進行切去操作是很安全的，因為ch在這兒 必須實際上已經是一個簡單的位元組了。

2.1.3 補數運算

就像我們早些時候看到的，add指令執行加法操作，而sub指令的執行減法操作。在FLAGS寄存器中的兩位能被這些指令設置，它們是：overflow(溢出位) 和carry flag(進位元標誌位元)。如果操作的正確結果太大了以致於不匹配有符號數運算的目的運算元，溢出標誌位元將被置位元。如果在加法中的最高有效位有一個進位或在減法中的最高有效位有一個借位，進位元標誌位元將被置位元。因此，它可以用來檢查無符號數運算的溢出。進位元標誌位元在有符號數運算中的使用將看起來非常簡單。補數的一個最大的優點是加法和減法的規則實際上就與無符號整形的

[1]這是一個普通的誤解：在檔的最後有一個EOF字元。 這是 不 正確的!
[2]對於這個要求的原因將在下面介紹。

一樣。因此， add和sub可以同時被用在有符號和無符號整形上。

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

這兒有一個進位產生，但是它不是結果的一部分。

這有兩個不同的乘法和除法指令。首先，使用MUL或IMUL 指令來進行乘法運算。 MUL指令用於無符號數之間相乘，而 IMUL指令用於有符號數之間相乘。為什麼需要兩個不同的指令呢？無符號數和有符號數補數的乘法規則是不同的。為什麼會這樣？考慮位元組FF乘以它本身產生一個字的結果。使用無符號乘法這就是255乘上255，得65025 (或十六進位的FE01)。使用有符號數乘法這就是 -1 乘上 -1，得1 (或十六進位的0001)。

這兒有乘法指令的幾種格式。最老的格式是像這樣的：

```
mul   source
```

source要麼是一個寄存器，要麼是一個指定的記憶體。它不可以是一個立即數。實際上，乘法怎麼執行取決於源運算元的大小。如果運算元大小是一個位元組，它乘以在AL寄存器中的位元組，而結果被儲存到了16位元寄存器AX中。如果源運算元是16位，它乘以在AX中的字，而32位元的結果被儲存到了DX:AX。如果源運算元是32位的，它乘以在EAX中的數，而結果被儲存到了EDX:EAX。

IMUL指令擁有與MUL指令相同的格式，但是同樣增加了其他一些指令格式。這有兩個和三個運算元的格式：

```
imul   dest (目的運算元), source1(源運算元1)
imul   dest (目的運算元), source1(源運算元1), source2(源運算元2)
```

表 2.2展示可能的組合。

兩個除法運算符是DIV和IDIV。它們分別執行無符號整形和有符號整形的除法。普遍的格式是：

```
div   source
```

如果源運算元為8位，那麼AX就除以這個運算元。商儲存在AL中，而餘數儲存在AH中。如果源運算元為16位，那麼DX:AX就除以這個運算元。商儲

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

存在AX中，而餘數儲存在DX中。如果源運算元為32位，那麼 EDX:EAX就除以這個運算元，同時商儲存在EAX中，餘數儲存在EDX中。IDIV 指令以同樣的方法進行工作。這沒有像IMUL指令一樣的特殊的IDIV指令。 如果商太大了，以致於不匹配它的寄存器，或除數為0,那麼這個程式將被中斷和中止。一個普遍的錯誤是在進行除法之前忘記了初始化DX或EDX。

NEG 指令通過計算它的單一的運算元補數來得到這個運算元的相反數。它的運算元可以是任意的8位，16位或32位元寄存器或記憶體區域。

2.1.4 程式例子

math.asm

```
1 %include "asm_io.inc"
2 segment .data          ; 輸出字串
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               ; 開始運行程式
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          ; 保存結果到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                         ; 通過正負號延伸初始化edx
51          mov     ecx, 100            ; 不可以被立即數除
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                 ; 求這個餘數的相反數
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 擴充精度運算

組合語言同樣提供允許你執行大於雙字的數的加減法的指令。 這些指令使用了進位元標誌位元。就像上面規定的，ADD 和SUB 指令在進位元或借位產生時分別修改了進位元標誌位元。儲存在進位元標誌位元裏的資訊可以用來做大的數位的加減法，通過將這些運算元分成小的雙字(或更小)塊。

ADC 和SBB 指令使用了進位元標誌位元裏的資訊。ADC指令執行下麵的操作：

operand1 = operand1 + carry flag + operand2

SBB執行下麵的操作：

operand1 = operand1 - carry flag - operand2

這些如何使用？考慮在EDX:EAX和EBX:ECX中的64位整形的總數。下面的代碼將總數儲存到EDX:EAX中:

```
1   add     eax, ecx        ; 低32位相加
2   adc     edx, ebx        ; 高32位帶以前總數的進位相加
```

減法也是一樣的。下面的代碼用EDX:EAX減去EBX:ECX:

```
1   sub     eax, ecx        ; 低32位相減
2   sbb     edx, ebx        ; 高32位帶借位相減
```

對於實際上大的數字，可以使用一個迴圈(看 小節 二)。對於一個求和的迴圈，對於每一次重複(替代所有的，除了第一次重複)使用ADC指令將會非常便利。通過在迴圈開始之前使用CLC (CLear Carry(清除進位元))指令初始化進位元標誌位元為0,可以使這個操作正確執行。如果進位元標誌位元為0,那麼ADD和 ADC指令就沒有區別了。這個在減法中也是一樣的。

第二节 控制結構

高階語言提供高級的控制結構(例如， if和while語句)來控制執行的順序。組合語言並沒有提供像這樣的複雜控制結構。它使用聲名狼藉的goto來替代，如果使用不恰當可能會導致非常複雜的代碼。但是，它是能夠寫出結構化的組合語言程式。基本的步驟是使用熟悉的高階語言控制結構來設計程式的邏輯，然後將這個設計翻譯成恰當的組合語言(就像一個編譯器要做的一樣)。

2.2.1 比較

控制結構決定做什麼是基於資料的比較的。在組合語言中，比較的結果儲存在FLAGS寄存器中，以便以後使用。80x86提供CMP指令來執行比較操作。FLAGS寄存器根據CMP指令的兩個運算元的不同來設置。具體的操作是相減，然後FLAGS根據結果來設置，但是結果是不在任何地方儲存的。如果你需要結果，可以使用SUB來代替CMP指令。

對於無符號整形，有兩個標誌位元(在FLAGS寄存器裏的位) 是非常重要的：零標誌位元(zero flag(ZF)) 和進位元標誌位元(carry flag(CF)) 。 如果比較的結果是0的話，零標誌位元將置成(1) 。進位元標誌位元在減法中當作一個借位來使用。考慮這個比較：

```
cmp     vleft, vright
```

vleft - vright的差別被計算出來，然後相應地設置標誌位元。如果CMP執行後得到差別為0，即vleft = vright那麼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，那麼ZF就不被置位而且CF也不被置位(沒有借位)。如果 vleft < vright，那麼ZF就不被置位，而CF就被置位了(有借位)。

對於有符號整形，有三個標誌位元非常重要：零標誌位元(zero flag (ZF))，溢出標誌位元(overflow flag(OF))和符號標誌位元(sign flag (SF))。如果一個操作的結果上溢(下溢)，那麼溢出標誌位元將被置位元。如果一個操作的結果為負數，那麼符號標誌位元將被置位元。如果vleft = vright，那麼ZF將被置位元(正好與無符號整形一樣)。 如果vleft > vright，那麼ZF不被設置，而且 SF = OF。如果vleft < vright，那麼ZF不被設置而且SF ≠ OF。

如果vleft ¿ vright，為什麼SF = OF? 因為如果沒有溢出，那麼差別將是一個正確的值，而且肯定是非負的。因此，SF = OF = 0。但是，如果有溢出，那麼差別將不是一個正確的值(而且事實上將會是個負數)。因此，SF = OF = 1。

不要忘記其他的指令同樣會改變FLAGS寄存器，不僅僅CMP可以。

2.2.2 分支指令

分支指令可以將執行控制權轉移到一個程式的任意一點上。換言之，它們像goto一樣運作。有兩種類型的分支： 無條件的和有條件的。一個無條件的分支就跟goto一樣，它總會產生分支。一個有條件分支可能也可能不產生分支，它取決於在FLAGS寄存器裏的標誌位元。如果一個有條件分支沒有產生分支，控制權將傳遞到下一指令。

JMP (jump的簡稱)指令產生無條件分支。它唯一的參數通常是一個指向分支指向的指令的代碼標號。彙編器和連接器將用指令的正確位址來替代這個標號。這又是一個乏味的運算元，通過這個，彙編器使得程式師的日子不好過。能認識到在JMP指令後的指令不會被執行，除非另一條分支指令指向它，是非常重要的。

這兒有jump指令的幾個變更形式：

SHORT 這個跳轉類型局限在一小範圍內。它僅僅可以在記憶體中向上或向下移動128位元組。這個類型的好處是相對於其他的，它使用較少的記憶體。它使用一個有符號位元組來儲存跳轉的位移。位移表示向前或向後移動的位元組數(位移須加上EIP)。為了指定一個短跳轉，需在JMP指令裏的變數之前使用關鍵字SHORT。

NEAR 這個跳轉類型是無條件和有條件分支的缺省類型，它可以用來跳到一段中的任意地方。事實上，80386支援兩種類型的近跳轉。其中一個的位移使用兩個位元組。它就允許你向上或向下移動32,000個位元組。另一種類型的位移使用四個位元組，當然它就允許你移動到代碼段中的任意位置。四位元組類型是386保護模式的缺省類型。兩個位元組類型可以通過在JMP指令裏的變數之前放置關鍵字WORD來指定。

FAR 這個跳轉類型允許控制轉移到另一個代碼段。在386保護模式下，這種事情是非常鮮見的。

有效的代碼標號遵守與資料變數一樣的規則。代碼標號通過在代碼段裏把它們放在它們標記的聲明前面來定義它們。有一個冒號放在變數定義的地方的結尾處。這個冒號不是名字的一部分。

條件分支有許多不同的指令。它們都使用一個代碼標號作為它們唯一的運算元。最簡單的就是看FLAGS寄存器裏的一個標誌位元來決定是否要

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

表 2.3: 簡單條件分支

分支。看表 2.3得到關於這些指令的列表。(PF是奇偶標誌位元(parity flag)，它表示結果中的低8位1的位數值為奇數個或偶數個。)

下麵的虛擬碼:

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

可以寫成彙編形式，如：

```
1       cmp     eax, 0          ; 置標誌位元(如果eax - 0 = 0, ZF就被置位)
2       jz      thenblock       ; 如果ZF被置位了，就跳轉到thenblock
3       mov     ebx, 2          ; IF結構的ELSE部分
4       jmp     next            ; 跳過IF結構中的THEN部分
5   thenblock:
6       mov     ebx, 1          ; IF結構的THEN部分
7   next:
```

其他比較使用在表 2.3裏的條件分支並不是很容易。為了舉例說明，考慮下面的虛擬碼：

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

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

表 2.4: 有符號和無符號的比較指令

```
else
  EBX = 2;
```

如果EAX大於或等於5，ZF可能被置位或不置位，而SF將等於OF。這是測試這些條件的彙編代碼 (假定EAX是有符號的):

```
1        cmp     eax, 5
2        js      signon              ; 如果SF = 1，就跳轉到signon
3        jo      elseblock           ; 如果OF = 1而且SF = 0，就跳轉到elseblock
4        jmp     thenblock           ; 如果SF = 0而且OF = 0，就跳轉到thenblock
5  signon:
6        jo      thenblock           ; 如果SF = 1而且OF = 1，就跳轉到thenblock
7  elseblock:
8        mov     ebx, 2
9        jmp     next
10 thenblock:
11       mov     ebx, 1
12 next:
```

上面的代碼使用起來非常不便。幸運的是，80x86提供了額外的分支指令使這種類型的測試條件更容易些。每個版本都分為有符號和無符號兩種。表 2.4展示了這些指令。等於或不等於分支(JE和JNE)對於有符號和無符號整形是相同的。(事實上，JE和JZ， JNE和JNZ基本上完全相同 。) 每個其他的分支指令都有兩個同義字。例如：看JL (jump less than)和 JNGE

(jump not greater than or equal to)。有相同的指令這是因為：

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

無符號分支使用A代表大於而B代表小於，替換了L和G。

使用這些新的指令，上面的虛擬碼可以更容易地翻譯成組合語言：

```
1         cmp     eax, 5
2         jge     thenblock
3         mov     ebx, 2
4         jmp     next
5   thenblock:
6         mov     ebx, 1
7   next:
```

2.2.3 迴圈指令

80x86提供了幾條專門為實現像for一樣的迴圈而設計的指令。每一個這樣的指令帶有一個代碼標號作為它們唯一的運算元。

LOOP ECX自減，如果 ECX $\neq 0$，分支到代碼標號指向的位址

LOOPE, LOOPZ ECX自減(FLAGS寄存器沒有被修改)，如果 ECX $\neq 0$ 而且 ZF $= 1$，則分支

LOOPNE, LOOPNZ ECX自減(FLAGS沒有改變)，如果 ECX $\neq 0$ 而且 ZF $= 0$，則分支

最後兩個迴圈指令對於連續的查找迴圈是非常有用的。下麵的虛擬碼：

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

可以翻譯在組合語言，如:

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

第三节 翻譯標準的控制結構

這一小節講述在高階語言裏的標準控制結構如何應用到組合語言中。

2.3.1 If語句

下麵的虛擬碼:

if (條件)
  then_block;
else
  else_block ;

可以像這樣被應用:

```
1        ; 設置FLAGS的代碼
2        jxx    else_block    ; 選擇xx，如果條件為假，則分支
3        ; then模組的代碼
4        jmp    endif
5  else_block:
6        ; else模組的代碼
7  endif:
```

如果沒有else語句的話，那麼else_block分支可以用endif分支取代。

```
1        ; 設置FLAGS的代碼
2        jxx    endif          ; 選擇xx，如果條件為假，則分支
3        ; then模組的代碼
4  endif:
```

2.3.2 While迴圈

while迴圈是一個頂端測試迴圈：

while(條件) {
  循環體;
}

這個可以翻譯成：

```
1 while:
2       ; 基於條件的設置FLAGS的代碼
3       jxx     endwhile        ; 選擇xx，如果條件為假，則分支
4       ; 循環體
5       jmp     while
6 endwhile:
```

2.3.3 Do while迴圈

do while迴圈是一個末端測試迴圈：

do {
  循環體;
} **while**(條件);

這個可以翻譯成：

```
1 do:
2       ; 循環體
3       ; 基於條件的設置FLAGS的代碼
4       jxx     do              ; 選擇xx，如果條件為假，則分支
```

第四节 例子:查找素數

這一小節是一個查找素數的程式。根據回憶，素數是一個只能被1和它本身整除的數。沒有公式來做這件事情。這個程式使用的基本方法是在一

```
1  unsigned guess;    /* 當前對素數的猜測  */
2  unsigned factor;   /* 猜測數的可能的因數  */
3  unsigned limit;    /* 查找這個值以下的素數  */
4
5  printf("Find primes up to: ");
6  scanf("%u", &limit);
7  printf("2\n");     /* 把頭兩個素數當特殊的事件處理  */
8  printf("3\n");
9  guess = 5;         /* 初始化猜測數 */
10 while ( guess <= limit ) {
11   /* 查找一個猜測數的因數 */
12   factor = 3;
13   while ( factor*factor < guess &&
14           guess % factor != 0 )
15     factor += 2;
16   if ( guess % factor != 0 )
17     printf("%d\n", guess);
18   guess += 2;    /* 只考慮奇數 */
19 }
```

图 2.3:

個給定的範圍內查找所有奇數的因數[3]。 如果一個奇數沒有找到一個因數，那麼它就是一個素數。圖 2.3 展示了用C寫的基本的演算法。

這是組合語言版:

prime.asm

```
1  %include "asm_io.inc"
2  segment .data
3  Message         db      "Find primes up to: ", 0
4
5  segment .bss
6  Limit           resd    1               ; 查找這個值以下的素數
```

[3]2是唯一的偶數素數。

```
7  Guess           resd    1               ; 當前對素數的猜測
8
9  segment .text
10         global  _asm_main
11 _asm_main:
12         enter   0,0             ; 程式開始運行
13         pusha
14
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     ; 因為數位為無符號數, 所以使用jnbe
32
33         mov     ebx, 3              ; ebx等於factor = 3;
34 while_factor:
35         mov     eax,ebx
36         mul     eax                 ; edx:eax = eax*eax
37         jo      end_while_factor    ; 如果結果不匹配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

第三章　位元操作

第一节　移位元操作

組合語言允許程式師對資料的單個比特位元進行操作。一個最常見的的位元操作稱為移位元。移位元操作移動某一個資料的比特位元的位置。移位元可以是向左移(也就是：向最高有效位移動)，也可以向右移(最低有效位)。

3.1.1　邏輯移位元

邏輯移位元是移位元中最簡單的類型。它以一種最直接的方式進行移位元操作。圖 3.1展示了一個位元組的移位元操作的例子。

原始資料	1	1	1	0	1	0	1	0
向左移位元	1	1	0	1	0	1	0	0
向右移位元	0	1	1	1	0	1	0	1

图 3.1: 邏輯移位元

注意：新進來的比特位總是為0。SHL 和SHR 指令分別用來執行邏輯左移和邏輯右移。這些指令允許你移動任意的位數。位元數可以是一個常量，也可以是儲存在CL寄存器的值。最後從資料中移出的比特位元儲存在進位元標誌位元中。這有一些代碼例子：

```
1        mov    ax, 0C123H
2        shl    ax, 1            ; 向左移動一個比特位,    ax = 8246H, CF = 1
3        shr    ax, 1            ; 向右移動一個比特位,    ax = 4123H, CF = 0
4        shr    ax, 1            ; 向右移動一個比特位,    ax = 2091H, CF = 1
5        mov    ax, 0C123H
```

```
6       shl     ax, 2           ; 向左移動兩個比特位,    ax = 048CH, CF = 1
7       mov     cl, 3
8       shr     ax, cl          ; 向右移動三個比特位,    ax = 0091H, CF = 1
```

3.1.2 移位元的應用

快速的乘法和除法是移位元操作最普遍的應用。回憶在十進位系統中，乘以和除以10的幾次方是非常簡單的，只是移動位而已。在二進位中，乘以和除以2的幾次方也是一樣的。例如：要得到二進位數字1011_2(或十進位11)的兩倍，只需向左移動一位，得到10110_2 (或22)。一個除以2的幾次方的除法的商相當於一個右移操作的結果。僅僅是除以2，向右移動一位；除以4(2^2),向右移動2位；除以8(2^3，向右移動3位，等等。 移位元指令非常基礎而且比功能相同的MUL 和DIV 指令執行要快得多!

實際上，邏輯移位元只可以用於無符號數的乘法和除法。一般它們不能應用於有符號數。考慮兩個位元組的數值FFFF(有符號時為-1)。如果它向右邏輯移動一位元，結果是7FFF，也就是$+32,767$!另一種類型的移位元操作可以用在有符號數上。

3.1.3 算術移位元

這些移位元操作是為允許有符號數能快速地執行乘以和除以2的幾次方的操作而設計的。它們保證符號位能被正確對待。

SAL 算術左移(Shift Arithmetic Left) - 這條指令只是SHL的同義詞。它實際上被編譯成與SHL一樣的機器代碼。只要符號位元沒有因移位元而改變，結果就將是正確的。

SAR 算術右移(Shift Arithmetic Right) - 這是一條新的指令，它不會移動運算元的符號位元(也就是： 最高有效位)。其他位被正常移動，除了從左邊新進來的位元通過複製符號位元(也就是說，如果符號位元為1,那麼新的位值也同樣為1)得到 外。因此，如果一個位元組使用這條指令來移位元，只有低7位會被移動。就像其他移位元指令一樣，最後移出的位元儲存在進位元標誌位元中。

```
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 迴圈移位元

迴圈移位元指令像邏輯指令一樣運作，除了把從資料的一端移出的比特位又移入到另一端外。因此，資料好像被當作一個迴圈結構體一樣對待。ROL 和ROR 是兩個最簡單的迴圈移位元指令，它們分別執行左移和右移操作。就像其他移位元指令一樣，這些移位元指令把迴圈移出的最後一個比特位元複製到進位元標誌位元中。

```
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
```

有兩個額外的迴圈指令用來在資料和進位元標誌位元之間移動比特位，它們稱為RCL 和RCR。例如，如果AX寄存器用這些指令來移位元，那麼有17位用來得到AX，進位元標誌位元也包括在迴圈中。

```
1       mov     ax, 0C123H
2       clc                     ; 進位元標誌位元清零(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 簡單應用

這有一個代碼小片斷，它用來計算在EAX寄存器裏“on”(也就是：1)的比特位有多少個。

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

表 3.1: AND運算符

```
1       mov     bl, 0           ; bl將儲存ON的比特位元的總數
2       mov     ecx, 32         ; ecx是迴圈計數器
3 count_loop:
4       shl     eax, 1          ; 把比特位移入到進位元標誌位元中
5       jnc     skip_inc        ; 如果CF == 0，那麼跳轉到skip_inc處執行
6       inc     bl
7 skip_inc:
8       loop    count_loop
```

上面的代碼毀掉了在EAX中的初值(迴圈之後，EAX的值為0)。如果你想保留EAX中的值，那麼用rol eax, 1替換第四行即可。

第二节 布林型按位運算

有四個普遍的布林型運算符，它們是：AND， OR，XOR和 NOT。真值表展示了每一個可能的運算元得到的運算結果。

3.2.1 AND運算符

兩個比特位的AND運算結果只有當這兩位都是1時才為1，否則結果就為0，就像真值表 3.1展示的一樣。

處理器支援這些運算指令對資料的所有位元平等地進行獨立的運算。例如：如果對AL和BL裏的內容進行AND運算，那麼基本的AND運算將應用於在這兩個寄存器裏的8對比特位中的每一對，像圖 3.2 展示的一樣。下面

	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: 一個位元組的AND運算

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

表 3.2: OR運算

是一個代碼例子：

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

3.2.2 OR運算符

兩個比特位的包含OR運算結果只有當這兩位都是0時才為0，否則結果就為1，就像真值表 3.2展示的一樣。下面是一個代碼例子：

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

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

表 3.3: XOR運算

X	NOT X
0	1
1	0

表 3.4: NOT運算

3.2.3 XOR運算

兩個比特位的互斥XOR運算結果只有當這兩位相等時為0，否則結果就為1，就像真值表 3.3展示的一樣。下面是一個代碼例子：

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

3.2.4 NOT運算

NOT運算符是一元運算符(也就是說，它只對一個運算元進行運算，而不像二元運算符，如：AND)。 一個比特位的NOT運算結果是這個位值的相反數，像真值表 3.4展示的一樣。下面是一個代碼例子：

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

注意，NOT能得到一個數的補數。與其他按位運算不同的是，NOT指令並不修改在FLAGS寄存器裏的任何一位。

3.2.5 TEST指令

TEST指令執行一次AND運算，但是並不儲存結果。它會基於可能的結果對FLAGS寄存器進行設置(非常像CMP指令執行了一次減法操作但是只是設置了FLAGS)。例如：如果結果是0，那麼ZF就被置位了。

3.2.6 位元操作的應用

位元操作對於操縱資料單個位元而不修改其他位來說是非常有用的。表 3.5展示了這些操作的三個普遍的應用。下面是一些實現了這些想法的代碼例子。

開啟位i	將需修改的數與2^i(這是一個只有位i為on的二進位數字)進行OR運算
關閉位i	將需修改的數與只有位i為off的二進位數字進行AND運算。這個運算元通常稱為遮罩
求反位i	將需修改的數與2^i進行XOR運算

表 3.5: 布耳運算的使用

```
1   mov    ax, 0C123H
2   or     ax, 8           ; 開啟位3,    ax = C12BH
3   and    ax, 0FFDFH      ; 關閉位5,    ax = C10BH
4   xor    ax, 8000H       ; 求反位31,     ax = 410BH
5   or     ax, 0F00H       ; 開啟半個位元組,   ax = 4F0BH
6   and    ax, 0FFF0H      ; 關閉半個位元組,   ax = 4F00H
7   xor    ax, 0F00FH      ; 求反半個位元組,   ax = BF0FH
8   xor    ax, 0FFFFH      ; 補數,    ax = 40F0H
```

AND運算還可以用來得到除以2的幾次方之後的餘數。要得到除以數2^i之後的餘數，只需對這個數和等於$2^i - 1$的遮罩進行AND運算。這個遮罩從位0到位$2^i - 1$都為1，也就是這些位包含了餘數。AND運算的結果將保留這些位，而將其他位輸出為0。 下面是一個得到100除以16的商和餘數的代碼小片斷。

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

使用CL寄存器，就使得修改資料中的任何一位元變得有可能。下面是一個對EAX任意比特位置位(開啟)的例子。需要置位元的比特位元儲存在BH中。

```
1   mov    cl, bh          ; 首先需得到參加OR運算的數值
2   mov    ebx, 1
3   shl    ebx, cl         ; 向左移cl次
4   or     eax, ebx        ; 開啟位
```

