



鲁宏伟/luhw@hust.edu.cn

第四讲

理解编译后的算术运算

概述

- 在逆向工程中，某些算术运算指令序列是很容易解读的，但是在其他一些情况下，算术运算指令序列经过了编译器优化处理后，读起来难度就会大一些。

算术标志位

- 为了详细理解汇编语言中是如何实现算术和逻辑运算的，必须全面地理解各个标志位以及它们的使用方法。
- 几乎指令集中所有的算术指令都要用到标志位，因此，要真正理解汇编语言中算术指令序列的含义，必须理解每一个标志位的含义以及算术指令是怎样使用它们的。
- IA-32 处理器中的标志位都被集中存放在 EFLAGS 存储器中，它是一个由处理器管理的32 位寄存器，程序代码很少会直接访问它。EFLAGS 存储器中的大多数标志位是系统标志位，系统标志位的状态确定了处理器的当前状态。
- 除了这些系统标志位外，还有8个状态标志位，这8个标志位代表了处理器的当前状态，状态标志位的取值与最近一次执行的算术运算的结果有关。

算术标志位

- **溢出标志位** (CF和OF)
- 进位标志位 CF 和溢出标志位 OF 对于汇编语言中的算术指令和逻辑指令是非常重要的两个标志位。它们的功能以及它们之间的区别并不十分明显。
- CF 和 OF 都是溢出指示符，这就是说它们两都可以用于通知程序算术运算的结果太大了以至于无法把它全部表示在目标操作数中。这两个标志位的区别与程序所处理的数据类型有关。

算术标志位

- **溢出标志位** (CF和OF)
- 与大多数高级语言不同，汇编语言程序不会显式地指明当前所处理数据的类型细节。一些算术指令如 ADD 指令和 SUB 指令也不去管它们处理的操作数到底是有符号数还是无符号数，因为这对它们来说不重要——运算的二进制结果是一样的。
- 而其他一些指令，如 MUL 指令和 DIV 指令就有无符号数版本和有符号数版本两种——因为不同的数据类型对于乘法和除法来说会产生不同的二进制输出。

算术标志位

- **溢出标志位** (CF和OF)
- 有符号数表示和无符号数表示总是与一个问题有关，这个问题就是溢出。因为有符号整数要比同样长度的无符号整数少一位（因为这个特殊的位被用来存放符号），所以有符号数和无符号数的溢出触发条件是不一样的。这就是我们需要 CF和OF两个溢出标志位的原因。
- 处理器并没有为算术指令提供独立的有符号和无符号两个版本，而是简单地通过用两个溢出标志位报告溢出就把问题合理地解决了：这两个标志位一个用于有符号操作数，一个用于无符号操作数。

算术标志位

- **溢出标志位** (CF和OF)
- 加法或者减法运算可以用同一个版本的指令对有符号操作数和无符号操作数进行运算，并对两个标志位进行置位——置位后的标志位留待接下来的指令处理。
- 举个例子，考虑下面的算术运算代码，看看它会对这两个溢出标志位产生什么样的影响：

```
mov ax, 0x1126  
mov bx, 0x7200  
add ax, bx
```
- 上面的加法指令可能会产生不同的结果，具体的结果取决于是把目的操作数当作有符号数还是无符号数对待。

算术标志位

- **溢出标志位** (CF和OF)

```
mov ax, 0x1126
```

```
mov bx, 0x7200
```

```
add ax, bx
```

- 如果用十六进制数表示的话，结果是 0x8326——假定 AX 被当作一个无符号操作数。
- 如果把 AX 当作一个有符号操作数，你就会看到发生了溢出，因为任何最高位为1的有符号数是负数，0x8326就成了-31962了。显然，因为一个 16位的有符号数能表示的最大的数是 32767，因此，上面的代码对无符号数来说没有溢出，但看作是有符号数就溢出了。所以，前面这段代码会导致OF（表示在有符号操作数中溢出）置1，而 CF（表示在无符号操作数中溢出）被清零。

算术标志位

- **零标志位 (ZF)**
- 当算术运算的结果为0时，零标志位ZF将被置 1；如果结果不为 0，ZF则被清零。
- 在 IA-32汇编语言代码中，在好多种情况下会使用ZF标志位，但可能最常见情况就是比较两个操作数并测试它们是否相等。
- 比如用 CMP 指令将一个操作数减去另一个操作数，如果减法运算的伪结果（表示此结果并不写入目的操作数中）为0就将ZF标志位置1，表明两个操作数相等。如果两个操作数不相等，ZF被清零。

