

信号的表示和处理

- 信号的表示和处理
 - 编码
 - 无符号(Unsigned)数的编码
 - 补码(Two's-complement)数的编码
 - 其他编码方式
 - 反码(Ones' Complement)编码
 - 原码(Sign-Magnitude)编码
 - 无符号数与补码数的转换
 - 补码转无符号
 - 无符号转补码
 - 拓展一个数字的位表示
 - 无符号的拓展
 - 补码数的拓展
 - 截断数字
 - 截断无符号数
 - 截断补码数
 - 整数运算
 - 右移方式
 - 无符号加法
 - 无符号加法溢出的检测
 - 无符号的非
 - 补码加法
 - 补码加法溢出的判断
 - 补码的非
 - 快速求补码非的办法
 - 无符号乘法
 - 补码乘法
 - 乘以常数
 - 乘以 2 的次幂
 - 乘以任意常数
 - 除以 2 的幂
 - 取整的描述
 - 无符号除法
 - 补码除法
 - 浮点数

编码

无符号(Unsigned)数的编码

设一个整数数据有 ω 位，我们可以将其写成位向量 \vec{x} 来表示该向量，或者将其每一位都写出来

$$\vec{x} = [x_{\omega-1}, x_{\omega-2}, \cdots, x_1, x_0]$$

对于向量 $\vec{x} = [x_{\omega-1}, x_{\omega-2}, \cdots, x_1, x_0]$ ，有：

$$B2U_{\omega}(\vec{x}) = \sum_{i=0}^{\omega-1} x_i 2^i$$

补码(Two's-complement)数的编码

对于向量 $\vec{x} = [x_{\omega-1}, x_{\omega-2}, \cdots, x_1, x_0]$ ，有：

$$B2T_{\omega}(\vec{x}) = -x_{\omega-1} 2^{\omega-1} + \sum_{i=0}^{\omega-2} x_i 2^i$$

对于补码数的编码，其实就是**无符号数的最高位变为负数**，其余都是一样的

我们以 4 位的无符号数与补码数为例，讨论二者的表示范围：

对于 4 位的无符号数的范围，有：0 ~ 15

对于 4 位的补码数的范围，有：-8 ~ 7

对于补码数而言，负数的个数有 8 个，**非负数**的个数也有 8 个，这导致了： $|TMin_{\omega}| = |TMax_{\omega} + 1|$

其他编码方式

反码(Ones' Complement)编码

$$B2O_{\omega}(\vec{x}) = -x_{\omega-1}(2^{\omega-1} - 1) + \sum_{i=0}^{\omega-2} x_i 2^i$$

反码编码只是在**补码的基础上**将最高位的权重从 $-2^{\omega-1}$ 变为 $-(2^{\omega} - 1)$ ，其余都跟补码一样

原码(Sign-Magnitude)编码

$$B2S_{\omega}(\vec{x}) = (-1)^{x_{\omega-1}} + \sum_{i=0}^{\omega-2} x_i 2^i$$

原码的最高位为**符号位**，其余位来决定该数的大小

这两种编码都有一个问题是，对于数字 0 会有两种解释方式

在原码中 $[00 \cdots 0]$ 被解释为 $+0$ ， $[10 \cdots 0]$ 被解释为 -0

在反码中 $[00 \cdots 0]$ 被解释为 $+0$ ， $[11 \cdots 1]$ 被解释为 -0

实际上，浮点数的编码方式就是才有原码编码

无符号数与补码数的转换

对于补码和无符号数之间的转换，**底层的位值表示是不变的**，只是改变了解释这些位的方式

补码转无符号

$$T2U_{\omega}(\vec{x}) = \begin{cases} x + 2^{\omega} & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$

无符号转补码

$$U2T_{\omega}(\vec{x}) = \begin{cases} x & \text{if } x \leq TMax \\ x - 2^{\omega} & \text{if } x > TMax \end{cases}$$