關閉一位稍微有點麻煩。

```
1       mov     bl, 0           ; bl將儲存值為ON的位元數
2       mov     ecx, 32         ; ecx是迴圈計數器
3   count_loop:
4       shl     eax, 1          ; 移動一位元到進位元標誌位元中
5       adc     bl, 0           ; bl加上進位元標誌位元
6       loop    count_loop
```

图 3.3: 用ADC計算位數

```
1       mov     cl, bh          ; 首先需得到參加AND運算的數值
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        ; 關閉位
```

求反任意一個比特位的代碼留給了讀者，做為一個習題。

在一個80x86程式中看到這個莫名其妙的指令是很平常的：

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

一個數字與自己進行XOR運算，其結果總是0。這條指令被使用是因為它產生的機器代碼比功能相同的MOV指令要少。

第三节 避免使用條件分支

現代處理器使用了非常尖端的技術來盡可能快地執行代碼。一個普遍技術稱為 預測執行。這種技術使用CPU的並行處理能力來同時執行多條指令。條件分支與這項技術有衝突。一般說來，處理器是不知道分支是否會執行。如果執行了，跟沒有執行相比執行的是一組不同的指令。處理器試著預測分支是否執行。如果預測錯誤，處理器就浪費了它的時間去執行了一些錯誤的代碼。

一個避免這個問題的辦法就是如果可能儘量避免使用條件分支。在3.1.5的樣例代碼中就提供了一個可以這樣做的簡單的例子。在以前的例

子中，EAX寄存器中的值為 “on”的位被計算出來。它使用了一個分支跳轉到INC指令。圖 3.3展示了如何使用ADC指令直接加上進位元標誌位元來替代這個分支。

SETxx指令提供了在一定情況下替換分支的方法。基於FLAGS寄存器的狀態，這些指令將一個位元組的寄存器或記憶體空間中的值置為0或1。在SET後的字元與條件分支使用的是一樣的。如果SETxx條件為真，那麼儲存的結果就為1,如果為假，則儲存的結果就為0。例如：

```
setz    al          ; AL = 如果ZF標誌位置位元了則為1，否則為0。
```

使用這些指令，你可以開發出一些進行數值運算的精巧的技術，而不需要使用分支。

例如，考慮查找兩個數的最大數的問題。這個問題的標準解決方法是使用一條CMP指令再使用條件分支對最大值進行操作。下面這個例子展示了不使用分支如何找到最大值。

```
1  ; file: max.asm
2  %include "asm_io.inc"
3  segment .data
4
5  message1 db "Enter a number: ",0
6  message2 db "Enter another number: ", 0
7  message3 db "The larger number is: ", 0
8
9  segment .bss
10
11 input1  resd    1         ; 鍵入的第一個數值
12
13 segment .text
14         global  _asm_main
15 _asm_main:
16         enter   0,0               ; 程式開始運行
17         pusha
18
```

```
19          mov     eax, message1       ; 顯示第一條消息
20          call    print_string
21          call    read_int            ; 輸入第一個數值
22          mov     [input1], eax
23
24          mov     eax, message2       ; 顯示第二條消息
25          call    print_string
26          call    read_int            ; 輸入第二個數值(在eax中)
27
28          xor     ebx, ebx            ; ebx = 0
29          cmp     eax, [input1]       ; 比較第一和第二個數值
30          setg    bl                  ; ebx = (input2 > input1) ?          1 : 0
31          neg     ebx                 ; ebx = (input2 > input1) ? 0xFFFFFFFF : 0
32          mov     ecx, ebx            ; ecx = (input2 > input1) ? 0xFFFFFFFF : 0
33          and     ecx, eax            ; ecx = (input2 > input1) ?     input2 : 0
34          not     ebx                 ; ebx = (input2 > input1) ?      0 : 0xFFFFFFFF
35          and     ebx, [input1]       ; ebx = (input2 > input1) ?          0 : input1
36          or      ecx, ebx            ; ecx = (input2 > input1) ?     input2 : input1
37
38          mov     eax, message3       ; 顯示結果
39          call    print_string
40          mov     eax, ecx
41          call    print_int
42          call    print_nl
43
44          popa
45          mov     eax, 0              ; 返回到C中
46          leave
47          ret
```

這個訣竅是產生一個可以用來選擇出正確的最大值的俺碼。在30行的SETG指令如果第二個輸入值是最大值就將BL置為1,否則就置為0。這不是真正需要的遮罩。為了得到真正需要的遮罩，31行對整個EBX寄存器

使用了 NEG指令。(注意：EBX在前面已經置為0了。)如果EBX等於0,那麼這條指令就不做任何事情；但是如果EBX為1,結果就是-1的補數表示或0xFFFFFFFF。這正好是需要的位元遮罩。剩下的代碼就是使用這個位遮罩來選擇出正確的輸入的最大值。

另外一個可供選擇的訣竅是使用DEC語句。在上面的代碼中，如果用DEC代替NEG，那麼結果同樣會是0或0xFFFFFFFF。但是，與使用NEG相比，得到值將是反過來的。

第四节 在C中進行位元操作

3.4.1 C中的按位運算

不同於某些高階語言的是，C提供了按位元操作的運算符。AND運算符用二元運算符&來描述。[1]。OR運算符用二元運算符|來描述。而NOT運算符是用單項運算符~ 來描述。

C中的二元運算符<<和>>執行移位元操作。運算符<<執行左移操作而運算符 >>執行右移操作。這些運算符有兩個運算元。左邊的運算元是需要移位元的數值，右邊的運算元是需要移的位元數。如果需要移位元的數值是無符號類型，那麼就執行了一次邏輯移位元。如果需要移位元的數值是有符號類型(比如：int)，那麼就執行了一次算術移位元。下面是一些使用了這些運算符的C代碼例子：

```
1 short int s;          /* 假定short int類型為16位 */
2 short unsigned u;
3 s = -1;               /* s = 0xFFFF (補數) */
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 (邏輯移位元) */
9 s = s >> 2;           /* s = 0xFFA5 (算術移位元) */
```

[1]這個運算符不同於二元運算符&&和單項運算符&!

宏	含意
S_IRUSR	用戶可讀
S_IWUSR	用戶可寫
S_IXUSR	用戶可執行
S_IRGRP	組用戶可讀
S_IWGRP	組用戶可寫
S_IXGRP	組用戶可執行
S_IROTH	其他用戶可讀
S_IWOTH	其他用戶可寫
S_IXOTH	其他用戶可執行

表 3.6: POSIX檔許可權宏

3.4.2 在C中使用按位運算

在C中使用按位元運算的目的與在組合語言中使用按位元運算的目的是一樣的。它們可以允許你運算元據的單個比特位，而且可以用在快速乘除法中。事實上，一個好的C編譯器應該可以自動用移位元來進行乘法運算如：`x *= 2`。

許多作業系統的API[2](例如：POSIX[3]和Win32)包含了一些函式，這些函式使用的運算元含有按位元編碼的資料。例如：POSIX系統就為三種不同類型的用戶保留了檔的許可權：user (用戶，owner可能是一個更好的名字)， group(組用戶)和others(其他用戶)。每一種類型的用戶可以被授予進行讀，寫和/或執行一個檔的許可權。要改變一個檔的許可權，要求C程式師進行單個的位元操作。POSIX定義了幾個宏來做這件事(看表3.6)。chmod函式可以用來設置檔的許可權。這個函式有兩個參數，一個是表示需設置的檔檔案名的字串，另外一個是為需要的許可權設置了正確位的整形 [4] 。 例如，下面的代碼設置了這樣的許可權：允許檔的owner用戶對檔可讀可寫，在group中的用戶許可權為可讀而others用戶沒有許可權訪問。

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

[2]Application Programming Interface，應用程式介面
[3]代表電腦環境的可移植作業系統介面。IEEE在UNIX上標準開發出來的。
[4]實際上就是一個mode_t類型的參數，mode_t是一個整形類型的類型定義。

POSIX中stat函式可以用來得到檔的當前許可權位。與chmod函式一起使用，它可以用來改變某些許可權而不影響到其他許可權。下面是一個移除檔的others用戶的寫許可權和增加owner用戶的讀許可權的例子。同時，其他許可權沒有被改變。

```
1 struct stat file_stats ; /* stat()使用的結構體 */
2 stat("foo", & file_stats ); /* 讀檔資訊file_stats.st_mode中有許可權位 */
3 chmod("foo", ( file_stats .st_mode & ~S_IWOTH) | S_IRUSR);
```

第五节 Big和Little Endian表示法

第一章介紹了多位元組資料的big和little endian表示法的概念。但是，作者發現這個主題弄得很多人非常迷惑。這一節將詳細介紹這一主題。

讀者可能會回憶起endian表示法指的是一個多位元組資料的單個位元組元素儲存在記憶體中的順序。Big endian是最直接的方法。它首先儲存的是最高有效位元組，然後是第二個有效位元組，以些類推。換句話說就是，大的位元被首先儲存。Little endian以一個相反的順序來儲存位元組(最小的有效位元組最先被儲存)。x86家族的處理器使用的就是little endian表示法。

看一個例子：考慮雙字12345678_{16}的表示。如果是 big endian表示法，這些位元組將像這樣儲存：12 34 56 78。如果是 little endian表示法，這些位元組就像這樣儲存：78 56 34 12。

現在讀者可能會這樣問自己：一個理智的晶片設計者怎麼會使用little endian表示法？在Intel公司裏的工程師是不是虐待狂？因為他們使廣大的程式師承受了這種混亂的表示法。像這樣，CPU看起來會因為在記憶體中向後儲存位元組而做額外的工作(而且從記憶體中讀出時又要顛倒它們)。答案是CPU使用little endian格式讀寫記憶體是不需要做額外的工作的。你必須認識到CPU是由許多電子電路組成，簡單地工作在位值上。位元(和位元組)在CPU中不需要有任何的順序的。

考慮2個位元組的寄存器AX。這可以分解成單個位元組的寄存器：AH和AL。在CPU中的電路保留了AH 和 AL的值。在一個CPU中，電路是沒有任何順序的。也就是說，儲存AH的電路可以在儲存AL的電路前面或後面。mov指令

```
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: 如何確定Endian格式

複製AX的值到記憶體中，首先複製AL的值，接著是AH。CPU做這件事一點也沒有比先儲存AH難。

同樣的討論還可以用到一個位元組的單個比特位元上。它們在CPU電路(或就此而言，記憶體)裏並不真的有一定順序。但是，因為在CPU或記憶體中單個的比特位並沒有編址，所以沒有辦法知道(或關心)CPU在內部對它們是如何排序的。

在圖3.4中的C代碼展示了如何確定CPU的Endian格式。p指標把word變數當作兩個元素的字元陣列來看待。因此，word在記憶體裏的第一個位元組賦值給了p[0]，而這取決於CPU的Endian格式。

3.5.1 什麼時候需要在乎Little和Big Endian

對於典型的編程，CPU的Endian格式並不是很重要。它很重要的大多數時刻是在不同的電腦系統間傳輸二進位資料時。此時使用的要麼是某種類型的物理資料媒介(例如一塊硬碟)要麼是網路。因為ASCII資料是單個位元組的，Endian格式對它來說是沒有問題的。

當多位元組字元來到時，比如：UNICODE，Endian格式甚至對於文本資料都是非常重要的。UNICODE支持任意一種Endian格式而且有一個能指出使用了哪一種Endian格式從而來描述資料的技巧。

所有的內部的TCP/IP消息頭都以big endian的格式來儲存整形。 (稱為網路位元組續). TCP/IP 庫提供了可移植處理Endian格式問題的方法的C函式。例如：htonl() 函式把一個雙字(或長整形)從主機格式轉換成了網路格式。 ntohl()函式執行一個相反的交換。[5]對於一個big endian系統，這兩個函式僅僅是無修改地返回它們的輸入。這就允許你寫出的網路程式可以在任何的Endian格式系統上成功編譯和運行。要得到關

[5]實際上，轉換一個整形的Endian格式就是簡單地顛倒一下位元組；因此從big轉換成little和從little轉換成big是同樣的操作。所以所有這些函式都在做同樣的事。

```
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];    /* 逐個位元組地進行交換 */
8    ip [1] = xp[2];
9    ip [2] = xp[1];
10   ip [3] = xp[0];
11
12   return invert ;    /* 返回順序相反的位元組 */
13 }
```

图 3.5: invert_endian Function

於Endian格式和網路編程更多的資訊，請看W. Richard Steven寫的優秀的書籍： UNIX Network Programming。

圖 3.5展示了一個轉換雙字Endian格式的C函式。486處理器提供了一個名為BSWAP的新的指令來交換任意32位元寄存器中的位元組。例如：

```
bswap   edx             ; 交換edx中的位元組
```

這條指令不可以使用在16位的寄存器上。但是 XCHG 指令可以用來交換可以分解成8位寄存器的16寄存器中的位元組。例如：

```
xchg    ah,al           ; 交換ax中的位元組
```

第六节 計算位數

前面介紹了一個簡單的技術來計算一個雙字中“on”的位數有多少。這一節來看看其他不是很直接來做這件事的方法，就當作使用在這一章中討論的位元操作的練習。

```
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: 計算位數:方法一

3.6.1 方法一

第一個方法很簡單，但不是很明顯。圖 3.6展示了代碼。

這個方法為什麼會起作用？在迴圈中的每一次重複，data中就會有一個比特位被關閉了。當所有的位都關閉了(也就是說，當 data 等於0了)，迴圈就結束了。使data等於0需要重複的次數就等於原始值data中比特位為1的位數。

第6行表示data中的一個比特位被關閉了。這個是如何起作用的？考慮data二進位表示法最普遍的格式和在這種表示法中最右邊的1。根據上面的定義，在這個1後面的所有位都為0。現在，data - 1的二進位表示是什麼樣的？最右邊的1的所有左邊的位與data是一樣的，但是在最右邊的1這一點的位將會是data原始位的反碼。例如：

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

x表示對於在這個位上這兩個數的值是相等的。當data和data - 1進行AND運算後，在data中的最右邊的1這一位的結果就會為0，而其他比特位沒有被改變。

3.6.2 方法二

查找表同樣可以用來計算出任意雙字的位數。這個直接方法首先要算

出每個雙字的位數，還要把位元數儲存到一個陣列中。但是，有兩個與這個方法相關的問題。雙字的值大約有40億。這就意味著陣列將會非常大而且會浪費很多時間在初始化這個陣列上。(事實上，除非你確實打算使用一個超過40億的陣列，否則花在初始化這個陣列的時間將遠遠大於用第一種方法計算位數的時間。

一個更現實的方法是提前算出所有可能的位元組的位元數，並把它們儲存到一個陣列中。然後，雙字就可以分成四個位元組來求。這四個位元組的位元數通過查找陣列得到，然後將它們相加就得到原始雙字的位數。圖 3.7展示了如何用代碼實現這個方法。

`initialize_count_bits`函式必須在第一次調用`count_bits`函式之前被調用。這個函式初始化了`byte_bit_count`全局陣列。`count_bits`函式並不是以一個雙字來看對待`data`變數，而是以把它看成四個位元組的陣列。`byte`指標作為一個指向這個四個位元組陣列的指標。因此，`byte[0]`是`data`中的一個位元組(是最低有效位元組還是最高有效位元組取決於硬體是使用little還是big endian。)。當然，你可以像這樣使用一條指令：

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

來得到最高有效位元組值，可以用同樣的方法得到其他位元組；但是這些指令會比引用一個陣列要慢。

最後一點，使用`for`迴圈來計算在22和23行的總數是簡單的。但是，`for`迴圈就會包含初始化一個迴圈變數，在每一次重複後比較這個變數和增加這個變數的時間開支。通過清楚的四個值來計算總數會快一些。事實上，一個好的編譯器會將`for`迴圈形式轉換成清楚的求和。這個簡化和消除迴圈重複的處理是一個稱為迴圈展開的編譯器優化技術。

3.6.3 方法三

現在還有一個更聰明的方法來計算在資料裏的位元數。這個方法逐位元地相加資料的0位元和1位。得到的總數就等於在這個資料中1的位元數。例如，考慮計算儲存在`data`變數中的一個位元組中為1的位元數。第一步是執行下面這個操作：

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

這個做了些什麼？十六進位常量0x55的二進位表示為01010101。在這個加法的第一個運算元中，data與這個常量進行了 AND運算，奇數的位就被拿出來了。第二運算元((data >> 1) & 0x55)，首先移動所有的偶數位到奇數位上，然後使用相同的遮罩得到這些相同的位。現在，第一個運算元含有data的奇數位而第二個運算元含有偶數位。把這兩個運算元相加就相當於把data的奇數位和偶數位相加。例如，如果data等於 10110011_2，那麼：

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

顯示在右邊的加法展示了實際的位相加。這個位元組的位元被分成了四個2位的欄位來展示實際上執行了四個獨立的加法。因為所有這些欄位的最大總數為2，總數超過它自身的欄位且影響到其他欄位的總數是不可能的。

當然，總的位數還沒被計算出來。但是，可以使用跟上面一樣的技術，經過同樣的步驟來計算總數。下一步應該是：

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

繼續上面的例子(謹記：data現在等於 01100010_2)：

$$\begin{array}{r} \texttt{data \& } 00110011_2 \\ +\ \texttt{(data >> 2) \& } 00110011_2 \\ \hline \end{array} \quad \text{or} \quad \begin{array}{r|cc} + & 0010 & 0010 \\ & 0001 & 0000 \\ \hline & 0011 & 0010 \end{array}$$

現在有兩個4位的欄位被單獨地相加。

下一步是通過把這兩位的總數相加來得到最終的結果：

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

使用上面的例子(data等於00110010_2)：

$$\begin{array}{r} \texttt{data \& } 00001111_2 \\ +\ \texttt{(data >> 4) \& } 00001111_2 \\ \hline \end{array} \quad \text{or} \quad \begin{array}{r} 00000010 \\ +\ 00000011 \\ \hline 00000101 \end{array}$$

現在data等於5，這正好是正確的結果。圖 3.8實現了用這個方法來計算一個雙字的位數。它使用了一個 for迴圈來計算總數。把迴圈展開可能會更快一點，但是使用迴圈能清晰地看到這個方法產生的不同大小的資料。

```
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

```
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 ;    /* 向右移動的位數 */
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

第四章　副程式

本章主要著眼於使用副程式來構成模組化程式和得到與高階語言(比如說C)的介面。函式和進程是高階語言中副程式的例子。

調用了一個子程式的代碼和這個子程式必須協商它們之間的資料如何傳輸。資料如何傳輸的這些規則稱為調用約定。這一章的很大一部分都是在討論使用在彙編副程式和C程式介面上的標準C調用約定。這個約定(和其他約定)通常都是通過傳遞資料的位址(也就是指標)來允許副程式訪問記憶體中的資料。

第一节　間接定址

間接定址允許寄存器像指標變數一樣運作。要指出寄存器像一個指標一樣被間接使用，需要用方括號([])將它括起來。例如：

```
1    mov    ax, [Data]      ; 一個字的標準的直接記憶體位址
2    mov    ebx, Data       ; ebx = & Data
3    mov    ax, [ebx]       ; ax = *ebx
```

因為AX可以容納一個字，所以第三行代碼從EBX儲存的位址開始讀取一個字。如果用AL替換AX，那麼只有一個位元組會被讀取。認識到寄存器不像在C中的變數一樣有類型是非常重要的。到底EBX具體指向什麼完全取決於使用了什麼指令。而且，甚至EBX是一個指標這個事實都完全取決於使用的指令。如果EBX錯誤地使用了，通常不會有編譯錯誤；但是，程式將不會正確運行。這就是為什麼相比於高階語言組合語言程式較容易犯錯的原因之一。

所有的32位通用寄存器(EAX， EBX， ECX， EDX)和指標寄存器(ESI, EDI)都可以用來間接定址。一般來說，16位或8位的寄存器是不可以的。

第二节 副程式的簡單例子

副程式是代碼中的一個的獨立的單元，它可以使用在程式的不同的地方。換句話說，一個子程式就像一個C中的函式。可以使用跳轉來調用副程式，但是返回會是一個問題。如果子程式要求能使用在程式中的任何地方，它必須要返回到調用它的代碼段處。因此，副程式的跳轉返回最好不要硬編碼為標號。下面的代碼展示了如何使用JMP指令的間接方式來做這件事。此指令方式使用一個寄存器的值來決定跳轉到哪(因此，這個寄存器與C中的函式指標非常相似。) 下面使用副程式的方法來重寫第一章中的第一個程式。

sub1.asm

```
1  ; file: sub1.asm
2  ; 副程式的簡單例子
3  %include "asm_io.inc"
4
5  segment .data
6  prompt1 db    "Enter a number: ", 0       ; 不要忘記空結束符
7  prompt2 db    "Enter another number: ", 0
8  outmsg1 db    "You entered ", 0
9  outmsg2 db    " and ", 0
10 outmsg3 db    ", the sum of these is ", 0
11
12 segment .bss
13 input1  resd 1
14 input2  resd 1
15
16 segment .text
17         global  _asm_main
```

```
18 _asm_main:
19         enter   0,0             ; 程式開始運行
20         pusha
21
22         mov     eax, prompt1    ; 顯示提示資訊
23         call    print_string
24
25         mov     ebx, input1     ; 儲存input1的位址到ebx中
26         mov     ecx, ret1       ; 儲存返回位址到ecx中
27         jmp     short get_int   ; 讀整形
28 ret1:
29         mov     eax, prompt2    ; 輸出提示資訊
30         call    print_string
31
32         mov     ebx, input2
33         mov     ecx, $ + 7      ; ecx = 當前地址 + 7
34         jmp     short get_int
35
36         mov     eax, [input1]   ; eax = 在input1中的雙字
37         add     eax, [input2]   ; eax += 在input2中的雙字
38         mov     ebx, eax        ; ebx = eax
39
40         mov     eax, outmsg1
41         call    print_string    ; 輸出第一條資訊
42         mov     eax, [input1]
43         call    print_int       ; 輸出input1
44         mov     eax, outmsg2
45         call    print_string    ; 輸出第二條資訊
46         mov     eax, [input2]
47         call    print_int       ; 輸出input2
48         mov     eax, outmsg3
49         call    print_string    ; 輸出第三條資訊
50         mov     eax, ebx
```

```
51          call    print_int           ; 輸出總數(ebx)
52          call    print_nl            ; 換行
53
54          popa
55          mov     eax, 0              ; 返回到C中
56          leave
57          ret
58  ; 副程式 get_int
59  ; 參數:
60  ;   ebx - 儲存整形雙字的位址
61  ;   ecx - 返回指令的位址
62  ; 注意:
63  ;   eax的值被已經被破壞掉了
64  get_int:
65          call    read_int
66          mov     [ebx], eax          ; 儲存輸入到記憶體中
67          jmp     ecx                 ; 返回到調用處
```

sub1.asm

副程式get_int使用了一個簡單，基於寄存器的調用約定。它認為EBX寄存器中存的是輸入雙字的儲存位址而ECX寄存器中存的是跳轉返回指令的位址。25行到28行，使用了ret1標號來計算返回位址。在32行到34行，使用了$運算子來計算返回的地址。$運算子返回出現$這一行的當前地址。$ + 7運算式計算在36行的MOV指令的位址。

這兩種計算返回位址的方法都是不方便的。第一種方法要求為每一次子程式調用定義一個標號。第二種方法不需要標號，但是需要仔細的思量。如果使用了近跳轉來替代短跳轉，那麼與$相加的數就不會是7！幸運的是，有一個更簡單的方法來調用副程式。這種方法使用堆疊。

第三节 堆疊

許多CPU都支援內置堆疊。堆疊是一個先進後出(LIFO)的佇列。它是以這種方式組織的一塊記憶體區域。PUSH指令添加一個資料到堆疊中而

POP指令從堆疊中移除數據。移除的資料就是最後入堆疊的資料(這就是稱為先進後出佇列的緣故)。

SS段寄存器指定包含堆疊的段(通常它與儲存資料的段是一樣)。ESP寄存器包含將要移除出堆疊資料的位址。這個資料也被稱為堆疊頂。資料只能以雙字的形式入堆疊。也就是說，你不可以將一個位元組推入堆疊中。

PUSH指令通過把ESP減4來向堆疊中插入一個雙字[1]，然後把雙字儲存到[ESP]中。 POP指令從[ESP]中讀取雙字，然後再把ESP加4.下面的代碼演示了這些指令如何工作，假定在ESP初始值為1000H。

```
1  push   dword 1     ; 1儲存到0FFCh中, ESP = 0FFCh
2  push   dword 2     ; 2儲存到0FF8h中,  ESP = 0FF8h
3  push   dword 3     ; 3儲存到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
```

堆疊可以方便地用來臨時儲存資料。它同樣可以用來形成副程式調用和傳遞參數和局部變數。

80x86同樣提供一條PUSHA指令來把EAX， EBX， ECX， EDX， ESI， EDI
和EBP寄存器的值推入堆疊中(不是以這個順序)。 POPA指令可以用來將它們移除出堆疊。

第四节 CALL和RET指令

80x86提供了兩條使用堆疊的指令來使副程式調用變得快速而簡單。CALL指令執行一個跳到副程式的無條件跳轉，同時將下一條指令的位址推入堆疊中。RET指令彈出一個位址並跳轉到這個位址去執行。使用這些指令的時候，正確處理堆疊以便RET指令能彈出正確的數值是非常重要的!

