发布时间：2015-11-08 16:54

# 深入理解计算机系统：信息的处理和表示（二）整数四则运算

四则运算 (/search/%E5%9B%9B%E5%88%99%E8%BF%90%E7%AE%97/1.htm)

参考自：http://csapp.cs.cmu.edu/ (http://csapp.cs.cmu.edu/)

开篇说明一下，本文不是介绍四则运算的具体执行过程，想了解具体过程的孩子们自己去看看计算机组成。

好了，话不多说。

## 1. 加减法

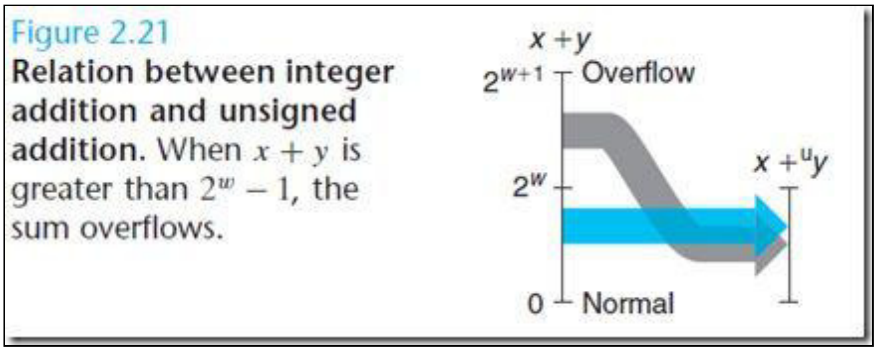
加法和减法没有区别，以下内容专注于加法。

### 1.1 无符号数加法

无符号数加法会出现溢出问题，当发生溢出的时候直接扔掉溢出进位。比如1111 + 0001 = 10000 直接扔掉进位1，结果为0000。考虑到溢出之后，我们可以总结出这样的计算式：

$$x +_w^u y = \begin{cases} x + y, & x + y < 2^w \\ x + y - 2^w, & 2^w \le x + y < 2^{w+1} \end{cases}$$

(http://images.cnitblog.com/blog/465162/201404/072133000905052.png)




(http://images.cnitblog.com/blog/465162/201404/072133010433894.png)

那么我们怎么来判断溢出呢？

**对于s= x+ y; 当s <x 或者 s < y的时候就是溢出了**。原因在于溢出的时候$s=x+y-2^w$。而x,y都是小于$2^w$的所以s<x ，s<y。

于是，我们可以写出函数来提前判断两个无符号数想加是否会溢出：

```
/* Determine whether arguments can be added without overflow */

int uadd_ok(unsigned x, unsigned y){

    unsigned  temp= x+y;

    return temp>=x;

}
```

