"Twos complement" representation of integers

To motivate our representation of negative numbers, let me take you back to your childhood again. Remember driving in your parent's car and looking at the odometer.[1] Remember the excitement you felt when the odometer turned from 009999 to 010000. Even more exciting was to see the odometer go from 999999 to 000000, if you were so lucky. This odometer model turns out to be the key to how computers represent negative numbers.

If we are working with six digit decimal numbers only, then we might say that 999999 behaves the same as -1. The reason is that if we add 1 to 999999 then we get 000000 which is zero.

$$\begin{array}{rl} & 999999 \\ + & \underline{000001} \\ & 000000 \end{array}$$

Notice that we are ignoring a seventh digit in the sum (a carry over). We do so because we are restricting ourselves to six digits.

Take another six digit number 328769. Let's look for the number that, when added to 328769, gives us 000000. By inspection, we can determine this number to be 671231.

$$\begin{array}{rl} & 328769 \\ + & \underline{651231} \\ & 000000 \end{array}$$

Thus, on a six digit odometer, 651231 behaves the same as -328769.

Let's apply the same idea to binary representations of integers. Consider eight bit numbers. Let's look at $26_{ten} = 00011010_{two}$. How do we represent -26_{ten} ?

$$\begin{array}{rl} & 00011010 \\ + & \underline{????????} \\ & 00000000 \end{array}$$

To find -26, we need to find the number that, when added to 00011010 gives us 00000000. We use the following trick. If we "invert" each of the bits of 26 and add it to 26, we get

$$\begin{array}{rll} & 00011010 & \leftarrow 26 \\ + & \underline{11100101} & \leftarrow \text{inverted bits} \\ & 11111111 & \end{array}$$

This is not quite right, but its close. We just need to add 1 more. (Remember the odometer).

$$\begin{array}{rl} & 11111111 \\ + & \underline{00000001} \\ & 00000000 \end{array}$$

Note we have thrown away the ninth bit because we are restricting ourself to eight bits only. Thus,

[1]Ok, ok, odometers were used back in the 20th century, before you were born..

$$\begin{array}{cll} & 00011010 & \leftarrow 26 \\ & 11100101 & \leftarrow \text{inverted bits} \\ + & \underline{00000001} & \leftarrow +1 \\ & 00000000 & \end{array}$$

Alternatively, we add 1 to the inverted bit representation and this must give us -26.

$$\begin{array}{cll} & 11100101 & \leftarrow \text{inverted bits} \\ + & \underline{00000001} & \leftarrow +1 \\ & 11100110 & \end{array}$$

$$\begin{array}{ll} 00011010 & \\ \underline{11100110} & \leftarrow \text{This is } -26\ _{10}. \\ 00000000 & \end{array}$$

Thus, *to represent the negative of a binary number, we invert each of the bits and then add 1.* This is called the *twos complement* representation.

Special cases

One special case is to check is that the twos complement of 00000000 is indeed 00000000.

$$\begin{array}{ll} 00000000 & \\ \underline{11111111} & \leftarrow \text{invert bits} \\ 11111111 & \end{array}$$

And adding 1 gets us back to zero. This makes sense, since -0 = 0.

Another special case is the decimal number 128, and again we assume a 8 bit representation. If you write 128 in 8-bit binary, you get 10000000. Note that taking the twos complement gives you back the same number 10000000.

$$\begin{array}{cll} & 10000000 & \\ + & \underline{01111111} & \leftarrow \text{invert bits} \\ & 11111111 & \end{array}$$

And adding 1 gets us back to zero. So, what is -128 ?

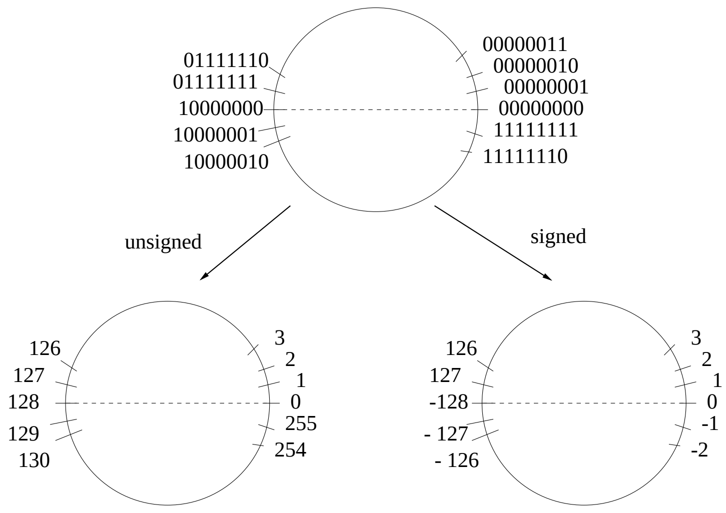
$$\begin{array}{cll} & 01111111 & \leftarrow \text{the inverted bits} \\ + & \underline{00000001} & \leftarrow \text{adding 1} \\ & 10000000 & \end{array}$$

Note that 128 has the same representation as -128. Of course, we can't have both: we have to decide on one. Either 10000000 represents 128 or it represents -128. How does that work?