拓展一个数字的位表示

我们主要讨论从一个较小的类型转换到较大的类型

无符号的拓展

无符号的拓展被称为零拓展(zero extention)，只需要简单的在开头添加 0 即可，其原理如下：

定义长度为 ω 的位向量 $\vec{u} = [u_{\omega-1}, u_{\omega-2}, \cdots, u_1, u_0]$ 和长度为 ω' 的位向量 $\vec{u}' = [0, 0, \cdots, 0, u_{\omega-1}, u_{\omega-2}, \cdots, u_1, u_0]$ ，其中 $\omega' > \omega$

此时有： $B2U(\vec{u}) = B2U(\vec{u}')$

补码数的拓展

补码数的拓展被称为符号拓展(sign extention)，即在前面**添加最高有效位**

最高有效位为 0，那么添加 0；最高有效位为 1，那么添加 1

定义长度为 ω 的位向量 $\vec{u} = [u_{\omega-1}, u_{\omega-2}, \cdots, u_1, u_0]$ 和长度为 ω' 的位向量 $\vec{u}' = [u_{\omega-1}, u_{\omega-1}, \cdots, u_{\omega-1}, u_{\omega-2}, \cdots, u_1, u_0]$ ，其中 $\omega' > \omega$

我们有： $B2T(\vec{u}) = B2T(\vec{u}')$

推导：

我们只需证明： $B2T([u_{\omega-1}, u_{\omega-2}, \cdots, u_1, u_0]) = B2T([u_{\omega-1}, u_{\omega-1}, u_{\omega-2}, \cdots, u_1, u_0])$ ，便可以说明对于任意位数的符号拓展，都是满足条件的

我们按照补码数的定义展开，有：

$$B2T([u_{\omega-1}, u_{\omega-2}, \cdots, u_1, u_0]) = -u_{\omega-1}2^{\omega} + \sum_{i=0}^{\omega-2} u_i 2^i = -u_{\omega-1}2^{\omega} + u_{\omega-1}2^{\omega-1} + \sum_{i=0}^{\omega-2} u_i 2^i = -u_{\omega-1}2^{\omega-1} + \sum_{i=0}^{\omega-2} u_i 2^i = B2T([u_{\omega-1}, u_{\omega-2}, \cdots, u_1, u_0])$$

本质上是使用属性： $2^{\omega} - 2^{\omega-1} = 2^{\omega-1}$

截断数字

截断无符号数

设 \vec{x} 表示位向量 $[x_{\omega-1}, x_{\omega-2}, \cdots, x_1, x_0]$ ， \vec{x}' 为其**截断为 k 位**的结果： $\vec{x}' = [x_{k-1}, x_{k-2}, \cdots, x_1, x_0]$ ，我们有： $x' = x \bmod 2^k$

$$B2U_{\omega}([x_{\omega-1}, x_{\omega-2}, \dots, x_1, x_0]) \bmod 2^k = \left(\sum_{i=0}^{\omega-1} x_i 2^i \right) \bmod 2^k = \left(\sum_{i=0}^{k-1} x_i 2^i \right) \bmod 2^k = B2U_{\omega}([x_{k-1}, x_{k-2}, \dots, x_1, x_0])$$

本质上是对于任意的 $i \geq k$ ，都有： $2^i \bmod 2^k = 0$

截断补码数

设 \vec{x} 表示位向量 $[x_{\omega-1}, x_{\omega-2}, \dots, x_1, x_0]$ ， $\vec{x'}$ 为其截断为 k 位的结果： $\vec{x'} = [x_{k-1}, x_{k-2}, \dots, x_1, x_0]$ ，我们有： $x' = U2T(x \bmod 2^k)$