前面的例子可以使用這些新的指令來重寫。把25行到34行改成：

[1]實際上，多字入堆疊也是可以的，但是在32位元保護模式下，在堆疊上最好只操作單個雙字。

```
    mov     ebx, input1
    call    get_int

    mov     ebx, input2
    call    get_int
```

同時把副程式get_int改成:

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

CALL和RET指令有幾個優點:

- 它們很簡單!
- 它們使副程式嵌套變得簡單。注意:副程式get_int調用了read_int。這個調用將另一個位址壓入到堆疊中了。在read_int代碼的末尾是一條彈出返回位址的RET指令,通過執行指令重新回到get_int代碼中去執行。然後,當 get_int的RET指令被執行時,它彈出跳回到asm_main的返回地址。這個之所以能正確運行,是因此堆疊的LIFO特性。

記住彈出壓入到堆疊的所有資料是非常重要的。例如,考慮下面的代碼:

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

這個代碼將不會正確返回!

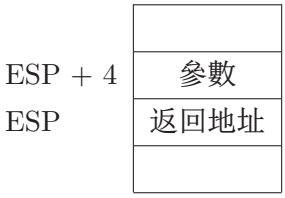

图 4.1:

第五节 調用約定

當調用了一個子程式，調用的代碼和副程式(被調用的代碼)必須協商好在它們之間如何傳遞資料。高階語言有標準傳遞資料的方法稱為調用約定。要讓高階語言介面於組合語言，組合語言代碼就一定要使用與高階語言一樣的約定。不同的編譯器有不同的調用約定或者說不同的約定可能取決於代碼如何被編譯。(例如：是否進行了優化)。一個廣泛的約定是：使用一條CALL指令來調用代碼再通過RET指令返回。

所有PC的C編譯器支持的調用約定將在本章的後續部分階段進行描述。這些約定允許你創建可重入的副程式。一個可重入的副程式可以在程式中的任意一點被安全調用(甚至在副程式內部)。

4.5.1 在堆疊上傳遞參數

給副程式的參數需要在堆疊中傳遞。它們在CALL指令之前就已經被壓入堆疊中了。和在C中是一樣的，如果參數被副程式改變了，那麼需要傳遞的是資料的位址，而不是值。如果參數的大小小於雙字，它就需要在壓入堆疊之前轉換成雙字。

在堆疊裏的參數並沒有由副程式彈出，取而代之的是：它們自己從堆疊中訪問本身。為什麼？考慮

- 因為它們在CALL指令之前被壓入堆疊中，所以返回時首先彈出的是返回位址(然後修改堆疊指標使其指向參數入堆疊以前的值)。

- 參數往往將會使用在副程式中幾個的地方。通常在整個程式中，它們不可以保存在一個寄存器中，而應該儲存在記憶體中。把它們留在堆疊裏就相當於把資料複製到了記憶體中，這樣就可以在副程式的任意一點訪問資料。

當使用了間接定址，80x86通過看間接定址運算式裏使用了什麼寄存器來決定訪問哪不同的段。ESP(和EBP)使用堆疊段，而EAX，EBX，ECX和EDX使用資料段。但是，這個對於保護模式通常是不重要的，因為對於保護模式資料段和堆疊段是同一段。

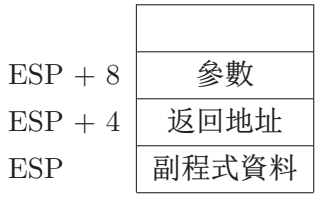

图 4.2:

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

图 4.3: 副程式的一般格式

通過堆疊傳遞了一個參數的副程式。當副程式被調用了，堆疊狀態如圖 4.1。這個參數可以通過間接定址訪問到。(`[ESP+4]` [2])。

如果在副程式內部使用了堆疊儲存資料，那麼與ESP相加的數將要改變。例如： 圖 4.2展示了如果一個雙字壓入堆疊中後堆疊的狀態。現在參數在`ESP + 8`中，而不在`ESP + 4`中。因此，引用參數時若使用ESP就很容易犯錯了。為了解決這個問題，80386提供使用另外一個寄存器：EBP。這個寄存器的唯一目的就 是引用堆疊中的資料。C調用約定要求副程式首先把EBP的值保存到堆疊中，然後再使EBP的值等於ESP。當資料壓入或彈出堆疊時，這就允許ESP值被改變的同時EBP不會被改變。在副程式的結束處，EBP的原始值必須恢復出來 (這就是為什麼它在副程式的開始處被保存的緣故。)圖 4.3展示了遵循這些約定的副程式的一般格式。

圖 4.3中的第2行和第3行組成了一個子程式的大體上的開始部分。第5行和第6行組成了結束部分。圖 4.4展示了剛執行完開始部分之後堆疊的狀態。現在參數可以在副程式中的任何地方通過`[EBP + 8]`來訪問，而不用擔心在副程式中有什麼資料壓入到堆疊中了。

[2]使用間接定址時，寄存器加上一個常量是合法的。許多更複雜的運算式也是合法的。這個話題將在下一章中介紹。

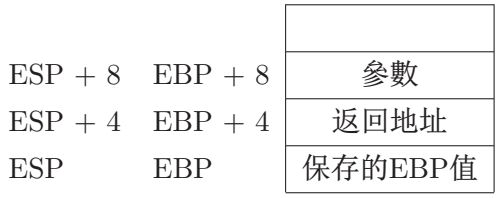

图 4.4:

```
1       push    dword 1         ; 傳遞參數1
2       call    fun
3       add     esp, 4          ; 將參數移除出堆疊
```

图 4.5: 副程式調用示例

執行完副程式之後，壓入堆疊中的參數必須移除掉。C調用約定規定調用的代碼必須做這件事。其他約定可能不同。例如：Pascal 調用約定 規定副程式必須移除參數。(RET指令的另一種格式可以很容易做這件事。)一些C編譯器同樣支持這種約定。關鍵字pascal用在函式的原型和定義中來告訴編譯器使用這種約定。事實上，MS Windows API的C函式使用的stdcall調用約定同樣以這種方式運作。這種方式有什麼優點？它比C調用約定更有效一點。那為什麼所有的C函式都使用C調用約定呢？一般說來，C允許一個函式的參數為變化的個數(例如：printf和scanf函式)。對於這種類型的函式，將參數移除出堆疊的操作在這次函式調用中和下次函式調用中是不同的。C調用約定能使指令簡單地執行這種不同的操作。Pascal和stdcall調用約定執行這種操作是非常困難的。因此， Pascal調用約定(和Pascal語言一樣)不允許有這種類型的函式。MS Windows只有當它的API函式不可以攜帶變化個數的參數時才可以使用這種約定。

圖 4.5展示了一個將被調用的副程式如何使用C調用約定。第3行通過直接操作堆疊指標將參數移除出堆疊。同樣可以使用POP指令來做這件事，但是常常使用在要求將無用的結果儲存到一個寄存器的情況下。實際上，對於這種情況，許多編譯器常常使用一條POP ECX來移除參數。編譯器會使用POP指令來代替ADD指令，因為ADD指令需要更多的位元組。但是，POP會改變ECX的值。下面是一個有兩個副程式的例子，它們使用了上面討論的C調用約定。54行(和其他行)展示了多個資料和文本段可以在同一個原始

檔案中聲明。進行連接處理時，它們將會組合成單一的資料段和文本段。把資料和文本段分成單獨的幾段就允許資料定義在副程式代碼附近，這也是副程式經常做的。

sub3.asm

```
%include "asm_io.inc"

segment .data
sum     dd   0

segment .bss
input   resd 1

;
; 虛擬碼演算法
; i = 1;
; sum = 0;
; while( get_int(i, &input), input != 0 ) {
;   sum += input;
;   i++;
; }
; print_sum(num);
segment .text
        global  _asm_main
_asm_main:
        enter   0,0               ; 程式開始運行
        pusha

        mov     edx, 1            ; edx就是虛擬碼裏的i
while_loop:
        push    edx               ; 保存i到堆疊中
        push    dword input       ; 把input的位址壓入堆疊
        call    get_int
        add     esp, 8            ; 將i和&input移除出堆疊
```

```
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]         ; 將總數壓入堆疊
42          call    print_sum
43          pop     ecx                 ; 將[sum]移除出堆疊
44
45          popa
46          leave
47          ret
48
49  ; 副程式get_int
50  ; 參數(順序壓入堆疊中)
51  ;   輸入的個數(儲存在[ebp + 12]中)
52  ;   儲存輸入字的位址(儲存在[ebp + 8]中)
53  ; 注意:
54  ;   eax和ebx的值被毀掉了
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          ; 將輸入儲存到記憶體中
72
73          pop     ebp
74          ret                         ; 返回到調用代碼
75
76  ; 副程式print_sum
77  ; 輸出總數
78  ; 參數:
79  ;   需要輸出的總數(儲存在[ebp+8]中)
80  ; 注意: eax的值被毀掉了
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
```

```
subprogram_label:
      push    ebp                 ; 保存原始EBP值到堆疊中
      mov     ebp, esp            ; 新EBP的值 = ESP
      sub     esp, LOCAL_BYTES    ; = #局部變數需要的位元組數
; subprogram code
      mov     esp, ebp            ; 釋放局部變數
      pop     ebp                 ; 恢復原始EBP值
      ret
```

图 4.6: 帶有局部變數的副程式的一般格式

```
96
97          pop     ebp
98          ret
```

sub3.asm

4.5.2 堆疊上的局部變數

堆疊可以方便地用來儲存局部變數。這實際上也是C儲存普通變數(或C lingo中的自動變數)的地方。如果你希望副程式是可重入的，那麼使用堆疊存儲變數是非常重要的。一個可重入的副程式不管在任何地方被調用都能正常運行，包括副程式本身。換句話說，可重入副程式可以嵌套調用。儲存變數的堆疊同樣在記憶體中。不儲存在堆疊裏的資料從程式開始到程式結束都使用記憶體(C稱這種類型的變數為 總體變數或靜態變數)。儲存在堆疊裏的資料只有當定義它們的副程式是活動的時候才使用記憶體。

在堆疊中，局部變數恰好儲存在保存的EBP值之後。它們通過在副程式的開始部分用ESP減去一定的位元組數來分配存儲空間。 圖 4.6展示了副程式新的骨架。EBP用來訪問局部變數。考慮圖 4.7中的C函式。 圖 4.8 展示如何用組合語言編寫等價的副程式。

圖 4.9展示了執行完圖 4.8中程式的開始部分後的堆疊狀態。這一節的堆疊包含了參數，返回資訊和局部變數，這樣堆疊稱為一個堆疊幀。C函式的每一次調用都會在堆疊上創建一個新的堆疊幀。

儘管ENTER和LEAVE指令事實上簡化了開始部分和結束部分，但是它們並沒有經常被使用。這是為什麼呢？因為與等價的簡單的指令相比，它們執行較慢！當你並不認為執行一隊指令序列比執行一條複合的指令要快的時候，這就是一個榜樣。

```
1  void calc_sum( int n, int * sump )
2  {
3    int i, sum = 0;
4
5    for( i=1; i <= n; i++ )
6      sum += i;
7    *sump = sum;
8  }
```

图 4.7: 求總數的C語言版

可以使用兩條專門的指令來簡化一個子程式的開始部分和結束部分，它們是為這個目的而專門設計的。ENTER指令執行開始部分的代碼，而 LEAVE指令執行結束部分。ENTER指令攜帶兩個立即數。對於C調用約定，第二個運算元總是為0。第一個運算元是局部變數所需要的位元組數。LEAVE指令沒有運算元。 圖 4.10展示了如何使用這些指令。注意程式skeleton(圖 1.7)同樣使用了ENTER和LEAVE指令。

第六节 多模組程式

多模組程式是由不止一個目標檔組成的程式。這裏出現的所有程式都是多模組程式。它們由C驅動目標檔和彙編目標檔(加上C庫目標檔)組成。回憶一下連接程式將目標檔組合成一個可執行程式。連接程式必須把在一個模組(也就是 目標檔)中引用的每個變數匹配到定義該變數的模組。 為了讓模組A能使用定義在模組B裏的變數，就必須使用extern(外部)指示符。在extern 指示符後面是用逗號隔開的變數列表。這個指示符告訴編譯器把這些變數視為是模組外部的。也就是說，這些變數可以在這個模組中使用，但是卻定義在另一模組中。asm_io.inc檔中就將read_int等程式定義為外部的。

在編譯語言中，缺省情況下變數不可以由外部程式訪問。如果一個變數可以被一個模組訪問，而這個模組又不是定義它的,那麼在定義它的模

組中，它一定被聲明為global(全局的)。global 指示符就可以用來做這件事情。圖 1.7的程式skeleton中的第13行定義了一個總體變數 _asm_main。若沒有這個聲明，就可能會出錯。為什麼？因為C代碼將會找不到內部的_asm_main變數。

下面是用兩個模組重寫的以前例子的代碼。副程式(get_int和print_sum)在不同的原始檔案中，而不是在_asm_main程式中。

main4.asm

```
1  %include "asm_io.inc"
2
3  segment .data
4  sum     dd   0
5
6  segment .bss
7  input   resd 1
8
9  segment .text
10         global  _asm_main
11         extern  get_int, print_sum
12 _asm_main:
13         enter   0,0               ; 程式開始運行
14         pusha
15
16         mov     edx, 1            ; edx就是虛擬碼中的i
17 while_loop:
18         push    edx               ; 保存i到堆疊中
19         push    dword input       ; 把input的位址壓入堆疊
20         call    get_int
21         add     esp, 8            ; 將i和&input移除出堆疊
22
23         mov     eax, [input]
24         cmp     eax, 0
25         je      end_while
26
```

```
27          add     [sum], eax          ; sum += input
28
29          inc     edx
30          jmp     short while_loop
31
32  end_while:
33          push    dword [sum]         ; 將總數壓入堆疊
34          call    print_sum
35          pop     ecx                 ; 將總數[sum]移除出堆疊
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          ; 將輸入儲存到記憶體中
20
21          leave
22          ret                         ; 返回到調用代碼
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碼變數；同樣，全局資料變數也可以使用一模一樣的方法。

第七节 C與彙編的介面技術

現今，完全用彙編書寫的程式是非常少的。編譯器能很好地將高階語言轉換成有效的機器代碼。因為用高階語言書寫代碼非常容易，所以高階語言變得很流行。此外，高階語言比組合語言更容易移植！

當使用組合語言時，我們經常將它使用在代碼中的一小部分上。有兩種使用組合語言的方法：在C中調用彙編副程式或內嵌彙編。內嵌彙編允

許程式師把彙編語句直接放入到C代碼中。這樣是非常方便的；但是，內嵌彙編同樣存在缺點。組合語言的書寫格式必須是編譯器使用的格式。目前沒有一個編譯器支持NASM格式。不同的編譯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機上，調用彙編副程式是更標準的技術。

在C中使用組合語言程式通常是因為以下幾個原因：

- 需要直接訪問電腦的硬體特性，而用C語言很難或不可能做到。
- 程式執行必須盡可能地快，而且相比于編譯器，程式師手動優化的代碼更好。

最後一個原因不像它以前一樣有根據。因為這些年編譯器技術提高了，而且編譯器通常可以產生非常有效的代碼 (特別是當開啟編譯器優化的時候)。調用組合語言程式的缺點：可攜性和可讀性減弱了。

絕大部分的C調用約定已經確定了。但是，還需要描述一些額外的特徵。

4.7.1 保存寄存器

關鍵字register可以使用在一個C變數聲明中來暗示編譯器：這個變數使用一個寄存器而不是記憶體空間。這些變數稱為寄存器變數。現在的編譯器會自動做這件事而不需要給它暗示。

首先， C假定副程式保存了下面這幾個寄存器的值：EBX，ESI，EDI，EBP，CS，DS，SS，ES。這並不意味著不能在副程式內部修改它們。相反，它表示如果子程式改變了它們的值，那麼在副程式返回之前必須恢復它們的原始值。EBX，ESI和EDI的值不能被改變，因為C將這些寄存器用於寄存器變數。通常都是使用堆疊來保存這些寄存器的原始值。

4.7.2 函式名

大多數C編譯器都在函式名和全局或靜態變數前附加一個下劃線字元。例如，函式名f將指定為_f。因此，如果這是一個組合語言程式，那麼它必須標記為_f，而不是f。Linux gcc編譯器並不附加任何字元。 在可執行的Linux ELF下，對於C函式f，你只需要簡單使用函式名f即可。但是，DJGPP的gcc卻附加了一個下劃線。注意，在組合語言程式skeleton中 (圖1.7)，主程序函式名是_asm_main。

[3]GAS是所有的GNU編譯器使用的組合語言程式。它使用AT&T語法，這是一種完全不同於MASM，TASM 和NASM的語法。

4.7.3 傳遞參數

按照C調用約定，一個函式的參數將以一定順序壓入堆疊中，這個順序與它們出現在函式調用裏的順序相反。

考慮這條C語句：`printf("x = %d\n",x);` 圖 4.11展示了如何編譯這條語句(用等價的NASM格式)。圖 4.12展示了執行完`printf`函式的開始部分後，堆疊的狀態。`printf`函式一個可以攜帶任意個參數的C語言庫函式。C調用約定的規則就是專門為允許這些類型的函式而規定的。 因為format字串的位址最後壓入堆疊，所以不管有多少參數傳遞到函式，它在堆疊裏的位置將總是 `EBP + 8`。然後`printf`代碼就可以通過看format字串的位置來決定需要傳遞多少參數和在堆疊上如何找到它們。

當然，如果有錯誤發生， `printf("x = %d\n")`， `printf`代碼仍然會將`[EBP + 12]`中的雙字值輸出，而這並不是`x`的值！

沒有必要使用組合語言來處理C中的可變參數函式。可以通過移植`stdarg.h`頭檔中定義的宏來處理它們。看一本好的C語言的書來得到更詳細的資訊。

4.7.4 計算局部變數的位址

找到定義在`data`或`bss`段的變數的位址是非常容易的。基本上，連接程式做的就是這件事情。但是，要計算出在堆疊上的一個局部變數(或參數)的位址就不簡單了。可是，當調用副程式的時候，這種需求是非常普通的。考慮傳遞一個變數(讓我們稱它為`x`)的位址到一個函式(讓我們稱它為`foo`)的情況。如果`x`處在堆疊的EBP − 8的位置，你不可以這樣使用：

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

為什麼？因為指令`MOV`儲存到EAX裏的值必須能由彙編器計算出來(也就是說，它最後必須是一個常量)。但是，有一條指令能做這種需求的計算。它就是 `LEA` (即Load Effective Address，載入有效位址)。下面的代碼就能計算出`x`的位址並將它儲存到EAX中：

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

現在EAX中存有了`x`的位址，而且當調用函式`foo`的時候，就可以將其壓入到堆疊中。不要搞混了，這條指令看起來是從[EBP−8]中讀數據；然而，這並不正確。`LEA`指令永遠不會從記憶體中讀數據。它僅僅計算出一個將會被其他指令使用到的位址，然後將這個位址儲存到它的第一個運算元裏。因為它並沒有實際讀記憶體，所以不指定記憶體大小(例如： `dword`)是必須的或說是允許的。

4.7.5 返回值

返回值不為空的C函式執行完後會返回一個值。C調用約定規定了這個要如何去做。返回值需通過寄存器傳遞。所有的整形類型(char，int， enum， 等)通過EAX寄存器返回。如果它們小於32位元，那麼儲存到EAX的時候，它們將被擴展成32位。 (它們如何擴展取決於是有符號類型還是無符號類型。) 64位的值通過EDX:EAX寄存器對返回。浮點數儲存在數學輔助運算器中的ST0寄存器中。 (這個寄存器將在浮點數這一章來討論。)

4.7.6 其他調用約定

所有的80x86 C編譯器中都支援上面描述的標準C調用約定的規則。通常編譯器也支援其他調用約定。當與組合語言進行介面時，知道編譯器調用你的函式時使用的是什麼調用約定是非常重要的。通常，缺省時，使用的是標準的調用約定；但是，並不總是這一種情況[4]。 使用多種約定的編譯器通常都擁有可以用來改變缺省約定的命令行開關。它們同樣提供擴展的C語法來為單個函式指定調用約定。但是，各個編譯器的這些擴展標準可以是不一樣的。

GCC編譯器允許不同的調用約定。一個函式的調用約定可以通過擴展語法__attribute__ 明確指定。例如，要聲明一個返回值為空的函式f，它帶有一個int參數，使用標準調用約定，需使用下面的語法來聲明它的原型：

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

GCC同樣支援標準call 調用約定。通過把cdecl替換成stdcall，上面的函式可以指定為使用這種約定。stdcall約定和cdecl約定的不同點是stdcall要求副程式將參數移除出堆疊(和Pascal調用約定一樣)。因此，stdcall調用約定只能使用在帶有固定參數的函式上 (也就是說，不可以是函式printf和scanf)。

GCC同樣支持稱為regparm 的約定，這種約定告訴編譯器前3個整形參數通過寄存器傳遞給函式，而不是通過堆疊。這是許多編譯器支援的一個共同的優化模式。

[4]Watcom C編譯器就是一個在缺省情況下不使用標準調用的例子。看Watcom的樣例原始檔案代碼來得到更詳細的資訊

Borland和Microsoft使用一樣語法來聲明調用約定。它們在C代碼中加上關鍵字__cdecl和 __stdcall。這些關鍵字用來修飾函式。在原型聲明中，它們出現在函式名的前面例如，上面的函式f用Borland和Microsoft定義如下：

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

每種調用約定都有各自的優缺點。cdecl調用約定的主要優點是它非常簡單而且非常靈活。它可以用於任何類型的C函式和C編譯器。使用其他約定會限制副程式的可攜性。它的主要缺點是與其他約定相比它執行較慢而且使用更多的記憶體(因為函式的每次調用都需要用代碼將參數移除出堆疊。)。

stdcall調用約定的主要優點是相比於cdecl它使用較少的記憶體。在CALL指令之後 ，不需要清理堆疊。它的主要缺點是它不能使用於可變參數的函式。

使用寄存器傳遞參數的調用約定的優點是速度非常快。主要缺點是這種約定太複雜。有些參數可能在寄存器中，而另一些可能在堆疊中。

4.7.7 樣例

下面是一個展示組合語言程式如何與C程式介面的例子。(注意：這個程式並沒有使用skeleton組合語言程式(圖 1.7)或driver.c模組。)

main5.c

```
1  #include <stdio.h>
2  /* 組合語言程式的原型聲明 */
3  void calc_sum( int, int * ) __attribute__ ((cdecl));
4
5  int main( void )
6  {
7     int n, sum;
8
9     printf("Sum integers up to: ");
10    scanf("%d", &n);
```

```
11    calc_sum(n, &sum);
12    printf("Sum is %d\n", sum);
13    return 0;
14  }
```

main5.c

sub5.asm

```
1  ; 副程式 _calc_sum
2  ; 求整形1到n的和
3  ; 參數:
4  ;   n - 從1加到多少(儲存在[ebp + 8])
5  ;   sump - 指向總數儲存位址的整形指標(儲存在[ebp + 12])
6  ; C虛擬碼:
7  ; void calc_sum( int n, int * sump )
8  ; {
9  ;   int i, sum = 0;
10 ;   for( i=1;i <= n; i++ )
11 ;     sum += i;
12 ;   *sump = sum;
13 ; }
14
15 segment .text
16         global  _calc_sum
17 ;
18 ; 局部變數:
19 ;   儲存在[ebp-4]裏的sum值
20 _calc_sum:
21         enter   4,0               ; 在堆疊上為sum分配空間
22         push    ebx               ; 重要!
23
24         mov     dword [ebp-4],0   ; sum = 0
25         dump_stack 1, 2, 4        ; 輸出堆疊中從ebp-8到ebp+16的值
26         mov     ecx, 1            ; ecx是虛擬碼中的i
```

```
27 for_loop:
28         cmp     ecx, [ebp+8]       ; 比較i和n
29         jnle    end_for            ; 如果i > n,則退出迴圈
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                ; 恢復ebx的值
41         leave
42         ret
```

sub5.asm

為什麼程式sub5.asm中的第22行非常重要？因為C調用約定要求EBX的值不能被調用的函式更改。如果沒有做這個，程式很可能不會正確運行。

第25行演示了巨集dump_stack如何運作。它的第一個參數只是一個數字標號，第二個參數決定需顯示EBP以下多少個雙字而第三個參數決定需顯示EBP以上多少個雙字。圖4.13展示了這個程式的運行示例。對於這次轉儲，你可以看到儲存總數的雙字位址是FBFFFFB80 (儲存在EBP + 12)；n值為0000000A (儲存在EBP + 8)；程式的返回位址為08048501 (儲存在EBP + 4)；保存的EBP的值為BFFFFB88 (儲存在EBP)；局部變數的值為 0(儲存在EBP - 4)；最後保存的EBX的值為4010648C (儲存在EBP - 8)。

calc_sum函式可以這樣重寫：把sum當作返回值返回，替代使用的指針參數。因為sum是一個整形值，所以應保存到EAX寄存器中。main5.c檔中的第11行應該改成：

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

同樣，calc_sum的原型也需要改變。下面是修改後的彙編代碼：

sub6.asm