Unsigned vs. signed numbers

If treat all the 8 bit numbers as positive, but we ignore the carry of the leftmost bit in our sum (the *most significant bit, or MSB)*, then adding 1 to the binary number 11111111 (which is 255 in decimal) takes us back to 0. See the figure below on the left. This representation is called *unsigned.* Unsigned numbers are interpreted as positive.

To allow for negative numbers, we use the twos complement representation. Then we have the situation of the circle on the right. This is called the *signed* number representation. *Note that the MSB indicates the sign of the number. If the MSB is 0, then the number is non-negative. If the MSB is 1, then the number is negative.*



Unsigned and signed n-bit numbers

The set of *unsigned* n-bit numbers is represented on a circle with 2^n steps. The numbers are $\{\ 0,\ 1,\ 2, \ldots,\ 2^n - 1\ \}$. It is common to use $n = 16,\ 32,\ 64$ or 128, though any value of n is possible. The *signed* n bit numbers are represented on a circle with 2^n steps, and these numbers are $\{-\ 2^{n-1}, \ldots,\ 0,\ 1,\ 2, \ldots,\ 2^{n-1} - 1\ \}$. Signed n bit numbers are represented using twos complement. For example, if n=8, then the signed numbers are $\{-128, -127, \ldots,\ 0,\ 1,\ 2, \ldots,\ 127\}$ as we saw earlier. Consider the following table for the 8 bit numbers.

binary	signed	unsigned
00000000	0	0
00000001	1	1
:	:	:
01111111	127	127
10000000	-128	128
10000001	-127	129
:	:	:
11111111	-1	255

If n=16, the corresponding table is:

binary	signed	unsigned
0000000000000000	0	0
0000000000000001	1	1
:	:	:
0000000001111111	127	127
0000000010000000	128	128
0000000010000001	129	129
:	:	
0111111111111111	$2^{15} - 1$	$2^{15} - 1$
1000000000000000	-2^{15}	2^{15}
1000000000000001	$-2^{15} + 1$	$2^{15} + 1$
:	:	
1111111101111111	-129	$2^{16} - 129$
1111111110000000	-128	$2^{16} - 128$
1111111110000001	-127	$2^{16} - 127$
:	:	
1111111111111111	-1	$2^{16} - 1$

A surprising example! (Java)

Take $n = 32$. The largest signed integer is thus $2^31 - 1$. In Java (and C), the type int defines a 32 bit signed number. Let's explore the limits on this representation.

First, note that $2^{10} = 1024 \approx 10^3$, i.e. one thousand, and $2^{20} \approx 10^6$ or one million, and $2^{30} \approx 10^9$ or one billion. So, $2^{31} \approx 2,000,000,000$ or two billion and in fact 2^{31} is a bit more than two billion.

What if we declare:

```
int  j = 4000000000;                 // 4 billion  >  2^31
```

This gives a compiler error. "The literal of type int is out of range." The compiler knows that 4,000,000,000 is greater than $2^{31} - 1$. Now try:

```
int  j = 2000000000;               //   2 billion <  2^31
System.out.println( 2 * j );
```

This prints out -294967296. To understand why these particular digits are printed, you would need to convert 4000000000 to binary, which I don't recommend because it is tedious. The point is that it is a negative number! This can easily happen if you are not careful, and obviously it can lead to problems.

Floating point

Let's next talk about binary representations of fractional numbers, that is, numbers that lie between the integers. Take a decimal number such as 22.63. We write this as:

$$22.63 \ = \ 2 * 10^1 + 2 * 10^0 + 6 * 10^{-1} * 3 * 10^{-2}$$

The "." is called the *decimal point*. We can use a similar representation for fractional binary numbers. For example,

$$(110.011)_{two} \;=\; 1*2^2 + 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2} + 1*2^{-3}$$

where "." is called the *binary point*. If we convert to decimal, we get

$$4 + 2 + 0.25 + 0.125 \;=\; 6.375$$

Binary to decimal conversion

Just as with integers, we can convert a binary number into a decimal number using the brute force method, namely remember or figure out the powers of 2 and then add up all contributions from 1 bits. This is relatively easy to do for simple examples such as above. What about for examples that have far more bits to the right of the binary point?

Decimal to binary conversion

Take the example 22.63 above. We can convert the number to the left of the decimal point from decimal to binary, using the method from lecture 1, namely $22 = (10110)_2$. But how do we convert the fractional part (.63) to binary? The idea is to use the fact that multiplication by 2 in binary produces a shift to the left by one bit. (i.e. Multiplying by 2 in binary just adds 1 to each exponent in the sum of powers of 2, and this corresponds to a shift of bits to the left.)

