第 3 章　算术运算指令

3.1　加法与减法运算

学习完了如何往寄存器中装载数据以及内存单元之间的数据传递，完成了 I/O 操作，下面来学习一下算术运算操作。学习使用汇编语言来完成算术运算的最简单方法之一，就是把一个高级语言语句转换成等价的汇编语句。假设整数变量 num1 和 num2 中已经存储了数值，如下 C 语言语句如何使用汇编语言来实现？

```
Sum = num1 + num2;
```

正如前面讨论的数据移动操作，不能把一个内存单元的内容直接复制到另一个内存单元中，只能通过寄存器来完成上述操作；在进行加法运算的时候，遵循同样的原理，不能直接在两个内存单元中完成上述操作，如表 3.1 所示。

表 3.1　加法指令

指　　令	意　　义
add mem,imm	将立即数的值与内存单元的数值相加，和值放到内存单元中
add reg,mem	将内存单元中的值与寄存器中的值相加，和值放到寄存器中
add mem,reg	将寄存器中的值与内存单元中的值相加，和值放到内存单元中
add reg,imm	将立即数的值与寄存器中的值相加，和值放到寄存器中
add reg,reg	将源（第二个）寄存器的值与目的（第一个）寄存器的值相加，和值放到目的寄存器中

请再次注意，正如之前介绍的没有 mov mem, mem 指令一样的，上述列表中也没有 add mem, mem 指令。因此，在进行计算的时候，必须先把内存单元的数据移动到寄存器中，然后再把另一个内存单元的数据与寄存器中的数据相加后存储到寄存器中，最后再把寄存器中的加法结果保存到指定的内存单元中。上述 C 语言语句可以使用如下汇编语言代码来实现：

```
; sum = num1 + num2
mov  eax,num1     ; 将 num1 中的内容复制到寄存器 eax 中
add  eax,num2     ; 把 num2 中的内容与 eax 中的内容相加后存储到 eax 中
mov  sum,eax      ; 把 eax 中的内容存储到 sum 中
```

同前面一样，把 C 语言语句放到前面作为如下汇编语句的注释。在上述代码段中，num1 中的内容复制到寄存器 eax 中，然后 num2 的内容与 eax 中的内容相加，结果放到 eax 中，最后，把 eax 的内容复制到变量 sum 中。假设 num1 的初始值为 5，num2 的初始值位 7，图 3.1 给出了前面代码段的运算结果，寄存器 eax 和变量 sum 的最终结果都为 12。

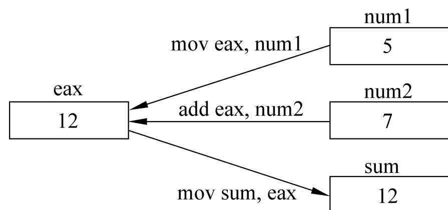

图 3.1　加法运算结果

虽然可以使用另外的三个寄存器来完成上述操作，但是最好还是使用累加器来进行，即使用寄存器 eax，因为算术运算指令使用寄存器 eax 会占用更少的内存，运算速度也更快一些，第 10 章将进行介绍。另外，如同高级语言一样，解决同一个问题可以有多种方式，低级语言也是这种情况。进一步来说，如同某些解决方案在高级语言中是更好的方案一样，在低级语言中也存在这种情况。例如，前面的汇编语言代码可以重新编写为如下形式：

```
mov sum,0         ; 把 sum 初始化为 0
mov eax, num1     ; 把 num1 的内容装载到寄存器 eax 中
add sum, eax      ; 把寄存器 eax 的内容与 sum 相加，结果放到 sum 中
mov eax, num2     ; 把 num2 的内容装载到寄存器 eax 中
add sum, eax      ; 把寄存器 eax 的内容同 sum 的内容相加，结果放到 sum 中
```

虽然上述代码可以正常工作，sum 的最终结果为 num1 和 num2 的和。但是上述代码并不是最初的 C 语言语句的实现，而是如下 C 语言语句的实现：

```
sum = 0;
sum = sum + num1;
sum = sum + num2;
```

表 3.2　减法指令

指　　令	意　　义
sub mem,imm	将内存单元的数值减去立即数的值，结果值放到内存单元中
sub reg,mem	将寄存器中的值减去内存单元中的值，结果值放到寄存器中
sub mem,reg	将内存单元中的值减去寄存器中的值，结果值放到内存单元中
sub reg,imm	将寄存器中的值减去立即数的值，结果值放到寄存器中
sub reg,reg	将目的（第一个）寄存器的值减去源（第二个）寄存器的值，结果值放到目的寄存器中

虽然上面两个汇编语言代码和 C 语言代码段都可以正常工作，计算出变量 num1 和 num2 的和，结果存储到 sum 中，但是第二个代码段在内存占用率和运算效率方面没有第一个高。对于内存来说，由于第二段代码包含的指令更多，因此它占用的内存空间也相应的变大了，同样，由于它包含更多的指令，这个代码段运行的时候花费的时间也更多。

正如上述示例，要想写出更简洁的代码出来，没必要首先考虑使用汇编语言来实现，可以先用高级语言来考虑实现问题，然后再把相应的高级语言程序转换成低级的汇编语言实现即可。虽然有时候这种做法由于寄存器的原因可能会引起低级语言代码的低效问题，但是最终的实现通常不会像上个例子那么糟糕，因此无论使用什么层级的程序设计语言都需要考虑效率的问题。

类似于加法指令，表 3.2 给出了相关的减法指令，其中减法指令的用法格式与加法指令的用法格式一样的。

再次注意，这里也没有内存单元与内存单元之间的减法指令存在。如前一样，高级语言指令如下：

```
difference = num2 -num1;
```

相应的汇编语言指令如下：

```
; difference = num2 -num1
mov eax,num2         ; 将 num2 的内容装载到寄存器 eax 中
sub eax,num1         ; 将 eax 中存储的值减去 num1 的内容
mov difference,eax   ; 将 eax 的内容存储到 difference 中
```