算术标志位

- **符号标志位 (SF)**
- 符号标志位记录结果的最高位（不管结果是有符号数还是无符号数）。
- 对于有符号整数，SF相当于整数的符号。
- 符号标志位SF为1表明结果是负数，为0表示结果是正数（或者是0）

算术标志位

- **奇偶标志位 (PF)**
- 奇偶标志位 (极少使用) 记录算术运算结果低8位的二进制奇偶校验。
- 二进制奇偶校验就是指指数中置为1的位数是奇数还是偶数，它与数的奇偶性完全是两回事儿。
- PF为1表明运算结果的低8位中1的个数是偶数，而 PF为零则表明结果的低8位中1的个数是奇数。

基本的整数算术运算

- 需要指出的是，对于任意一款正常的编译器来说，任何涉及两个常量操作数的算术运算都会在编译阶段被全部清除并代之以它们的运算结果（你在汇编语言代码中只能看到这个运算结果）。
- 因此，我们讨论的算术优化只应用于运算中至少包括一个事先不知道取值得变量的情形。

基本的整数算术运算

- **加法和减法**

- 整数的加法和减法通常是用 ADD 和 SUB 指令来实现的，这两条指令可以接收几种不同类型的操作数：寄存器名，立即（硬编码）数或者内存地址。
- 这些类型的操作数具体怎么组合取决于编译器，而且通常并不能反映出任何有关源代码的具体信息，但有一种情况很明显——如果是加上、减去一个立即操作数的话，那么它通常反映到源代码中的一个硬编码的常数（当然，在某些情况下编译器会为了其他目的将常数放到寄存器中再进行加减运算，而不是按照源代码所指示的方式处理）。
- 要指出的是，两条指令都是将运算结果存放在左操作数中。

基本的整数算术运算

- **加法和减法**

- 加、减法运算都是非常简单的运算，在现代 IA-32 处理器中执行效率非常高，并且通常编译器是用直接的方式实现的。
- 在早期的IA-32处理器中，LEA指令的执行效率比 ADD和SUB指令高，这导致了許多编译器都优先选用 LEA 指令来实现快速加法和移位操作。
- 下面是用LEA 指令执行算术运算的一个例子。

```
lea ecx, DWORD PTR [edx+edx]
```

- 需要指出的是，尽管大多数反汇编器会在操作数前加上 "DWORD PTR"，但事实上 LEA 指令并不能区别指针和整数。LEA不执行任何真正意义上的内存访问。

基本的整数算术运算

- **乘法和除法**

- A-32 处理器提供了几种不同类型的乘法、除法指令，但是它们的运算速度都相对比较慢。因此，编译器通常倾向于使用其他方式来完成乘法和除法运算。
- 用2 的整数次方乘或除一个数是最适合于计算机的运算，因为这 在二进制的整数中再好表示不过了。这就像我们可以轻松地完 乘以或除以 10的整数次方的运算一样——只需要移小数点或者 补零就可以了。
- 计算机处理乘法和除法运算和我们的处理方法基本上是一样的。 总的方法是试着将除数或乘数尽可能精确地转化成易于二进制系 统表示的数值。这样，就可以执行相对简单的计算了，并找出 将除数或乘数的其余部分用到计算中的方法。

基本的整数算术运算

- **乘法和除法**
- 对于 IA-32处理器，等效于移小数点或者补零的操作是执行二进制移位——二进制移位可以用SHL 和SHR 指令完成的。
- 移位操作完成后，编译器通常还会用加法和减法对结果做一些必要的补偿。

基本的整数算术运算

- 当一个变量乘以另外一个变量的时候，通常 MUL / IMUL 指令是最有效的工具。不过，当乘数是一个常数时，大多数编译器决不会使用MUL / IMUL 指令。
- 比如说当一个变量乘以常数 3时，编译器通常是先将变量左移1位然后加上原来的值。完成这一运算可以使用SHL指令和ADD指令，也可以只使用 LEA 指令，如下所示：

```
lea eax, DWORD PTR [eax+eax*2]
```

- 在更加复杂的情况下，编译器会组合使用LEA 和 ADD 指令。例如下面这段代码——实现的是乘以 32 的运算：

```
lea eax, DWORD PTR [eax+edx]  
add eax, eax  
add eax, eax  
add eax, eax
```

基本的整数算术运算

- 这段代码是由 Intel 的编译器生成的，当你有没有感觉奇怪：
 - 首先，就代码长度而言它很长——这段代码用了1个LEA指令和4个ADD指令，这比只用一条SHL指令来实现的代码长多了。
 - 其次，令人吃惊的是，这段代码的实际执行速度比只执行一条左移5位的SHL指令还要快，尽管我们认为SHL指令已经算是执行效率非常高的指令了。
- 其中的原因是：LEA指令和ADD指令都是执行时间短、高吞吐量的指令。
- 实际上，**执行完这段代码可能不到三个时钟周期**（虽然这取决于具体的处理器和其他环境方面的因素）。比较而言，**执行一条SHL指令需要4个时钟周期**，这就是为什么用它实现不如用上面的代码效率高的原因。

