A Note on Two's Complement Representation of Integers

Li Yue

March 9, 2022

This note addresses the following questions:

- The modern way to compute the two's complement of a binary integer is to invert all its bits and then add 1 to the least significant bit (LSB) (e.g., the two's complement of 0010 is 1110, that is, from 0010 to 1101 and then to 1110). But what does this have to do with getting a complement w.r.t. *two*?
- What happens when intuitively the sum of two intergers (both in two's complement form) are out of the range ? And how to tackle this situation? What we should expect from two's complement addition ?

1 The Syntactic-Semantic Parallel Domains

The key is that we are thinking in two parallel domains : the *syntactic* domain and the *semantic* domain. The semantic domain Q is rational numbers[1] such as -1.01 (minus one and a quarter), 0.11 (three quarters) and 10.00 (two) and their sums such as $(-1.01) + 10.00 = 0.11$ and $(-1.00) + (-0.11) = (-1.11)$. The syntactic domain W is fixed-length (say, three-bit) binary words such as 001 and 111 and their sums such as $001 +' 111 = 000$ (the last carry bit is lost) and $010 +' 101 = 111$ — note that we use $+$ for addition in the semantic domain and $+'$ for addition in the syntactic domain. Entities in the syntactic domain are given interpretations in terms of entities in the semantic domain. The validity of a computer relies on commuting of the diagrams below.

$$\begin{array}{ccc} Q \times Q & \xrightarrow{+} & Q \\ \downarrow{\scriptstyle (f,f)} & & \downarrow{\scriptstyle f} \\ W \times W & \xrightarrow{+'} & W \end{array} \qquad \begin{array}{ccc} Q \times Q & \xrightarrow{+} & Q \\ \uparrow{\scriptstyle (f^{-1},f^{-1})} & & \uparrow{\scriptstyle f^{-1}} \\ W \times W & \xrightarrow{+'} & W \end{array}$$

In the diagrams, which highlight the syntactic-semantic distinction, the mapping f is called *representation* and its inverse f^{-1} is *interpretation*. The particular choice for f is left unspecified. In practice f could be fixed-point or floating-point, unsigned or signed and in the last case sign-magnitude or two's complement, or whatever useful scheme of representation. The idea is the agreement between "sum then represent" and "represent then sum"; or equivalently, between "interpret then sum" and "sum then interpret".

[1] We use binary notation for convenience; also note that if the semantic domain becomes R — the real numbers, our discussion would be very different.

2 Complements

The complement $C_a(b)$ of a number b w.r.t. a is defined $a - b$; we call $a - b$ the a's complement of b. For example, $C_2(1.4) = 0.6$ — the two's complement of 1.4 is 0.6; $C_{13}(6) = 7$ — the thirteen's complement of six is seven.

For any $n' \in W$, we define $C(n')$ as the word in W obtained by flipping all 1's and 0's in n' and then add 1 bit to the LSB; e.g. $C(000) = 000$, $C(111) = 001$ and $C(010) = 110$.

3 Historical Context

A group of researchers in the 1950s were developing a binary circuit computer[2] and they were focusing on computation with magnitudes no more than 1; two's complement was then proposed for representing numbers in the range $[-1, 1)$. All our discussion below is situated in this setting: we focus on computation of numbers with magnitudes no greater than one.

3.1 Unsigned Numbers and One's Complement

Conceptually the representation of non-negative numbers in the *unsigned* way is more basic, which we discuss now. We want to represent numbers in the range $Q_1 = [0, 1) \subseteq Q$. For example, $0.11 \in Q_1$ is represented as $110 \in W$ by removing the prefix "0." and pad zeros (or drop digits) in the end to fill the word length; inversely, a word is interpreted by setting a binary point to the left of the first significant bit; for example, to interprete $101 \in W$, we assume a binary point to the left of the MSB, and we get $.101 \in Q_1$, which is a half with one eighth. We use $f_u : Q_1 \mapsto W$ to denote unsigned representation of number's in Q_1; its inverse is denoted ${f_u}^{-1}$. Note that word length of W is unspecified as long as it is fixed uniformly for all elements of W.