3.2　乘法运算与除法运算指令

虽然加法运算与减法运算看起来比较简单，但是乘法运算与除法运算就稍微有点复杂了。当计算两个数值之和时，可能会遇到这种情况，它们的和值需要占用的存储空间大于寄存器所能提供的存储空间，或者大于内存所能使用的存储空间，此时将会引起内存溢出错误。例如，在十进制中，数值 999 加上数值 999，结果为 1998，此时结果值所占的位数要比两个原始数值所占的位数多一位。同样，在二进制中，数值 111 与数值 111 的和为 1110（详见附录 B）。不过，当使用 32 位符号双字的时候，除非和值大于 2147483647（见表 1.2），否则并不会引起溢出的问题。

但是对于乘法来说，这种情况就严重的多了。例如，在十进制中，数值 999 乘以数值 999，乘积结果为 998001，此时结果值占用的位数不是多出了一位，而是多出了一倍。二进制也是同样地情况，数值 111 与数值 111 相乘的话，结果值为 110001，结果值所占的比特位同样比原始数值所占的比特位多出了一倍。

因此，在计算机中出现乘法运算的时候，比如当两个 32 位寄存器中的数值相乘的时候，就需要额外的存储空间来存储结果值，因为两个 32 位寄存器中的数值相乘，有可能结果值需要占用 64 个比特位。虽然乘法运算有多种类型，可以有两个操作数或三个操作数，不过本书中只介绍一个操作数的指令，这种指令从第一个 intel 处理器开始就在使用（那时的 intel 处理器是 16 位的）。另外，单操作数乘法指令非常类似于单操作数除法指令，而除法指令只有这一种指令形式。因此，这里类似的知识有利于我们学习后面的除法指令。虽然无符号乘法指令 mul 可以计算稍微大一些的数值，但是它不能用于计算负数。因此，本书只介绍带符号乘法指令，这个指令以字母 i 打头，称为 imul。乘法指令 imul 的两种格式见表 3.3 所示。

表 3.3　imul 指令格式

指　令	意　义
imul reg	将寄存器 eax 的值与寄存器中的整数值相乘
imul mem	将内存单元中的整数值与寄存器 eax 中的值相乘

这两个单操作数带符号乘法运算指令的运算方式是先把需要乘积的数值（被乘数）装载到寄存器 eax 中。然后把另一个乘数（乘数）放到寄存器中或者放到内存空间中。注意单操作数指令 imul，它没有立即数的形式，且该指令默认使用寄存器 eax 来存储被乘数。

在乘法指令 imul 执行时，寄存器 eax 中的数值将与指定寄存器中的数值或内存空间中的数值相乘，乘积结果存储到寄存器对 edx:eax 中。请回忆一下第 1 章中介绍的内容，edx 是数据寄存器，它可以用于各种类型的算术运算指令，而乘法指令 imul 就属于这一类算术运算指令。前面介绍过，乘法运算的结果值需要占用的比特位可能是乘数或被乘数所占用的比特位的两倍，因此乘积值的低位部分存储到寄存器 eax 中，而高位部分存储到寄存器 edx 中。目前来说，我们不打算让乘积结果值所占的比特位超过 32 位，或者说乘积结果的正数值不超过 2147483647，或负整数值不超过 2147483648（详见第 1 章或附录 B）。无论如何，我们需要清楚，寄存器 edx 将存储数值的高位部分的比特值；在本例中，依赖于乘积结果是正数还是负数，寄存器 edx 中存储的要么是一堆 0，要么是一堆 1。此时，之前在寄存器 edx 中存储的内容将被覆盖掉。基于以上分析，如下为使用 C 语言实现：

```
product = num1 * num2;
```

如下为使用汇编语言实现：

```
; product = num1 * num2
mov eax,num1         ; 将 num1 的内容存储到寄存器 eax 中
imul num2            ; 将寄存器 eax 的内容乘以 num2 的内容
mov product,eax      ; 将 eax 的内容存储到 product 中
```

同样，在如上代码段中，寄存器 edx 中的内容也会被覆盖的。假设 num1 的值为正数 2，num2 的值为正数 5，那么寄存器对 edx:eax 存储的结果为：寄存器 eax 中第 31 位（最左边位）存储的值为 0，这个值将复制到寄存器 edx 全部的 32 个比特位中。如果 num1 的值为负数 2，num2 的值仍然为正数 5，此时寄存器对 edx:eax 存储的结果为：寄存器 eax 中第 31 位（最左边位）存储的值为 1，这个值将复制到寄存器 edx 全部的 32 个比特位中，如图 3.2 所示（附录 B 详细地介绍了正数和负数相关内容）。这就是为什么不要在寄存器中存储数值的原因，数值应该存储到内存中。

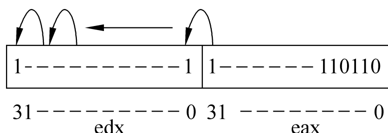

图 3.2　imul 指令运行后寄存器 edx:eax 的存储内容

基于上面对于指令 imul 的介绍，下面的 C 语言语句如何使用汇编语言来实现？

```
product = num1 * 2;
```

虽然上述 C 语言语句可以通过 imul 指令的双操作数或三操作数版本来实现，但是通过单操作数指令仍然比较容易解决这个问题。首先把立即数存储到一个空寄存器中，此时可以把该寄存器用于 imul 指令中。另外，后面用到 idiv 指令时，可以仿照这里的代码来写。

```
; product =  num1 * 2
mov eax, num1        ; 将 num1 的内容存储到寄存器 eax 中
mov ebx,2            ; 将数值 2 存储到寄存器 ebx 中
imul ebx             ; 将寄存器 eax 的内容乘以 ebx 的内容
mov product, eax     ; 将寄存器 eax 的内容存储到 product 中
```

正如前面过，有一个针对无符号数值的乘法指令 mul；类似地，也有一个针对无符号数值的除法指令（div）。尽管这个指令可以除以稍微大一些的数值，但是它不能除以负数，本书只讨论带符号除法指令 idiv。如前所述，idiv 指令的格式与前面介绍的单操作数指令 imul 一样。idiv 的指令格式如表 3.4 所示。