We convert 0.63 to binary as follows. To the left of the binary point, we represent the number in binary. (Assume the number is positive. We'll deal with negative numbers next lecture.) To the right of the binary point, we represent the fractional part in decimal. To go from one line to the next, we multiply by 2 and divide by 2. Specifically we multiple the fractional decimal part by 2, and keep track of the number of divisions by 2 by incrementing the exponent of a power of 2. To the left of the binary point, we shift by one bit and we fill the least significant bit with 1 or 0 depending on whether the (doubled) fractional part is greater than 1. You should verify why this makes sense, namely think what happens when we multiply by 2.

$$\begin{array}{crcll}
 & & . & 63 & \\
= & (1)_2 & . & 26 & \times 2^{-1} \\
= & (10)_2 & . & 52 & \times 2^{-2} \\
= & (101)_2 & . & 04 & \times 2^{-3} \\
= & (1010)_2 & . & 08 & \times 2^{-4} \\
= & (10100)_2 & . & 16 & \times 2^{-5} \\
= & (101000)_2 & . & 32 & \times 2^{-6} \\
= & (1010000)_2 & . & 64 & \times 2^{-7} \\
= & (10100001)_2 & . & 28 & \times 2^{-8} \\
= & (101000010)_2 & . & 56 & \times 2^{-9} \\
= & \text{etc} & . & \text{etc} &
\end{array}$$

Notice that the number of bits on the left of the binary point is 9, corresponding to the shift by 9 bits which is implied by the exponent of 2^{-9}. Finishing the example, ... since 22 in binary is 10110, we have

$$22.63 \;\approx\; (10110.101000010)_2$$

We write $\approx$ because we have rounded down, namely by chopping off any trailing bits. These trailing bits are of the form $\sum_{i=-9}^{-\infty} b_i 2^i$. Note that we have negative powers of 2 here.

What if we don't chop off the trailing bits and instead we generate bits *ad infinitum*. What happens? Interestingly, at some point you will generate a sequence of bits that repeats itself over and over. To see why, first consider a simple example. We write 0.05 in binary.

$$\begin{array}{crcll}
 & & . & 05 & \\
= & (0) & . & 1 & \times 2^{-1} \\
= & (00) & . & 2 & \times 2^{-2} \\
= & (000) & . & 4 & \times 2^{-3} \\
= & (0000) & . & 8 & \times 2^{-4} \\
= & (00001) & . & 6 & \times 2^{-5} \\
= & (00\underline{0011}) & . & 2 & \times 2^{-6} \\
= & (00\underline{0011}0) & . & 4 & \times 2^{-7} \\
= & (00\underline{0011}00) & . & 8 & \times 2^{-8} \\
= & (00\underline{0011}001) & . & 6 & \times 2^{-9} \\
= & (00\underline{0011}0011) & . & 2 & \times 2^{-10} \\
= & \text{etc} & . & \text{etc} &
\end{array}$$

Note this pattern 0011 will be repeated over and over as the part to the right of the binary point cycles back to "2". This gives:

$$(.05)_{ten} = .00\underline{0011}\cdots$$

where the underlined part repeats *ad infinitum*.

In a more general case, we have N digits to the right of the binary point. If "run the algorithm" I have described, then you will find that the number to the right of the binary/decimal point will eventually repeat itself. Why? Because there are only 10^N possible numbers of N digits. And once the algorithm hits an N digit number it has hit before, this defines a loop that will repeat over and over again. If the algorithm takes k steps to repeat, then k bits are generated. These k bits will similarly repeat *ad infinitum*. Of course, the repeating bits might be all 0's, as in the case of 0.375 which was in the lecture slides. (Do it for yourself if you are studying from these notes.)

Hexadecimal representation of binary strings

When we write down binary strings with lots of bits, we can quickly get lost. No one wants to look at 16 bit strings, and certainly not at 32 bit strings. Yet we will need to represent such bit strings often. What can we do?

The most common solution is to use *hexadecimal*, which is essentially a base 16 representation. We group bits into 4-tuples ($2^4 = 16$) starting at rightmost bit (i.e. least significant bit). Each 4-bit group can code 16 combinations and we typically write them down as: `0,1,...,9,a,b,c,d,e,f`. The symbol `a` represents 1010, `b` represents 1011, `c` represents 1100, ..., `f` represents 1111.

You may find yourself saying `a` represents the decimal number 10 and is written in 4-bit binary as 1010, `b` represents the decimal number 11 and is written in 4-bit binary as 1011, ..., `f` represents the decimal number 15 and is written in 4-bit binary as 1111. If you do this, then you are first converting 4-bit binary to decimal, and then converting decimal to hexadecimal. Its fine to do this, but its not necessary. The 4-tuples don't always stand for numbers from 0 to 15.

We commonly write hexadecimal numbers as `0x`______ where the underline is filled with characters from `0,...,9,a,b,c,d,e,f`. For example,

$$\texttt{0x2fa3} = 0010\ 1111\ 1010\ 0011.$$

Sometimes hexadecimal numbers are written with capital letters. In that case, a large `X` is used as in the example `0X2FA3`.

If the number of bits is not a multiple of 4, then you group starting at the rightmost bit (the least significant bit). For example, if we have six bits string 101011 , then we represent it as `0x2b`. Note that looking at this hexadecimal representation, we don't know if it represents 6 bits or 7 or 8, that is, 101011 or 0101011 or 00101011. But usually this is clear from the context.