谈补码

张帅

2014 年 1 月 25 日

摘要

现在一谈到补码，很多人都会立刻说**补码就是反码加一**，虽然没说错，但是却没说道点子上。我更希望能够听到有人说**补码就是负数对应原码的表示方式**。这篇文章主要论述补码的本质，以及有符号整数二进制表示的原因。

计算机科学不像其他的自然科学，比如物理，是由观察到的现象总结出结论，而是完全由人为规定发展而来的一门科学。因此，计算机科学中的每一个现象都是有着合理的解释的，而不能说我观察到的现象就是这样来解释。以下这些问题你可能遇到过，但是却没有深思过，以至于学过了就过去了，过不了多久就忘得干干净净，没有真正的理解：

1. 为什么补码 $=$ 反码 $+1$？
2. 为什么有符号整数的符号位用 0 表示正整数，用 1 表示负数？
3. 为什么 32 位整数的表示范围是 $[-2^{31}, 2^{31}-1]$，最小值的绝对值比最大值的绝对值大 1？
4. 为什么有**零扩展**和**符号扩展**两种不同的运算？
5. 为什么左移运算只有一种，而右移运算有**算术右移**和**逻辑右移**两种？

这些问题其实都跟补码这一概念有关，都能够从补码的概念去解释。下面我们将依次解答这些问题。

一 从历史的角度看补码

下面谈到的历史都是我杜撰的，但是不是我随意编的，而是根据计算机组成原理[1, 2]的知识脑补的。我估计跟历史的发展相差不大，欢迎各位纠正或补充资料。

首先，我们知道可以用**原码**，以二进制的形式来表示一个十进制自然数。

计算机硬件的基础是数字电路，数字电路中最基础的元件是**逻辑门**[3]。要想完成两个数字相加的运算，需要一个叫做**加法器**[4]的元件。显然，加法器是由逻辑门构成的，这是一个比较复杂的过程。

如果我们在计算机中只使用原码，那么为了进行减法运算，我们还需要重新设计一个减法器。上面已经提到过了，加法器的设计就已经很麻烦了，减法又需要借位等操作，重新设计起来很麻烦。想一想我们学习十进制数的过程，首先我们认识了自然数，然后，我们认识了负数。我们知道，$A-B=A+(-B)$。因此，如果有一种二进制格式可以表示负数，我们就无需再重新设计一个减法器来做减法运算了，而是可以通过加上一个数的负数来计算减法。

补码就是这个用于表示负数的二进制编码。

换句话说，假设 A 是一个十进制数，则 $(A)_{原}+(A)_{补}=0$。这里需要注意到一个重要的事实：若只是进行自然数的加法，除非两个操作数都是 0 不然不可能结果为 0。那我们怎么做到让 $(A)_{原}+(A)_{补}=0$ 呢？我们知道，计算机中表示的整数都是有限精度的，例如，一个 32 位无符号整数的范围是 $[0,2^{32}-1]$。如果运算结果是超出其表示范围的整数，则会发生**溢出**，通常对溢出的处理办法是截取，例如 4 位二进制无符号整数加法运算中 $[1101]+[0111]=[10100]$，而 $[10100]$ 无法使用 4 位寄存器保存，因此发生溢出，截取低四位结果 $[0100]$。这一奇妙的性质使得我们有可能令 $(A)_{原}+(A)_{补}=[1|0\ldots0]=0$。

为方便起见，我们以 4 位二进制数运算为例，则有 $(A)_{补}=[10000]-(A)_{原}$。由于 $(A)_{反}$ 中的每一个二进制位都和 $(A)_{原}$ 不同，因此 $(A)_{反}+(A)_{原}$ 的每一个二进制位都不会有进位且恰好为 1，即 $(A)_{反}+(A)_{原}=[1111]$。因此，我们可以得到：

$$\begin{aligned}(A)_{补} &= [10000] - (A)_{原} \\ &= [0001] + [1111] - (A)_{原} \\ &= 1 + (A)_{反}\end{aligned} \tag{1}$$

也就是**补码等于反码加一**。

这下，我们既有正数又有负数，可以统一的使用加法器来做整数的加减法了，这是一个非常振奋的消息。但是别着急，我们现在有两套整数的表示方法——原码和补码，对于给定的二进制序列，我们怎么知道这是原码还是补码呢？例如 [0110] 如果按照原码来解释的话，就是十进制的 6，但是如果按照补码来解释的话，就是十进制的 −10。我们将在下一节解决这一问题。