表 3.4　idiv 指令

指　　令	意　　义
idiv mem	将内存中的值拆分开存储到寄存器对 edx:eax 中
idiv reg	将寄存器中的值拆分开存储到寄存器对 edx:eax 中

除法指令的工作方式非常类似于乘法指令的工作方式，唯一不一样的是，乘法运算的结果值比乘数值和被乘数值都要大，而除法运算的结果值（商）和余数要小于原始数值（被除数）。结果是乘法运算的乘积结果要存储到寄存器对 edx:eax 中，而除法运算中，在使用除法指令 idiv 之前，被除数要首先放到寄存器对 edx:eax 中。idiv 指令运行之后，寄存器 eax 中存储的是商，寄存器 edx 中存储的是余数。

但问题是，如何把一个数值，无论它是立即数还是内存中的数据，放到寄存器对 edx:eax 中？实际上不需要特别的指令就可以完成上述工作。例如，要实现如下 C 语言语句：

```
answer = number / amount;
```

首先，变量 number 中的内容将被存储到寄存器 eax 中，然后，假设变量 number 的值为正数，将存储 0 到寄存器 edx 中。但是如果变量 number 的值为负数，此时不是把 0 存储到寄存器 edx 中，而是把–1 存储到寄存器 edx 中。最终把寄存器 edx 的所有比特位都存储成二进制的 1，把符号位也设置为 1。不过，这里需要用到选择结构，目前还没有介绍到选择结构，该方法属于比较笨的解决方案。幸运的是，intel 处理器的设计工程师们早已考虑到了这个问题，他们设计了特殊的指令，可以把符号位从小一些的寄存器转换为大一点的寄存器。表 3.5 给出了这些指令。

表 3.5　转换指令

操作码	意　　义	描　　述
cbw	字节转换为字	寄存器 al 的内容扩展到寄存器 ax
cwd	字转换为双字	寄存器 ax 的内容扩展到寄存器 eax
cdq	双字转换为 4 个字	寄存器 eax 的内容扩展到寄存器对 edx:eax

这里需要用到的是表 3.5 中最后一条指令。指令 cdq 允许符号位，无论它是 0 或 1，都能够传送到整个 edx 寄存器，该指令能够有效地避免上面方法的不足。例如，如果寄存器 eax 初始值为–2，然后寄存器 eax 的符号位（第 31 位，存储着二进制 1）将被复制到寄存

器 edx 中的每一个比特位，如图 3.3 所示。

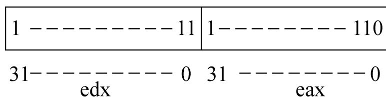

图 3.3　将寄存器 eax 的符号位传送到寄存器 edx 中

上述 C 语言代码相对应的汇编代码如下：

```
; answer = number / amount
mov eax,number  ; 将 number 的内容存储到寄存器 eax 中
cdq             ; 将符号位的值传送到寄存器 edx 中
idiv amount     ; 将寄存器对 edx:eax 的内容除以 amount 的内容
mov answer,eax  ; 将寄存器 eax 的内容存储到 answer 中
```

假设变量 number 中存储的值为 5，变量 amount 中存储的值为 2，那么上述代码执行之后，寄存器对 edx:eax 的存储详情为：余数 1 存储在寄存器 edx 中，而商为 2 存储在寄存器 eax 中，如图 3.4 所示。

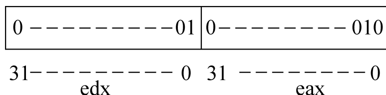

图 3.4　idiv 指令执行后，寄存器对 edx:eax 的存储详情

虽然 imul 指令有双操作数或三操作数的格式能够用于处理立即数，但是 idiv 指令却没有这种指令格式，它的单操作数只能是内存空间或者寄存器。那么如何实现立即数的运算呢（比如 answer=number/2）？提示一下，参考在乘法运算中介绍过的立即数操作。另外，考虑一下如何实现 C 语言中的%操作符，比如 answer= number % amount？如果读者还记得%运算符的意义（mod 或余数函数），请复习一下 idiv 指令的工作方式，上述两个问题的答案就比较显而易见的了，这几个问题我们留作本章练习题。

3.3　一元运算：递增、递减和求反

在高级语言中，前两节所介绍的算数运算符称为二元运算符，之所以这么称谓它们，并不是因为它们操作的是二进制数值，而是因为它们都有两个操作数，比如 X+Y。但是一些高级语言运算符只有一个操作数，比如负号–y，我们称这种运算符为一元运算符。

虽然可以使用前面介绍过的指令来实现一元运算符所能够完成的所有算术运算，但是在汇编语言中有这样一些算数运算指令，它们占用更少的内存，运算速度也更快，可以让汇编程序员的程序编写工作更加轻松。另外，在介绍这些指令的同时，本节还将介绍关于运算符优先级方面的相关内容。

例如，如果需要将变量 x 加 1，或者变量 y 减 1，即

```
X=x+1;
Y=y-1;
```

或者，使用递增或递减运算符：

```
x++ 或者 ++x;
y--或者--y;
```

对上面独立的语句来说，++或--出现在变量前面或后面都一样。上面的语句可以使用 add 或 sub 指令来实现：

```
add x,1
sub y,1
```

虽然上面的所有操作方法都是可行的，但是 Intel 处理器的设计者已经设计好了两条内置指令用于递增加 1 或递减减 1。这两条指令格式见表 3.6 所示。

表 3.6 递增与递减指令

指 令	指 令
inc reg	dec reg
inc mem	dec mem

本节开头提到过，内置的递增与递减指令比 add 和 sub 指令占用更少的内存空间。事实上，在早先的 16 位处理器上，如果需要对寄存器进行加 2 操作或者减 2 操作，使用两次 inc 或 dec 指令比使用一次 add 或 sub 指令直接加 2 或减 2 执行效率更高。但是对于新型的 32 位处理器来说，情况并非如此，只能说一次 inc 或 dec 指令的执行效率比使用 add 或 sub 指令加 1 或减 1 的执行效率更高（见第 10 章）。结果如下：

```
inc x
dec y
```