下面给出一张图，全面展示四位无符号二进制树之间的想加情况：

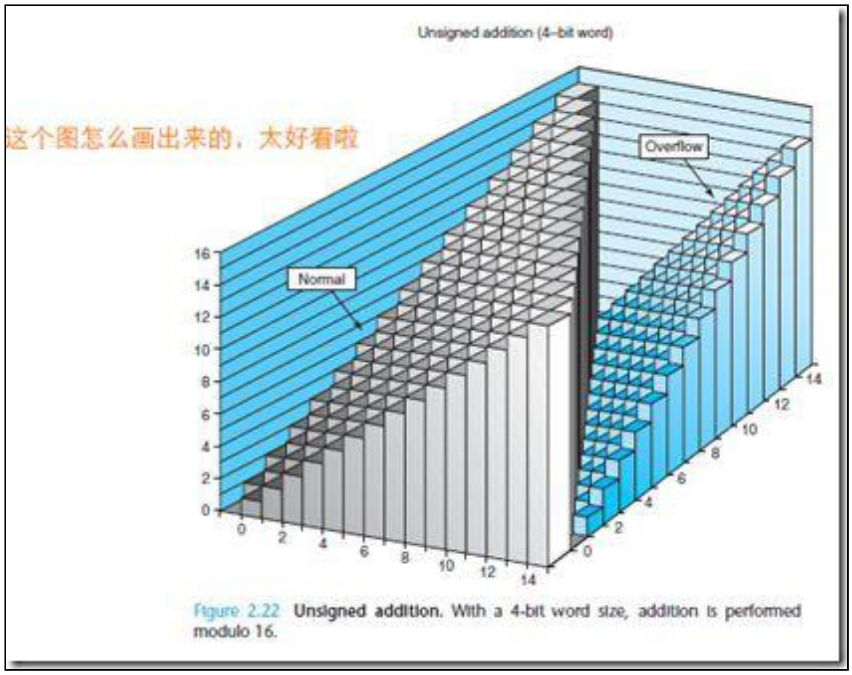



Figure 2.22 Unsigned addition. With a 4-bit word size, addition is performed modulo 16.

(http://images.cnitblog.com/blog/465162/201404/072133036222206.png)

好了，之前提到加法和减法没什么区别，那么我们怎么**计算无符号数减法**呢？要知道无符号数可是没有“相反数”这个东西的。

为了解决这个问题，我们引入一个 **加法逆元**(additive inverse) 的概念。假设x的加法逆元是y，当且仅当$(x+y)\%2^w=0$，其中x,y为w位的二进制数。

加法逆元的计算公式：

$$-_w^u x = \begin{cases} x, & x = 0 \\ 2^w - x, & x > 0 \end{cases}$$

(http://images.cnitblog.com/blog/465162/201404/072133068566917.png)

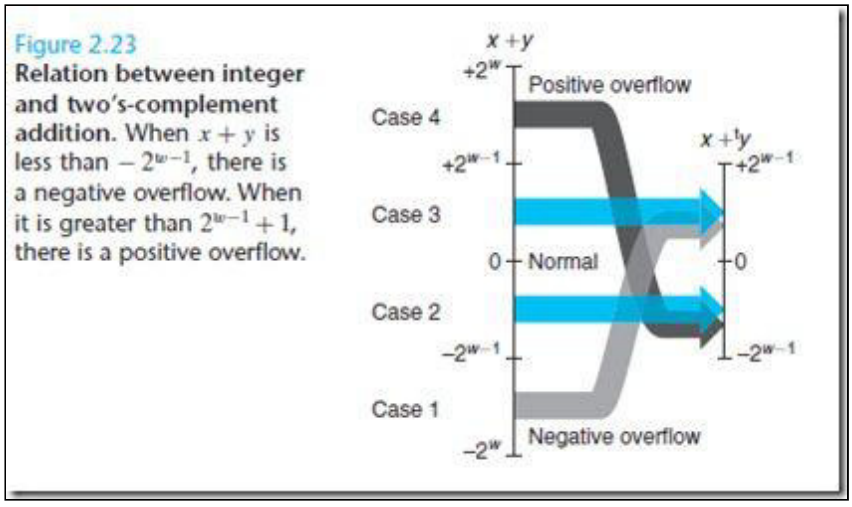
以后计算减法的时候，就将被减数转换为其加法逆元，这样再相加就好了。

## 1.2 有符号数加法

有符号加法和无符号加法产生的二进制是一样的，它也会产生溢出，由于符号的原因它有两种溢出 **正溢出和负溢出**。正溢出是指两个两个正数想家结果为负数，比如0111 + 0001 =1000。负溢出就是两个负数相加为正数1001 + 1001 =0010。计算公式如下：

$$x +_w^t y = \begin{cases} x + y - 2^w, & 2^{w-1} \le x + y & \text{Positive overflow} \\ x + y, & -2^{w-1} \le x + y < 2^{w-1} & \text{Normal} \\ x + y + 2^w, & x + y < -2^{w-1} & \text{Negative overflow} \end{cases}$$

(http://images.cnitblog.com/blog/465162/201404/072133075753531.png)




