The Algebra Of Two's Complements Binary Number Representation

Óscar Pereira*

Abstract. The mathematical rationale for two's complements is provided.

Prerequisites. The reader is assumed familiarised with converting unsigned (positive) integers to their binary representation, and vice-versa.

Keywords: binary representation, two's complements, modular arithmetic.

1 Introduction: Rules Of Two's Complements

First, notational conventions: given an n-bit number, we refer to any specific bit by referring to its corresponding power of two, when interpreting it as an unsigned integer and translating it to decimal. So for example, in the 4 binary number 1000_2, the only bit set to 1 is the 2^3 bit. Additionally, as this examples illustrates, we assume that the most significant bit is the leftmost one, and the least significant bit is the rightmost one. Hence, moving leftwards increases the "significancy" of bits, and moving rightwards decreases it. Lastly, given a binary string, its conversion to decimal as if it were an unsigned number will be referred to as the "standard way" or the "usual way."

The rules of two's complements are the following: using n-bit strings, instead of representing integers from 0 up to and including $2^n - 1$, we reserve the most significant bit to indicate the *sign* (positive or negative) of the integer, and use the remaining bits to represent numbers from -2^{n-1} up to and including $2^{n-1} - 1$. In particular, negative numbers have the most significant bit set to 1, while 0 and the positive numbers have that bit set to 0—which is also why we can represent one less positive number than the number of negative numbers. In other words, we get half (2^{n-1}) of the full space (2^n) for negative numbers, and half for zero plus the positive numbers. Throughout the rest of this section, we tacitly assume that the number of bits n is big enough to represent whatever integer we

*CONTACT: {https://, oscar@}randomwalk.eu. DATE: May 14, 2024.
Updated versions of this document and other related information can be found at https://randomwalk.eu/scholarship/twos-complements/.

wish to represent as two's complements (the question of how big is big enough is dealt with in §5).

So given a positive integer a that can be represented with n-bit two's complements, we just represent it the usual (unsigned) way. For example, with $n = 3$, 1 is presented as 001_2 and 3 as 011_2. Note that the most significant bit will always be set to 0. If now a is a negative integer, compute $2^n - (-a)$, and then represent it as usual. For example, again with 3 bits, to represent -1, we compute $2^3 - 1 = 7$, and then represent it the usual way: 111_2. To take another example, -4, we have $2^3 - 4 = 4$, the standard representation of which is 100_2.

Conversely, given an n-bit string that represents an integer in two's complements, we convert it do decimal as usual, ***except*** *that the power* 2^{n-1} *is replaced with* -2^{n-1} (note the minus sign). It is immediate the observation that this has no effect on non-negative numbers, for which the 2^{n-1} bit is always zero. As for negative numbers, take the same examples as above: $111_2 = 1 \times (-2^2) + 1 \times 2^1 + 1 \times 2^0 = -1_{10}$ and $100_2 = 1 \times (-2^2) + 0 \times 2^1 + 0 \times 2^0 = -4_{10}$.

Given the two's complements binary representation of a number, we can obtain the representation of its additive inverse in the same scheme, by flipping all the bits, and then adding 1 (binary addition). For example with $n = 3$, we have $2_{10} = 010_2$, and flipping all the bits yields 101, and adding 1 yields $110_2 = -2_{10}$ (table 1). Note that with 0_{10}, in the result of the addition the 2^3 bit will be set to 1—*and it must be discarded*: $0_{10} = 000_2$, and flipping all bits yields 111, and adding 1, yields 1000 (table 2).

	0	0	1	(carry)
		1	0	1
+		0	0	1
	0	1	1	0 (-2_{10})

Table 1: In two's complements with 3 bits, the bit of power 2^2 corresponds to -2^2.

	1	0	0	(carry)
		1	1	1
+		0	0	1
	1	0	0	0 $(000_2 = 0_{10})$

Table 2: In two's complements with 3 bits, when the bit of power 2^3 is 1, we discard it.

A more expedite way to obtain the same result is the following: starting with the least significant bit (leftmost), keep everything equal up to and including the first 1; and then flip all other bits to the right. It is a simple exercise to see that this is equivalent to flipping all bits and then adding one, discarding the 2^n bit, if it is set.

WARNING. This procedure *does not work* for the smallest negative number representable with n bits (i.e., -2^{n-1})—instead of yielding its additive inverse, it yields itself! This is because the additive inverse, 2^{n-1} cannot be represented with n bits: the largest positive integer representable is

$2^{n-1} - 1$. Taking the 3 bit case again, $-4_{10} = 100_2$, flipping everything produces 011, and adding 1 yields again $100_2 = -4_{10}$.