```
1  ; 副程式 _calc_sum
2  ; 求整形1到n的和 ; 參數:
3  ;   n - 從1加到多少(儲存在[ebp + 8]
4  ; 返回值:
5  ;   sum的值
6  ; C虛擬碼:
7  ; int calc_sum( int n )
8  ; {
9  ;   int i, sum = 0;
10 ;   for( i=1; i <= n; i++)
11 ;     sum += i;
12 ;   return sum;
13 ; }
14 segment .text
15         global  _calc_sum
16 ;
17 ; 局部變數:
18 ;   儲存在[ebp-4]裏的sum值
19 _calc_sum:
20         enter   4,0               ; 在堆疊上為sum分配空間
21
22         mov     dword [ebp-4],0   ; sum = 0
23         mov     ecx, 1            ; ecx是虛擬碼中的i
24 for_loop:
25         cmp     ecx, [ebp+8]      ; 比較i和n
26         jnle    end_for           ; 如果i > n,則退出迴圈
27
28         add     [ebp-4], ecx      ; sum += i
29         inc     ecx
30         jmp     short for_loop
31
32 end_for:
33         mov     eax, [ebp-4]      ; eax = sum
```

```
34
35          leave
36          ret
```
sub6.asm

4.7.8 在組合語言程式中調用C函式

C與彙編介面的一個主要優點是允許彙編代碼訪問大型C庫和用戶寫的函式。例如，如果你想調用一下scanf函式來從鍵盤讀一個整形，該怎麼辦？圖 4.14展示了完成這件事的代碼。需要記住的非常重要的一點就是scanf函式遵循字面意義的C調用標準。這就意味著它保存了EBX，ESI和
EDI寄存器的值；但是， EAX，ECX和EDX寄存器的值可能會被修改。事實上，EAX肯定會被修改，因為它將保存scanf調用的返回值。至於與C介面的其他例子，可以看用來產生asm_io.obj的asm_io.asm檔中的代碼。

第八节 可重入和遞迴副程式

一個可重入副程式必須滿足下面幾個性質：

- 它不能修改代碼指令。在高階語言中，修改代碼指令是非常難的；但是在組合語言中，一個程式要修改自己的代碼並不是一件很難的事。例如：

  ```
  mov     word [cs:$+7], 5        ; 將5複製到前面七個位元組的字中
  add     ax, 2                   ; 前面的語句將2改成了5!
  ```

  這些代碼在實模式下可以運行，但是在保護模式下的作業系統上不行，因為代碼段被標識為唯讀。在這些作業系統上，當執行了上面的第一行代碼，程式將被終止。這種類型的程式從各個方面來看都非常差。它很混亂，很難維護而且不允許代碼共用(看下面)。

- 它不能修改總體變數(比如在data和 bss段裏的數據)。所有的變數應儲存在堆疊裏。

書寫可重入性代碼有幾個好處。

- 一個可重入副程式可以遞迴調用。
- 一個可重入程式可以被多個進程共用。在許多多工作業系統上，如果一個程式有許多實例正在運行，那麼只有一份代碼的拷貝在記憶體中。共用庫和DLL(Dynamic Link Libraries，動態連結程式庫)同樣使用了這種技術。
- 可重入副程式可以運行在多線程 [5] 程式中。 Windows 9x/NT和大多數類 UNIX作業系統(Solaris， Linux，等)都支援多線程程式。

4.8.1 遞迴副程式

這種類型的副程式調用它們自己。遞迴可以是 直接的或是間接的。當一個名為foo的副程式在foo內部調用自己就產生直接遞迴。當一個子程式雖然自己沒有直接調用自己，但是其他副程式調用了它，就產生間接遞迴。例如：副程式foo可以調用 bar且bar也可以調用foo。

遞迴副程式必須有一個終止條件。當這個條件為真時，就不再進行遞迴調用了。如果一個子程式沒有終止條件或條件永不為真，那麼遞迴將不會結束(非常像一個無窮迴圈)。

圖 4.15展示了一個遞迴求n!的函式。在C中它可以這樣被調用：

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

圖 4.16展示了上面的函式調用的最深點的堆疊狀態。

圖 4.17展示了另一個更複雜的遞迴樣例的C語言版而4.18展示了它的組合語言版。對於f(3)，輸出是什麼？注意：每一次遞迴調用，ENTER指令都會在堆疊上給新的i值分配空間。因此，f的每一次遞迴調用都有它自己獨立的變數i。若是在data段定義i為一雙字，結果就不一樣了。

4.8.2 回顧一下C變數的儲存類型

C提供了幾種變數儲存類型。

global，全局 這些變數定義在任何函式的外面，且儲存在固定的記憶體空間中(在data或bss段)，而且從程式的開始一直到程式的結束都存

[5] 一個多線程程式同時有多條線程在執行。也就是說，程式本身是多工的。

在。缺省情況下，它們能被程式中的任何一個函式訪問；但是，如果它們被聲明為static，那麼只有在同一模組中的函式才能訪問它們(也就是說， 依照彙編的術語，這個變數是內部的，不是外部的)。

static，靜態 在一個函式中，它們是被聲明為靜態的局部變數。(不幸的是，C使用關鍵字static有兩種目的！)這些變數同樣儲存在固定的記憶體空間中(在data或bss段)，但是只能被定義它的函式直接訪問。

automatic，自動 它是定義在一個函式內的C變數的缺省類型。當定義它們的函式被調用了，這些變數就被分配在堆疊上，而當函式返回了又從堆疊中移除。因此，它們沒有固定的記憶體空間。

register，寄存器 這個關鍵字要求編譯器使用寄存器來儲存這個變數的資料。這僅僅是一個要求。編譯器並不一定要遵循。如果變數的位址使用在程式的任意的地方，那麼就不會遵循(因為寄存器沒有位址)。同樣，只有簡單的整形資料可以是寄存器變數。結構類型不可以；因為它們的大小不匹配寄存器！C編譯器通常會自動將普通的自動變數轉換成寄存器變數，而不需要程式師給予暗示。

volatile，不穩定 這個關鍵字告訴編譯器這個變數值隨時都會改變。這就意味著當變數被更改了，編譯器不能做出任何推斷。通常編譯器會將一個變數的值暫時存在寄存器中，而且在出現這個變數的代碼部分使用這個寄存器。但是，編譯器不能對不穩定類型的變數做這種類型的優化。一個不穩定變數的最普遍的例子就是：它可以被多線程程式的兩個線程修改。考慮下面的代碼：

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

如果x可以被另一個線程修改。那麼其他線程可以會在第1行和第3行之間修改x的值，以致於z將不會等於10.但是， 如果x沒有被聲明為不穩定類型，編譯器就會推斷x沒有改變，然後再將z置為10。

不穩定類型的另一個使用就是避免編譯器為一個變數使用一個寄存器。

```
1  cal_sum:
2        push   ebp
3        mov    ebp, esp
4        sub    esp, 4                ; 為局部變數sum分配空間
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: 求總數的組合語言版

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     ; = #局部變數需要的位元組數
; subprogram code
      leave
      ret
```

图 4.10: 通過使用ENTER和LEAVE指令，帶有局部變數的副程式的一般格式

```
segment .data
x             dd     0
format        db     "x = %d\n", 0

segment .text
...
      push    dword [x]      ; 將x的值壓入堆疊中
      push    dword format   ; 將format字串的位址壓入堆疊中
      call    _printf        ; 注意下劃線!
      add     esp, 8         ; 從堆疊中移除參數
```

图 4.11: 調用printf

EBP + 12	x的值
EBP + 8	format字串的位址
EBP + 4	返回地址
EBP	保存的EBP值

图 4.12: printf的堆疊結構

```
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程式的運行示例

```
1  segment .data
2  format       db "%d", 0
3
4  segment .text
5  ...
6        lea    eax, [ebp-16]
7        push   eax
8        push   dword format
9        call   _scanf
10       add    esp, 8
11 ...
```

图 4.14: 在組合語言程式中調用scanf函式

```
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, 則終止
10       dec    eax
11       push   eax
12       call   _fact            ; eax = fact(n-1)
13       pop    ecx              ; 結果在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!的遞迴函式

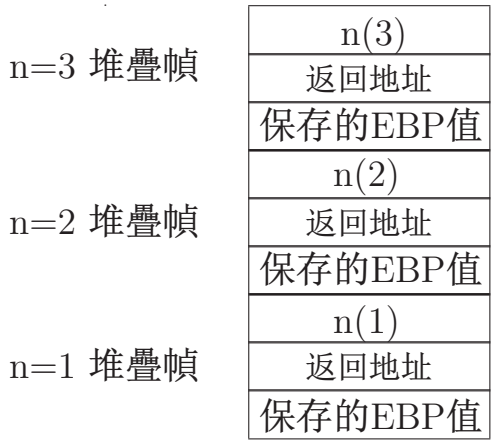

图 4.16: n!函式的堆疊幀

```
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: 另一個例子(C語言版)

```
1  %define i ebp-4
2  %define x ebp+8          ; useful macros
3  segment .data
4  format       db "%d", 10, 0     ; 10 = '\n'
5  segment .text
6        global _f
7        extern _printf
8  _f:
9        enter  4,0           ; 在堆疊上為i分配空間
10
11       mov    dword [i], 0  ; i = 0
12 lp:
13       mov    eax, [i]      ; is i < x?
14       cmp    eax, [x]
15       jnl    quit
16
17       push   eax           ; 調用printf
18       push   format
19       call   _printf
20       add    esp, 8
21
22       push   dword [i]     ; 調用f
23       call   _f
24       pop    eax
25
26       inc    dword [i]     ; i++
27       jmp    short lp
28 quit:
29       leave
30       ret
```

图 4.18: 另一個例子(組合語言版)

第五章　陣列

第一节　介紹

一組陣列是記憶體中的一個連續資料列表塊。在這個列表中的每個元素必須是同一種類型而且使用恰好同樣大小的記憶體位元組來儲存。因為這些特性，陣列允許通過資料在陣列裏的位置(或下標)來對它進行有效的訪問。如果知道了下面三個細節，任何元素的位址都可以計算出來：

- 陣列第一個元素的位址
- 每個元素的位元組數
- 這個元素的下標

0(正如在C中)作為陣列的第一個元素的下標是非常方便的。使用其他值作為第一個下標也是可能的，但是這將把計算弄得很複雜。

5.1.1　定義陣列

5.1.1.1　在data和bss段中定義陣列

在data段定義一個初始化了的陣列，可以使用標準的db，dw，等等 指示符。NASM同樣提供了一個有用的指示符，稱為TIMES ，它可以用來反復重複一條語句，而不需要你手動來複製它。圖 5.1展示關於這些的幾個例子。

在bss段定義一個示初始化的陣列，可以使用 resb，resw，等等 指示符。記住，這些指示符包含一個指定保留多少個記憶體單元的運算元。圖5.1同樣展示了關於這種類型定義的幾個例子。

```
segment .data
; 定義10個雙字的陣列並初始化為1,2,..,10
a1              dd 1, 2, 3,4, 5, 6, 7, 8, 9, 10
; 定義10個字的陣列並初始化為0
a2              dw 0, 0, 0, 0,0, 0, 0, 0, 0, 0
; 像以前一樣使用TIMES
a3              times 10 dw 0
;定義一個位元組陣列,其中包含200個0和100個1
a4              times 200 db 0
                times 100 db 1

segment .bss
; 定義10個雙字的陣列, 示未始化
a5              resd  10
;定義10了個字的陣列, 示未始化
a6              resw  100
```

图 5.1: 定義陣列

5.1.1.2 以局部變數的方式在堆疊上定義陣列

在堆疊上定義一個局部陣列變數沒有直接的方法。像以前一樣，你可以首先計算出所有局部變數需要的全部位元組，包括陣列，然後再用ESP減去這個數值(或者直接使用ENTER指令)。例如，如果一個函式需要一個字元變數，兩個雙字整形和一個包含50個元素的字陣列，你將需要$1 + 2 \times 4 + 50 \times 2 = 109$個位元組。但是，為了保持ESP在雙字的邊界上，被ESP減的數值必須是4的倍數(這個例子中是112。)圖 5.2展示了兩種可能的方法。第一種排序的未使用部分用來保持雙字在雙字邊界上，這樣可以加速記憶體的訪問。

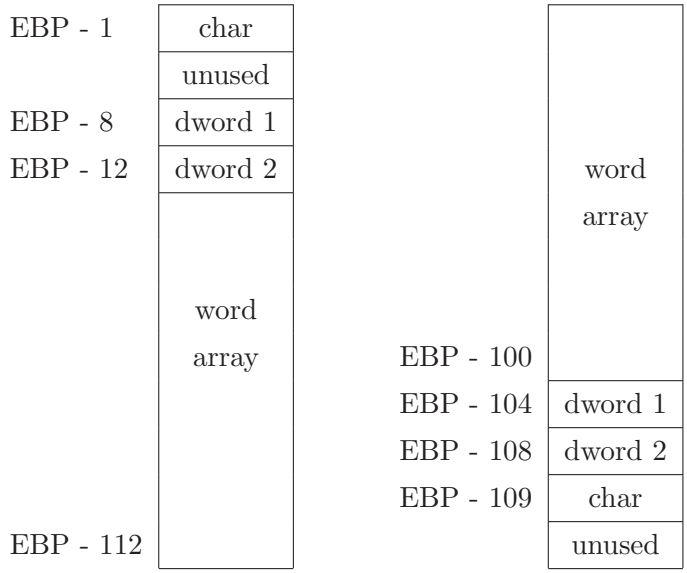

图 5.2: 堆疊上的陣列排列

5.1.2 訪問陣列中的元素

跟C不同的是，在組合語言中沒有[ ]運算符。要訪問陣列中的一個元素，必須將它的位址計算出來。考慮下面兩個陣列的定義：

```
array1      db      5, 4, 3, 2, 1       ; 位元組陣列
array2      dw      5, 4, 3, 2, 1       ; 字陣列
```

下面是使用這些陣列的例子：

```
1       mov     al, [array1]            ; al = array1[0]
2       mov     al, [array1 + 1]        ; al = array1[1]
3       mov     [array1 + 3], al        ; array1[3] = al
4       mov     ax, [array2]            ; ax = array2[0]
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，引用了字陣列中的元素1,而不是元素2。為什麼？因為字是兩個位元組的單元，所以移動到字陣列的下一元素，你必須向前移動兩個位元組，而不是一個。第7行將從第一個元素中讀取一個位元組再從第二個元素

```
1       mov     ebx, array1             ; ebx = array1的地址
2       mov     dx, 0                   ; dx將儲存總數
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: 對一組陣列的元素求和(版本1)

中讀取一個位元組。在C中，編譯器會根據一個指標的類型來決定使用指標運算的運算式需移動多少位元組，而程式師就不需要管這些了。然而，在組合語言中，當需要從一個元素移動到另一個元素時，它取決於程式師認為的陣列元素的大小。

圖 5.3展示了一個代碼片段：對前面樣例代碼中的陣列array1中的元素進行了求和。第 7行，AX與DX相加。為什麼不是AL？首先，ADD指令的兩個運算元必須為同樣的大小。其次，這樣做對於對位元組求和後得到一個太大以致不能匹配一個位元組的總數是非常容易的。通過使用DX，達到65,535的總數是允許。然而，認識到AH同樣被相加了是非常重要的。這就是第3行為什麼AH被置為0的緣故了。[1]

圖 5.4和圖5.5展示了兩種可以替換的方法來計算總數。斜體字的行替換了圖 5.3中的第6行和第7行。

5.1.3 更高級的間接定址

不要驚訝，間接定址經常與陣列一起使用。最普遍的間接記憶體引用格式為：

[1]置AH為0隱含地表示AL是一個無符號數值。如果它是有符號的，恰當的操作是在第6行和第7行之間插入一條CBW指令。

```
1       mov     ebx, array1             ; ebx = array1的地址
2       mov     dx, 0                   ; dx將儲存總數
3       mov     ecx, 5
4   lp:
5       add     dl, [ebx]               ; dl += *ebx
6       jnc     next                    ; 若沒有進位，則跳轉到next
7       inc     dh                      ; inc dh
8   next:
9       inc     ebx                     ; bx++
10      loop    lp
```

图 5.4: 對一組陣列的元素求和(版本2)

[base reg(基址寄存器) + factor(係數)*index reg(變址寄存器) + constant(常量)]

其中：

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

係數 可以是1，2，4或8。(如果是1，係數是可以省略的。)

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

常量 為一個32位的常量。這個常量可以是一個變數(或變數運算式)。

5.1.4 例子

這有一個使用陣列並將它傳遞給函式的例子。它使用 array1c.c程式(下面列出的)作為驅動程式，而不是 driver.c程式。

array1.asm

```
1   %define ARRAY_SIZE 100
2   %define NEW_LINE 10
```

```
1       mov     ebx, array1             ; ebx = array1的地址
2       mov     dx, 0                   ; dx將儲存總數
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: 對一組陣列的元素求和(版本3)

```
3
4   segment .data
5   FirstMsg        db   "First 10 elements of array", 0
6   Prompt          db   "Enter index of element to display: ", 0
7   SecondMsg       db   "Element %d is %d", NEW_LINE, 0
8   ThirdMsg        db   "Elements 20 through 29 of array", 0
9   InputFormat     db   "%d", 0
10
11  segment .bss
12  array           resd ARRAY_SIZE
13
14  segment .text
15          extern  _puts, _printf, _scanf, _dump_line
16          global  _asm_main
17  _asm_main:
18          enter   4,0     ; 在EBP - 4處的局部雙字變數
19          push    ebx
20          push    esi
21
22  ; 將陣列初始化為100, 99, 98, 97,  ...
23
```

```
24          mov     ecx, ARRAY_SIZE
25          mov     ebx, array
26  init_loop:
27          mov     [ebx], ecx
28          add     ebx, 4
29          loop    init_loop
30
31          push    dword FirstMsg          ; 顯示FirstMsg
32          call    _puts
33          pop     ecx
34
35          push    dword 10
36          push    dword array
37          call    _print_array            ; 顯示陣列的前10個元素
38          add     esp, 8
39
40  ; 提示用戶輸入元素下標
41  Prompt_loop:
42          push    dword Prompt
43          call    _printf
44          pop     ecx
45
46          lea     eax, [ebp-4]       ; eax = 局部雙字變數的位址
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  ; 轉儲當前行的剩餘部分並重新開始
55          jmp     Prompt_loop           ; 如果輸入無效
56
```

```
57  InputOK:
58          mov     esi, [ebp-4]
59          push    dword [array + 4*esi]
60          push    esi
61          push    dword SecondMsg       ; 顯示元素的值
62          call    _printf
63          add     esp, 12
64
65          push    dword ThirdMsg        ; 顯示元素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  ; 程式_print_array
82  ; 調用的C程式把雙字陣列的元素當作有符號整形來顯示。
83  ; C函式原型:
84  ; void print_array( const int * a, int n);
85  ; 參數:
86  ;   a - 指向需要顯示的陣列的指標(堆疊上ebp+8處)
87  ;   n - 需要顯示的整數個數(堆疊上ebp+12處)
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 = 陣列的位址
102 print_loop:
103         push    ecx                     ; printf將會改變ecx!
104
105         push    dword [ebx + 4*esi]     ; 將array[esi]壓入堆疊
106         push    esi
107         push    dword OutputFormat
108         call    _printf
109         add     esp, 12                 ; 移除參數(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 /* 函式dump_line* 轉儲輸入緩衝區中當前行的所有字元*/
14 void dump_line()
15 {
16   int ch;
17
18   while( (ch = getchar()) != EOF && ch != '\n')
19     /*空程式體*/ ;
20 }
```

array1c.c

5.1.4.1 再看一下LEA指令

LEA指令不僅僅可以用來計算位址，也可以用作其他目的。一個相當普遍的目的是快速計算。考慮下面的代碼：

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

這條代碼有效地將5 × EAX的值儲存到EBX中。相比於使用MUL指令，使用LEA既簡單又快捷。但是，你必須認識到在方括號裏的運算式必須是一個合法的間接位址。因此，例如，這個指令就不可以用來快速乘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          ; 將結果儲存到x中(在ebp - 52中)
```

图 5.6: x = a[i][j]的組合語言表示法

5.1.5 多維陣列

多維陣列和已經討論的普遍的一維陣列相比，差異並不是很大。事實上，在記憶體中，它們的描述方法和普遍的一維陣列是一樣的。

5.1.5.1 二維陣列

不要感到意外，最簡單的多維陣列就是二維陣列。一個二維陣列通常以網格的形式來表示元素。每個元素通過兩個下標來確定。按照慣例，第一個下標用來確定元素的行值，而第二下標用來確定元素的列值。

考慮一個三行二列的陣列，像這樣定義：

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

C編譯器將為這個陣列保留$6(= 2 \times 3)$個整形數的空間，而且像下面一樣來映射元素：

下標	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]

這張表試圖展示的是a[0][0]引用的元素儲存在6個元素的一維陣列的開始。元素a[0][1]儲存在下一個位置(下標 1)，以此類推。在記憶體中，二維陣列的各個行都是連續儲存的。一行的最後一個元素後面緊跟著下一行的第一個元素。這就是所謂的陣列 依行表示法，這也是C/C++編譯器表示陣列的表示方法。

在依行表示法中，編譯器如何確定a[i][j]出現在哪？一個簡單的公式將通過i和j來計算下標。在這個例子中，公式為$2i + j$。並不難看出如何得到這個公式。每一行有兩個元素大小；所以行i的第一個元素的位置為$2i$。

然後該行的j列的位置可以通過j和 $2i$相加得到。這個分析同樣展示了如何產生N列陣列的公式：$N \times i + j$。注意，這個公式並不依賴於行的總數。

作為一個例子，我們來看看gcc如何編譯下面的代碼(使用上面定義的陣列a)：

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

圖 5.6展示了這條語句翻譯成的組合語言。因此編譯器實質上將代碼轉換成：

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

而且事實上，程式師可以以這種方法來書寫，也可以得到同樣的結果。

選擇依行的陣列表示法並沒有什麼魔力。依列的表示法同樣可以工作：

下標	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，各列被連續儲存。元素[i][j]儲存在$i + 3j$位置中。其他語言 (例如：FORTRAN)使用依列表示法。當與多種語言進行代碼介面時，這是非常重要的。

5.1.5.2 大於二維

對於大於二維的陣列，應用了同樣原理的想法。考慮一個三維陣列：

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

在記憶體中，這個陣列將被當作四個大小為[3][2]的二維陣列被連續儲存。下面的表展示了它如何運作：

下標	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]
下標	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]

計算b[i][j][k]的位置的公式是$6i + 2j + k$。其中6由[3][2]陣列的大小決定。一般來說，對於維數為a[L][M][N]的陣列，元素a[i][j][k]的位置將是$M \times N \times i + N \times j + k$。再次需要注意的是，第一個維的元素個數(L)並沒有出現在公式中。

對於更高維的陣列，可以通過推廣來做同樣的處理。對於一個從D_1到D_n的n陣列，下標為i_1到i_n的元素的位置可以由下面這個公式得到：

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

或者對於超級的數學技客，它可以用更簡潔地書寫為：

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

第一維D_1，並沒有出現在公式中。

這裏你可以斷定作者是物理專業的。(或者說是引用了FORTRANO語言的一個免費樣品？)

對於依列表示法，普遍的公式將是：

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

或表示為超級數學技客的書寫方法：

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

在這種情況下，是最後一維D_n，不出現在公式中。

5.1.5.3 在C語言中，傳遞多維陣列參數

多維陣列的依列表示法在C編程有一個直接的效果。對於一維的陣列，當任何具體的元素被放置到記憶體中時，陣列的大小並不需要計算出來。但這對於多維陣列是不正確的。為了訪問這些陣列的元素，除了第一維的元素個數，編譯器必須知道其他所有維數的元素個數。當一個函式的原型帶有一個多維陣列參數時，這就變得很明顯了。下面的代碼將不會被編譯：

```
void f( int a[ ][ ] );  /* 沒有維數資訊 */
```

但是，下面的代碼就會被編譯：

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

任何有兩列的二維陣列可以傳遞給這個函式。第一維的元素個數是不需要的[2]。

不要被這類函式的原型搞混了：

[2]它可以在這裏指定，但是會被編譯器忽略。

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: 從串取和存入串指令

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

它定義了一個一維的整形指標陣列。(它可以附帶用來創建一個像二維陣列一樣運作的陣列。)