前面介绍过，在 C/C++/Java 语言中，对于单独的语句来说，递增或递减符号出现在变量的前面或后面都行。但是，读者在其他的计算机科学课程中可能学习过，当把这种运算符与其他运算符一同使用时，情况就大不一样了。虽然对于入门级程序员来说，应该尽量避免使用这种组合运算式，但是偶尔还是会碰到这种情况，因此最好能够理解清楚这种情况在高级语言与低级语言中是如何实现的。作为复习或介绍，请思考如下两条指令有何不同？

```
x = y++;  x = ++y;
```

在上面左边这条指令中，变量 y 先把值赋给变量 x，然后变量 y 的值加 1。在上面右边的指令中，变量 y 先进行加 1 操作，然后才把值赋给变量 x。假设在上述两种情况下，变量 y 的初始值为 2，图 3.5 给出了指令运行之后的结果情况。

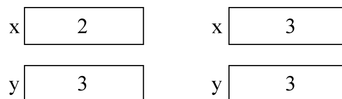

图 3.5 x= y++ 与 x= ++y 的执行结果

很明显，这两条语句是不一样的。上面两条语句可以替换成如下语句：

```
x = y; y = y+1;
y = y+1; x = y;
```

前面两条语句使用汇编语言实现如下：

```
mov eax,y  inc y
mov x,eax  mov eax,y
inc y  mov x,eax
```

如果想要求出一个数值的负数，或者求出一个数的 2 的补码（详见附录 B），并把结果存储到另一个内存单元中，该如何操作呢？例如，

```
x = -y;
```

介绍递增与递减之前，这里先使用已学过的指令来实现求负数：

```
mov eax,0
sub eax,y
mov x,eax
```

如前面介绍的一样，也有一个指令 neg，用它来求负数更简便、计算速度更快，表 3.7 给出了该指令的格式。

表 3.7　求负指令

指　　令	意　　义
neg reg	求寄存器中值的 2 的补码
neg mem	求内存单元值的 2 的补码

上面的代码段使用 neg 指令重新编写如下：

```
mov eax,y
neg eax
mov x,eax
```

注意，这里变量 y 并没有被求负，因为这里只是对寄存器的值进行了求负操作。即，变量 y 的值首先复制到寄存器中，然后对寄存器的值进行了求负操作，最后将求负后的值复制到了变量 x 中。虽然第 2 段代码与第 1 段代码的指令条数相同，但是 neg 指令比 sub 指令占用的内存空间更小。

在表达式中使用递增或递减指令可能会引起疑惑，但是求负指令简单得多。只需要知道求负时的单个减号符号的优先级要高于其他二元运算符号+、–、*和/的优先级。下面语句中，左边的语句在进行加法之前先对变量 y 进行了求负操作；如果想要先进行加法运算，需要使用括号把加法运算先括起来，如右边语句所示：

```
x = -y+z; x = -(y + z);
```

上述两条语句使用汇编语言实现如下：

```
mov eax,y mov eax,y
```

```
neg eax add eax,z
add eax,z neg eax
mov x,eax mov x,eax
```

3.4　一元运算符与二元运算符的优先级

尽管前面章节中介绍了运算符优先级方面的知识，但是只涉及了一元运算符。本节将更加深入地介绍算术运算符的优先级问题，其中包括二元运算符的优先级问题以及它与一元运算符复合运算等问题。为了更好地理解运算符的优先级以及提高使用汇编语言算术运算指令的能力，本节将讨论一些稍微复杂点的代数运算语句。同样地，还是先给出高级语言实现的语句：

```
answer = num1 + 3 -num2;
```

回忆一下 C、C++和 Java 语言中的算术运算优先级，因为加法和减法的优先级相同，因此上述语句的运算顺序是从左到右的。在这个例子中，加法运算先执行，然后进行减法运算。虽然有些编译程序和经验丰富的程序员有时为了提高机器代码执行的效率会更改算术运算的计算顺序，但是本书为了运算优先级的一致性将遵从预先定义的规则以帮助读者熟悉运算符优先级方面的相关内容。

首先，变量 num1 的内容将被装载到寄存器 eax 中，然后数值 3 将累加到寄存器 eax 中，其中数值 3 并不是存在于内存中，而是一个立即数。接下来，要从寄存器 eax 中减去变量 num2 的值，最后，把寄存器 eax 中的值复制到变量 answer 中，相应代码如下所示：

```
; answer = num1 + 3-num2
mov eax,num1     ; 将变量 num1 中的内容装载到寄存器 eax 中
add eax,3        ; 把寄存器 eax 的值加 3
sub eax,num2     ; 把寄存器 eax 的值减去变量 num2 的值
mov answer,eax   ; 把计算结果存储到变量 answer 中
```

如前所述，在汇编语言中，解决一个问题通常不止只有一种方法，如下代码同样可以完成上述工作：

```
add num1,3       ; 把变量 num1 的值加 3
mov eax,num1     ; 把变量 num1 的内容装载到寄存器 eax 中
sub eax,num2     ; 把寄存器 eax 的值减去变量 num2 的值
mov answer,eax   ; 把计算结果存储到变量 answer 中
```

首先，第 2 段代码看起来与第 1 段代码一样都非常不错。它们的指令条数一样，都把正确的计算结果放置到内存位置 answer 中。但是，需要注意的是，第 2 段代码比第 1 段代码多引用了一次内存，因此它的执行时间要稍长一些。不过，除了执行速度和内存空间占用情况之外，上述代码段还有其他的问题，如涉及原始 C/C++/Java 代码的实现问题。请注意，在原始的高级语言指令 answer= num1+ 3 –num2 中，唯一有变化的只有变量 answer，即等号左边的变量。上述语句中变量 num1 和变量 num2 的值是不变的。不过，在第 2 段汇

编语言代码中，变量 num1 的值加上了立即数 3，它的内容变化了。因此，上述代码段实现的并不是原始的语句 answer= num1+ 3 –num2，而实现的是如下 C 语言语句：

```
num1 = num1 + 3;
answer = num1 -num2;
```