To add numbers represented in two's complements with n bits, just add them as normal, **discarding the 2^n bit if it is 1** (just as done in table 2): the result will be within the representable range for the n bits, *as long as the two leftmost carry bits are **equal***. Some examples are provided below, and §4 looks specifically at overflow situations. (Keep in mind that when adding n bit parcels, we always have n carry bits.)

Remark 1.1 ("Two's complements" is a misnomer). In reality, the system we have described computes—for the negative numbers—the complements of 2^n, rather than of 2, as the name implies. But "two's complements" is the standard name, and so we shall stick to it. △

This covers the basics of *how* two's complements arithmetic functions. The rest of the text explains *why* it functions. The task requires modular arithmetic—the fundamentals of which are reviewed in §2. Its application to two's complements is done in §3, and in §4 we cover how to detect overflows. Finally, in §5 we cover extending two's complements from n to $n+1$ bits, and how to compute the number of bits needed to represent a given number (positive or negative) in two's complements binary form.

We will rely heavily on examples of two's complementation with 3 bits (i.e., $n = 3$).

2 Modular Arithmetic

Feel free to skip this section if you are already comfortable with modular arithmetic: it is just a review of fundamental notions, needed in the sequel.

to be done... XXX. In the meantime, a good reference is Shoup [1], especially chapters 1 and 2.

3 Modular Arithmetic In Binary

Consider the binary numbers from 000_2 to 111_2 in the standard representation—i.e., the integers from 0 to 7. If, to the customary way of adding them, one adds the rule that whenever the 2^3 bit is set to 1, we discard it, then we obtain addition modulo 8: for from $1000_2 = 8_{10}$ comes $000_2 = 0_{10}$, from $1001_2 = 9_{10}$ comes $001_2 = 1_{10}$, and so on. That is to say, we have addition modulo 8 of the elements in the set $\{0, \ldots, 7\}$. Put differently, we have addition in $\mathbb{Z}_8$, but where each equivalence class is represented by one (and only one) of the integers in the set $\{0, \ldots, 7\}$.

Definition 3.1. *We denote the set* $\{0, \ldots, 7\}$*—sometimes called the* **canonical representatives** *of the equivalence classes of* $\mathbb{Z}_8$*—by* $\mathbb{Z}_8^+$.

The representation in two's complements leverages the fact that in $\mathbb{Z}_8$ arithmetic, each equivalence class has an additive inverse. In fact, given $x \in \mathbb{Z}_8^+$, the additive inverse of $[x]$ is $[8 - x]$. So the additive inverse of $[1]$ is $[7]$, of $[2]$ is $[6]$ and of $[3]$ is $[5]$—and vice-versa, as modular addition, similarly to "normal" addition, is commutative. The inverse of $[0]$ is $[8]$, or equivalently, $[0]$, and that of $[4]$ is also the same element, viz. $[4]$. When we take the usual binary representation of the elements in $\mathbb{Z}_8^+$, complement their bits and (binary) add 1 (cf. §1), what we are, in effect, doing, is going from x to $8 - x$ (because complementing the bits gives us $7 - x$). When we discard the 2^3 bit if it is 1, we doing a modular reduction—which in this case, amounts to subtracting 8—that is, we are ensuring the resulting integer belongs to $\mathbb{Z}_8^+$.

The idea of two's complements, is to use the most significant bit—2^2 in our case—to store the *sign* of the integer. Now we naturally want our two's complements representations of *non-negative* numbers to coincide with their unsigned representations, which means we want the representation of non-negative numbers to have the 2^2 bit set to 0. And thus, negative numbers must have that bit set to 1. In $\mathbb{Z}_8^+$, the numbers that have that bit set to 1 are $4, 5, 6, 7$—and thus, this means we want the equivalence classes $[4]$, $[5]$, $[6]$ and $[7]$ to represent negative numbers—so we want a negative representative for each. Moreover, just as $\mathbb{Z}_8^+$ consists of sequential integers, we want the new set consisting of the new representatives of these classes, plus the same representatives of the other classes, to also consist of sequential integers. Otherwise we could end up with a system where we might be able to, say, represent -3 and -1, but not -2... The easiest way to do this, is to subtract 8: i.e., in the elements of $\mathbb{Z}_8^+$, 4 is replaced by -4, 5 by -3, 6 by -2 and 7 by -1, all the other elements remaining equal. Observe that we are still dealing with the same equivalence classes, because we have $4 \equiv -4$, $5 \equiv -3$, $6 \equiv -2$ and $7 \equiv -1$, all modulo 8. We just represent some of them with different elements, which gives us a new set:

Definition 3.2. *We denote the set* $\{-4, \ldots, 0, \ldots, 3\}$ *by* $\mathbb{Z}_8^-$.