本质上是先将位向量截断为 k 位，然后在从补码的角度诠释位表示

整数运算

右移方式

右移方式有两种：逻辑右移与算术右移

前者会直接补 0，后者会补最高有效位；如果最高有效位为 0，那么便补 0，如果最高有效位为 1，那么便补一

对于无符号数，右移均为逻辑右移

对于补码数，右移均为算术右移

无符号加法

设 $0 \leq x, y \leq 2^{\omega}$ ，有：

$$\begin{aligned} x +_{\omega} y &= \begin{cases} x + y, & \text{if } x + y \leq 2^{\omega} \\ x + y - 2^{\omega}, & \text{if } x + y > 2^{\omega} \end{cases} \\ &\quad \text{正常} \\ &\quad \text{溢出} \end{aligned}$$

两个 ω 位的无符号数相加，其结果可能需要 $\omega + 1$ 位来表示

若两个 ω 位的无符号数相加，如果加和会自动截断为 ω 位

无符号加法溢出的检测

设 $0 \leq x, y \leq UMax_{\omega}$ ，令 $s = x +_{\omega} y$ ，当且仅当 $s < x$ （或等价条件 $s < y$ ）成立时，发生了溢出

代码判断：

```
//未发生溢出时返回true，发生溢出时返回false
bool uadd_ok(unsigned x, unsigned y)
{
    unsigned sum = x + y;
    return sum >= x;
}
```

无符号的非

设 $0 \leq x < 2^{\omega}$ ，其 ω 位的逆元 $-_{\omega}^u x$ 有下式给出：

$$-_{\omega}^u x = \begin{cases} x, & \text{if } x = 0 \\ 2^{\omega} - x, & \text{if } x > 0 \end{cases}$$

求无符号的非，本质上是对于一个无符号数 x ，找到一个 y 使得：

$$(x +_{\omega}^u y) \bmod 2^{\omega} = 0$$

因此直接在 x 的基础上加一个使其溢出的数即可

补码加法

设 $-2^{\omega-1} \leq x, y \leq 2^{\omega-1} - 1$ ，有：

$$\begin{aligned} x +_{\omega} y &= \begin{cases} x + y, & \text{if } x + y \geq 2^{\omega-1} \\ x + y - 2^{\omega-1}, & \text{if } x + y < 2^{\omega-1} \end{cases} \\ &\quad \text{正溢出} \\ &\quad \text{负溢出} \end{aligned}$$

对于 ω 位的补码数可表示的最小值为 $-2^{\omega-1}$ ，可表示的最大值为 $2^{\omega-1} - 1$

因此如果两个 ω 位的补码数相加，其结果可能需要 $\omega + 1$ 位来表示

如果结果大于 $2^{\omega-1} - 1$ ，则发生**正溢出**， $\omega - 1$ 位的值由 0 变为 1，数值上相当于加和结果**减去** $2 \times 2^{\omega-1} = 2^\omega$

如果结果小于 -2^ω ，则发生**负溢出**， ω 位的会溢出为 1，由于我们需要将该位舍去，因此数值上相当于加和结果**加上** 2^ω

补码加法溢出的判断

设 $TMin \leq x, y \leq TMax$ ，令 $s = x +_\omega y$

当且仅当 $x > 0, y > 0$ 而 $s \leq 0$ 时，发生**正溢出**

当且仅当 $x < 0, y < 0$ 而 $s \geq 0$ 时，发生**负溢出**

代码判断：

```
//返回-1表示负溢出，1表示正溢出，0表示正常
int tadd_ok(int x, int y)
{
    int sum = 0;
    if(x < 0 && y < 0 && sum >= 0) return -1;
    else if(x > 0 && y > 0 && sum <= 0) return 1;
    else return 0;
}
```

该程序不能写成：

```
int tadd_ok(int x, int y)
{
    int sum = x + y;
    return (sum - x == y) && (sum - y == x);
}
```

这是因为，**无论补码加法是否溢出， $\text{sum} - x == y$ 都始终成立**（补码加法是模数加法，具有循环的特性）

如果需要判断 $x - y$ 是否发生溢出，不能写成：

```
int tsub_ok(int x, int y)
{
    return tadd_ok(x, -y);
}
```