如果在编写程序的时候，变量 num1 的值在后续不需要用了，那么问题可能不大。但是，如果并不能确定该变量在程序的后续执行过程中是否要用到的话，那么通常的做法是最好不要随意更改变量的值。另外，最好把原始的高级语言语句作为注释，并确保汇编语言实现的结果与该高级语言语句实现的情况是一样的。

为了进一步学习运算符的优先级，请看如下 C 语言语句：

```
answer = num1 + 3 * num2;
```

首先请注意，乘法运算的优先级要高于加法运算的优先级，因此数值 3 要首先装载到寄存器 eax 中，然后乘以变量 num2 的值。接下来把 num1 的值加到寄存器 eax 中，最后把 eax 的值存储到变量 answer 中，相应的汇编语言代码如下：

```
; answer = num1 + 3 * num2
mov eax,3          ; 把数值 3 装载到寄存器 eax 中
imul num2          ; 把寄存器 eax 的值乘以变量 num2 的值
add eax,num1       ; 把寄存器 eax 值加上变量 num1 的值
mov answer,eax     ; 把寄存器 eax 的内容存储到变量 answer 中
```

另外，请记住 imul 指令将更改寄存器 edx 的值。另一个例子，请考虑如下 C 语言语句：

```
result = num3 / (num4 -2);
```

尽管除法的优先级要高于减法的优先级，但是请记住括号中的表达式要优先进行计算，因此上述语句中减法要比除法先执行。然后变量 num3 的值将被减法的结果值相除，最后把寄存器 eax 的值存储到变量 result 中：

```
; result = num3 / (num4 -2);
mov ebx,num4       ; 将 num4 的内容装载到寄存器 ebx 中
sub ebx,2          ; 将寄存器 ebx 的内容减 2
mov eax,num3       ; 将 num3 的内容装载到寄存器 eax 中
cdq                ; 将符号位的内容传送到寄存器 edx 中
idiv ebx           ; 将寄存器对 edx:eax 的内容除以 amount 的内容
mov result,eax     ; 将寄存器 eax 的内容存储到 result 中
```

注意寄存器 ebx 用于存储减法运算的临时结果值，因此差值可以用于后续的除法运算。下面把上一节介绍的一元运算指令与本节介绍的二元运算指令组合使用，考虑如下 C 语言代码：

```
v = -w + x * y-z++;
```

上述语句的执行顺序是先把变量 w 的值进行求负值操作，然后计算变量 x 与变量 y 的乘积值，接着再把乘积结果加上负的 w 值，然后把结果值减去变量 z 的值，并把结果值赋

给变量 v，最后把变量 z 的值加 1，因为 z 后面有++运算符，相应的汇编程序代码如下：

```
;v = -w+x * y-z++
mov ebx,w
neg ebx
mov eax,x
imul y
add ebx
sub z
mov v,eax
inc z
```

再次注意，变量 w 的值并没有真正的变为自身的负值，在把结果值赋给变量 v 之后，只有变量 z 的值变化了，z 的值加 1 了。熟悉运算符优先级的最好方法就是自己动手实践。更多的练习题请见 3.7 节。

3.5 完整程序示例：实现 I/O 与算术运算

结合第 1～3 章学到的内容，可以考虑编写一个完整的程序，程序会输出提示信息，让用户输入各种数值，然后完成一定的计算工作，最后输出相应的结果。衍生于第 2 章最后一个程序，本程序仍然相对简单，但是它可以作为更加复杂程序的模板用于测试各种算术运算式，而且有助于实现本章结尾的部分练习题。

例如，在已知伏特值与欧姆值的情况下，如何计算出安培值？解决办法就是使用欧姆定律，即公式 E=IR，其中 E 为电动势伏特，I 为电流安培，R 为电阻欧姆。很显然，上述公式不能计算出安培值需要对上述公式进行变换，即 I=E/R。因为该程序只针对整数运算，结果意义不大，它的主要作用在于示例说明一个完整程序。与前面一样，先看一下使用 C 语言实现的代码段：

```
#include <stdio.h>
int main(){
  int volts, ohms, amperes;
  printf("\n%s", "Enter the number of volts: ");
  scanf("%d", &volts);
  printf("%s", "Enter the number of ohms: ");
  scanf("%d", &ohms);
  amperes = volts / ohms;
  printf("\n%s%d\n\n","The number of amperes is:",amperes);
  return 0;
}
```

相应的汇编语言代码段如下：

```
        .386
        .model flat, c
        .stack 100h
```

```
printf   PROTO arg1:Ptr Byte, printlist:VARARG
scanf    PROTO arg2:Ptr Sdword, inputlist:VARARG
         .data
in1fmt   byte "%d",0
msg1fmt  byte 0Ah,"%s",0
msg2fmt  byte "%s",0
msg3fmt  byte 0Ah,"%s%d",0Ah,0Ah,0
msg1     byte "Enter the number of volts: ",0
msg2     byte "Enter the number of ohms: ",0
msg3     byte "The number of amperes is: ",0
volts    sdword ?            ; 数值 volts
ohms     sdword ?            ; 数值 ohms
amperes  sdword ?            ; 数值 amperes
         .code
main     proc
         INVOKE printf, ADDR msg1fmt, ADDR msg1
         INVOKE scanf, ADDR in1fmt, ADDR volts
         INVOKE printf, ADDR msg2fmt, ADDR msg2
         INVOKE scanf, ADDR in1fmt, ADDR ohms
         ; amperes = volts/ohms
         mov eax,volts       ; 将 volts 的值装载到寄存器 eax 中
         cdq                 ; 扩展符号位的内容
         idiv ohms           ; 将寄存器 eax 的内容除以 ohms 的内容
         mov amperes,eax     ; 将寄存器 eax 的内容存储到 amperes 中
         INVOKE printf, ADDR msg3fmt, ADDR msg3, amperes
         ret
main     endp
         end
```