(http://images.cnitblog.com/blog/465162/201404/072133091373786.png)

给出四位二进制有符号数相加的全部情况图：

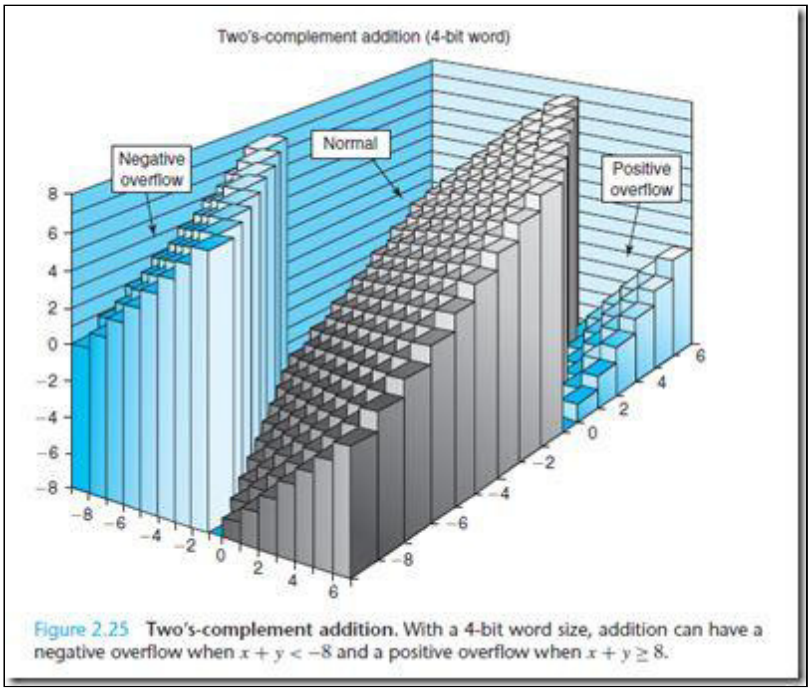



Figure 2.25 **Two's-complement addition.** With a 4-bit word size, addition can have a negative overflow when $x + y < -8$ and a positive overflow when $x + y \geq 8$.

(http://images.cnitblog.com/blog/465162/201404/072133124347712.png)那么怎么判断是否溢出呢？

原理就是 **负负得正，正正得负 就是溢出啦。**

```
/* Determine whether arguments can be added without overflow */

int tadd_ok(int x, int y) {

    int sum = x+y;

    int neg_over = x < 0 && y < 0 && sum >= 0;

    int pos_over = x >= 0 && y >= 0 && sum < 0;

    return !neg_over && !pos_over;

}
```

当然了，如果有人神经叨叨的给你看他的实现方法：

```
/* Determine whether arguments can be added without overflow */

/* WARNING: This code is buggy. */

int tadd_ok(int x, int y) {

    int sum = x+y;

    return (sum-x == y) && (sum-y == x);

}
```

这段代码毫无疑问是错误的，为什么呢？因为我们知道加法是符合交换律的，那么(x+y)-x = (x-x)+y，也就是说上述判断条件始终成立。不信你可以验证。

说完加法，我们来看看怎么将减法转换为加法。当然是**取相反数**了，关键是，怎么取相反数？这是一个很有学问的地方，技巧妙的一谈糊涂。

x相反数是y，当且仅当x+y=0。那么我们显然得到y=-x这个公式了，但是要是有人问你最小负数的相反数是多少，你怎么办？举个例子，对于8位二进制而言，最小负数-128，-(-128)=128>127(8位二进制最大正数)。

哈哈，怎么办，怎么办。

结论是最小负数的相反数就是它自己！！你可以算一算1000 0000 + 1000 0000 =0000 0000 符合定义！

于是就有我们的求负数的公式了：

$$-_w^t x = \begin{cases} -2^{w-1}, & x = -2^{w-1} \\ -x, & x > -2^{w-1} \end{cases}$$