對於更高維的陣列，除了第一維的元素個數，陣列參數的其他維數的必須指定。例如，一個四維的陣列參數可以像這樣被傳遞：

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

第二节 陣列/串處理指令

80x86家族的處理器提供了幾條與陣列一起使用的指令。這些指令稱為串處理指令。它們使用變址寄存器(ESI和EDI)來執行一個操作，然後這兩個寄存器自動地進行增1或減1操作。FLAGS寄存器裏的方向標誌位元(DF)決定了這些變址寄存器是增加還是減少。有兩條指令用來修改方向標誌位元：

CLD 清方向標誌位元。這種情況下，變址寄存器是自動增加的。

STD 置方向標誌位元。這種情況下，變址寄存器是自動減少的。

80x86編程中的一個非常普遍的錯誤就是忘記了把方向標誌位元明確地設置為正確的狀態。這就經常導致代碼大部分情況下能正常工作(當方向標誌位元恰好就是所需要的狀態時)，但並不能正常工作在所有情況下。

```
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                    ; 別忘記這個!
9        mov    esi, array1
10       mov    edi, array2
11       mov    ecx, 10
12 lp:
13       lodsd
14       stosd
15       loop  lp
```

图 5.8: 從串取和存入串的例子

5.2.1 讀寫記憶體

最簡單的串處理指令是讀或寫記憶體或同時讀寫記憶體。它們可以每次讀或寫一個位元組，一個字或一個雙字。圖 5.7中的小段虛擬碼展示了這些指令的作用。這有點需要注意的。首先，ESI是用來讀的，而EDI是用來寫的。如果你能記得SI代表Source Index，源變址寄存器和DI代表Destination Index，目的變址寄存器，那麼這個就很容易記住了。 其次，注意包含資料的寄存器是固定的(AL，AX或 EAX)。最後，注意存入串指令使用ES來決定需要寫的段，而不是DS。在保護模式下，這通常不是問題，因為它只有一個資料段，而ES應自動地初始化為引用DS(和DS一樣)。但是，在實模式下，將ES初始化為正確的段值對於程式師來說是非常重要的[3]。圖 5.8展示了一個使用這些指令將一個陣列複製到另一陣列的例子。

[3]另一複雜的地方是你不可以直接使用一條MOV指令來將DS寄存器中的值複製到ES寄存器中。取而代之的是，你需要使用兩條MOV指令，先把DS寄存器的值複製到一個通用寄存器(比如：AX)中，然後再把這個通用寄存器的值複製到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: 串傳送指令

LODSx和STOSx指令的聯合使用(如圖 5.8中的13和14行)是非常普遍的。事實上，一條MOVSx串處理指令可以用來完成這個聯合使用的功能。圖 5.9描述了這些指令執行的操作。圖 5.8的第13和14行可以用MOVSD指令來替代，能得到同樣的效果。唯一的區別就是在迴圈時EAX寄存器根本就不會被使用。

5.2.2 REP首碼指令

80x86家族提供了一個特殊的首碼指令[4]，稱為REP，它可以與上面的串處理指令一同使用。這個首碼告訴CPU重複執行下條串處理指令一個指定的次數。ECX寄存器用來計算重複的次數 (和在LOOP指令中的使用是一樣的)。使用REP首碼，在圖 5.8中的12到15行的循環體可以替換成一行：

```
rep movsd
```

圖 5.10展示了另一個例子：得到一個零陣列。

5.2.3 串比較指令

圖 5.11展示了幾個新的串處理指令：它們可以用來比較記憶體和記憶體或記憶體和寄存器。在比較和查找陣列方面，它們是很有用的。它們會

[4]首碼指令並不是一個指令，它是放置在串處理指令前面的一個特殊的位元組，用來修改指令的行為。其他首碼同樣可以用來進行跨段記憶體訪問

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

图 5.10: 零陣列的例子

像CMP指令一樣設置FLAGS寄存器。CMPSx 指令比較相應的記憶體空間，而SCASx 根據一指定的值掃描記憶體空間。

圖 5.12展示了一個代碼小片斷：在一個雙字陣列中查找數位12。行10裏的SCASD指令總是對EDI進行加4操作，即使找到了需要的數值。因此，如果你想得到陣列中數12的位址，就必須用EDI減去4(正如 行 16所做的)。

5.2.4 REPx首碼指令

還有其他可以用在串比較指令中的，像REP一樣的首碼指令。圖 5.13展示了兩個新的首碼並描述了它們的操作。REPE 和 REPZ作用是一樣的(REPNE和REPNZ也是一樣)。如果重複的串比較指令因為比較的結果而終止了，變址寄存器同樣會進行增量操作而ECX也會進行減1操作；但是FLAGS寄存器將仍然保持著重複終止時的狀態。 因此，使用ZF標誌位元來確定重複的比較是因為一次比較而結束，還是因為ECX等於0而結束是可能的。

在重複比較之後，為什麼你不能單單看ECX是否等於0?

圖 5.14展示了一個樣例子代碼片斷：確定兩個記憶體塊是否相等。例子中行 7中的JE指令是用來檢查前面指令的結果。如果重複的比較是因為它找到了兩個不相等的位元組而終止，那麼ZF標誌位元就為0,也就不會執行分支；但是，如果比較是因為ECX等於0而結束，那麼ZF標誌位元就為1,而且代碼將分支到equal標號處。

CMPSB	比較位元組[DS:ESI]和[ES:EDI] ESI = ESI ± 1 EDI = EDI ± 1
CMPSW	比較字[DS:ESI]和[ES:EDI] ESI = ESI ± 2 EDI = EDI ± 2
CMPSD	比較雙字[DS:ESI]和[ES:EDI] ESI = ESI ± 4 EDI = EDI ± 4
SCASB	比較AL和[ES:EDI] EDI ± 1
SCASW	比較AX和[ES:EDI] EDI ± 2
SCASD	比較EAX和[ES:EDI] EDI ± 4

图 5.11: 串比較指令

5.2.5 樣例

這一節包含了一個彙編原始檔案，原始檔案中包含了幾個應用串處理指令進行陣列操作的函式。其中大部分都是C語言庫中的常見函式的複製品。

memory.asm

```
1  global _asm_copy, _asm_find, _asm_strlen, _asm_strcpy
2
3  segment .text
4  ; 函式 _asm_copy
5  ; 複製記憶體塊
6  ; C原型
7  ; void asm_copy( void * dest, const void * src, unsigned sz);
8  ; 參數:
9  ;   dest - 指向複製操作的目的緩衝區的指標
10 ;   src  - 指向複製操作的源緩衝區的指標
```

```
segment .bss
array        resd 100

segment .text
      cld
      mov    edi, array    ; 指向陣列開始部分的指標
      mov    ecx, 100      ; 元素的個數
      mov    eax, 12       ; 需掃描的個數
lp:
      scasd
      je     found
      loop   lp
 ; 若沒有找到, 執行的代碼
      jmp    onward
found:
      sub    edi, 4         ; edi現在指向陣列中的12
 ; 若找到了, 執行的代碼
onward:
```

图 5.12: 查找的例子

```
;   sz   - 需要複製的位元組數

; 下面, 定義了一些有用的變數

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

REPE, REPZ	當ZF標誌位元為1或重複次數不超過ECX時，重複執行指令
REPNE, REPNZ	當ZF標誌為0或重複次數不超過ECX時，重複執行指令

图 5.13: REPx首碼指令

```
segment .text
      cld
      mov    esi, block1         ; block1的地址
      mov    edi, block2         ; block2的地址
      mov    ecx, size           ; block以位元組表示的大小
      repe   cmpsb               ; 當ZF為1時，重複執行
      je     equal               ; 如果ZF為1,跳轉到equal
   ; 如果block不相等，執行的代碼
      jmp    onward
equal:
   ; 如果相等，執行的代碼
onward:
```

图 5.14: 比較記憶體塊

```

        mov     esi, src        ; esi = 複製操作的源緩衝區的位址
        mov     edi, dest       ; edi = 複製操作的目的緩衝區的位址
        mov     ecx, sz         ; ecx = 需要複製的位元組數

        cld                     ; 清方向標誌位元
        rep     movsb           ; 執行movsb操作ECX次

        pop     edi
        pop     esi
        leave
        ret

```

```
35
36  ; 函式 _asm_find
37  ; 根據一給定的位元組值查找記憶體
38  ; void * asm_find( const void * src, char target, unsigned sz);
39  ; 參數:
40  ;   src    - 指向需要查找的緩衝區的指標
41  ;   target - 需要查找的位元組值
42  ;   sz     - 在緩衝區中的位元組總數
43  ; 返回值:
44  ;   如果找到了target, 返回指向在緩衝區中第一次出現target的地方的指標
45  ;   否則
46  ;     返回NULL
47  ; 注意: target是一個位元組值, 但是被當作一個雙字壓入堆疊中。
48  ;       位元組值儲存在低8位元上。
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           ; 掃描直到ECX == 0或[ES:EDI] == AL才停止
64
65          je      found_it        ; 如果ZF為1,則找到了相應的值
66          mov     eax, 0          ; 如果沒有找到, 返回NULL指標
67          jmp     short quit
```

```
68  found_it:
69          mov     eax, edi
70          dec     eax                 ; 如果找到了, 則返回(DI - 1)
71  quit:
72          pop     edi
73          leave
74          ret
75
76
77  ; 函式 _asm_strlen
78  ; 返回字串的大小
79  ; unsigned asm_strlen( const char * );
80  ; 參數:
81  ;   src - 指向字串的指標
82  ; 返回值:
83  ;   字串中的字元數(以0結束, 若碰到0,則不再計數) (儲存在EAX中)
84
85  %define src [ebp + 8]
86  _asm_strlen:
87          enter   0,0
88          push    edi
89
90          mov     edi, src            ; edi = 指向字串的指標
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               ; 掃描終止符0
96
97  ;
98  ; repnz將會多執行一步, 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 ; 函式 _asm_strcpy
109 ; 複製一個字串
110 ; void asm_strcpy( char * dest, const char * src);
111 ; 參數:
112 ;   dest - 指向進行複製操作的目的字串
113 ;   src  - 指向進行複製操作的源字串
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                   ; 載入AL & inc si
127         stosb                   ; 儲存AL & inc di
128         or      al, al          ; 設置條件標誌位元
129         jnz     cpy_loop        ; 如果沒到終止符0,則繼續
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); /* 複製源字串的所有30個字元到目的字
   串 */
20   printf("%s\n", st2);
21
22   printf("Enter a char: "); /* 在字串中查找一位元組值 */
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 );       /* 複製源字串的有效字元到目的字串 */
36    printf("%s\n", st2 );
37
38    return 0;
39 }
```

memex.c

第六章　浮點

第一节　浮點表示法

6.1.1　非整形的二進位數字

在第一章討論數制的時候，我們只討論了整形。顯然，和十進位一樣，其他進制必須也能表示非整形數。在十進位中，在小數點右邊的數字關聯了10的負乘方值：

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

不必驚訝，二進位也是以同樣的方法表示：

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

這個辦法與第一章中的整形辦法相結合就可以用來轉換一個一般數值：

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

將十進位轉換成二進位也不是很難。一般來說，需將十進位數字分成兩塊：整數部分和分數部分。使用第一章中的方法來將整數部分轉換成二進位。分數部分的轉換可以使用下面描述的方法。

考慮一個用$a, b, c, \ldots$標記比特位元的二進位分數。這個數用二進位表示為：

$$0.abcdef\ldots$$

將此數乘2.新得到的數的二進位表示將是：

$$a.bcdef\ldots$$

注意，第一個比特位現在在權值為1的位置。用0替換a得到：

$$0.bcdef\ldots$$

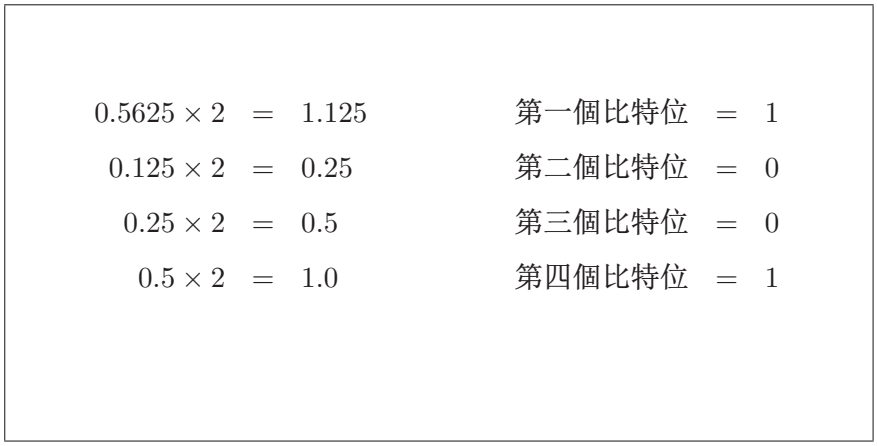

图 6.1: 將0.5625轉換成二進位

再乘以2得到：

$$b.cdef\ldots$$

現在第二個比特位(b)在權值為1的位置。重複這個過程，直到得到了需要的盡可能多的比特位。圖 6.1展示了一個實例：將0.5625轉換成二進位。這種方法當分數部分為0了才停止。

另一個例子，將23.85轉換成二進位。將整數部分($23 = 10111_2$)轉換成二進位是容易的，但是分數部分呢？圖 6.2展示了這個計算的開始部分。如果你仔細看了這個數值，就會發現一個無限迴圈。這就意味著0.85是一個無限迴圈的二進位數字(與基數為10的無限迴圈十進位數字相對應)[1]。這裏顯示了這個數的計算模式。在這個模式中，你可以看到$0.85 = 0.11\overline{0110}_2$。因此， $23.85 = 10111.11\overline{0110}_2$。

上面計算的一個重要結論是23.85不可以用有限的比特位來精確表示成二進位數字。 (就像$\frac{1}{3}$不能表示成有限的十進位數字。)正如這一章展示的，C語言中的`float`和`double`變數是以二進位儲存的。因此，類似23.85的數值不能精確地儲存到這些變數中。只能儲存23.85的近似值。

為了簡化硬體，採用固定的格式來儲存浮點數。這種格式採用科學計數法(但是是在二進位中，是2的乘方,不是10)。例如，23.85或$10111.11011001100110\ldots_2$將儲存為：

$$1.0111110110011001 10\ldots \times 2^{100}$$

[1]不要大驚小怪，一個數值在一種數制下是一個無限迴圈，而在另一種數制下可能不是。考慮下 $\frac{1}{3}$，以十進位表示，它是一個無窮數，但是以三進制(基數為3)表示，它就為 0.1_3。

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

图 6.2: 將0.85轉換成二進位

(其中指數(100)是二進位形式)。規範的浮點數有下面的形式：

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

其中1.*sssssssssssss*是有效數而*eeeeeeee*是 指數。

6.1.2 IEEE浮點表示法

IEEE(Institute of Electrical and Electronic Engineers，電氣與電子工程師學會)是一個國際組織，它已經設計了存儲浮點數的特殊的二進位格式。這種格式應用在大多數(但不是全部)現在的電腦上。通常電腦本身的硬體就支援它。例如，Intel的數學輔助運算器(從Pentium開始，就嵌入到所有它的CPU中了)就使用它。IEEE為不同的精度定義了不同的格式：單或雙精度。在C語言中，`float`變數使用單精確度，而`double`變數使用雙精度。

Intel數學輔助運算器使用第三種，更高的精度，稱為 擴展精度。事實上，在數學輔助運算器自身裏的所有資料都是這種格式。當資料從輔助運算器儲存到記憶體中時，將自動轉換成單或雙精度。[2]跟IEEE的浮點雙精度格式相比，擴展精度使用了一種有細微差別的格式，所以將不在這討論。

[2] 有些編譯器的(例如Borland) `long double`類型使用這種擴展精度。但是，其他的編譯器的`double`和`long double`都使用雙精度。(在ANSI C就允許這樣做。)

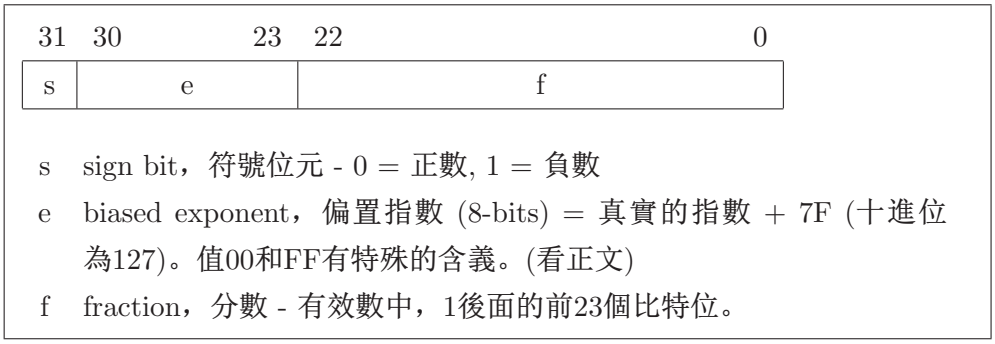

图 6.3: IEEE單精確度

6.1.2.1 IEEE單精確度

單精確度浮點使用32個比特位元來編碼數位。通常它精確到小數點後七位。相比於整數，浮點數的儲存格式更複雜。圖 6.3展示了IEEE單精確度數的基本格式。這種格式有幾個古怪的地方。負的浮點數並不使用補數表示法。它們使用符號量值表示法。如圖顯示，第31位元決定數的符號。

二進位的指數並不會直接儲存。取而代之的是將指數和7F的和儲存到位23 30中。這個 偏置指數總是非負的。

分數部分假定是一個規範的有效數(格式為 1.*sssssssss*)。因為第一個比特位總是1,所以領頭的1是 不儲存的！這就允許在後面儲存一額外的比特位，稍微地擴展了精度。這個想法稱為 隱藏一的表示法.

你必須永遠記住：這些位元組41 BE CC CD可以用不同的方法解釋，使用什麼方法解釋取決於程式如何使用它們。因為，當作為一個單精確度浮點數時，它表示23.850000381，但是當它作為一個雙字整形時，它表示1,103,023,309！CPU並不知道哪種才是正確的解釋！

怎樣儲存23.85呢？ 首先，它是個正數，所以符號位元為0。其次，真實的指數為4,所以偏置指數為7F + 4 = 83_{16}。最後，分數部分應表示為01111101100110011001100 (記住領頭的1是隱藏的)。把這些放到一起得到 (為了幫助澄清浮點格式的不同部分，符號位元和分數部分都加了下劃線，而且所有的比特位都分成了四個比特位一組。)：

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

這不是準確的23.85(因為它是一個無限迴圈的二進位數字)。如果你將上面的數值轉換回十進位形式，你會發現它大約等於 23.849998474。這個數與23.85非常接近，但是它並不準確。實際上，在C語言中，23.85的描述和上面的是一樣的。因為該數的精確描述被截去後的最左邊的位為1,所以最後一個比特位經四捨五入後為1。因此單精確度數23.85將表示成十六進位 41 BE CC CD。將這個轉換成十進位得23.850000381，這個數就更接

$e = 0$ and $f = 0$	表示數0(它不可以被規範化)。注意這兒有+0和-0之分。
$e = 0$ and $f \neq 0$	表示一個非規範數。它們將在下一節中討論。
$e =$ FF and $f = 0$	表示無窮大(∞)，包括正無窮大和負無窮大。
$e =$ FF and $f \neq 0$	表示一個不可以定義的結果，稱為NaN (Not a Number，不是數)。

表 6.1: f和e的特殊值

近23.85。

怎麼描述-23.85呢？只需要改變符號位元得：C1 BE CC CD。不要使用補數！

IEEE浮點格式中，e和f的某些組合有特殊的含義。表 6.1描述了這些特殊的值。溢出或除以0將產生一個無窮數。一個無效的操作將產生一個不確定的結果，例如：試圖求一個負數的平方根，將兩個無窮數相加，等等。

規範的單精確度數的數量級範圍為從 1.0×2^{-126} ($\approx 1.1755 \times 10^{-35}$) 到 $1.11111\ldots \times 2^{127}$ ($\approx 3.4028 \times 10^{35}$)。

6.1.2.2 非規範化數

非規範化數可以用來表示那些值太小了以致於不能以規範格式描述的數(也就是小於1.0×2^{-126})。例如：考慮下數$1.001_2 \times 2^{-129}$ ($\approx 1.6530 \times 10^{-39}$)。在約定的規範格式中，這個指數太小了。但是，它可以用非規範的格式來描述：$0.01001_2 \times 2^{-127}$。為了儲存這個數，偏置指數被置為0(看表 6.1)，而且分數部分是以2^{-127}方式書寫得到的所有有效數 (也就是說儲存了所有的比特位，包括小數點左邊的1).1.001×2^{-129}將表示成：

0 000 0000 0 001 0010 0000 0000 0000 0000

6.1.2.3 IEEE雙精度

IEEE雙精度使用64位元來表示數字，而且通常精確到小數點後15位。如圖 6.4所示，基本的結構和單精確度是非常相似的。只是相比於單精確度，它使用了更多的位來描述偏置指數(11)和分數(52)。

图 6.4: IEEE雙精度

更大範圍的偏置指數會導致兩個後果。一是計算的將是真實指數和3FF(1023)的和(而不是單精確度中的 7F)。二是，允許描述更大範圍的真實的指數(因此也可以描述更大範圍的數量級)。雙精度的數量級範圍大約為從10^{-308}到10^{308}。

雙精度值中增加的有效位元是增大分數欄位的原因。

作為一個例子，再次考慮下。偏置指令用十六進位表示為 $4 + 3FF = 403$。因此，該數用雙精度表示為：

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

或在十六進位中為40 37 D9 99 99 99 99 9A。如果你將它轉換回十進位，你將得到23.8500000000000014 (這有12個0！)，這個數就更接近23.85。

雙精度和單精確度一樣有一些特殊的值[3]。非規範化數同樣也是一樣的。最主要的區別是雙精度的非規範數使用2^{-1023}替換2^{-127}。

第二节 浮點運算

電子電腦裏的浮點運算和持續精確的數學運算是不同的。數學中，所有的數都可以精確表示。但就如前面的章節所示，在電子電腦裏，許多數不能用有限個比特位來描述。所有的計算都在一定的精度下執行。在這節的例子中，為了簡單化，將使用8位的有效數。

6.2.1 加法

要將兩個浮點數相加，它們的指數必須是相等的。如果它們並不相等，那麼通過移動較小指數的數的有效數來使它們相等。例如：考

[3]唯一的區別是：對於無窮數和不確定的值，偏置指數是7FF，而不是FF。

慮$10.375 + 6.34375 = 16.71875$或在十進位中：

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

這兩個數位的指數不一樣，所以通過移動有效數使指數相同，然後再相加：

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

注意，移位丟掉了1.1001011×2^2中的末尾的1,經過四捨五入後得0.1100110×2^3。加法的結果，10.0001100×2^3 (or 1.00001100×2^4)等於 10000.110_2或16.75。這個數並不等於準確的答案 (16.71875)！它只是一個近似值，是在進行加法操作時四捨五入後的應有誤差。

認識到在電子電腦(或計算器)裏的浮點運算得到的結果經常是近似值是非常重要的。對於電子電腦裏的浮點運算，算術法則不總是對的。算術中假定的無窮精度是任何電子電腦都無法做的。例如，算術法則告訴我們$(a + b) - b = a$；但是，在電子電腦裏，並不能完全保證它正確。

6.2.2 減法

減法和加法一樣運作，而且有和加法一樣的問題。作為一個例子，考慮$16.75 - 15.9375 = 0.8125$：

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

移位1.1111111×2^3後得到(四捨五入) 1.0000000×2^4

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

$0.0000110 \times 2^4 = 0.11_2 = 0.75$ 它並不完全正確的。

6.2.3 乘法和除法

對於乘法，有效數執行乘法操作而指數執行相加操作。考慮$10.375 \times 2.5 = 25.9375$：

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

當然，真正的結果需四捨五入成8位，得：

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

除法更複雜，但是也有同樣的四捨五入的誤差問題。

6.2.4 分支程式設計

這一節的重點是浮點運算的結果並不準確。程式師必須意識到這點。一個程式師經常犯的浮點運算錯誤就是在假定一個運算是精確的情況下，用它們去比較。例如，考慮一個執行複雜運算的f(x)函式和一個求這個函式的根的程式[4]。你可能會試圖用下面的語句來檢查x是不是一個根：

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

但是，如果f(x)返回1×10^{-30}又該怎麼辦呢？這個數的最合適的含義是x是一個實根的非常好的近似值。可能沒有一個IEEE浮點值x能恰好返回0，因為f(x)的四捨五入誤差。

一個比較好的方法是使用：

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

其中的EPS是一個宏，定義為一個非常小的正數 (比如說1×10^{-10})。當f(x)非常接近0時，它就為真。一般來說，一個浮點數(譬如 x)和另一個浮點數(y)的比較，需使用：

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

[4]函式的根是滿足$f(x) = 0$條件的x值

第三节 數字輔助運算器

6.3.1 硬體

早期的Intel處理器並沒有提供支援浮點操作的硬體。這並不意味著它們不可以執行浮點操作。它僅僅表示它們需要通過由許多非浮點指令組成的程式來執行這些操作。對於早期的系統，Intel提供了一片額外的稱為數學輔助運算器的晶片。相比於使用軟體程式，數學輔助運算器擁有能快速執行許多浮點操作的機器指令(在早期的處理器上，至少快10倍！)。8086/8088的輔助運算器為8087。80286的輔助運算器為80287，80386的為80387。80486DX處理器將數學輔助運算器內置到80486中了。[5]從Pentium開始，所有生產的80x86
處理器都內置數學輔助運算器；但是，它依然被規劃成好像它是一個分離的單元。即使是早期沒有輔助運算器的系統都可以安裝一個類比數學輔助運算器的軟體。當一個程式執行了一條輔助運算器指令時，這個類比套裝軟體將自動啟動並執行一個軟體程式來得到與真實輔助運算器一樣的結果(雖然，毫無疑問，會比較慢)。

數學輔助運算器有八個浮點數寄存器。每個寄存器儲存著80位元的資料。在這些寄存器中，浮點數總是儲存成80位元的擴展精度。這些寄存器稱為ST0，ST1，ST2，... ST7。浮點寄存器與主CPU中的整形寄存器的使用方法是不同的。浮點寄存器被當作一個堆疊來管理。回想一下堆疊是一個後進先出 (LIFO)佇列。ST0總是指向堆疊頂的值。所有新的數都被加入到堆疊頂中。已經存在的數被壓入到堆疊中，為了為新來的數提供空間。

在數學輔助運算器中同樣有一個狀態寄存器。它有幾個標誌位元。只有4個用來比較的標誌位元將會提到：C_0, C_1, C_2 and C_3。這些位的使用將在以後討論。

6.3.2 指令

為了很容易地將普通的CPU指令和輔助運算器指令區分開來，所有的輔助運算器助詞符都是以F開頭。

[5]但是，80486SX並沒有內置數學輔助運算器。這些機器有分離的80487SX晶片。

6.3.2.1 導入和儲存

用來將資料導入到輔助運算器寄存器堆疊頂的指令有幾條：

FLD source 從記憶體導入一個浮點數到堆疊頂。 source可以是單，雙或擴展精度數或是一個輔助運算器寄存器。

FILD source 從記憶體中讀出一個整形數，將它轉換成浮點數，再將結果儲存到堆疊頂。source可以是字，雙字或四字。

FLD1 將1儲存到堆疊頂。

FLDZ 將0儲存到堆疊頂。

將堆疊中的資料儲存到記憶體的指令同樣也有幾條。其中有幾條指令當它們儲存好一個數後，會將這個數從堆疊中彈出(也就是刪除)。

FST dest 將堆疊頂的值(ST0)儲存到記憶體中。 dest可以是單，雙精度數或是一個輔助運算器寄存器。

FSTP dest 像FST一樣，將堆疊頂的值儲存到記憶體中；但是，當儲存完這個數後，它的值將被彈出堆疊。 dest可以是單，雙或擴展精度數或是一個輔助運算器寄存器。

FIST dest 將堆疊頂的值轉換成整形後再儲存到記憶體中。dest可以是字或雙字。堆疊本身的值是不改變的。浮點數如何轉換成整形取決於輔助運算器的控制字中的某些比特位。這是一個特殊的(非浮點)字寄存器，用來控制輔助運算器如何工作。缺省情況下，控制字會被初始化，以便於當需要轉換成整形時，它會四捨五入成最接近的整形數。但是， FSTCW (Store Control Word，儲存控制字)和FLDCW (Load Control Word，導入控制字)指令可以用來改變這種行為。

FISTP dest 它和FIST是一樣，除了兩件事：堆疊頂的值會被彈出，dest同樣可以是四字的。

同樣有兩條其他的指令用來從堆疊自身中移動或刪除資料。

FXCH STn 將堆疊中的ST0的值和STn的值相互交換 (其中n是一個從1到7的寄存器號)。

FFREE STn 通過標記寄存器為未被使用或為空來釋放堆疊中的一個寄存器。