3.6　本 章 小 结

- 注意不要更改赋值符号右边的变量值。
- 请注意寄存器 edx 中存储的是乘法结果的高位部分的比特值。
- 在进行除法运算之前，不要忘记使用 cdq 指令。
- 在实现算术运算指令的时候，按照如下运算符优先级进行：
  - 最内层括号中的表达式先计算；
  - 一元负号优先于乘法和除法运算符；
  - 乘法和除法优先于加法和减法运算符；
  - 表达式从左向右进行计算。
- 请注意递增和递减（++与--）运算符：
  - 递增与递减运算符单独出现的时候，运算符在变量的前面和后面结果都一样；
  - 在赋值语句中，前缀形式时，递增或递减的运算优先于赋值操作进行，后缀形式时，递增或递减操作在赋值操作之后进行。

3.7 练　习　题

注意：带有*号练习题的参考答案见附录 E。

1.请指出如下语句哪些在语法上是正确的，哪些是错误的。如果不正确，请指出错在哪里：

*A. `inc eax,1`

B. `add ebx,ecx`

*C. `add dog,cat`

D. `idiv 3`

*E. `sub 2,number`

F. `imul eax`

2. 请把如下 C 语言算术运算语句转换为相应的 intel 汇编语言程序（提示：如本章所介绍的，在进行 imul 指令与 idiv 指令的时候，不要忘记首先把立即数装载到寄存器中）。

*A. `product = 3. number;`

*B. `result = number % amount;`

*C. `answer = number / 2;`

*D. `difference = 4 -number;`

3. 按照 C 语言的运算优先级，把如下算术运算语句转换为相应的汇编语言语句。注意不要更改出现在赋值符号右边的任何变量的值。

*A. `x = x. y+z. 2;`

*B. `a = b-c/3;`

*C. `total = num1 / num2 -(num3. num4);`

*D. `r = -s + t++;`

*E. `m = n. ((i -j). k);`

4. 按照 C 语言的运算优先级，把如下算术运算语句转换为相应的汇编语言语句。注意一元负号、递增和递减运算符。

*A. `--i;`

*B. `j = ++k -m;`

*C. `z = -(x + y);`

*D. `a = ++b -c++;`

*E. `x = -y + z--;`

5. 编写一个完整的汇编语言程序来实现如下 C 语言代码段。

```
#include <stdio.h>
int main(){
    int number;
    printf("\n%s","Enter an integer: ");
    scanf("%d",&number);
    number=7-number*3;
    printf("\n%s%d\n\n","The integer is: ",number);
    return 0;
}
```

6. 已知本章结尾完整程序中的欧姆定律和瓦特定律 W=IE，其中 W 表示瓦特值，请编写一个完整的汇编程序，实现的程序要能够提示用户输入安培值和欧姆值，然后计算出伏特值与瓦特值。输入与输出的格式如下所示，请注意水平与垂直空白区：

```
Input and Output
Enter the number of amperes: 5
Enter the number of ohms: 4

The number of volts is: 20
The number of watts is: 100
```

7. 编写一个完整的汇编语言程序提示用户输入华氏温度，然后计算出摄氏温度，然后输出该摄氏温度值。相应的公式为 C=(F−32)/9*5，其中 C 表示摄氏温度，F 表示华氏温度。注意使用整数的结果和相应的算术运算优先级。输入与输出的格式如下所示，请注意水平与垂直空白区：

```
Input and Output
Enter the degrees in Fahrenheit: 100
The degrees in Celsius is: 35
```

第 4 章　选 择 结 构

4.1　引　　言

假如读者已经学习过计算机科学的入门课程了，那么可能会知道无论何种编程语言，一定存在有两种类型的控制结构。这两种类型的控制结构是选择结构与迭代结构，通常称为 ifs 与 loops。

在低级程序设计语言中，可以使用 if 语句和分支语句来实现控制结构。有两种类型的分支语句，分别是条件分支语句与无条件分支语句，前者需要满足一定地条件（比如等于 0 的时候），而后者与条件无关。高级程序设计语言中通常避免使用无条件分支语句与 goto 语句，但是在低级程序设计语言中，很难做到不去使用无条件分支语句与 goto 语句，因为只有使用了 goto 语句才能通过编译器和解释器来创建其他高级控制语句。因此，大多数汇编语言都会在程序中使用到类似 goto 语句一样的语句。

不过，MASM 比较特殊，它能够让程序员使用等价于高级控制结构的语句。尽管这一点在某种程度上否定了需要使用汇编语言的理由，但是它提供了高级语言特性到低级语言特性之间的延续性，而且给我们了一个机会，让我们认清楚高级语言控制结构是如何在低级语言中实现的。分析完低级程序设计语言中的高级控制结构之后，将分析一下这些控制结构的低级语言实现情况。

对于选择结构来说，最常用的是 if-then 结构和 if-then-else 结构。另外，这两种结构还可以嵌套出 if-then-else-if 结构与 if-then-if 结构来，前者比后者更为常用一些。最后，还有一类结构称为 case 结构，在 C/C++/Java 语言中称为 switch 结构。虽然在 MASM 中没有可用的高级版本，但是可以使用条件分支与无条件分支语句来构造出这种结构来。4.2 节首先通过最长的篇幅来介绍 if-then 结构，因为这一部分基本介绍所有相关的内容，其他结构只不过是这个结构的变体而已。

4.2　if-then 结构

在 C 语言中，if-then 语句的一般结构如下，其中当 then 后面只有一条语句的时候，括弧是可选的。但是如果 then 后面有一条以上的语句时，就必须使用括号把这些语句括起来，如下右边代码所示：

```
if (number == 0)
   number--;
```

```
if (amount != 1) {
   count++;
   amount = amount + 2;
}
```

假设变量 number 已经声明为 sdword 类型，那么上述 C 语言代码相对应的 MASM 代码如下：

```
.if number == 0
dec number
.endif
```

```
.if amount != 1
inc count
add amount,2
.endif
```

首先，请注意在 if 和 endif 前面有一个小数点，它表示后面的单词不是一条可以直接执行的命令，而是一条汇编指令，用来告诉编译程序在这个位置插入相应的代码以实现这条编译指令，与第 1 章介绍过的.data 和.code 编译指令是一样的。另外，需要注意的是，在条件关系判断这里没有圆括号，这里与 C 语言不同，在 MASM 中条件关系判断这里的圆括号是可选的，但是在 C 语言中任何类型的条件关系判断在 MASM 中都是可用的。还有就是必须要使用汇编指令.endif，无论在 then 后面是一条语句还是多条语句，都要使用.endif 汇编指令。目前来说，第 3 章介绍的算术运算语句应该是比较熟悉了，如果对 C 语言中的 if 语句有一定的理解和认识，以上内容就比较容易理解了。