(http://images.cnitblog.com/blog/465162/201404/072133152473954.png)

哈哈，看完这个我们再来看看**怎么验证两个数相减会不会溢出**，先看看下述代码：

```
/* Determine whether arguments can be subtracted without overflow */

/* WARNING: This code is buggy. */

int tsub_ok(int x, int y) {

    return tadd_ok(x, -y);

}
```

这个对不对呢？

应该是不对的，因为**Tmin的相反数还是他自己**，当x为正数，y为Tmin的时候，这个函数是会报错的，但是实际上是不会溢出的。

下面接着来说怎么实际操作来取相反数：一种方法是 各位取反后加一

| $\vec{x}$ | | $\sim\vec{x}$ | | $incr(\sim\vec{x})$ | |
|---|---|---|---|---|---|
| [0101] | 5 | [1010] | −6 | [1011] | −5 |
| [0111] | 7 | [1000] | −8 | [1001] | −7 |
| [1100] | −4 | [0011] | 3 | [0100] | 4 |
| [0000] | 0 | [1111] | −1 | [0000] | 0 |
| [1000] | −8 | [0111] | 7 | [1000] | −8 |

(http://images.cnitblog.com/blog/465162/201404/072133160129852.png)

第二种方法：我们仔细看看这个我们能看出一个规律，就是说我们沿着二进制从右往左走，知道遇到第一个1，然后将第一个1之后的每个位取反就好了。

# 2. 乘法

关于具体怎么做乘法自己去查阅计算机组成原理课本。我们先来看具体的二进制乘法的例子：

| Mode | $x$ | | $y$ | | $x \cdot y$ | | Truncated $x \cdot y$ | |
|---|---|---|---|---|---|---|---|---|
| Unsigned | 5 | [101] | 3 | [011] | 15 | [001111] | 7 | [111] |
| Two's comp. | −3 | [101] | 3 | [011] | −9 | [110111] | −1 | [111] |
| Unsigned | 4 | [100] | 7 | [111] | 28 | [011100] | 4 | [100] |
| Two's comp. | −4 | [100] | −1 | [111] | 4 | [000100] | −4 | [100] |
| Unsigned | 3 | [011] | 3 | [011] | 9 | [001001] | 1 | [001] |
| Two's comp. | 3 | [011] | 3 | [011] | 9 | [001001] | 1 | [001] |

Figure 2.26 Three-bit unsigned and two's-complement multiplication examples. Although the bit-level representations of the full products may differ, those of the truncated products are identical.

(http://images.cnitblog.com/blog/465162/201404/072133176067566.png)

我们发现一个现象，就是说**有符号二进制乘法和无符号二进制乘法，他们获得的最终结果的二进制表示是一样的！！、**

好吧，处于严谨考虑，我们来证明一下下：

$$
\begin{aligned}
(x' \cdot y') \bmod 2^w &= [(x + x_{w-1}2^w) \cdot (y + y_{w-1}2^w)] \bmod 2^w \\
&= [x \cdot y + (x_{w-1}y + y_{w-1}x)2^w + x_{w-1}y_{w-1}2^{2w}] \bmod 2^w \\
&= (x \cdot y) \bmod 2^w
\end{aligned}
$$

(http://images.cnitblog.com/blog/465162/201404/072133192628792.png)

那么我们**怎么来判断乘法会不会发生溢出**呢？我们注意到，上面那个图中的乘法都是溢出的。

先给出代码，大家可以看看这个代码对不对？

```
/* Determine whether arguments can be multiplied without overflow */

int tmult_ok(int x, int y) {

    int p = x*y;

    /* Either x is zero, or dividing p by x gives y */

    return !x || p/x == y;

}
```

