Representing Numbers

Shaan Fulton

July 1, 2025

Unsigned Numbers Are Easy

We use a sequence of bits and the idea of a positional number system to create a bijection between any set of characters and our simple binary transistors. Because our array of bits is just a base-2 number system, it already represents numbers. However, there are some considerations to keep in mind when thinking of how to represent negative numbers in such a way that we can easily perform hardware computation on them.

Summation with Bits

We can easily implement a summation operation on two unsigned binary numbers.

	1	1	0	1	
+	1	0	1	1	
carry:	1	1	1	0	
sum:	1	1	0	0	0

Unsigned Numbers Are Harder

But what about negative numbers? An intuitive method of representing negatives is deciding that the left leading bit will represent our sign. A 1 might correlate to a negative signed number, and a 0 might correlate to a positive signed number.

Sign Bit	Magnitude Bits	Full Binary ($0b$)	Value
0	101	0b0101	+5 (positive)
1	101	0b1101	−5 (negative)

This is the sign-magnitude encoding for numbers. It is terrible for numerical operations. Try to add two sign-magnitude numbers and you'll see why: the result makes no sense because as the magnitude digits of a sign magnitude number increases, the actual value represented may be decreasing or increasing depending on the sign. Thus it is totally incompatible with our simple summation operation described above.

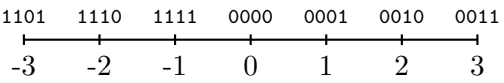
Two's Complement and Biased Encodings

We present two alternatives to the sign-magnitude approach that are beautifully compatible with our simple summation operation. This is critical, as it makes building the hardware summation operation much simpler.

The **two's complement** approach is as follows:

1. **Check the leftmost bit (sign bit):**
   - If it is `0`, the number is positive. Read it as a normal binary number.
   - If it is `1`, the number is negative.
2. **If the number is negative (sign bit is 1):**
   (a) Imagine that you invert (flip) all the bits (change `0` to `1` and `1` to `0`).
   (b) Add 1 to the result.
   (c) The answer is the magnitude, but the number is negative.

Why is this useful? Two's complement allows positive and negative numbers to be added and subtracted using the same binary addition rules. There are not two ways to write zero (as with sign-magnitude), and the ordering of numbers is continuous—making math operations simpler for computers.



Biased encoding is even simpler. The idea is we just use unsigned numbers, however, we shift them as we need to. So we might create an encoding with a bias of 127. `0b00000000`, or binary 0, is now -127. We can go all the way up to positive 127, `0b11111111`. All summation operations work perfectly.

Encoding Summary

- Sign-Magnitude: Never used, problematic for numerical operations
- Two's Complement: Standard for `int`
- Biased Encoding: Manually implemented on top of `uint`