Proposition 3.1. *The diagrams does <u>not</u> commute.*

$$\begin{array}{ccc} Q_1 & \xrightarrow{C_1} & Q_1 \\ \downarrow{\scriptstyle f_u} & & \downarrow{\scriptstyle f_u} \\ W & \xrightarrow{C} & W \end{array} \qquad \begin{array}{ccc} Q_1 & \xrightarrow{C_1} & Q_1 \\ {\scriptstyle {f_u}^{-1}}\uparrow & & \uparrow{\scriptstyle {f_u}^{-1}} \\ W & \xrightarrow{C} & W \end{array}$$

Proof. We first discuss about the diagram on the left. Assume the word length is i. A typical $n \in Q_1$ has the shape

$$n = 0 \ . \ b_1 \ b_2 \ b_3 \ \cdots \ b_i \ b_{i+1} \ b_{i+2} \ \cdots \ b_{i+j}$$

Define $\bar{1} = 0$ and $\bar{0} = 1$, we have

$$C_1(n) = 0 \ . \ \bar{b}_1 \ \bar{b}_2 \ \bar{b}_3 \ \cdots \ \bar{b}_i \ \bar{b}_{i+1} \ \bar{b}_{i+2} \ \cdots \ \bar{b}_{i+j} \uparrow$$

where the up-arrow $\uparrow$ denotes adding 1 bit to the LSB $\bar{b}_{i+j}$. Then $(f_u \circ C_1)(n)$ is collecting the first i bits to the right of the binary point in $C_1(n)$.

[2]See, e.g. *Arithemtic Operations in a Binary Computer*, Robert F. Shaw, 1950. This article is available from *Computer Arithmetic - Volumn I*, Earl E. Swartzlander (ed.), 2014.

On the other hand,

$$(C \circ f_u)(n) = \bar{b}_1\ \bar{b}_2\ \bar{b}_3\ \cdots\ \bar{b}_i \uparrow$$

We argue that $(f_u \circ C_1)(n)$ and $(C \circ f_u)(n)$ are equal only when the effect of $\uparrow$ in the expression of $C_1(1)$ crosses the boundary between the i and $i+1$ bit; that means all $b_{i+1}, \ldots, b_{i+j}$ must be zero — this is in general not the case, therefore the diagram does not commute.

But if we replace Q_1 by $Q_1 - X$ where X collects those in Q_1 whose bit-length to the right of the binay point is longer than the word length of W, the new diagram would commute.

For the second diagram, note that

$$({f_u}^{-1} \circ C)(000) = 0.00$$

but

$$(C_1 \circ {f_u}^{-1})(000) = 1.00$$

Therefore the diagram does not commute. But if we replace W by $W - \{000\}$, the new diagram would commute. □

3.2 Signed Numbers and Two's Complement

We want to represent numbers in the range $Q_2 = [-1, 1) \subseteq Q$. For example, the positive number $0.10 \in Q_2$ is represented as $\underline{0}10 \in W$ by setting the sign bit (underlined) to 0, followed by the unsigned representation of its magnitude, with necessary bit padding or trimming; the binary point is assumed to the right of the sign bit when interpreting a word; the negative number $-0.01 \in Q_2$ is represented as $\underline{1}11 \in W$ by setting the sign bit to 1, followed by the unsigned representation of the one's complement 0.11 of its magnitude 0.01. If we use a binary point to delimit the sign bit from the significant bits in a word, the notation of a word that represents a positive number would coincide with the notation of the number being represented; the notation of a word that represents a negative number would coincide with the notation of two's complement of the magnitude of the number being represented. For example, $0.01 \in Q_2$ is represented by $0.01 \in W$; $-0.01 \in Q_2$ is represented by $1.11 \in W$ — if we interpret 1.11 not as a word but as a number, then it is the two's complement of 0.01 (the magnitude of -0.01).

The operation C is taking two's complement if a binary point is assumed to the right of the sign bit.