当 y 取除 $TMin$ 以外的任何值时，都是满足条件的，但当 $y = TMin$ 时，此函数做出了与我们期望相反的行为

当 $y = TMin$ 时，该函数认为只要 $x \geq 0$ 时就不会发生溢出，只要 $x < 0$ 时就会发生溢出，但实际与此刚好相反

当 $y = TMin$ 时， $-y = TMin$ ，若 $x \geq 0$ ，其实际计算结果为**正数**，但位表示却为负数，因此发生了溢出；同理，若 $x < 0$ ，其实际计算结果**依旧为正数**，在位表示上，溢出的一位会被舍去，而符号位为 0，即位表示为正数，因此没有发生溢出

正确写法：

```
//溢出返回1，未溢出返回0
int tsub_ok(int x, int y)
{
    int sub = x - y;
    return (x > 0 && y < 0 && sub < 0) || (x < 0 && y > 0 && sub > 0);
}
```

补码的非

设 $TMin \leq x \leq TMax$ ， x 的补码非 $-_\omega x$ 由下式给出：

$$-_\omega x = \begin{cases} TMin - x & \text{if } x \geq 0 \\ x + TMax + 1 & \text{if } x < 0 \end{cases}$$

补码的非本质上同无符号的非一样，都是找一个数 y 使得 $(y +_\omega x) \bmod 2^\omega = 0$ ，本质上都是模数加法的溢出

在位表示上，对于任意值 x ， $\sim x + 1$ 所求得的就是 x 的非，即 $-x$

快速求补码非的办法

在 x 的二进制表示中, 设 k 为最右边的 1 的位置, 即 x 的位表示为 $[x_{\omega-1}, x_{\omega-2}, \dots, x_{k+1}, 1, 0, \dots, 0]$ (只要 x 不为 0 就总能找到这样的 k)

这个值的非的位表示为 $[\sim x_{\omega-1}, \sim x_{\omega-2}, \dots, \sim x_{k+1}, 1, 0, \dots, 0]$

也就等同于, 将 k 的左边所有位均**取反**

无符号乘法

对于 $0 \leq x, y \leq UMax$, 无符号乘法结果由下式给出:

$$x *_{\omega}^u y = (x * y) \bmod 2^{\omega}$$

两个 ω 位的数相乘, 结果可能需要 2ω 位才能表示, 因此无符号程序会将结果**截断为 ω 位**

补码乘法

对于 $TMin \leq x, y \leq TMax$, 补码乘法结果由下式给出:

$$x *_{\omega}^t y = U2T((x * y) \bmod 2^{\omega})$$

其实就是将乘法结果**截断为 ω 位**, 然后再从**补码的角度解释**

乘以常数

乘以 2 的次幂

我们先讨论乘以 2 的次幂的情况

无论是无符号数还是补码数, 乘以 2^k 相当于将位表示左移 k 个单位, 然后再对其进行截断, 即:

$$x *_{\omega}^t (x \ll k) = U2T((x \ll k) \bmod 2^{\omega})$$

乘以任意常数

对于乘以任意常数 K 的情况, 我们将 K 的二进制表示写出来: $[(0 \dots 0)(1 \dots 1)(0 \dots 0)]$, 必然会是**连续的 0, 1 交替**的情况

例如, 14 可以写为 $[(0 \dots 0)(111)(0)]$, 考虑**位位值**从 n 到 m 的一组连续的 1 ($n \geq m$, 且二者均从 0 开始)。(对 14 来说, $n = 3, m = 1$) 我们可以用以下两种方式来将乘以任意常数 K 转换成加法、减法与移位操作:

- $(x \ll n) + (x \ll (n-1)) + \dots + (x \ll m)$
- $(x \ll (n+1)) - (x \ll m)$

问题: 如果加法、减法、移位所需的时间相同, 那么当 n, m 取不同值时, 编译器该如何决定选择哪一种操作?