虽然 MASM 有一些特殊的功能，但是它也有一些使用上的限制。就特殊功能来说，它除了能够完成上述变量内容与文字的比较之外，还能够实现寄存器内容与文字的比较以及两个寄存器内容的比较。但是对于后两种情况有个限制就是编译程序默认情况下认为作比较的两个值都是无符号的，否则就会引起逻辑错误。在这两种应用情况下，请确保比较的就是两个非负数值，假如真是需要比较两个负数值，请确保这两个负数值至少有一个是存储在 sdword 类型的变量中。关于另一个限制，请读者考虑一下如何像 C 语言一样比较两个内存位置的内容？

```
if (count > number)
    flag = -1;
```

可能读者已经猜到了，与第 1 章介绍的 mov 指令一样，汇编指令.if 不能够直接比较两个内存位置的内容。为什么不能直接比较两个内存位置的内容呢，原因是比较指令 cmp 是由汇编指令.if 所产生的。同 mov 指令一样，比较指令 cmp 不能够引用两个内存位置。下面将会对 cmp 指令进行介绍，两个内存位置的内容至少有一个需要复制到寄存器中，才能够进行比较操作，即实现寄存器内容与内存位置中的内容进行直接比较：

```
mov eax,count
.if eax > number
mov flag,-1
.endif
```

尽管上述代码写起来非常不方便，但是使用 MASM 汇编指令可以让上述 if-then 结构的编码过程简单化。但是假如 MASM 没有相应的高级汇编指令的话，如何来实现 if-then 结构呢？有人可能会问，既然 MASM 已经提供了相应的高级汇编指令，为什么还要考虑这个问题呢？因为并不是所有的低级语言都提供了高级汇编指令，因此理解清楚高级结构是如何被实现的将是非常有益的，能够帮助读者学习其他汇编语言。另外，无论是高级语言

也好，低级语言也罢，它能够帮助读者理解清楚高级控制结构最终是如何通过低级语言来实现的。

在第 1 章介绍过，程序员可以直接使用 4 个通用的寄存器。虽然还有另外一些寄存器程序员不能够直接访问，但是这些寄存器都是可以间接被访问的。这些寄存器中有一个最重要的寄存器就是 eflag 寄存器，它控制着 CPU 的方方面面，记录了任意时刻 CPU 的状态。在各种类型指令执行的时候，会在 eflag 寄存器中用 1 到 2 个比特位作为 flag 标记。与其他许多类型的寄存器不一样的是，这里不用逻辑指令来访问单独的比特位，而是为每个 flag 标记都指定 2 个字母的缩写名，其中一些标记的内容可以使用高级操作符通过高级控制结构来访问，本章后面将对该部分内容进行详细介绍。如下所示，方向标记不包含高级操作符，因为它是一个控制标记，有特殊的指令来控制这个标记，第 9 章将详细介绍该内容。表 4.1 给出了汇编语言程序员比较常用的标记。

表 4.1 常用的标记

标志	简写	高级运算符	比特位位置	当值被设置为 1 时的意义
carry	CF	CARRY?	0	无符号整数进位
parity	PF	PARITY?	2	偶数位被设置
zero	ZF	ZERO?	6	运算的结果为 0
sign	SF	SIGN?	7	结果为负
direction	DF		10	从高比特位向低比特位处理字符串
overflow	OF	OVERFLOW?	11	符号整数溢出

这里介绍两个最为重要的标记，它们用来表示最后一条指令执行的结果是零、负或者正。如果结果是零，zero 标记（ZF）将会被设置为 1，sign 标记（SF）将被设置为 0。一开始 zero 标记将被设置为 1，咋看起来有点不对，但是如果 1 表示真，那么表示的意思就对了。然后，如果结果为负，那么 SF 标记将被设置为 1，而 ZF 标记将被设置为 0。最后，如果结果为正，那么 ZF 标记和 SF 标记将都被设置为 0，如表 4.2 所示。

表 4.2 ZF 与 SF 标记

结果	ZF	SF
zero	1	0
negative	0	1
positive	0	0

更改这些标记的一种简便方式就是使用 cmp 指令，cmp 指令来比较两个操作数并设置相应的标记。CPU 完成上述比较操作，它实际上对两个操作数进行了一个隐性地减法操作，然后设置了相应的标记值，这里所谓的隐性地减法操作指的是在运算过程中两个操作数都没有任何变化。例如，如果两个操作数相等，减法的结果为 0；如果第一个操作数比第二个操作数大，减法结果将为正。表 4.3 给出了 cmp 指令的格式。

如表 4.3 所示，以及前面介绍的内容，可以把寄存器内容与立即数进行比较，可以把寄存器内容与内存内容比较，可以把立即数与内存内容比较，也可以比较两个寄存器的内容，唯一不能直接比较的就是两个内存位置的内容。一旦两个操作数比较完成，相应的标

记将被设置一定的值，然后程序就可以依据标记的值跳转到相应的分支，表 4.4 给出了两种条件跳转指令。

表 4.3　cmp 指令

指　　令	意　　义
cmp reg,imm	将寄存器中的值与立即数的值进行比较
cmp imm,reg	将立即数的值与寄存器中的进行比较
cmp reg,mem	将寄存器中的值与内存中的值进行比较
cmp mem,reg	将内存中的值与寄存器的值进行比较
cmp mem,imm	将内存中的值与立即数的值进行比较
cmp imm,mem	将立即数的值与内存中的值相比较
cmp reg,reg	将寄存器的值与寄存器的值相比较

表 4.4　Je 与 jne 指令

指令	意　　义
je	当相等时，发生跳转
jne	当不相等时，发生跳转

je 指令与 jne 指令既可以用于符号数据也可以用于非符号数据。对于符号数值数据的条件跳转指令如表 4.5 所示。

表 4.5　符号数值数据的条件跳转指令

指令	意义	指令	意义
jg	大于时跳转	jnle	不小于或等于时跳转
jge	大于或等于时跳转	jnl	小于时跳转
jl	小于时跳转	jnge	不大于时或等于时跳转
jle	小于或等于时跳转	jng	不大于时跳转