二 有符号整数

现在，一个整数具有两种编码形式，正好可以分别用二进制的 0 和 1 来表示，我们不妨在最高位额外添加一位，用于标记该编码是使用原码表示的还是补码表示的，并称这一位为**符号位**。（我知道这部分和书上讲的不太一样，别着急，接着往下看）

现在有一个问题，我们应该用 0 来指示接下来的二进制序列是原码还是补码呢？理论上讲都可以，但是也许有一种情况更加的方便。

[1] 中有这样一个函数：

$$\mathrm{B2T}_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \tag{2}$$

用于将补码转换为十进制。对比一下将原码转换为十进制的函数：

$$\mathrm{B2S}_w(\vec{x}) \doteq (-1)^{x_{w-1}} \cdot \left(\sum_{i=0}^{w-2} x_i 2^i\right)$$

最大的不同在于 $\mathrm{B2S}_w$ 函数中，最高位仅用于确定符号为正还是为负，而 $\mathrm{B2T}_w$ 函数中最高位参与运算。例如：

$$\begin{aligned}\mathrm{B2T}_5([11011]) &= -1\cdot 2^4 + \sum_{i=0}^{3} x_i 2^i \\ &= -2^4 + 2^3 + 2^1 + 2^0 \\ &= -5\end{aligned} \tag{3}$$

突然觉得这个公式有点眼熟，不就是上面的式 1 吗？将式 1 变换一下，可以得到如下公式：

$$-(A)_{原} = -[10\ldots0] + (A)_{补} \tag{4}$$

正好是上面式 3 中用到的形式。而如果符号位为 0 的话，则恰好 $\mathrm{B2T_w}(\vec{\mathrm{x}})$ 和 $\mathrm{B2S_w}(\vec{\mathrm{x}})$ 完全相同，都是我们非常熟悉的将二进制转换为十进制的方法。因此，我们将符号位解释为负权，恰好能够统一的解决二进制转换为十进制的问题。

当然，我们也可以用符号位的 0 来表示接下来的是补码表示的负数，但是这就需要将符号位的 0 映射为 −1 并且将 1 映射为 0，才能够统一的使用和式 $\mathrm{B2U}_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i$；而不像现在这样，只需将符号位的 1 映射为 −1 即可满足。

三　有符号整数的相关问题

我们刚刚已经回答了最初的两个问题：

1. 为什么补码 = 反码 + 1？
2. 为什么有符号整数的符号位用 0 表示正整数，用 1 表示负数？

接下来，我们还要回答这样三个问题：

3. 为什么 32 位整数的表示范围是 $[-2^{31}, 2^{31}-1]$，最小值的绝对值比最大值的绝对值大 1？
4. 为什么有**零扩展**和**符号扩展**两种不同的运算？
5. 为什么左移运算只有一种，而右移运算有**算术右移**和**逻辑右移**两种？

3.1 有符号整数的表示范围

我们需要注意这样一个事实：0 的负值还是 0。也就是说，在刚才所说的有符号整数的定义中，含有两个零：0 和 −0。在 3 位二进制整数中，$(0)_{原} \doteq [000]$，而 $(-0)_{补} \doteq [000]$。如果只是将符号位解释为接下来的编码是原码还是反码的话，我们将得到两个零：$+0 = [0\,000]$ 和 $-0 = [1\,000]$。但是实际上我们只需要使用 +0 就足够了，方便又省事。这样就有一个问题：−0 怎么办？

我们来看一下下面几个式子：

$$\begin{aligned}(-7)_{补} &= [1\,001]\\(-1)_{补} &= [1\,111]\\(-7)_{补} + (-1)_{补} &= [1\,001] + [1\,111]\\\text{（溢出结果）} &= [1\,000]\end{aligned}$$

也就是说，我们正好可以用 −0 的位置来表示 −8（即 -2^{w-1}），这也正好与式 2 不谋而合。我并不清楚这两者究竟是谁促成了谁，抑或这只是一次完美的巧合，但是这样去解释有符号整数的最小值的绝对值恰比最大值大 1 这一问题，恰到好处。

3.2 整数的扩展

整数的**扩展**发生在这样一种情况下。例如，我们现在正处在 32 位处理器向 64 位处理器过度的时期，有大量的代码是针对 32 位处理器编写的。为了保证兼容性，通常可以使用 64 位处理器运行这些代码。但是 64 位处理器的寄存器都是 64 位的，各种指令也是用于处理 64 位整数的。此时，在将数据从内存载入到寄存器的时候，就需要进行整数的**扩展**。