Note that we are still dealing with the same binary representations—000_2 to 111_2—as well with the same way of adding them—i.e., addition as usual followed by discarding the 2^3 bit if it is 1. And if, when we assign them to the integers of $\mathbb{Z}_8^+$ the usual way, the operation we have is addition in $\mathbb{Z}_8$, then when we assign them to the integers in $\mathbb{Z}_8^-$ as just described, the operation is still addition in $\mathbb{Z}_8$—because the changed integers were replaced with other integers in the same equivalence class. In

other words, we are still doing addition in $\mathbb{Z}_8$, but now using as representatives the integers in $\mathbb{Z}_8^-$.

Remark 3.3. While it is correct to say that addition using two's complements with 3 bits computes addition in $\mathbb{Z}_8$, one **cannot** say that it computes addition modulo 8. Because, for example, $3+2$ is 5 modulo 8, but with two's complements the result of $3+2$ is -3—in fact, this is an example of an **overflow**, of which much more in §4. △

We can now also understand why with n bits, in two's complement the 2^{n-1} bit is actually "worth" -2^{n-1} (cf. §1), rather than 2^{n-1}, as in the standard representation: with 3 bits, when the most significant bit ($2^{n-1} = 2^2$) is 1, subtracting 8 (i.e., subtracting $2^n = 2^3$) effectively means that when converting to decimal, that bit is now multiplied by -2^2, because $2^2 - 2^3 = -2^2$ (or more generically, $2^{n-1} - 2^n = -2^{n-1}$).

Remark 3.4 (Smallest negative.). It should now be clear why applying the rule to obtain the additive inverse by complementing and then adding 1, fails for the smallest negative number (-4 with 3 bits, -2^{n-1} with n bits), and instead yields the same number (cf. the warning given in §1): -2^{n-1} in n-bit two's complements is represented by the same binary string that represents 2^{n-1} in the standard representation. And as we have seen, complementing and adding one computes $2^n - 2^{n-1} = 2^{n-1}$, the unsigned representation of which, in two's complements, corresponds again to -2^{n-1}... △

Just as $\mathbb{Z}_8$ addition "wraps around" in $\mathbb{Z}_8^+$—for instance, $7+1 = 8 \equiv 0 \pmod 8$—so too we have $3+1 = 4 \equiv -4$ and $-3+(-2) = -5 \equiv 3$, all congruences modulo 8. That is, $\mathbb{Z}_8$ addition also wraps around in $\mathbb{Z}_8^-$. For completeness, the "bit version" (in two's complements) of those additions is provided in tables 3 and 4.

	0	1	1		(carry)
		0	1	1	(3)
+		0	0	1	(1)
	0	1	0	0	$(4 \equiv -4)$

Table 3: Recall that in two's complements with 3 bits, the bit of power 2^2 corresponds to -2^2.

	1	0	0		(carry)
		1	0	1	(-3)
+		1	1	0	(-2)
	1	0	1	1	$(-5 \equiv 3)$

Table 4: Recall that in two's complements with 3 bits, when the bit of power 2^3 is 1, we discard it.

There will be much more to say on wrapping around on §4, but before that, note the following: for a given binary sum, whether or not wrapping around happens **depends on the representatives used!** So for example, in $\mathbb{Z}_8^-$, $(-3)+(-1) = -4$, without any wrapping around. However, if we "translate" the same binary addition into the elements of $\mathbb{Z}_8^+$, we have $5+7 = 12 \equiv 4 \pmod 8$—where we've had to wrap around (i.e., modularly reduce) the result, in order to get an element of $\mathbb{Z}_8^+$.

4 To Wrap Or Not To Wrap (Around)

Our goal with the representation of numbers in two's complements, is *not* modular addition, but rather addition in $\mathbb{Z}$. Which coincides with addition in $\mathbb{Z}_8$ **only when there is no wrap around!** In particular, the examples of remark 3.3, and of the tables 3 and 4—all of which involve wrapping around $\mathbb{Z}_8^-$—are *wrong* in $\mathbb{Z}$, because $5 \neq -3$, $-4 \neq 4$ and $-5 \neq 3$. When such wrapping around has occurred, we say we have an **overflow**. We want to be able to detect, after having performed binary addition, whether or not an overflow has occurred. Which is straightforward with the elements of $\mathbb{Z}_8^+$ and the standard representation, but not so much with those of $\mathbb{Z}_8^-$ (represented in two's complements)—let us now see why.