好吧，我承认我顽皮了，因为大部分同学（包括曾经的我）根本看不懂这个是什么东东！脑子里就俩个字：尼玛！

我们从理论上证明一下下：

首先我们知道对于w位的x和y，x*y是可以用2w位二进制来表示的，我们将这2w位二进制拆为两部分，高w位（用v来表示）和低w位（用u来表示），于是我们得到$x*y=v*2^w + u$. u代表了代码中p的二进制表示，根据无符号数转换为有符号数的公式我们得到：$u = p + p_{w-1} 2^w$，于是我们可以将$x * y = p + t * 2^w$. 根据溢出的定义，该乘法发生溢出当且仅当 t != 0。

其次，当x!=0的时候我们可以将p表示为p = x * q + r， |r| < |x|。于是我们有x * y = x * q + r + t * 2w。我们需要推导出t == 0的等价条件。当t = 0的时候，我们有x * y = x * q + r，而|r| < |x|，所以必有q=y；反过来,q=y的时候我们也可以推出t=0。也即是说t=0这个条件等价于q=y这个条件（前提是x不为0）。

q=p/x；

于是我们就得到了我们代码里面的判断条件： !x || p/x == y

那么如果我们不允许使用除法，但是允许你使用long long这个数据类型，你怎么来做乘法的溢出检查呢？

哈哈，其实这个就简单了。

```
/* Determine whether arguments can be multiplied without overflow */

 int tmult_ok(int x, int y) {

     /* Compute product without overflow */

     long long pll = (long long) x*y;

     /* See if casting to int preserves value */

     return pll == (int) pll;

 }
```

哈哈，说完这些之后我们来看看怎么将乘法转换为 移位和加法操作， 因为这两种操作时非常快速的。

我们来看，怎么将通过移位的方法一个数扩大两倍？很显然是左移一位。扩大四倍呢？左移两位。

其实我们就类比十进制乘法 列竖的计算式子就可以了。

这里面有个可以被优化的技巧，我们来看。加入被乘数是[(0 . . . 0)(1 . . . 1)(0 . . . 0) ].那么我们还要一个个的移位相加么？其中连续的1，起于从右到左的第m位，终于第n位。

我们可以这样，(x<<n+1) - (x<<m)。

好了，乘法也就这么多了，下面是除法。

# 3. 除法

除法这一块我们只涉及除以一个$2^k$次方。对于正数以及无符号数而言，这意味着右移k位，高位补0。下图中斜体的0都是后来补的0。

| k | >> k (Binary) | Decimal | $12340/2^k$ |
|---|---|---|---|
| 0 | 0011000000110100 | 12340 | 12340.0 |
| 1 | 0001100000011010 | 6170 | 6170.0 |
| 4 | 0000001100000011 | 771 | 771.25 |
| 8 | 0000000000110000 | 48 | 48.203125 |

Figure 2.27 **Dividing unsigned numbers by powers of 2.** The examples illustrate how performing a logical right shift by k has the same effect as dividing by $2^k$ and then rounding toward zero.

(http://images.cnitblog.com/blog/465162/201404/072133203252836.png)

而对于负数而言，我们需要采用逻辑右移，也就是说高位补1。例子见下图：

| k | >> k (Binary) | Decimal | $-12340/2^k$ |
|---|---|---|---|
| 0 | 1100111111001100 | −12340 | −12340.0 |
| 1 | 1110011111100110 | −6170 | −6170.0 |
| 4 | 1111110011111100 | −772 | −771.25 |
| 8 | 1111111111001111 | −49 | −48.203125 |

Figure 2.28 **Applying arithmetic right shift.** The examples illustrate that arithmetic right shift is similar to division by a power of 2, except that it rounds down rather than toward zero.