显然的，扩展的基本原则就是：**扩展之后的数和扩展之前的数必须解释为同一个数**。

对于原码表示来说，我们无论在高位添加多少个 0，都不会改变数值。但是对于补码而言，是否仍然如此呢？答案是否定的，因为原本的最高位解释为

负权，但是扩展之后解释为正权，因此会相差 $2\cdot 2^{w-1}=2^{w}$，恰为新扩展出的最高位权重，因此我们只需将扩展后的符号位置 1，即可保证扩展之后的值和原值相同。这样递归的做下去，我们就可以扩展任意位数，并最终总结出这样的规律：

> 对于无符号整数，我们在扩展的时候总是将扩展的位置 0；对于有符号整数，我们在扩展的时候总是将扩展的位置为与符号位相同。

这也就是为什么，我们需要**零扩展**和**符号扩展**两种运算。

3.3 左移运算和右移运算

很久以前，我就注意到左移运算只有一种，但是右移运算却有两种。这种不对称的现象，令我百思不得其解。我们先来看这两种右移运算有什么区别：[5]

算术右移（SAR） In a right arithmetic shift, the sign bit is shifted in on the left, thus preserving the sign of the operand.

逻辑右移（SHR） Logical right-shift inserts value 0 bits into the most significant bit, instead of copying the sign bit.

这种区别，似乎跟位扩展非常相似。我们可以先假设认为，逻辑右移是用于无符号整数的，而算术右移是用于有符号整数的，也许我们能够为这种假设找到一些理由。

如果逻辑右移是用于无符号整数的，我们来看一下逻辑右移做了什么：$\mathrm{SHR}_k(n)=\lfloor\frac{n}{2^k}\rfloor$。相应的，算术右移很有可能是为了在有符号整数中保持这个性质而设计的。对于一个正整数而言，其符号位为零，因此算术右移等同于逻辑右移。这是因为，一个正整数除了符号位以外，都是使用原码表示的。而对于一个负整数呢？我们不妨来看这样一个例子：

$$
\begin{aligned}
\mathrm{B2T}_4([1\,a\,b\,c]) &= -2^3 + a \cdot 2^2 + b \cdot 2^1 + c \cdot 2^0 \\
&= -8 + 4a + 2b + c \\
\mathrm{SAR}_1([1\,a\,b\,c]) &= [s\,1\,a\,b] \\
\mathrm{B2T}_4([s\,1\,a\,b]) &= s \cdot -2^3 + 1 \cdot 2^2 + a \cdot 2^1 + b \cdot 2^0 \\
&= -8s + 4 + 2a + b
\end{aligned}
$$

令

$$
\mathrm{B2T}_4(\mathrm{SAR}_1([1\,a\,b\,c])) = \left\lfloor \frac{\mathrm{B2T}_4([1\,a\,b\,c])}{2} \right\rfloor
$$

即

$$
\begin{aligned}
-8s + 4 + 2a + b &= \left\lfloor \frac{-8 + 4a + 2b + c}{2} \right\rfloor \\
&= -4 + 2a + b + \left\lfloor \frac{c}{2} \right\rfloor
\end{aligned}
$$

化简得

$$
\begin{aligned}
-8s + 8 &= \left\lfloor \frac{c}{2} \right\rfloor \\
\because c &= 0 \quad \text{or} \quad 1 \\
\therefore \left\lfloor \frac{c}{2} \right\rfloor &= 0 \\
\therefore -8s + 8 &= 0 \\
\therefore s &= 1
\end{aligned}
$$

容易证明这一结论的推广情况。也就是说，为了维持上面提到的性质，在对有符号整数进行右移的时候，需要在前方补充和最初符号位相同的值，而不能仅仅填充 `0`。

参考文献

[1] Randal Bryant, *Computer systems: A Programmer's Perspective*. Addison-Wesley, Massachusetts, 2nd Edition, 2010.

[2] 唐朔飞, 计算机组成原理. 高等教育出版社, 北京, 第 2 版, 2008.

[3] 逻辑门. http://zh.wikipedia.org/wiki/%E9%80%BB%E8%BE%91%E9%97%A8.

[4] 加法器. http://zh.wikipedia.org/zh/%E5%8A%A0%E6%B3%95%E5%99%A8

[5] Bitwise operation, http://en.wikipedia.org/wiki/Bitshift