```
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         ; 移動到下個雙字
12       loop   lp
13       fstp   qword sum      ; 將結果儲存到sum中
```

图 6.5: 陣列求和的例子

6.3.2.2 加法和減法

每一條加法指令都是計算ST0和另一個運算元的和。結果總是儲存到一個輔助運算器寄存器中。

FADD src	ST0 += src。src可以是任何輔助運算器寄存器或記憶體中的單或雙精度數。
FADD dest, ST0	dest += ST0。 dest可以是任何輔助運算器寄存器。
FADDP dest or FADDP dest, ST0	dest += ST0然後再被彈出堆疊。dest可以是任何輔助運算器寄存器。
FIADD src	ST0 += (float) src。ST0和一個整形相加。 src必須是記憶體中的字或雙字。

減法指令是加法指令的兩倍，因為在減法中，運算元的次序是重要的。(也就是說， $a+b=b+a$，但是，$a-b\neq b-a$！)。對於每一條指令，都有一條跟它次序相反的反向指令。這些反向指令要都是以R或RP結尾。圖6.5展示了一小段代碼：對一個雙字陣列的元素求和。在第10和第13行中，你必須指定記憶體運算元的大小。否則彙編器將不會知道記憶體運算元是一個單精確度浮點數(雙字)還是雙精度數(四字)。

FSUB src	ST0 -= src。src可以是任何輔助運算器寄存器或記憶體中單，雙精度數。
FSUBR src	ST0 = src - ST0。src可以是任何輔助運算器寄存器或記憶體中單，雙精度數。
FSUB dest, ST0	dest -= ST0。dest可以是任何輔助運算器寄存器。
FSUBR dest, ST0	dest = ST0 - dest。dest可以是任何輔助運算器寄存器。
FSUBP dest or FSUBP dest, ST0	dest -= ST0然後被彈出堆疊。dest可以是任何輔助運算器寄存器。
FSUBRP dest or FSUBRP dest, ST0	dest = ST0 - dest然後被彈出堆疊。dest可以是任何輔助運算器寄存器。
FISUB src	ST0 -= (float) src。用ST0減去一個整數。 src必須是記憶體中的一個字或雙字。
FISUBR src	ST0 = (float) src - ST0。 用一個整數減去ST0。src必須是記憶體中的一個字或雙字。

6.3.2.3 乘法和除法

乘法指令和加法指令完全類似。

FMUL src	ST0 *= src。src可以是任何輔助運算器寄存器或記憶體中的單或雙精度數。
FMUL dest, ST0	dest *= ST0。dest可以是任何輔助運算器寄存器。
FMULP dest or FMULP dest, ST0	dest *= ST0然後被彈出堆疊。dest可以是任何的輔助運算器寄存器。
FIMUL src	ST0 *= (float) src。ST0與一個整數相乘。 src必須是記憶體中的一個字或雙字。

不要驚訝，除法指令和減法指令非常類似。除以0結果將是一個無窮數。

FDIV src	ST0 /= src。src可以是任何輔助運算器寄存器或記憶體中的單或雙精度數。
FDIVR src	ST0 = src / ST0。src可以是任何輔助運算器寄存器或記憶體中的單或雙精度數。
FDIV dest, ST0	dest /= ST0。dest可以是任何輔助運算器寄存器。
FDIVR dest, ST0	dest = ST0 / dest。dest可以是任何輔助運算器寄存器。
FDIVP dest or FDIVP dest, ST0	dest /= ST0然後被彈出堆疊。dest可以是任何輔助運算器寄存器。
FDIVRP dest or FDIVRP dest, ST0	dest = ST0 / dest然後被彈出堆疊。dest可以是任何輔助運算器寄存器。
FIDIV src	ST0 /= (float) src。ST0除以一個整數。 src必須是記憶體中的一個字或雙字。
FIDIVR src	ST0 = (float) src / ST0。一個整數除以ST0。The src必須是記憶體中的一個字或雙字。

6.3.2.4 比較

輔助運算器同樣能執行浮點數的比較操作。 FCOM家族的指令就是執行比較操作的。

FCOM src	比較ST0和src。src可以是輔助運算器寄存器或記憶體中的單或雙精度數。
FCOMP src	比較ST0和src，然後再彈出堆疊。src可以是輔助運算器寄存器或記憶體中的單或雙精度數。
FCOMPP	比較ST0和ST1，然後執行兩次出堆疊操作。
FICOM src	比較ST0和(float) src。src可以是記憶體中的一個整形字或整形雙字。
FICOMP src	比較ST0和(float)src，然後再彈出堆疊。src可以是記憶體中的一個整形字或整形雙字。
FTST	比較ST0和0。

這些指令會改變輔助運算器狀態寄存器中的C_0，C_1，C_2和C_3比特位的值。不幸的是，CPU直接訪問這些位是不可能的。條件分支指令使用FLAGS寄存器，而不是輔助運算器中的狀態寄存器。但是，使用幾條新

```
1  ;      if ( x > y )
2  ;
3         fld     qword [x]         ; ST0 = x
4         fcomp   qword [y]         ; 比較ST0和y
5         fstsw   ax                ; 將C狀態標誌位元儲存到FLAGS中
6         sahf
7         jna     else_part         ; 如果x不大於y，則跳轉到else_part
8  then_part:
9         ; then部分的代碼
10        jmp     end_if
11 else_part:
12        ; else部分的代碼
13 end_if:
```

图 6.6: 比較指令的例子

的指令可以相當容易地將狀態字的比特位元傳遞到FLAGS寄存器上相同的比特位中。

FSTSW dest	存儲輔助運算器狀態字到記憶體的一個字或AX寄存器中。
SAHF	將AH寄存器中的值儲存到FLAGS寄存器中。
LAHF	將FLAGS寄存器中的比特位導入到AH寄存器中。

圖 6.6展示了一小段樣例代碼。第5行和第6行將C_0，C_1，C_2和C_3比特位傳遞到FLAGS寄存器相同的比特位中了。傳遞了這些比特位，所以它們就類似於兩個無符號整形的比較結果。這也是為什麼第7行使用JNA指令的緣故。

Pentium處理器(和它以後的處理器(Pentium II and III))支援兩條新比較指令，用來直接改變CPU中FLAGS寄存器的值。

FCOMI src	比較ST0和src。src必須是一個輔助運算器寄存器。
FCOMIP src	比較ST0和src，然後再彈出堆疊。src必須是一個輔助運算器寄存器。

圖 6.7展示了一個子程式例子：使用FCOMIP指令來找出兩個雙精度數的較大值。不要把這些指令和整形比較函式(FICOM 和FICOMP)混起來。

6.3.2.5 雜項指令

這一節包括了輔助運算器提供的其他雜項指令。

FCHS	ST0 = - ST0改變ST0的符號位元
FABS	ST0 = |ST0| 求ST0的絕對值
FSQRT	ST0 = $\sqrt{\text{ST0}}$ 求ST0的平方根
FSCALE	ST0 = ST0 × $2^{\lfloor \text{ST1} \rfloor}$ 快速執行ST0乘以2的幾次方的操作。ST1並不會從輔助運算器堆疊中移除。圖 6.8展示了一個如何使用這些指令的例子。

6.3.3 樣例

6.3.4 二次方程求根公式

第一個例子展示了如何用組合語言編寫二次方程求根公式。回憶一下如何用求根公式計算二次方程等式的根：

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

公式本身給出兩個根x：x_1和x_2。

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

在平方根$(b^2 - 4ac)$裏的運算式稱為 判別式。這個值在判定是下面三種可能的根的情況中的哪一種時非常有用。

1. 只有一個實根。當$b^2 - 4ac = 0$時
2. 有兩個實根。當$b^2 - 4ac > 0$時
3. 有兩個複根。當$b^2 - 4ac < 0$時

這是一個使用彙編副程式的小的C程式：

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

彙編副程式如下：

quad.asm

```
1  ; 函式 quadratic
2  ; 求二次等式的根:
3  ;       a*x^2 + b*x + c = 0
4  ; C函式原型:
5  ;   int quadratic( double a, double b, double c,
6  ;                  double * root1, double *root2 )
7  ; 參數:
8  ;   a, b, c - 二次等式中各次方的系統(看上面)
9  ;   root1   - 指向存儲第一個根的雙精度變數的指標
10 ;   root2   - 指向存儲第二個根的雙精度變數的指標
11 ; 返回值:
12 ;   如果存在實根, 則返回1,否則返回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         ; 分配兩個雙精度數大小的空間(disc & one_over_2a)
31         push    ebx             ; 必須保存原始的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, 則沒有實根
46         fsqrt                   ; stack: sqrt(b*b - 4*a*c)
47         fstp    disc            ; 儲存然後再彈出堆疊
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]     ; 儲存到*root2中
66          mov     eax, 1          ; 返回值為1
67          jmp     short quit
68
69  no_real_solutions:
70          mov     eax, 0          ; 返回值為0
71
72  quit:
73          pop     ebx
74          mov     esp, ebp
75          pop     ebp
76          ret
```

quad.asm

6.3.5 從檔中讀數組

在這個例子中，有一個從檔中讀取雙精度數的組合語言程式。這是一個簡短的C測試程式：

readt.c

```
1  /*這個程式用來測試32位元的read_doubles()組合語言程式。它從stdin中讀
   取雙精度數。(使用重定向從檔中讀取。)*/
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

這是組合語言程式：

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 ; 函式 _read_doubles
16 ; C函式原型:
17 ;   int read_doubles( FILE * fp, double * arrayp, int array_size );
18 ; 這個函式從一個文字檔案中讀取雙精度數，並將它們儲存到一個陣列裏，直到遇到
19 ; EOF或陣列滿了。
20 ; 參數:
21 ;   fp         - 指向需要讀取的檔的指標(必須允許輸入)
22 ;   arrayp     - 指向寫入的雙精度陣列的指標
23 ;   array_size - 陣列的元素個數
24 ; 返回值:
25 ;   儲存到陣列中的雙精度數的個數(保存在EAX中)
26
27 _read_doubles:
28         push    ebp
29         mov     ebp,esp
30         sub     esp, SIZEOF_DOUBLE      ; 在堆疊中定義一個雙精度數
31
32         push    esi                     ; 保存esi
33         mov     esi, ARRAYP             ; esi = ARRAYP
34         xor     edx, edx                ; edx = 陣列的下標(最開始為0)
35
36 while_loop:
37         cmp     edx, ARRAY_SIZE         ; edx < ARRAY_SIZE?
38         jnl     short quit              ; 如果不是，退出迴圈
39 ;
40 ; 調用fscanf()函式讀一個雙精度數到TEMP_DOUBLE中
41 ; fscanf()會改變edx，所以需要保存它
42 ;
```

```
43          push    edx                     ; 保存edx
44          lea     eax, TEMP_DOUBLE
45          push    eax                     ; 將&TEMP_DOUBLE壓入堆疊中
46          push    dword format            ; 將&format壓入堆疊中
47          push    FP                      ; 將檔案指針壓入堆疊中
48          call    _fscanf
49          add     esp, 12
50          pop     edx                     ; 恢復edx的值
51          cmp     eax, 1                  ; fscanf函式是否返回1?
52          jne     short quit              ; 如果不是, 則退出迴圈
53
54  ;
55  ; 複製TEMP_DOUBLE到ARRAYP[edx]中
56  ; (8個位元組的雙精度數是通過分成兩個4位元組的數來完成複製的)
57  ;
58          mov     eax, [ebp - 8]
59          mov     [esi + 8*edx], eax      ; 首先複製低4位元組
60          mov     eax, [ebp - 4]
61          mov     [esi + 8*edx + 4], eax  ; 接著複製高4位元組
62
63          inc     edx
64          jmp     while_loop
65
66  quit:
67          pop     esi                     ; 恢復esi
68
69          mov     eax, edx                ; 將返回值儲存到eax中
70
71          mov     esp, ebp
72          pop     ebp
73          ret
```

read.asm

6.3.6 查找素數

最後一個例子又是查找素數的例子。這次的實現方法比以前的方法更有效。它將找到的素數儲存到一個陣列中，而且在查找新的素數時，只除以它已經找到的素數，而不是去除以每一個奇數。

另一個區別是它會計算出猜想的下一個素數的平方根來決定查找因數時應停在哪一個數。它修改了輔助運算器的控制字，所以當它把平方根當作一個整數來儲存時，是通過直接截去來得到整數，而不是四捨五入。這是由控制字中的第10位和第11位來控制的。這些位稱為RC(Rounding Control，四捨五入控制)位。如果這兩位都是0(缺省值)，則輔助運算器轉換成整數時，採用四捨五入的方法。如果是1,則通過直接截去來得到整數。注意：程式必須小心保存好控制字的原始值，當它返回時須恢復它的值。

這是C驅動程式：

fprime.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  /** 函式 find_primes* 查找給定範圍的素數* 參數:* a - 保存素數的陣列* n
   - 找到的素數的個數*/
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

下麵是組合語言程式:

prime2.asm

```
1  segment .text
2          global  _find_primes
3  ;
4  ; 函式 find_primes
5  ; 查找給定範圍的素數
6  ; 參數:
7  ;   array  - 保存素數的陣列
8  ;   n_find - 找到的素數的個數
9  ; C函式原型:
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          ; 目前為止找到的素數的個數
15  %define isqrt         ebp - 8          ; 猜想的下一個素數開平方後得到的整數
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                   ; 為局部變數分配空間
21
22          push    ebx                    ; 保存可能的寄存器變數
23          push    esi
24
25          fstcw   word [orig_cntl_wd]    ; 得到當前控制字
26          mov     ax, [orig_cntl_wd]
27          or      ax, 0C00h              ; 設置RC位為11(截去)
28          mov     [new_cntl_wd], ax
29          fldcw   word [new_cntl_wd]
30
31          mov     esi, [array]           ; esi指向陣列
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  ; 這個外部的迴圈用來查找一個新的素數，新的素數將被加到
38  ; 陣列的末尾。跟以前的查找素數程式不同的是，這個函式
39  ; 並不是通過除以所有的奇數來決定它是不是素數。它僅僅
40  ; 除以已經找到的素數。(這也是為什麼它們
41  ; 被儲存到陣列中的緣故。)
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用來表示陣列的下標
49          push    ebx                     ; 將猜想的素數儲存到堆疊中
50          fild    dword [esp]             ; 將猜想的素數導入到輔助運算器堆疊中
51          pop     ebx                     ; 將猜想的素數移除出堆疊
52          fsqrt                           ; 求sqrt(guess)
53          fistp   dword [isqrt]           ; isqrt = floor(sqrt(quess))
54  ;
55  ; 這個內部的迴圈用猜想的素數(ebx)除以已經找到的素數,
56  ; 直到找到一個猜想的素數的因數(也就意味著這個猜想的素數不是素數),
57  ; 或直到猜想的素數除以的找到的素數大於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                             ; 試下一個素數
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        ; 將猜想的素數加到陣列的末尾
```

```
77          inc     eax
78          mov     [n], eax                        ; inc n
79
80  quit_factor_not_prime:
81          add     ebx, 2                          ; 試下一個奇數
82          jmp     short while_limit
83
84  quit_limit:
85
86          fldcw   word [orig_cntl_wd]             ; 恢復控制字
87          pop     esi                             ; 恢復寄存器變數
88          pop     ebx
89
90          leave
91          ret
```

prime2.asm

```
1  global _dmax
2
3  segment .text
4  ; 函式 _dmax
5  ; 返回兩個參數中的較大的一個
6  ; C函式原型
7  ; double dmax( double d1, double d2 )
8  ; 參數:
9  ;   d1   - 第一個雙精度數
10 ;   d2   - 第二個雙精度數
11 ; 返回值:
12 ;   d1和d2中較大的一個 (儲存在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                 ; 從堆疊中彈出d2
23         fld     qword [d1]          ; ST0 = d1
24         jmp     short exit
25 d2_bigger:                      ; 如果d2不是較大的那個數, 不做任何事
26 exit:
27         leave
28         ret
```

图 6.7: FCOMIP指令的例子

```
1  segment .data
2  x            dq  2.75          ; 轉換成雙精度格式
3  five         dw  5
4
5  segment .text
6        fild   dword [five]       ; ST0 = 5
7        fld    qword [x]          ; ST0 = 2.75, ST1 = 5
8        fscale                    ; ST0 = 2.75 * 32, ST1 = 5
```

图 6.8: FSCALE指令的例子

第七章　結構體與C++

第一节　結構體

7.1.1　簡介

在C語言中的結構體用來將相關的資料集合到一個組合變數中。這項技術有幾個優點：

1. 通過展示定義在結構體內的資料是緊密相聯的來使代碼變得清晰明瞭。

2. 它使傳遞資料給函式變得簡單。代替單獨地傳遞多個變數，它通過傳遞一個單元來傳遞多個變數。

3. 它增加了代碼的 局部性[1] 。

從組合語言的觀點看，結構體可以認為是擁有不同大小的元素的陣列。而真正的陣列的元素的大小和類型總是一樣的。如果你知道陣列的起始位址，每個元素的大小和需要的元素的下標，有這個特性就能計算出這個元素的位址。

結構體中的元素的大小並不一定要是一樣的(而且通常情況下是不一樣的)。因為這個原因，結構體中的每個元素必須清楚地指定而且需要給每個元素一個標號(或者名稱)，而不是給一個數字下標。

在組合語言中，結構體中的元素可以通過和訪問陣列中的元素一樣的方法來訪問。為了訪問一個元素，你必須知道結構體的起始位址和這個元素相對於結構體的相對偏移位址。但是，和陣列不一樣的是：不可以通過元素的下標來計算該偏移位址，結構體的元素的位址需要通過編譯器來賦值。

[1]可以看任何作業系統書中關於虛擬記憶體管理中的討論這個術語的部分。

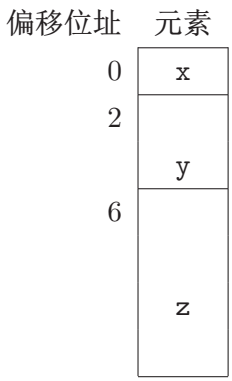

图 7.1: 結構體S

例如，考慮下面的結構體：

```
struct S {
  short int x;     /* 2個位元組的整形 */
  int       y;     /* 4個位元組的整形 */
  double    z;     /* 8個位元組的浮點數 */
};
```

圖 7.1展示了一個S結構體變數在電腦記憶體中是如何儲存的。ANSI C標準規定結構體中的元素在記憶體中儲存的順序和在struct定義中的順序是一樣的。它同樣規定第一個元素需恰好在結構體的起始位址中(也就是說偏移位址為0)。它同樣在stddef.h頭檔中定義了另一個有用的宏offsetof()。這個巨集用來計算和返回結構體中任意元素的偏移位址。這個巨集攜帶兩個參數，第一個是結構體類型的變數名，第二個是需要得到偏移位址的元素名。因此，圖 7.1中的，offsetof(S, y)的結果將是2。

7.1.2 記憶體位址對齊

回想一下一個位址如果除以了4以後，位址是處在雙字界上的。

如果在gcc編譯器中你使用offsetof宏來得到y的偏移地址，那麼它們將找到並返回4，而不是2！為什麼呢？ 因此gcc(和其他許多編譯器)，在缺省情況下，變數是對齊在雙字界上的。在32位元保護模式下，如果資料是從雙字界開始儲存的，那麼CPU能快速地讀取記憶體。圖 7.2展示了如果使用gcc，那麼S結構體在記憶體中是如何儲存的。編譯器在結構體中插入了兩個沒有使用的位元組，用來將y(和z)對齊在雙字界上。這就表明了在C中

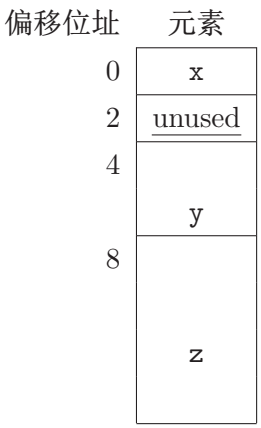

图 7.2: 結構體S

定義的結構體，使用offsetof計算偏移來代替元素自己來計算自己的偏移為什麼是一個好的想法。

當然，如果只是在組合語言程式中使用結構體，程式師可以自己決定偏移位址。但是，如果你需要使用C和彙編的介面技術，那麼在彙編代碼和C代碼中約定好如何計算結構體元素的偏移位址是非常重要的！一個麻煩的地方是不同的C編譯器給出的元素的偏移位址是不同的。例如：就像我們已經知道的， gcc編譯器創建結構體S如圖 7.2；但是，Borland的編譯器將創建結構體如圖 7.1。C編譯器提供了指定資料對齊的方法。但是，ANSI C標準並沒有指定它們該如何完成，因此不同的編譯器使用不同的方法來完成記憶體位址對齊。

gcc編譯器有一個靈活但是複雜的方法來指定位址對齊。它允許你使用特殊的語法來指定任意類型的位址對齊。例如，下麵一行：

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

定義了一個名為unaligned_int的新類型，它採用的是位元組界對齊方式。(是的，所以在__attribute__後面的括弧都是需要的！)aligned的參數1可以用其他的2的乘方值來替代，用來表示採用的是其他對齊方式。(2為字邊界，4表示雙字界， 等等。)如果結構體裏的y元素改為unaligned_int類型，那麼gcc給出的y的偏移位址為2.但是，z依然處在偏移位址8的位置，因為雙精度類型的缺省對齊方式為雙字對齊。要想z的偏移位址為6，那麼還要改變它的類型定義。

gcc編譯器同樣允許你壓縮一個結構體。它告訴編譯器使用盡可能小的