基本的整数算术运算

- 我们来看另一段乘法代码：

```
lea eax, DWORD PTR [esi+esi*2]  
sal  eax, 2  
sub  eax, esi
```

- 上面这段代码是用GCC编译器生成的，它先用LEA指令实现 ESI 乘以3，然后用SAL指令（SAL指令与SHL 指令相同——它们共用同一个操作码）把结果进一步乘以4。这两次运算将操作数乘以了12。然后，代码又将结果减去该操作数。
- 实际上，这段代码是将操作数乘以 11。用数学表达式可以将这段代码表示为： $y = (x + x*2)*4 - x$ 。

基本的整数算术运算

- **除法**

- 对于计算机来说，除法是整数算术运算中最复杂的运算。
- 在处理器中实现的内置除法指令有 DIV 指令和 IDIV 指令，相对于其他指令来说，它们的执行速度非常慢，执行时间超过了50个时钟周期，而加法和减法指令执行时间不到一个时钟周期。
- 对于除数是未知数的情况，编译器只能使用 DIV / IDIV 指令。这会影响到程序的执行性能，但对于逆向工程人员来说却是个好消息，因为这使得代码可读性强、且很直观。
- 如果除数是常数的话，情况就变得复杂多了。编译器会根据除数的具体数值选用一些非常有创造性的技术来实现更为高效的除法。
- 问题出现了——这样做会导致代码的可读性变差。

基本的整数算术运算

- **倒数相乘**

- 倒数相乘的思想是**利用乘法来代替实现除法运算**。在 IA-32 处理器上，乘法指令的运算速度比除法指令要快4到6倍，因此我们会在某些情况下尽量避免使用除法指令，而代之以乘法指令。其思路就是**将被除数乘以除数的倒数**。
- 举例来说，如果要用 30 除以3，可以容易地计算出3的倒数，" $1 \div 3$ "，计算的结果近似为0.333，再用30乘以0.333，就可以得到正确的结果 10。
- 在整数算术运算中实现倒数相乘要比这个例子复杂得多，因为能使用的数据类型只有整型。

基本的整数算术运算

- **倒数相乘**

- 为了解决这一问题，编译器采用了一种被称为“定点运算”的方法。
- 定点算法使得我们可以不用“浮动的”可动小数点分数和实数。有了定点运算，我们表示小数时不再用阶码（即小数点在浮点数据类型中的位置），而是要保持小数点的位置不变。
- 这和硬件浮点数机制截然不同。硬件浮点数机制是由硬件负责给整数部分和小数部分分配可用的位。有了这种机制，浮点数就可以表示很大范围的数——从极小的数（在0到1之间的实数）到极大的数（在小数点前有数十个0）。

基本的整数算术运算

- **倒数相乘**
- 要用整型数来近似地表示实数，我们要在整数中定义一个假想的小数点，用它来确定哪一部分表示的是该实数的整数值而哪一部分表示的是该实数的小数部分。
- 实数的整数值就用划分后用于表示整数部分的"位"（分配的数量）按照普通的整型数的方式来表示。实数的小数部分尽可能精确地用可用的位数表示出来。
- 显然，很多实数值都不能精确地表示，只能近似地表示。举例来说，要表示 0.5，小数值应为 0x80 00 00 00（假定是 32 位的小数值），要表示 0.1，小数值应为 0x20 00 00 00（0.125）。

基本的整数算术运算

- **倒数相乘**
- 再回到我们原来的问题上，为了用乘法来实现除 32 位被除数的除法运算，编译器会为被除数乘上一个 32 位的倒数，相乘的结果是一个 64 位的数，其中低 32 位是余数（也是用一个小数值来表示的），高 32 位就是我们要的结果。
- 看一下左边的代码，这段代码干了什么事？

```
argc= dword ptr 4
argv= dword ptr 8
envp= dword ptr 0Ch

call    ds:rand
lea     ecx, [eax+0FFFFh]
mov     eax, 68D88BADh
imul    ecx
sar     edx, 0Ch
mov     eax, edx
shr     eax, 1Fh
add     eax, edx
push    eax
push    offset Format    ; "a=%d\n"
call    ds:printf
add     esp, 8
xor     eax, eax
retn
```