分类讨论:

如果 $n = m$, 那么第一种只需要一次移位, 第二种需要一次减法和两次移位

如果 $n = m + 1$, 那么第一种需要两次移位和一次加法, 第二种需要两次移位和一次减法

如果 $n > m + 1$, 那么第二种只会需要两次移位和一次减法, 第一种需要 $n - m + 1$ 次移位和 $n - m$ 次加法

综上, 我们有如下结论:

- 若 $n = m$, 选第一种
- 若 $n = m + 1$, 哪种都行
- 若 $n > m + 1$, 选第二种

除以 2 的幂

取整的描述

对于除法, 我们期望其结果**总是舍去小数**, 这一点对正数和负数我们都期望是成立的

例如, 我们期望: $5 \div 2 = 2, -5 \div 2 = -2$

这种舍入我们称为**向零舍入**, 即直接舍去小数位

为了更便于描述该过程, 我们需要引入两个符号: 下取整与上取整

对于任意实数 x , 我们定义 $\lfloor x \rfloor$ 生成唯一**整数** x' 使得 $x' \leq x < x' + 1$, $\lceil x \rceil$ 生成唯一**整数** x' 使得 $x' - 1 < x \leq x'$

例如:

$\lfloor 2.5 \rfloor$ 生成 $x' = 2$, 使得 $x' \leq 2.5 < x' + 1$

$\lfloor -2.5 \rfloor$ 生成 $x' = -3$, 使得 $x' \leq -2.5 < x' + 1$

$\lceil 2.5 \rceil$ 生成 $x' = 3$, 使得 $x' - 1 < 2.5 \leq x'$

$\lceil -2.5 \rceil$ 生成 $x' = -2$, 使得 $x' - 1 < -2.5 \leq x'$

我们发现，对于整数的**向零舍入**，恰好用**下取整**便可以描述，而对于负数的**向零舍入**，我们需要**上取整**才可以描述

无符号除法

若一个无符号数除以 2^k ，则对其执行**算术右移 k 个单位**，所得二进制表示便是**下取整**的结果

例如： $5 = [0101]$ ，将其右移一位（除以 2^1 ），所得位表示为 $2 = [0010]$ ，我们发现这恰好变式**下取整**

对于无符号数值 x 和整数 k 而言， $x \gg k = \lfloor x/2^k \rfloor$

补码除法

对于补码的除法，我们需要在右移的过程中保证符号不变，因此必须执行**算术右移**

我们从一个例子出发，考虑对负数直接执行算术右移的操作后，其结果如何

如果我期望将 $-5 = [1011]$ 右移一位除以 2^1 ，那么所得位表示为 $-3 = [1101]$

我们发现，不论是逻辑右移还是算术右移，其结果**都是下取整**。这对于正数是合理的，但对于负数却不是我们希望的，我们需要一种办法将下取整转换为上取整

我们利用如下等式：

$$\lceil x/y \rceil = \lfloor (x + y - 1)/y \rfloor$$

例如，当 $x = -30, y = 4$ 时， $\lceil -30/4 \rceil = -7$ ，此时我们有 $x + y - 1 = -27$ ，而 $\lfloor (x + y - 1)/y \rfloor = \lfloor -27/4 \rfloor = -7$

对于补码数 x 和整数 k 而言， $x \gg k = \lceil x/2^k \rceil = \lfloor (x + 2^k - 1)/2^k \rfloor = (x + (1 \ll k) - 1) \gg k$

到此为止，我们已经讨论了除以 2 的幂次的情况，但不幸的是，除法无法像乘法那样推广到除以任意常数 K 。换句话说，对于任意常数 K ，我们无法将其简单地转为**除以多个 2 的幂次相加的情况**

你也许会想，任意一个数都可以分解为 2 的幂次，并且乘法与除法都满足**分配律**，为什么除法不能像乘法那样分解呢？

这里的问题在于，除法会有**舍入**的问题（而乘法不具有这种问题），如果将其简单的分解为多个数相除，那么小数的部分将会全部被舍去，这将会导致最终的结果与我们所预期的相差过大

浮点数