Beginning with the easy case, in $\mathbb{Z}_8^+$ we have an overflow if and only if the result after addition has the 2^3 bit set to 1. Otherwise the resulting integer can be represented with only 3 bits, and hence, no overflow.

The case of $\mathbb{Z}_8^-$ is much thornier, and reason can be seen in table 3: we have an overflow with the 2^3 bit of the result set to 0. Conversely, consider the addition of -1 plus -2, depicted in table 5.

	1	1	0	(carry)
		1	1	1	(-1)
+		1	1	0	(-2)
	1	1	0	1	$(13 \equiv -3)$

Table 5: Recall that in two's complements with 3 bits, the bit of power 2^2 corresponds to -2^2, and also that when the bit of power 2^3 is 1, we discard it.

Here we do *not* have an overflow, even though the result after addition contains the 2^3 bit set! The solution out of this conundrum, is to take each possible case separately:

- Addition of a negative integer with a non-negative integer. Here overflow is not possible, because the result will always be equal to or greater equal than -4, and smaller than or equal to 3—which can both be represented in two's complements using 3 bits, meaning that so can every other possible result.
- Addition of two non-negative integers. Here overflow occurs when the result is greater than or equal to 4, which by wrapping around means it is in the set $\{-4, -3, -2\}$. It cannot go "further" than -2 because that is the result (in $\mathbb{Z}_8^-$) of adding to itself the greatest possible positive number, viz. $3 + 3$. This leads to the key observation: the 2^2 bit in both parcels is 0, but *in the result* it will be 1. In other words, overflow

happens if and only if the 2^2 bit goes from 0 in the parcels, to 1 in the result.

- Addition of two negative integers. Here overflow happens when the result is smaller than or equal to -5. Now, again by wrapping around, this means the result will belong to the set $\{3, 2, 1, 0\}$—it cannot go "further" than 0 because that is the result (in $\mathbb{Z}_8^-$) of adding the smallest possible numbers, i.e. $(-4) + (-4)$. And analogously to above, we see that overflow happens if and only if the 2^2 bit goes from 1 in the parcels, to 0 in the result. (Note that when adding two negatives, the 2^3 bit of the result will always be 1, whether there is or isn't an overflow.)

So we know how to detect when overflow occurs—but there is another way to do that, which is more "implementation friendly." Which is the following: overflow occurs if and only if out the 3 carry bits, **the two leftmost are different**. First, we show that when adding two integers of different signs (or one integer and zero)—where overflow cannot occur—the two leftmost carry bits are always equal. In this scenario, we have a sum like the one shown in table 6 (the x are arbitrary values).

	z	y	0		(carry)
		1	x	x	
+		0	x	x	

Table 6: We omit the result line because it is unneeded for our analysis.

It should be straightforward to see that if $y = 0$, then it will also be that $z = 0$. And if $y = 1$, then in that column we will have the sum $1 + 1 + 0$, the result of which is 0, with a carry of 1—i.e., $z = 1$. In either case, the two leftmost carry bits are equal.

Now, in the case of addition of two positive numbers the leftmost carry bit is always 0, and for the addition if two negative numbers, it is always 1. Thus we have sketch sums like in tables 7. In both cases it is immediate to see that the 2^2 bit of the result differs from that of the parcels if and only if the two leftmost carry bits are different.

	0	y	x		(carry)
		0	x	x	
+		0	x	x	
	0	y	x	x	

	1	y	x		(carry)
		1	x	x	
+		1	x	x	
	1	y	x	x	

Table 7: Left: addition of two positives. Right: addition of two negatives.

It is a straightforward (albeit cumbersome) exercise to show that *all* of the analysis above carries over—*mutatis mutandis*—to two's complements

addition with any arbitrary number n of bits. In particular, it is still the case that **overflow occurs if and only if the two leftmost carry bits are different**. One of things that changes, is the interval of numbers that can be represented with n bits—which is the topic of §5.