```
struct S {
  short int x;     /* 2個位元組的整形 */
  int       y;     /* 4個位元組的整形 */
  double    z;     /* 8個位元組的浮點數 */
} __attribute__ ((packed));
```

图 7.3: 使用gcc的壓縮結構體

空間來儲存這個結構體。圖 7.3展示了S如何以這種方法來定義。這種形式下的S將使用可能的最少的位元組數，14個位元組。

Microsoft和Borland的編譯器都支援使用#pragma指示符的方法來指定對齊方式。

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

上面的指示符告訴編譯器採用位元組界的對齊方式來壓縮結構體中的元素。(也就是說，沒有額外的填充空間)。其中的1可以用2，4，8或16代替，分別用來指定對齊方式為字邊界，雙字界，四字界和節邊界。這個指示符在被另一個指示符置為無效之前保持有效。這就可能會導致一些問題，因為這些指示符通常使用在頭檔中。如果這個頭檔在包含結構體的其他頭檔之前被包含到程式中，那麼這些結構體的放置方式將和它們缺省的放置方式不同。這將導致非常嚴重的查找錯誤。程式中的不同模組將會將結構體元素放置在不同的地方。

有一個方法來避免這個問題。Microsoft和Borland都支持這個方法：保存當前對齊方式狀態值和隨後恢復它。圖 7.4展示了如何使用這種方法。

7.1.3 位元域s

位元域允許你指定結構體中的成員的大小為只使用指定的比特位數。比特位數的大小並不一定要是8的倍數。一個位域成員的定義和unsigned int或int的成員定義是一樣，只是在定義的後面增加了冒號和位數的大小。圖 7.5展示了一個例子。它定義了一個32位元的變數，它由下面的幾部分組成：

8個比特	11個比特	10個比特	3個比特
f4	f3	f2	f1

```
#pragma pack(push)  /* 保存對齊方式的狀態值 */
#pragma pack(1)     /* 設置為位元組界    */

struct S {
  short int x;     /* 2個位元組的整形 */
  int       y;     /* 4個位元組的整形 */
  double    z;     /* 8個位元組的浮點數   */
};

#pragma pack(pop)   /* 恢復原始的對齊方式 */
```

图 7.4: 使用Microsoft或Borland的壓縮結構體

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

图 7.5: 關於位域的例子

第一個位域被指定到此雙字的最低有效位處。[2]

但是，如果你看了這些比特位元實際上在記憶體中是如何儲存的，你就會發現格式並不是如此簡單。難點發生在當位元域跨越位元組界時。因為在little endian處理器上的位元組將以相反的順序儲存到記憶體中。例如，S結構體在記憶體中將如下所示：

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

f2l變數表示f2位域的末尾五個比特位(也就是，五個最低有效位)。f2m變數表示f2的五個最高有效位。雙垂直線的地方表示位元組界。如果你將所有的

[2]實際上，ANSI/ISO C標準在實際上如何放置比特位方面給了編譯器少許靈活性。但是，普遍的C編譯器(gcc，Microsoft和 Borland)都將像這樣放置比特位域。

位元組 \ 位元	7	6	5	4	3	2	1	0
0	操作碼(08h)							
1	邏輯單元 #				LBA的msb			
2	邏輯塊位址的中間部分							
3	邏輯塊位址的lsb							
4	傳遞的長度							
5	控制字							

图 7.6: SCSI讀命令格式

位元組反向，f2和f3位域將重新結合到正確的位置。

實體記憶體的放置方式通常並不是很重要，除非有資料需要傳送到程式中或從程式中傳出(實際上這和位域是非常相同的)。硬體設備的介面使用奇數的比特位是非常普遍的，此時使用位域來描述是非常有用的。

SCSI[3]就是一個例子。SCSI設備的直接讀命令被指定為傳送一個六個位元組的資訊到設備，格式指定為圖 7.6中的格式。使用位域來描述這個的難點是 邏輯區塊位址(logical block address)，它在此命令中跨越了三個不同的位元組。從圖 7.6中，你可以看到資料是以big endian的格式儲存的。圖 7.7展示了一個試圖在所有編譯器中工作的定義。前兩行定義了一個宏，如何代碼是由Microsoft或Borland編譯器來編譯時，則它就為真。可能比較混亂的部分是11行到14行。首先，你可能會想為什麼lba_mid和 lba_lsb 位域要分開被定義，而不是定義成一個16位的域？原因是資料是以big endian順序儲存的。而編譯器將把一個16位元的域以little endian順序來儲存。其次，lba_msb和 logical_unit 位域看起來似乎方向反了；但是，情況並不是這樣。它們必須得以這樣的順序來擺放。圖 7.8展示了作為一個48位元的實體，它的位元域圖是怎樣的。(位元組界同樣是以雙垂直線來表示。)當它在記憶體中是以little endian的格式來儲存，那麼比特位元將以要求的格式來排列。 (圖 7.6).

考慮得複雜一點，我們知道SCSI_read_cmd的定義在Microsoft C編譯器中不能完全正確工作。如果 sizeof (SCSI_read_cmd)運算式被賦值了，Microsoft C將返回8,而不是6！這是因為Microsoft編譯器使用位域的類型來決定如何繪製比特圖。因為所有的位域都被定義為unsigned類型，所以編

[3]Small Computer Systems Interface，小型電腦系統介面，一個硬碟，等等的工業標準

譯器在結構體的末尾加了兩個位元組使得它成為一個雙字類型的整數。這個問題可以通過用unsignedshort替代所有的位域定義類型來修正。現在，Microsoft編譯器不需要增加任何的填充位元組，因為六個位元組是兩個位元組字類型的整數。[4]有了這個改變，其他的編譯器也能正確工作。圖 7.9展示了另外一種定義，能在所有的三種編譯器上工作。它通過使用unsignedchar避免了除2位的域以外的所有位域的問題。

如果發現前面的討論非常混亂的讀者，請不要氣餒。它本來就是混亂的！通過經常完全地避免使用位域而採用位操作來手動地檢查和修改比特位，作者發現能避免一些混亂。

7.1.4 在組合語言中使用結構體

就像上面討論，在組合語言中訪問結構體就類似於訪問陣列。作為一個簡單的例子，考慮一下你如何寫這樣一個組合語言程式：將0寫入到S結構體的y中。假定這個程式的原型是這樣的：

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

組合語言程式如下：

```
1 %define      y_offset  4
2 _zero_y:
3       enter  0,0
4       mov    eax, [ebp + 8]       ; 從堆疊中得到s_p(結構體的指標)
5       mov    dword [eax + y_offset], 0
6       leave
7       ret
```

C語言允許你把一個結構體當作數值傳遞給函式；但是，通常這都是一個壞主意。當以數值來傳遞時，在結構體中的所有資料都必須複製到堆疊中，然後在程式中再拿出來使用。用一個結構體指標來替代能有更高的效率。

C語言同樣允許一個結構體類型作為一個函式的返回值。很明顯，一個結構體不能通過儲存到EAX寄存器中來返回。不同的編譯器處理這種情況的

[4]混亂的不同類型的位域將導致非常混亂的行為！讀者需要自己去實驗。

方法也不同。一個編譯器普遍使用的解決方法是在內部重寫函式，讓它攜帶一個結構體指標參數。這個指標用來將返回值放入到結構體中，這個結構體是在調用的程式外面定義的。

大多數彙編器(包括NASM)都有在你的彙編代碼中定義結構體的內置支援。查閱你的資料來得到更詳細的資訊。

第二节 組合語言和C++

C++編程語言是C語言的一種擴展形式。許多C語言和組合語言介面的基本規則同樣適用於C++。但是，有一些規則需要修正。同樣，擁有一些組合語言的知識，你能很容易理解C++中的一些擴展部分。這一節假定你已經有一定的C++基礎知識。

7.2.1 重載函式和名字改編

C++允許不同的函式(和類成員函式)使用同樣的函式名來定義。當不止一個函式共用同一個函式名時，這些函式就稱為重載函式。在C語言中，如果定義的兩個函式使用的函式名是一樣，那麼連接器將產生一個錯誤，因為在它連接的目標檔中，一個符號它將找到兩個定義。例如，考慮圖7.10中的代碼。等價的彙編代碼將定義兩個名為_f的標號，而這明顯是錯誤的。

C++使用和C一樣的連接過程，但是通過執行名字改編或修改用來標記函式的符號來避免這個錯誤。在某種程式上，C也早已經使用了名字改編。當創建函式的標號時，它在C函式名上增加了一條下劃線。但是，C語言將以同樣的方法來改編圖 7.10中的兩個函式名，那麼將會產生一個錯誤。C++使用一個更高級的改編過程：為這些函式產生兩個不同的標號。例如：圖 7.10中的第一個函式將由DJGPP指定為標號_f__Fi，而第二個函式，指定為_f__Fd。這樣就避免了任何的連接錯誤。

不幸的是，關於在C++中如何改編名字並沒有一個標準，而且不同的編譯器改編的名字也不一樣。例如，Borland C++將使用標號@f$qi和@f$qd來表示圖 7.10中的兩個函式。但是，規則並不是完全任意的。改編後的名字編碼成函式的簽名。一個函式的簽名是通過它攜帶的參數的順序和類型來

定義的。注意，，攜帶了一個int參數的函式在它的改編名字的末尾將有一個i(對於DJGPP 和Borland都是一樣)，而攜帶了一個double參數的函式在它的改編名字的末尾將有一個d。如果有一個名為f的函式，它的原型如下：

```
void f( int x, int y, double z);
```

DJGPP將會把它的名字改編成_f__Fiid而Borland將會把它改編成 @f$qiid。

函式的返回類型並不是函式簽名的一部分，因此它也不會編碼到它的改編名字中。這個事實解釋了在C++中的一個重載規則。只有簽名唯一的函式才能重載。就如你能看到的，如果在C++中定義了兩個名字和簽名都一樣的函式，那麼它們將得到同樣的簽名，而這將產生一個連接錯誤。缺省情況下，所有的C++函式都會進行名字改編，甚至是那些沒有重載的函式。它編譯一個檔時，編譯器並沒有方法知道一個特定的函式重載與否，所以它將所有的名字改編。事實上，和函式簽名的方法一樣，編譯器同樣通過編碼變數的類型來改編總體變數的變數名。因此，如果你在一個檔中定義了一個總體變數為某一類型然後試圖在另一個檔中用一個錯誤的類型來使用它，那麼將產生一個連接錯誤。C++這個特性被稱為類型安全連接。 它同樣暴露出另一種類型的錯誤：原型不一致。當在一個模組中函式的定義和在另一個模組使用的函式原型不一致時，就發生這種錯誤。在C中，這是一個非常難調試出來的問題。C並不能捕捉到這種錯誤。程式將被編譯和連接，但是將會有未定義的操作發生，就像調用的代碼會將和函式期望不一樣的類型壓入堆疊中一樣。在C++中，它將產生一個連接錯誤。

當C++編譯器語法分析一個函式調用時，它通過查看傳遞給函式的參數的類型來尋找匹配的函式[5]。如果它找到了一個匹配的函式，那麼通過使用編譯器的名字改編規則，它將創建一個CALL來調用正確的函式。

因為不同的編譯器使用不同的名字改編規則，所以不同編譯器編譯的C++代碼可能不可以連接到一起。當考慮使用一個預編譯的C++庫時，這個事實是非常重要的！如果有人想寫出一個能在C++代碼中使用的組合語言程式，那麼他必須知道要使用的C++編譯器使用的名字改編規則(或使用下面將解釋的技術)。

[5]這個匹配並不一定要是精確匹配，編譯器將通過強制轉型參數來考慮匹配。這個過程的規則超出了本書的範圍。查閱一本C++的書來得到更詳細的資訊。

機敏的學生可能會詢問在圖 7.10中的代碼到底能不能如預期般工作。因為C++改編了所有函式的函式名，那麼 printf將被改編，而編譯器將不會產生一個到標號_printf處的CALL調用。這是一個非常正確的擔憂！如果printf的原型被簡單地放置在檔的開始部分，那麼這就將發生。原型為：

```
int  printf ( const char *,  ...);
```

DJGPP將會把它改編為_printf__FPCce。(F表示function，函式，P表示pointer，指針，C表示const，常量， c表示 char而e表示省略號。)那麼它將不會調用正規C庫中的printf函式！當然，必須有一種方法讓C++代碼用來調用C代碼。這是非常重要的，因為到處都有許多非常有用的舊的C代碼。除了允許你調用遺留的C代碼外，C++同樣允許你調用使用了正規的C改編約定的彙編代碼。

C++擴展了extern關鍵字，允許它用來指定它修飾的函式或總體變數使用的是正規C約定。在C++術語中，稱這些函式或總體變數使用了C鏈結。例如，為了聲明printf為C鏈結，需使用下面的原型：

```
extern "C" int  printf ( const char *,  ...  );
```

這就告訴編譯器不要在這個函式上使用C++的名字改編規則，而使用C規則來替代。但是，如果這樣做了，那麼printf將不可以重載。這就提供了一個簡易的方法用在C++和組合語言程式介面上：使用C鏈結定義一個函式，然後再使用C調用約定。

為了方便，C++同樣允許定義函式或總體變數塊的C鏈結。通常函式或總體變數塊用捲曲花括弧表示。

```
extern "C" {
  /* C鏈結的總體變數和函式原型 */
}
```

如果你檢查了當今的C/C++編譯器中的ANSI C頭檔，你會發現在每個頭檔上面都有下面這個東西：

```
#ifdef __cplusplus
extern "C" {
#endif
```

而且在底部有一個包含閉捲曲花括弧的同樣的結構。C++編譯器定義了宏__cplusplus(有兩條領頭的下劃線)。上面的代碼片斷如果用C++來編譯，那麼整個頭檔就被一個extern "C"塊圍起來了，但是如果使用C來編譯，就不會執行任何操作 (因為對於extern "C"，C編譯器將產生一個語法錯誤)。程式師可以使用同樣的技術用來在組合語言程式中創建一個能被C或C++使用的頭檔。

7.2.2 引用

引用是C++的另一個新特性。它允許你傳遞參數給函式，而不需要明確使用指標。例如，考慮圖 7.11中的代碼。事實上，引用參數是非常簡單，實際上它們就是指標。只是編譯器對程式師隱藏它而已(正如Pascal編譯器把var參數當作指針來執行)。當編譯器產生此函式調用的第7行代碼的彙編語句時，它將y的位址傳遞給函式。如果有人是用組合語言書寫的f函式，那麼他們操作的情況，就好像原型如下似的：[6]：

```
void f( int * xp);
```

引用是非常方便的，特別是對於運算符重載來說是非常有用的。運算符重載又是C++的另一個特性，它允許你在對結構體或類類型進行操作時賦予普通運算符另一種功能。例如，一個普遍的使用是賦予加號(+)運算符能將字串物件連接起來的功能。因此，如果a和b是字串，那麼a + b將得到a和b連接後的字串。實際上，C++可以調用一個函式來做這件事(事實上，上面的運算式可以用函式的表示法來重寫為：operator +(a,b))。為了提高效率，有人可能會希望傳遞字串的位址來代替傳遞他們的值。若沒有引用，那麼將需要這樣做：operator +(&a,&b)，但是若要求你以運算符的語法來書寫應為：&a + &b。這是非常笨拙而且混亂的。但是，通過使用引用，你可以像這樣書寫：a + b，這樣就看起來非常自然。

7.2.3 內聯函式

到目前為止，內聯函式又是C++的另一個特性[7]。內聯函式照道理應該可以取代容易犯錯誤的，攜帶參數的，基於預處理程式的巨集。回想一下在C中，書寫一個求數的平方的宏可以是這樣的：

[6]當然，他們可能想使用C鏈結來聲明函式，用來避免名字改編，就像小節 7.2.1中討論的

[7] C編譯器通常支持這種特性，把它當作ANSI C的擴展。

#define SQR(x) ((x)*(x))

因為預處理程式不能理解C而採用簡單的替換操作，在大多數情況下，圓括號裏要求是能正確計算出來的值。但是，即使是這個版本也不能給出SQR(x++)的正確答案。

宏之所以被使用是因為它除去了進行一個簡單函式的函式調用的額外時間開支。就像副程式那一章描述的，執行一個函式調用包括好幾步。對於一個非常簡單的函式來說，用來進行函式調用的時間可能比實際上執行函式裏的操作的時間還要多！內聯函式是一個更為友好的用來書寫代碼的方法，讓代碼看起來象一個標準的函式，但是它並不是CALL指令能調用的普通代碼塊。出現內聯函式的調用運算式的地方將被執行函式的代碼替換。C++允許通過在函式定義前加上inline關鍵字來使函式成為內聯函式。如果，考慮在圖 7.12中聲明的函式。第10行對f的調用將執行一個標準的函式調用(在組合語言中，假定x的地址為ebp-8而y地址為ebp-4)：

```
1       push    dword [ebp-8]
2       call    _f
3       pop     ecx
4       mov     [ebp-4], eax
```

但是，第11行對inline_f的調用將得到如下結果：

```
1       mov     eax, [ebp-8]
2       imul    eax, eax
3       mov     [ebp-4], eax
```

這種情況下，使用內聯函式有兩個優點。首先，內聯函式更快。沒有參數需要壓入堆疊中，也不需要創建和毀壞堆疊幀，也不需要進行分支。其次，內聯函式調用使用的代碼是非常少！後面一點對這個例子來說是正確的，但是並不是在所有情況下都是正確的。

內聯函式的主要優點是內聯代碼不需要連接，所以對於使用內聯函式的所有原始檔案來說，內聯函式的代碼都必須有效。前面的彙編代碼的例子展示了這一點。對於非內聯函式的調用，只要求知道參數，返回值類型，調用約定和函式的函式名。所有的這些資訊都可以從函式的原型中得

到。但是，使用內聯函式調用，就必須知道這個函式的所有代碼。這就意味著如果改變了一個內聯函式中的任何部分，那麼所有使用了這個函式的原始檔案必須重新編譯。回想一下對於非內聯函式，如果函式原型沒有改變，通常使用這個函式的原始檔案就不需要重新編譯。由於所有的這些原因，內聯函式的代碼通常放置在頭檔中。這樣做違反了在C語言中標準的穩定和快速準則：執行的代碼語句決不能放置在頭檔中。

7.2.4 類

C++中的類描述了一個物件類型。一個物件包括資料成員(data member)和函式成員(function member)[8]。換句話說就是，它是由跟它相關聯的資料和函式組成的一個`struct`結構體。考慮在圖 7.13中定義的那個簡單的類。一個`Simple`類型的變數非常類似於包含一個`int`成員的標準C`struct`結構體。這些函式並不會儲存到指定結構體的記憶體中。但是，成員函式和其他函式是不一樣的。它們傳遞了一個隱藏的參數。這個參數是一個指向成員函式能起作用的物件的指標。

事實上，C++使用`this`關鍵字從成員函式內部來訪問指向此函式能起作用的物件的指標。

例如，考慮圖 7.13中的`Simple`類的成員函式`set_data`。如果用C語言來書寫此函式，這個函式將像這樣：明確傳遞一個指向成員函式能起作用的物件的指標，如圖 7.14所示。使用DJGPP編譯器加上`-S`選項(gcc和Borland編譯器也是一樣)來告訴編譯器輸出一個包含此代碼產生的等價的組合語言代碼的原始檔案。對於DJGPP和gcc編譯器，此彙編原始檔案是以`.s`副檔名結尾的，但是不幸的是使用的語法是AT&T組合語言語法，這種語法和NASM和MASM語法區別非常大[9]。(Borland和MS編譯器產生一個以`.asm`副檔名結尾的原始檔案，使用的是MASM語法。) 圖 7.15展示了將DJGPP的輸出轉換成NASM語法後的代碼，增加了闡明語句目的的注釋。在第一行中，注意成員函式`set_data`的函式名被指定為一個改編後的標號，此標號是通過編碼成員函式名，類名和參數後得到的。類名被編碼進去的是因為其他類中可能也有名為`set_data`的成員函式，而這

[8]在C++中，通常稱之為成員函式(member function)或者更為普遍地稱之為方法(method)。

[9]gcc編譯系統包含了一個屬於自己的稱為gas的彙編器。gas彙編器使用AT&T語法，因此編譯器以gas的格式來輸出代碼。網頁中有好幾頁用來討論INTEL和AT&T語法的區別。同時有一個名為`a2i`的免費程式 (http://www.multimania.com/placr/a2i.html)，此程式將AT&T格式轉換成了NASM格式。

兩個成員函式必須使用不同的標號。參數之所以被編碼進去是為了類能通過攜帶其他參數來重載成員函式set_data，正如標準的C++函式。但是，和以前一樣，不同的編譯器在改編標號時編碼資訊的方式也不同。

下面的第2和第3行，出現了熟悉的函式的開始部分。在第5行，把堆疊中的第一個參數儲存到EAX中了。這並不是參數x！替代它的是那個隱藏的參數[10]，它是指向此函式能起作用的物件的指標。第6行將參數x儲存到EDX中了，而第7行又將EDX儲存到了EAX指向的雙字中。它是Simple物件中的data成員，也是這個類中的唯一的資料，它儲存在Simple結構體中偏移位址為0的地方。

7.2.4.1 樣例

這一節使用了這章中的思想創建了一個C++類：用來描述任意大小的無符號整形。因為要描述任意大小的整形，所以它需要儲存到一個無符號整形的陣列(雙字的)中。可以使用動態分配來實現任意大小的整形。雙字是以相反的方向儲存的[11] (也就是說，雙字的最低有效位的下標為0)。圖 7.16展示了Big_int類的定義[12]。Big_int的大小是通過測量unsigned陣列的大小得到的，用來儲存它的資料。此類中的size_資料成員的偏移位址為0，而number_成員的偏移為4。

為了簡單化這些例子，只有擁有大小相同的陣列的物件實例才可以相互進行加減操作。

這個類有三個構造函式(constructor)：第一個構造函式(第9行)使用了一個正常的無符號整形來初始化類實例；第二個構造函式(第18行)使用了一個包含一個十六進位值的字串來初始化類實例。第三個構造函式(第21行)是拷貝構造函式(copy constructor)。

因為這裏使用的是組合語言，所以討論的焦點在於加法和減法運算符如何工作。圖 7.17展示了與這些運算符相關的部分頭檔。它們展示了如何創建運算符來調用組合語言程式。因為不同的編譯器使用完全不同的名字改編規則來改編運算符函式，所以創建了內聯的運算符函式來調用C鏈結組合語言程式。這就使得在不同編譯器間的移植變得相對容易些，而且和直接調用速度一樣快。這項技術同樣免去了從彙編中拋出異常的必要！

[10] 像平常一樣，沒有東西能隱藏在彙編代碼中！
[11] 為什麼呢？因為加法運算將從陣列的開始處開始逐漸向前進行操作。
[12] 查閱樣例源代碼來得到這個例子的全部的代碼。本文中將只引用部分代碼。

為什麼在這裏使用的全部是組合語言呢？回想一下，在執行多倍精度運算時，進位必須從一個雙字移去與下一個有效的雙字進行加法操作。C++(和 C)並不允許程式師訪問CPU的進位元標誌位元。只有通過讓C++獨立地重新計算出進位元標誌位元後有條件地與下一個雙字進行加法操作，才能執行這個加法操作。使用組合語言來書寫代碼會更有效，因為它可以訪問進位元標誌位元，可以使用ADC指令來自動將進位元標誌位元加上，這樣做更有道理。

為了簡化，只有add_big_ints的組合語言程式將在這討論。下面是這個程式的代碼(來自big_math.asm)：

big_math.asm

```
1  segment .text
2          global  add_big_ints, sub_big_ints
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 ; 加法和減法程式的參數
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         ; 首先設置: 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          ; 要保證所有3個Big_int類型數有同樣的大小
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          ; 現在, 讓寄存器指向它們各自的陣列
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                                     ; 清進位元標誌位元
49          xor     edx, edx                        ; edx = 0
50          ;
51          ; 加法迴圈
52  add_loop:
53          mov     eax, [edi+4*edx]
54          adc     eax, [esi+4*edx]
55          mov     [ebx + 4*edx], eax
56          inc     edx                             ; 不要改變進位元標誌位元
```

```
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，到此刻為止讀者能明白大部分這裏的代碼。第25行到27行將Big_int物件傳遞的指標儲存到寄存器中。記住引用的僅僅是指針。第31行到35行檢查保證三個物件陣列的大小是一樣的。(注意， size_的偏移被加到指針中了，為了訪問資料成員。)第44行和第46行調整寄存器，讓它們指向被各自物件使用的陣列，用來替代使用物件本身。(同樣，number_的偏移被加到物件指標中了。)

在第52行到57行的迴圈中，將儲存在陣列裏的整形一起相加，首先加的是最低有效的雙字，然後是下一最低有效的雙字，等等。多倍精度運算必須以這樣的順序來完成(看小節 2.1.5)。第59行用來檢查溢出，一旦溢出，進位元標誌位元將由最後進行加法運算的最高有效位置位。因為陣列裏的雙字是以little endian順序儲存的，所以迴圈從陣列的開始處開始，依次向前直到結束。