4 Addition of Two's complement Numbers

Table 1 shows what to expect from two's complement addition (assuming word length three without loss of generality). We see that adding a positive number with a negative number always results in a correct answer; if adding two positives and the result is positive, the result is correct, otherwise incorrect; similarly, adding two negatives if the result is negative then it is correct, otherwise incorrect. Therefore we have a simple rule to decide if the addition result is correct. In practise, usually the imperfection of two's complement is tolerated

	0.00	0.01	0.10	0.11	1.00	1.01	1.10	1.11
0.00	0.00	0.01	0.10	0.11	1.00	1.01	1.10	1.11
0.01	0.01	0.10	0.11	1.00	1.01	1.10	1.11	0.00
0.10	0.10	0.11	1.00	1.01	1.10	1.11	0.00	0.01
0.11	0.11	1.00	1.01	1.10	1.11	0.00	0.01	0.10
1.00	1.00	1.01	1.10	1.11	0.00	0.01	0.10	0.11
1.01	1.01	1.10	1.11	0.00	0.01	0.10	0.11	1.00
1.10	1.10	1.11	0.00	0.01	0.10	0.11	1.00	1.01
1.11	1.11	0.00	0.01	0.10	0.11	1.00	1.01	1.10

Table 1: Sum of 3-bit signed fixed-point numbers using the standard binary adder. The binary point is located to the left of the bit that is the coefficient of 2^{-1}. The numbers assume two's complement interpretation. Results in red are incorrect. Results in green are correct.

for its benefit, and moreover, exactly three quarters of the answers are correct and the correctness is decidable.

Table 2 exhausts the possibilities of two's complement addition using three-bit word length. This explains how the incorrect answers are produced: the correct answer does not have a representation under the choice of word length.

n_1	n_2	$n_1 + n_2$	n_1'	n_2'	$n_1' +' n_2'$	$(n_1 + n_2)'$
0.00	-1.00	-1.00	0.00	1.00	1.00	1.00
	-0.11	-0.11		1.01	1.01	1.01
	-0.10	-0.10		1.10	1.10	1.10
	-0.01	-0.01		1.11	1.11	1.11
	0.00	0.00		0.00	0.00	0.00
	0.01	0.01		0.01	0.01	0.01
	0.10	0.10		0.10	0.10	0.10
	0.11	0.11		0.11	0.11	0.11
0.01	-1.00	-0.11	0.01	1.00	1.01	1.01
	-0.11	-0.10		1.01	1.10	1.10
	-0.10	-0.01		1.10	1.11	1.11
	-0.01	0.00		1.11	0.00	0.00
	0.01	0.10		0.01	0.10	0.10
	0.10	0.11		0.10	0.11	0.11
	0.11	1.00		0.11	1.00	-
-0.01	-1.00	-1.01	1.11	1.00	0.11	-
	-0.11	-1.00		1.01	1.00	1.00
	-0.10	-0.11		1.10	1.01	1.01
	-0.01	-0.10		1.11	1.10	1.10
	0.10	0.01		0.10	0.01	0.01
	0.11	0.10		0.11	0.10	0.10
0.10	-1.00	-0.10	0.10	1.00	1.10	1.10
	-0.11	-0.01		1.01	1.11	1.11
	-0.10	0.00		1.10	0.00	0.00
	0.10	1.00		0.10	1.00	-
	0.11	1.01		0.11	1.01	-
-0.10	-1.00	-1.10	1.10	1.00	0.10	-
	-0.11	-1.01		1.01	0.11	-
	-0.10	-1.00		1.10	1.00	1.00
	0.11	0.01		0.11	0.01	0.01
0.11	-1.00	-0.01	0.11	1.00	1.11	1.11
	-0.11	0.00		1.01	0.00	0.00
	0.11	1.10		0.11	1.10	-
-0.11	-1.00	-1.11	1.01	1.00	0.01	-
	-0.11	-1.10		1.01	0.10	-
-1.00	-1.00	-10.00	1.00	1.00	0.00	-

Table 2: Comparing intuitive addition and two's complement addition, for 3-bit signed fixed-point numbers. n' means to take two's complement representation of n.