5 Bounds & Padding

Extending n. Before dealing with bounds proper, one might wonder how, when the number of bits is increased, will the two's complements representation of integers *that could already be represented* vary. For instance, take again $n = 3$. How does the representation of integers that can be represented in 3-bit two's complement changes, when going to 4 bits? The case for non-negative integers should be easy to crack: they just get an extra 0 padded to the left. For instance, 001 becomes 0001, and 011 becomes 0011. As for negative numbers, they also get padded to the left, *but with 1's!* To see that this is so, observe that the n-bit two's complement representation of a negative integer a, is the standard representation of the positive integer $2^n - a$. And with $n + 1$ bits, that same negative integer is represented by the positive integer $2^{n+1} - a$. But $\left(2^{n+1} - a\right) - (2^n - a) = 2^n$—and as $2^n - a$ is at most $2^n - 1$, it can be represented using n bits. Thus, if when going from n to $n + 1$ bits we add 2^n, that means (p)adding an extra 1 to the left of those n bits (the binary representation of a). For example, -2 with 3 bits is represented by the standard representation of the integer $2^3 - 2 = 6_{10} = 110_2$. And with 4 bits we add 2^3: $2^3 - 2 + 2^3 = 2^4 - 2 = 14_{10} = 1110_2$ which can readily be checked to correspond to -2 in two's complement with 4 bits: $1 \times (-2^3) + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = -2$.

Bounds. With 3 bits we can represent integers from -4 up to and including 3. With n bits, we can go from -2^{n-1} up to and including $2^{n-1}-1$. An interesting question is the reverse one: given an integer, what is the minimum number of bits that are needed to represent it? For positive integers in the unsigned world, the smallest integer that requires at least n bits to represent, is the one where the most significant bit is 1, and all other bits are 0—i.e., 2^{n-1}. And the largest integer that can be represented without adding more bits, is the one where all the bits are 1, i.e., $2^n - 1$. In other words, the integers that require at least n bits to represent, are those verifying the condition $2^{n-1} \leq x < 2^n$—which resolves to $n = \lfloor \log_2 x \rfloor + 1$. This last step makes sense because given any positive integer x, we can always find another positive integer n such that the double inequality is verified. Similar assumptions will also hold for similar reasonings to be done in the next two paragraphs.

With two's complements, the most significant bit is reserved for the

sign, and hence the positive numbers that require at least n bits to represent are the ones verifying $2^{n-2} \leq x < 2^{n-1}$, with $n \geq 2$ (with 1 bit no positive integers can be represented, only 0 and -1; with 2 bits, the only representable positive integer is 1). This resolves to $n = \lfloor \log_2 x \rfloor + 2$. I.e., to represent a positive integer x in two's complements requires at least $\lfloor \log_2 x \rfloor + 2$ bits.

Lastly, negative integers. In two's complements with n bits, all representable negatives are of the form $-2^{n-1} + \sum_{i=n-2}^{0} b_i \cdot 2^i$, with $n \geq 1$ and where the b_i are the individual bits. Thus, the smallest negative representable is -2^{n-1}. Now, if $n = 1$, $-2^{n-1} = -1$ is also the *largest*—and thus, the *only*—negative integer representable. So let us now assume that $n \geq 2$.

To compute the largest negative that requires at least n bits to represent, we need the following observation: we saw above that we can add 1's to the left of the binary representation of a negative number, and we will still be left with the same negative number, although now represented in two's complement with a greater number of bits. *Conversely*, given the binary representation of a negative number, in which the two leftmost bits are 1, we can remove the leftmost 1, and still be left with the representation of the same number, albeit with less bits. Hence, the biggest negative number that requires at least n bits to represent, is one where the two leftmost bits are *not* 1. The largest such number is the one where the bit 2^{n-2} is 0, and all the other bits to its right are 1: $-2^{n-1} + \sum_{i=n-3}^{0} 2^i = -2^{n-1} + (2^{n-2} - 1) = -2^{n-2} - 1$. (Observe that if $n = 2$, the summation is done over an empty set, and thus equals 0—but the result remains valid.) All of which combined means that at least n bits are required to represent, in two's complements, the integers x verifying $-2^{n-1} \leq x < -2^{n-2}$, which resolves to $n = \lceil \log_2(-x) \rceil + 1$. This formula was deduced assuming that $n \geq 2$, but it is simple to observe that it also holds when $n = 1$, or equivalently, $x = -1$. Hence, the minimum number of bits needed to represent a negative integer x, is $\lceil \log_2(-x) \rceil + 1$.

References

1. **Shoup**, Victor, 2008. *A Computational Introduction to Number Theory and Algebra*. New York: Cambridge University Press, eBook edition edition. Cited on p. 3.