圖 7.18展示了Big_int的簡單應用的簡短的例子。注意，Big_int常量必須明確聲明，如第16行。這有兩個原因。首先，沒有轉換構造函式來將一個無符號整形轉換成Big_int類型。其次，只有相同大小的Big_int數才

能用來進行相加操作。這裏進行類型轉換是有問題的，因為要知道需轉換的大小是非常困難的。此類的一個更高級的實現將允許任意大小的數之間的相加。作者不打算因為要實現任意大小的數的相加而把這個例子弄得過度複雜。(但是，鼓勵讀者來實現它。)

7.2.5 繼承和多態

繼承(Inheritance)允許一個類繼承另一個類的資料和成員函式。例如，考慮圖 7.19中的代碼。它展示了兩個類，A和B，其中類B是通過繼承類A得到的。程式的輸出如下：

```
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，兩個類的資料成員ad(B通過繼承A得到的)在相同的偏移處。這是非常重要的，因為f函式將傳遞一個指標到一個 A物件或任意一個由A派生(也就是，通過繼承得到)的物件類型中。圖 7.20展示了此函式的(編輯過的)彙編代碼(gcc得到的)。

注意在輸出中，a和b對象調用的都是A的成員函式m。從組合語言程式中，我們可以看到對A::m()的調用被硬編碼到函式中了。對於真正的面向物件編程，成員函式的調用取決於傳遞給函式的物件類型是什麼。這就是所謂的多態。缺省情況下，C++關掉了這個特性。你可以使用virtual 關鍵字來啟動它。圖 7.21展示了如何修改這兩個類。其他代碼不需要修改。多態可以用許多方法來實現。不幸的是，當在以這種方法書寫的時候，gcc的實現方法正處在改變中，而且與它最初的實現方法相比，明顯變得更複雜了。為了簡單化討論的目的，作者只涉及基於Microsoft和Borland編譯器Windows使用的多態的實現方法。這種實現方法很多年沒有改變了，而且可能在未來幾年也不會改變。

有了這些改變，程式的輸出如下：

```
Size of a: 8 Offset of ad: 4
Size of b: 12 Offset of ad: 4 Offset of bd: 8
A::m()
```

```
B::m()
```

現在，對f的第二次調用調用了B::m()的成員函式，因為它傳遞了物件B。但是，這並不是唯一的修改的地方。A的大小現在為8(而B為12)。同樣，ad的偏移為4,不是0。在偏移0處是的什麼呢？這個問題的答案與如何實現多態相關。

含有任意虛成員函式的C++類有一個額外的隱藏的域，它是一張指向成員函式指標陣列的指標表。[13]。這個表通常稱為vtable。對於 A和B類，指標表儲存在偏移位址0處。Windows編譯器總是把此指針表放到繼承樹頂部的類的開始處。從擁有虛成員函式的程式版本(源自圖 7.19)中的f函式產生的彙編代碼(圖 7.22)中，你可以看到對成員函式m的調用不是使用一個標號。第9行來查找物件的vtable的位址。物件的位址在第11行中被壓入堆疊。第12行通過分支到vtable裏的第一個位址處來調用虛成員函式。 [14]。這次調用並不使用一個標號，它分支到EDX指向的代碼位址處。這種類型的調用是一個 晚綁定(late binding)的例子。晚綁定將調用哪個成員函式的判定延遲到代碼運行時。這就允許代碼為物件調用恰當的成員函式。標準的案例(圖 7.20)硬編碼某個成員函式的調用，也稱為早綁定(early binding) (因為這兒成員函式被早綁定了，在編譯的時候。)。

用心的讀者將會覺得奇怪為什麼在圖 7.21中的類的成員函式通過使用__cdecl關鍵字來明確聲明使用的是C調用約定。缺省情況下，Microsoft對於C++類成員函式使用的是不同的調用約定，而不是標準C調用約定。此調用約定將指向成員函式能起作用的物件的指標傳遞到ECX寄存器，而不是使用堆疊。成員函式的其他明確的參數仍然使用堆疊。修改為__cdecl告訴編譯器使用標準C調用約定。Borland C++缺省情況下使用的是C調用約定。

下面我們再看一個稍微複雜一點的例子。 (圖 ??)。在這個例子中，類A和B都有兩個成員函式：m1和m2。記住因為類B並沒有定義自己的成員函式m2，它繼承了A類的成員函式。圖 7.24展示了物件b在記憶體中如何儲存。圖 7.25展示了此程式的輸出。首先，看看每個物件的vtable的位

[13]對於沒有虛成員函式的類，C++編譯器通過一個包含同樣資料成員的標準C結構體來對這種類進行相容。

[14]當然，這個值已經在ECX寄存器中了。它是在第8行放置到該寄存器的，並且可以移除第10行，再把下一行改變為push ECX。這些代碼並不十分有效，因為它是在沒有開啟優化編譯選項的情況下產生的。

址。兩個B對象的vtable位址是一樣的，因此他們共用同樣的vtable。一張vtable表是類的屬性而不是一個物件(就如一個static資料成員)。其次，看看在vtable裏的地址。從組合語言程式的輸出中，你可以確定成員函式m1指標在偏移位址 0處 (或雙字 0)而m2在偏移位址 4處(雙字 1)。m2成員函式指標在類A和B的vtable中是一樣的，因為類B從類A繼承了成員函式m2。

第25行到32行展示了你可以通過從物件的vtable讀位址的方法來調用一個虛函式[15]。成員函式位址通過一個清楚的this指標儲存到了一個C類型函式指標中了。從圖 7.25的輸出中，你可以看到它確實可以運行。但是，請不要像這樣寫代碼！這只是用來舉例說明虛成員函式如何使用vtable。

從這裏我們可以學到一些實踐的教訓。一個重要的事實是當你讀或寫類變數到一個二進位原始檔案中時，你必須非常小心。你不可以在整個物件中僅僅使用一個二進位讀或寫，因為可能會讀或寫原始檔案之外的vtable指針！這是一個指向留在程式記憶體中的vtable的指標，而且不同的程式將不同。 同樣的問題會發生在C語言的結構中，但是在C語言中，結構體只有當程式師明確將指標放到結構體中時，結構體內部才有指標。類A或類B中，並沒有明顯地定義過指針。

再次，認識到不同的編譯器實現虛成員函式的方法是不一樣的是非常重要的。在In Windows中，COM(元件物件模型，Component Object Model) 類物件使用vtable來實現COM介面[16]。只有像Microsoft一樣用來實現虛成員函式的編譯器才可以創建COM類。這也是為什麼Borland採用和Microsoft一樣的實現方法的原因，也是為什麼不可以用gcc來創建COM類的原因之一。

虛成員函式的代碼和非常虛的成員函式的代碼非常相像。只是調用它們的代碼是不同的。如果彙編器能絕對保證調用哪個虛成員函式，那麼它可以忽略vtable，直接調用成員函式。(例如，使用早綁定)。

7.2.6 C++的其他特性

C++其他特性的工作方式(例如，除了處理繼承和多繼承，還有運行時類型識別)不屬於本書的範圍。如果讀者希望走得更遠一些，一個好的起點是Ellis 和Stroustrup寫的The Annotated C++ Reference Manual和Stroustrup寫的The Design and Evolution of C++。

[15]記住這些代碼只能在MS和Borland編譯器下運行，gcc不行。
[16]COM類同樣使用__stdcall 調用約定，而不是標準C調用約定。

```
#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;     /* 中間的比特位 */
  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讀命令格式的結構

8個比特	8個比特	8個比特	8個比特	3個比特	5個比特	8個比特
控制字	傳輸的長度	lba_lsb	lba_mid	logical_unit	lba_msb	opcode

图 7.8: `SCSI_read_cmd`的位域圖

```
1  struct SCSI_read_cmd {
2    unsigned char opcode;
3    unsigned char lba_msb : 5;
4    unsigned char logical_unit : 3;
5    unsigned char lba_mid;     /* 中間的比特位 */
6    unsigned char lba_lsb ;
7    unsigned char transfer_length ;
8    unsigned char control ;
9  }
10 #if defined(__GNUC__)
11     __attribute__ ((packed))
12 #endif
13 ;
```

图 7.9: 另一種SCSI讀命令格式的結構

```
1  #include <stdio.h>
2
3  void f( int x )
4  {
5    printf("%d\n", x);
6  }
7
8  void f( double x )
9  {
10   printf("%g\n", x);
11 }
```

图 7.10: 兩個名為f()的函式

```
1  void f( int & x )      // &表示是一個引用參數 { x++; }
2
3  int main()
4  {
5    int y = 5;
6    f(y);                // 傳遞了引用y，注意這裏沒有&!
7    printf ("%d\n", y);  // 顯示6!
8    return 0;
9  }
```

图 7.11: 引用的例子

```
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: 內聯函式的例子

```
1  class Simple {
2  public:
3    Simple();                  // 缺省的構造函式
4    ~Simple();                 // 析構函式
5    int get_data() const;      // 函式成員
6    void set_data( int );
7  private:
8    int data;                  // 資料成員
9  };
10
11 Simple::Simple()
12 { data = 0; }
13
14 Simple::~Simple()
15 { /* 空程式體 */ }
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: 一個簡單的C++類

```
void set_data( Simple * object, int x )
{
  object->data = x;
}
```

图 7.14: Simple::set_data()的C版本

```
1  _set_data__6Simplei:            ; 改編後的名字
2        push   ebp
3        mov    ebp, esp
4
5        mov    eax, [ebp + 8]    ; eax = 指向對象的指標(this)
6        mov    edx, [ebp + 12]   ; edx = 整形參數
7        mov    [eax], edx        ; data在偏移位址0處
8
9        leave
10       ret
```

图 7.15: 編譯Simple::set_data(int)的輸出

```
1  class Big_int {
2  public:
3     /*
4     * Parameters:
5     *     size              – 表示成正常無符號整數的整形大小
6     *
7     *     initial_value     – 將Big_int的值初始化為一個正常的無符號整形
8     */
9     explicit Big_int( size_t     size,
10                      unsigned initial_value = 0);
11    /*
12    * Parameters:
13    *     size              – 表示成正常無符號整數的整形大小
14    *
15    *     initial_value     – 將Big_int的值初始化為一個包含一個以十六進位
表示的值的字串
16    *
17    */
18    Big_int( size_t           size,
19             const char *  initial_value );
20
21    Big_int( const Big_int & big_int_to_copy );
22    ~Big_int();
23
24    // 返回Big_int的大小 (以無符號整形的形式)
25    size_t  size() const;
26
27    const Big_int & operator = ( const Big_int & big_int_to_copy );
28    friend Big_int operator + ( const Big_int & op1,
29                                const Big_int & op2 );
30    friend Big_int operator – ( const Big_int & op1,
31                                const Big_int & op2);
32    friend bool operator == ( const Big_int & op1,
33                              const Big_int & op2 );
34    friend bool operator < ( const Big_int & op1,
35                             const Big_int & op2);
36    friend ostream & operator << ( ostream &      os,
37                                   const Big_int & op );
38 private:
39    size_t        size_;     // 無符號陣列的大小
40    unsigned * number_; // 指向擁有數值的無符號陣列的指標
```

```
1  // 組合語言程式的原型
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類的算術代碼

```
#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的簡單應用

```
1  #include <cstddef>
2  #include <iostream>
3  using namespace std;
4
5  class A {
6  public:
7    void __cdecl m() { cout << "A::m()" << endl; }
8    int ad;
9  };
10
11 class B : public A {
12 public:
13   void __cdecl m() { cout << "B::m()" << endl; }
14   int bd;
15 };
16
17 void f( A * p )
18 {
19   p->ad = 5;
20   p->m();
21 }
22
23 int main()
24 {
25   A a;
26   B b;
27   cout << "Size of a: " << sizeof(a)
28        << " Offset of ad: " << offsetof(A,ad) << endl;
29   cout << "Size of b: " << sizeof(b)
30        << " Offset of ad: " << offsetof(B,ad)
31        << " Offset of bd: " << offsetof(B,bd) << endl;
32   f(&a);
33   f(&b);
34   return 0;
35 }
```


图 7.19: 簡單繼承

```
1  _f__FP1A:                          ; 改編後的函式名
2        push   ebp
3        mov    ebp, esp
4        mov    eax, [ebp+8]          ; eax指向對象
5        mov    dword [eax], 5        ; ad的偏移為0
6        mov    eax, [ebp+8]          ; 將物件的位址傳遞給A::m()
7        push   eax
8        call   _m__1A                ; A::m()改編後的成員函式名
9        add    esp, 4
10       leave
11       ret
```

图 7.20: 簡單繼承的彙編代碼

```
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: 多態繼承

```
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               ; 將"this"指針壓入堆疊中
12       call   dword [edx]       ; 調用在vtable裏的第一個程式
13       add    esp, 4            ; 清理堆疊
14
15       pop    ebp
16       ret
```

图 7.22: f()函式的彙編代碼

```
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: 圖 ??中程式的輸出

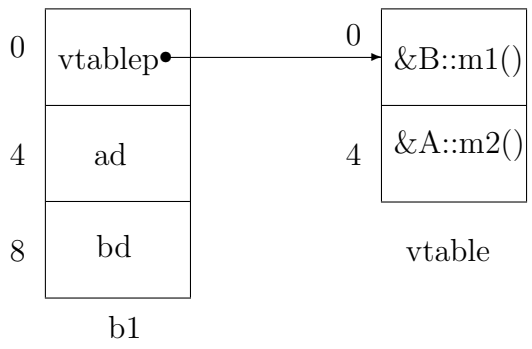

图 7.24: b1的內部表示

附件一　80x86指令

第一节　非浮點指令

這一節列出和描述了Intel 80x86CPU家族的非浮點指令的行為和格式。

這些格式使用下面的約定：

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	記憶體
M8	位元組
M16	字
M32	雙字
I	立即數

上面這些可以結合使用於多運算元指令。例如：格式R, R表示指令攜帶兩個寄存器器運算元。許多雙運算元指令允許同樣類型的運算元。約定O2可以用來表示這些運算元：R,R R,M R,I M,R M,I。如果一個運算元可以是8位元的寄存器或記憶體，則使用這樣的約定： R/M8。

這個表同樣展示了每一條指令如何影響FLAGS寄存器中的不同的位。如果列為空，則表示與它相應的位沒有被影響。如果這些位總是被改為一特定的值，則在相應的列中顯示一個1或0。如果位元的改變的值依賴於指令的運算元，則在相應的列中顯示為C。最後，如果位被某種未定義的形式修改，則在列中顯示?。因為改變方向標誌位元的唯一指令是CLD和STD，所以在FLAGS列中，它們沒有被列出來。

名稱	描述	格式	標誌位元 O	S	Z	A	P	C
ADC	帶進位相加	O2	C	C	C	C	C	C
ADD	整數相加	O2	C	C	C	C	C	C
AND	按位AND	O2	0	C	C	?	C	0
BSWAP	位元組次序變反	R32						
CALL	調用程式	R M I						
CBW	將位元組轉換成字							
CDQ	將雙字轉換成四字							
CLC	進位元標誌位元清0							0
CLD	方向標誌位元清0							
CMC	進位元標誌位元變反							C
CMP	整數比較	O2	C	C	C	C	C	C
CMPSB	位元組比較		C	C	C	C	C	C
CMPSW	字比較		C	C	C	C	C	C
CMPSD	雙字比較		C	C	C	C	C	C
CWD	將字轉換成雙字，並儲存到DX:AX中							
CWDE	將字轉換成雙字，並儲存到EAX中							
DEC	整數減一	R M	C	C	C	C	C	
DIV	無符號數相除	R M	?	?	?	?	?	?
ENTER	建立堆疊幀	I,0						
IDIV	有符號數相除	R M	?	?	?	?	?	?
IMUL	有符號數相乘	R M	C	?	?	?	?	C
		R16,R/M16						
		R32,R/M32						
		R16,I						
		R32,I						
		R16,R/M16,I						
		R32,R/M32,I						

名稱	描述	格式	標誌位元					
			O	S	Z	A	P	C
INC	整數加一	R M	C	C	C	C	C	
INT	產生中斷	I						
JA	如果大於則跳轉	I						
JAE	如果大於等於則跳轉	I						
JB	如果小於則跳轉	I						
JBE	如果小於等於則跳轉	I						
JC	如果進位為1則跳轉	I						
JCXZ	如果CX = 0則跳轉	I						
JE	如果等於則跳轉	I						
JG	如果大於則跳轉	I						
JGE	如果大於等於則跳轉	I						
JL	如果小於則跳轉	I						
JLE	如果小於等於則跳轉	I						
JMP	無條件跳轉	R M I						
JNA	如果不大於則跳轉	I						
JNAE	如果不大於行於則跳轉	I						
JNB	如果不小於則跳轉	I						
JNBE	如果不小於等於則跳轉	I						
JNC	如果沒有進位則跳轉	I						
JNE	如果不等於則跳轉	I						
JNG	如果不大於則跳轉	I						
JNGE	如果不大於等於則跳轉	I						
JNL	如果不小於則跳轉	I						
JNLE	如果不小於等於則跳轉	I						
JNO	如果不溢出則跳轉	I						
JNS	如果SF=0則跳轉	I						

名稱	描述	格式	標誌位元					
			O	**S**	**Z**	**A**	**P**	**C**
JNZ	如果ZF=0則跳轉	I						
JO	如果溢出則跳轉	I						
JPE	如果PF=1則跳轉	I						
JPO	如果PF=0則跳轉	I						
JS	如果SF=1則跳轉	I						
JZ	如果ZF=1則跳轉	I						
LAHF	將FLAGS的低位元組							
	載入到AH中							
LEA	載入有效的位址	R32,M						
LEAVE	釋放堆疊幀							
LODSB	載入位元組							
LODSW	載入字							
LODSD	載入雙字							
LOOP	迴圈	I						
LOOPE/LOOPZ	如果ZF=1則迴圈	I						
LOOPNE/LOOPNZ	如果ZF=0則迴圈	I						
MOV	移動資料	O2						
		SR,R/M16						
		R/M16,SR						
MOVSB	移動位元組							
MOVSW	移動字							
MOVSD	移動雙字							
MOVSX	符號擴展移動	R16,R/M8						
		R32,R/M8						
		R32,R/M16						
MOVZX	零擴展移動	R16,R/M8						
		R32,R/M8						
		R32,R/M16						
MUL	無符號數相乘	R M	C	?	?	?	?	C
NEG	求反	R M	C	C	C	C	C	C

名稱	描述	格式	標誌位元					
			O	**S**	**Z**	**A**	**P**	**C**
NOP	無操作							
NOT	非運算	R M						
OR	按位OR	O2	0	C	C	?	C	0
POP	出堆疊	R/M16						
		R/M32						
POPA	全部出堆疊							
POPF	出堆疊送FLAGS		C	C	C	C	C	C
PUSH	進堆疊	R/M16						
		R/M32 I						
PUSHA	全部進堆疊							
PUSHF	FLAGS進堆疊							
RCL	帶進位迴圈左移	R/M,I	C					C
		R/M,CL						
RCR	帶進位迴圈右移	R/M,I	C					C
		R/M,CL						
REP	重複執行							
REPE/REPZ	如果ZF=1則重複執行							
REPNE/REPNZ	如果ZF=0則重複執行							
RET	返回							
ROL	迴圈左移	R/M,I	C					C
		R/M,CL						
ROR	迴圈右移	R/M,I	C					C
		R/M,CL						
SAHF	將AH複製到FLAGS中			C	C	C	C	C
SAL	算術左移	R/M,I						C
		R/M, CL						
SBB	帶借位相減	O2	C	C	C	C	C	C
SCASB	掃描位元組		C	C	C	C	C	C
SCASW	掃描字		C	C	C	C	C	C
SCASD	掃描雙字		C	C	C	C	C	C

名稱	描述	格式	標誌位元 O	S	Z	A	P	C
SETA	如果大於則目的位元組置1	R/M8						
SETAE	如果大於等於則目的位元組置1	R/M8						
SETB	如果小於則目的位元組置1	R/M8						
SETBE	如果小於等於則目的位元組置1	R/M8						
SETC	如果進位元標誌位元為1則目的位元組置1	R/M8						
SETE	如果等於則目的位元組置1	R/M8						
SETG	如果大於則目的位元組置1	R/M8						
SETGE	如果大於等於則目的位元組置1	R/M8						
SETL	如果小於則目的位元組置1	R/M8						
SETLE	如果小於等於則目的位元組置1	R/M8						
SETNA	如果不大於則目的位元組置1	R/M8						
SETNAE	如果不大於等於則目的位元組置1	R/M8						
SETNB	如果不小於則目的位元組置1	R/M8						
SETNBE	如果不小於等於則目的位元組置1	R/M8						
SETNC	如果進位元標誌位元為0則目的位元組置1	R/M8						

名稱	描述	格式	標誌位元 O	S	Z	A	P	C
SETNE	如果不等於則目的位元組置1	R/M8						
SETNG	如果不大於則目的位元組置1	R/M8						
SETNGE	如果不大於等於則目的位元組置1	R/M8						
SETNL	如果不小於則目的位元組置1	R/M8						
SETNLE	如果不小於等於則目的位元組置1	R/M8						
SETNO	如果OF=0則目的位元組置1	R/M8						
SETNS	如果SF=0則目的位元組置1	R/M8						
SETNZ	如果ZF=0則目的位元組置1	R/M8						
SETO	如果OF=1則目的位元組置1	R/M8						
SETPE	如果PF=1則目的位元組置1	R/M8						
SETPO	如果PF=0則目的位元組置1	R/M8						
SETS	如果SF=1則目的位元組置1	R/M8						
SETZ	如果ZF=1則目的位元組置1	R/M8						
SAR	算術右移	R/M,I R/M, CL						C
SHR	邏輯右移	R/M,I R/M, CL						C

名稱	描述	格式	標誌位元					
			O	**S**	**Z**	**A**	**P**	**C**
SHL	邏輯左移	R/M,I						C
		R/M, CL						
STC	進位元標誌位置1							1
STD	方向標誌位置1							
STOSB	儲存位元組							
STOSW	儲存字							
STOSD	儲存雙字							
SUB	相減	O2	C	C	C	C	C	C
TEST	邏輯比較	R/M,R	0	C	C	?	C	0
		R/M,I						
XCHG	交換	R/M,R						
		R,R/M						
XOR	按位XOR	O2	0	C	C	?	C	0

第二节 浮點數指令

在這一節中，描述了許多80x86數位輔助運算器的指令。說明部分簡要地描述了指令的操作。為了節省空間，關於指令是否出堆疊的資訊並沒有在描述中給出。

格式列展示了每個指令可以使用的運算元類型。使用的是下面的約定：

STn	一個輔助運算器寄存器
F	記憶體中的單精確度數
D	記憶體中的雙精度數
E	記憶體中的擴展精度數
I16	記憶體中的整形字
I32	記憶體中的整形雙字
I64	記憶體中的整形四字

需要奔騰或更好的處理器的指令用星號標示出來了(*)。

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	比較ST0和src	STn F D
FCOMP src	比較ST0和src	STn F D
FCOMPP src	比較ST0和ST1	
FCOMI* src	比較並設置FLAGS	STn
FCOMIP* src	比較並設置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

指令	描述	格式
FFREE dest	Marks as empty	STn
FIADD src	ST0 += src	I16 I32
FICOM src	比較ST0和src	I16 I32
FICOMP src	比較ST0和src	I16 I32
FIDIV src	ST0 /= src	I16 I32
FIDIVR src	ST0 = src/ST0	I16 I32
FILD src	將src壓入堆疊中	I16 I32 I64
FIMUL src	ST0 *= src	I16 I32
FINIT	初始化輔助運算器	
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	將src壓入堆疊中	STn F D E
FLD1	將1.0壓入堆疊中	
FLDCW src	裝載控制字寄存器	I16
FLDPI	將π壓入堆疊中	
FLDZ	將0.0壓入堆疊中	
FMUL src	ST0 *= src	STn F D
FMUL dest, ST0	dest *= ST0	STn
FMULP dest[,ST0]	dest *= ST0	STn
FRNDINT	ST0取整	
FSCALE	$\mathtt{ST0} = \mathtt{ST0} \times 2^{\lfloor \mathtt{ST1} \rfloor}$	
FSQRT	$\mathtt{ST0} = \sqrt{\mathtt{ST0}}$	
FST dest	儲存ST0	STn F D
FSTP dest	儲存ST0	STn F D E
FSTCW dest	儲存控制字寄存器	I16
FSTSW dest	儲存狀態字寄存器	I16 AX
FSUB src	ST0 -= src	STn F D
FSUB dest, ST0	dest -= ST0	STn
FSUBP dest[,ST0]	dest -= ST0	STn

指令	描述	格式
FSUBR src	ST0 = src-ST0	STn F D
FSUBR dest, ST0	dest = ST0-dest	STn
FSUBP dest[,ST0]	dest = ST0-dest	STn
FTST	比較ST0和0.0	
FXCH dest	將ST0和dest內容交換	STn

索引