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    int v3; // eax

    v3 = rand();
    printf("a=%d\n", (v3 + 0xFFFF) / 10000);
    return 0;
}
```


基本的整数算术运算

• 倒数相乘

- 68DB8BADh与除数10000 (2710h) 什么关系?
- $10000 = 625 * 16$, $1/625 = 0.0016$
- 0.0016用32位的小数表示:

```
def dec2bin(x):
```

```
def dec2bin(x):
    x -= int(x)
    bins = []
    while x:
        x *= 2
        bins.append("1" if x>=1. else "0")
        x -= int(x)
    return bins

bins = dec2bin(.0016)
#ss = bins[8:i] for i in 32
binchar = ''.join(bins[8:40])
print(hex(int(binchar,2)))
print("length = %d"%len(bins))
print(bins[0:40])
```

```
0x68db8bac
length = 62
```

左移8位

$[0', 0', 0', 0', 0', 0', 0', 0']$, $0'$, $1'$, $1'$, $0'$, $1'$, $0'$, $0'$, $0'$, $1'$, $1'$, $0'$, $1'$,
 $1'$, $0'$, $1'$, $1'$, $1'$, $0'$, $0'$, $0'$, $1'$, $0'$, $1'$, $1'$, $0'$, $0'$

基本的整数算术运算

- 再回到我们原来的代码
- 代码中直接取edx相当于 $\gg 32$ ，然后再sar edx, 0Ch，总体右移了44位 ($32+12$)
- 为什么是右移12位呢，因为这个数是 $1/625$ 的，但 $10000=625*2^4$
- $0x68db8bad$ 是 $1/625$ 左移8位，再左移4位恰好就是 $1/10000$ 的2进制值

```
argc= dword ptr 4
argv= dword ptr 8
envp= dword ptr 0Ch

call ds:rand
lea ecx, [eax+0FFFFh]
mov eax, 68DB8BADh
imul ecx
sar edx, 0Ch
mov eax, edx
shr eax, 1Fh
add eax, edx
push eax
push offset Format ; "a=%d\n"
call ds:printf
add esp, 8
xor eax, eax
retn
```

基本的整数算术运算

- 在计算0.0016小数值时，如果把二进制值左移不是8位，例如7位呢（取[7:39]，则得到的16进制值为0x346dc5d6+1，是否能得到同样的结果呢？
- 反编译能够得到同样的结果！

```
argc= dword ptr 4
argv= dword ptr 8
envp= dword ptr 0Ch

call     ds:rand
lea      ecx, [eax+0FFFFh]
mov      eax, 346DC5D7h
imul     ecx
sar      edx, 0Bh
mov      eax, edx
shr      eax, 1Fh
add      eax, edx
push     eax
push     offset Format ; "a=%d\n"
call     ds:printf
add      esp, 8
xor      eax, eax
retn
```

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    int v3; // eax

    v3 = rand();
    printf("a=%d\n", (v3 + 0xFFFF) / 10000);
    return 0;
}
```

基本的整数算术运算


- 倒数相乘还可以换一种方法理解

- 考虑 $a/b = c$ b是常量
- 设 $b * k = 2^n$ 2的n次直接用移位运算即可
- $a * 2^n / (b * 2^n) = c$
- $a * b * k / b = c * 2^n$
- $a * k \gg n = c$
- 所以 $a * k \gg n = a/b$ //编译器会选择n的长度 从而保证后面位数舍弃时, 造成的影响最小
- 当 $b=625$ 时, $n=40$, $k=68DB8BAC_{16} + 1 = 68DB8BAD_{16}$
- 对不同的除数b, 选取合适的n和k, 除法运算就变成了乘法和移位运算。

基本的整数算术运算

- 32位魔术数的示例
- d 是除数
- M 是魔术数
- s 是右移的位数
- $a * M \gg s = a/d$

d	Signed		Unsigned		
	M(hex)	s	M(hex)	a	s
-5	99999999	1			
-3	55555555	1			
-2^k	7FFFFFFF	k-1			
1	-	-	0	1	0
2^k	80000001	k-1	2^{32-k}	0	0
3	55555556	0	AAAAAAAAAB	0	1
5	66666667	1	CCCCCCCCD	0	2
6	2AAAAAAB	0	AAAAAAAB	0	2
7	92492493	2	24924925	1	3
9	38E38E39	1	38E38E39	0	1
10	66666667	2	CCCCCCCCD	0	3
11	2E8BA2E9	1	BA2E8BA3	0	3
12	2AAAAAAB	1	AAAAAAAB	0	3
25	51EB851F	3	51EB851F	0	3
125	10624DD3	3	10624DD3	0	3
625	68DB8BAD	8	D1B71759	0	9



luhw@hust.edu.cn