(http://images.cnitblog.com/blog/465162/201404/072133223404721.png)

但是我们在图中发现一个问题，就是结果可能跟真实的值相差一，比如-12340/8应该为-48而不是-49，-7/2=-3而不是-4。怎么处理这个问题呢？

我们可以对负数做一点预处理，使得x=x+y-1。其中y就是我们的除数。

我们先来看看这么做对不对，(-7+2-1)/2 = (-6/2) =-3 没有问题啊。

证明如下：假设x = qy + r，其中0 ≤r <y。那么，(x + y − 1)/y = q + (r + y − 1)/y。当r=0的时候x/y=q，否则x/y=q+1。符合要求！！

**代码实现是这样的：(x<0 ? x+(1<<k)-1 : x) >> k。**

哈哈，妙吧！

那么如果仅仅允许使用移位和&操作以及加法，怎么代码实现除法呢？（注意除数为$2^k$）

```
int div16(int x) {

/* Compute bias to be either 0 (x >= 0) or 15 (x < 0) */

    int bias = (x >> 31) & 0xF;

    return (x + bias) >> 4;

}
```

看看这个代码写的！！亮瞎了吧。

当x为正数的时候,bias=0, 这没错。

当x为负数的时候，bias为15=16-1。又没问题！

好了，除法也就这些了！

# 4 总结练习

Assume we are running code on a 32-bit machine using two’s-complement arithmetic for signed values. Right shifts are performed arithmetically for signed values and logically for unsigned values. The variables are declared and initialized as follows:

```
int x = foo(); /* Arbitrary value */
int y = bar(); /* Arbitrary value */
unsigned ux = x;
unsigned uy = y;
```

For each of the following C expressions, either (1) argue that it is true (evaluates to 1) for all values of x and y, or (2) give values of x and y for which it is false (evaluates to 0):

A. (x > 0) || (x-1 < 0)

B. (x & 7) != 7 || (x<<29 < 0)

C. (x * x) >= 0

D. x < 0 || -x <= 0

E. x > 0 || -x >= 0

F. x+y == uy+ux

G. x*~y + uy*ux == –x

解答如下：

A. (x > 0) || (x-1 < 0)

False. Let x be −2,147,483,648 (TMin32). We will then have x-1 equal to 2147483647 (TMax32).

B. (x & 7) != 7 || (x<<29 < 0)

True. If (x & 7) != 7 evaluates to 0, then we must have bit x2 equal to 1. When shifted left by 29, this will become the sign bit.

C. (x * x) >= 0

False. When x is 65,535 (0xFFFF), x*x is −131,071 (0xFFFE0001).

D. x < 0 || -x <= 0

True. If x is nonnegative, then -x is nonpositive.

E. x > 0 || -x >= 0

False. Let x be −2,147,483,648 (TMin32). Then both x and -x are negative.

F. x+y == uy+ux

True. Two’ s-complement and unsigned addition have the same bit-level behavior, and they are commutative.

G. x*~y + uy*ux == -x

True. ~y equals -y-1. uy*ux equals x*y. Thus, the left hand side is equivalent to x*-y-x+x*y.

## 你可能感兴趣的

- **书其实只有三类 (/article/106.htm)** 西蜀石兰 类 (/search/%E7%B1%BB/1.htm)
- **《TCP/IP 详解，卷1：协议》学习笔记、吐槽及其他 (/article/233.htm)** bylijinnan tcp (/search/tcp/1.htm)
- **Linux—— 静态IP跟动态IP设置 (/article/360.htm)** eksliang linux (/search/linux/1.htm) IP (/search/IP/1.htm)
- **Informatica update strategy transformation (/article/487.htm)** 18289753290
- **使用Scrapy时出现虽然队列里有很多Request但是却不下载，造成假死状态 (/article/614.htm)** 酷的飞上天空 request (/search/request/1.htm)
- **利用预测分析技术来进行辅助医疗 (/article/741.htm)** 蓝儿唯美 医疗 (/search/%E5%8C%BB%E7%96%97/1.htm)
- **java 线程(一)：基础篇 (/article/868.htm)** DavidIsOK java (/search/java/1.htm) 多线程 (/search/%E5%A4%9A%E7%BA%BF%E7%A8%8B/1.htm) 线程 (/search/%E7%BA%BF%E7%A8%8B/1.htm)
- **Tomcat服务器框架之Servlet开发分析 (/article/995.htm)** aijuans servlet (/search/servlet/1.htm)