注意表 4.5 中，在同一行的左侧列的指令与同一行的右侧列的指令是一样的。虽然右侧列的指令功能一样，使用“不”这种方式容易引起迷惑，因此相对来说，大家在写程序的时候更多地使用左侧列的指令。为了帮助说明以上指令如何用于实现 if-then 结构，下面分析一下前面介绍过的示例：

```
.if number == 0
dec number
.endif
```

当 number 的值为 0 时，对 number 的值进行递减操作；否则不改变 number 的值。前面介绍过，想要实现.if 汇编指令，需要用到 cmp 指令以及上面介绍过的条件跳转指令。但是这里稍稍复杂一些：在高级层面的实现中，如果 if 语句中的条件为真，那么 if 语句后面的 then 语句部分代码将被执行；如果 if 语句中的条件为假，则程序控制流将跳过 then 部分的代码，不执行这一部分代码。但是，前面提到过的跳转语句情况是不一样的，当条件判

断结果为真的时候，程序并不执行紧接着的部分程序代码，而是跳转到其他程序分支。

解决上述问题有很多办法，其中最简单的办法就是把逻辑关系对调一下。上述代码可以用如下方式来实现，首先对变量 number 和 0 进行比较。如果两个值不相等，进行一个不相等时的跳转操作（jne），跳转到标记 endif01 处，如果两个值相等，程序控制流不变，直接往下执行，代码如下：

```
          ; if number == 0
if01:     cmp number,0      ; 将数值 number 和 zero 进行比较
jne       endif01           ; 当不等时跳转到 endif01
then01:   dec number        ; 将 number 的值减 1
endif01:  nop               ; if 结束，没有任何操作
```

关于上述代码段的一些注意事项。第一，由于跳转方式与高级语言中的执行方式正好相反，因此在汇编语言代码段的前面附上相应的高级语言代码作为一种注释是非常好的编程习惯。第二，这里使用了 endif01 来代替 endif，这是为了区别出到底要跳转到程序中哪一个 endif 标记处。另外使用 if01 代替 if1，因为 if1 将用于过程，即条件汇编，第 7 章将详细介绍。语句 nop 将占据一个字节的内存空间，它表示“无任何操作”，该语句什么工作都不做。乍一看，可能会觉得奇怪，但是实际上为 nop 指令指派一个标记是非常有用处的。在上述例子中，nop 指令可以忽略，opcode 部分可以留空白。但是，对于汇编语言的初学者来说，包含这么一个语句以方便未来能够添加其他语句，这种方式是一种非常好的工作方法。这里的标记 if01：和 then01:是可选的，因为程序控制流不会跳转到这里来，不过，这两个标记有助于指明 if-then 结构的开始位置以及中间位置，建议保留它们。最后请注意，标记后头有一个冒号，即 if01:，如果把冒号忘记了，将会引起程序语法错误。

如果希望扩展一下前面代码段的.if 汇编指令，了解一下最终的代码是个什么样子，那么可能会看到类似的 cmp 指令和 jne 指令。可以在程序的开头部分增加汇编指令.listall，并重新汇编一下就可以完成上述操作，最后打开.lst 文件就可以看到相应的结果，详见附录 A。

我们首先会注意到由汇编程序自动生产的标记@C0001。对于汇编程序生产的标记来说，数值是连续增加的，即下一个标记为@C0002。之所以在标记的开头使用@符号就是为了避免与程序员自定义的标记重复而引起程序错误。这就是为什么在第 1 章中建议程序员不要使用@符号来命名标记的原因之一。第二个需要注意的地方是，代码段的结尾部分并没有 nop 语句。前面我们介绍过，nop 语句是可选的，它将出现在程序员编写的代码段中，以增强程序的可读性，但是考虑到程序对于内存的占用以及运行效率的时候，会把它去掉：

```
        . cmp number,000h
          jne @C0001
          dec number
@C0001:
```

进一步练习一下，如何在不使用.if 汇编指令的情况下来实现如下这个前面介绍过的代码段？

```
mov eax,count
.if eax > number
```

```
mov flag,-1
.endif
```

首先，可能会有读者试图使用 jl（小于）跳转指令来完成上述操作，但是请注意了。大于的反义是什么呢？大于的反义并不是小于，而是小于或等于，这应该是初学汇编语言的程序员在编写底层语言实现的时候，最容易出现的错误之一。这也是比较难以排除的程序错误，因为除非出现了两个操作数相等的情况，否则这段代码会正确的执行。如下是上述代码正确的实现：

```
          ; if count > number
if02:     mov eax,count
          cmp eax,number
          jle endif02
then02:   mov flag,-1
endif02:  nop
```

4.3　if-then-else 结构

介绍完 if-then 结构之后，只需要简单地扩充一下就可以实现 if-then-else 结构了。例如，如下左边的 C 语言代码，可以使用.else 汇编指令来实现，如下右边代码所示：

```
if x>= y                  ;if  x>= y
    x--;                 mov eax,x
else                     .if eax >= y
    y--;                 dec x
                         .else
                          dec y
                         .endif
```

或者说，可以不用汇编指令，而直接使用比较指令、跳转指令和标记来实现上述代码。第一部的实现类似于前面介绍的 if-then 结构，只不过最后的分支跳转到了 endif03 标记处，程序控制流转移到标记 else03 处，如下所示：

```
            ;if x >= y
if03:       mov eax,x
            cmp eax,y
            jl else03
then03:     dec x
            jmp endif03
else03:     dec y
endif03:    nop
```

关于上述代码段最重要的事情是一定不要忘记在 then 部分的结尾使用无条件跳转指令（jmp），否则程序的控制流将绕过 else 部分，这样的话，肯定不会正确地实现 if-then-else 结构。无条件跳转指的是无论何种条件都要进行跳转，换句话说，无论标志寄存器 eflags