Tutorial: Number Systems and Signed Representations

Interpreting Numbers in Different Signed Representations

An arbitrary string of bits has no meaning unless we first "interpret" it. In spoken language you're probably familiar with the word "helmet" (English): a protective piece of headwear. However, in Finnish this same word "helmet" is equivalent to the English word "pearls". So, if you found a piece of paper on the ground that said "helmet", it would be meaningless until you knew what language the word was written in. The same is true for signed numbers.

For example, if I give you the binary string 101101, without telling you if it's unsigned, 1's complement, 2's complement, you wouldn't be able to tell me what that binary number represents in decimal. Once you know the number system, however, you can then convert it appropriately:

- Unsigned: 43
- 1's Complement: -18
- 2's Complement: -19

For the sake of simplicity, though, if you see a binary number and are not given the signed representation, you can assume it's simply unsigned (always positive).

So, we don't "convert" to/from 1's complement or 2's complement, we just interpret the binary string as such. Keep in mind, too, that just because a number is in 1's or 2's complement doesn't mean it can't be positive. 0110 is still 3 (decimal) in unsigned, 1's complement, and 2's complement. You would only apply the complement rules, if asked, when switching from positive to negative or negative to positive. I've been asked: "Well, I have 0110, but it's a two's complement number, so do I convert it to 1010?" If you're asked to convert 0110 to a negative number, then yes, you'll want to flip all the bits and add 1, but 0110, although positive, is a valid two's complement number by itself.

Now, let's say you're given a 1's complement number and asked to convert it to decimal: 10101. We know this is negative because we're using 1's complement and the leading bit is a 1, but we want to find out the decimal equivalent. Let's pull out the negative sign and then flip all the bits: -(01010). What's inside the parentheses is a +10 (decimal), but we must keep the negative sign because our original number was negative, so we now know that 10101 = -10 (decimal).

The rule is true for 2's complement as well. Let's convert 11001 (2's complement) to decimal. We pull out the negative sign and then flip all the bits and add 1: -(00111), So 11001 = -7 (decimal).

"Representing" Numbers in Different Number Systems

Finally, some notes on converting between hex/binary/decimal. We talked about "interpreting" the value of unsigned, 1's complement and 2's complement numbers above. That is how we look at the binary/hex/decimal "representation" and determine it's value. I say "representation" because converting between binary/hex/decimal does nothing to change the value of the numbers nor does it change how we interpret them. Switching a 2's complement number from hex to binary does not alter the value nor change the fact that it's a two's complement number. It's the same as writing 1 as 2/2 or 5-4, for example. It's just a different way of "representing" a number.

So, if I asked you only to convert 1100 0110 to hex, there is no need to know if it's unsigned, 1's complement, or 2's complement because you're not being asked to "interpret" it's value, only to write it in another "representation". So, 1100 0110 = C6, and you're done! Similarly, if you were asked to convert A4 (2's complement) to binary, the fact A4 is a 2's complement number is just extra information and can be ignored. A4 = 1010 0100. However, if you were asked to convert it to binary AND give the decimal equivalent, then you would use your 2's complement rules on the binary "representation" to "interpret" the value. Since A4 = 1010 0100, we convert 1010 0100 = -(0101 1100) = -92.

Overflow

Overflow occurs when we don't have enough bits to be able to represent a number. Remember, given a binary string of N bits, we can represent (2^N)-1 different values. So, for an unsigned number, if we have 5 bits we can represent 32 values ranging from 0-31. If we have a signed number though, and still only 5 bits, we can only represent –15 through +16 since half the values are positive and half are negative. So when does overflow happen? It occurs when we perform an operation, let's say addition or subtraction, and the result does not fit in the range of values we can represent with the given number of bits.

4-bit unsigned example: 1100 (12) + 1000 (8) = 10100 (20)

Notice we added 2 4-bit numbers but got a 5-bit result. This is overflow, womp. Since we can only keep 4 bits, we get the (incorrect) result: 0100 (4). We can think of it like this, too: I have 4 bits so I can store numbers 0-15. 20 is outside of the range of values I can represent, so overflow must have occurred.

4-bit signed (1's complement) example: 1010 (-4) - 0111 (7) = 10100 (-11)

Again, we can only represent –7 through +8 with 4 bits, but our result, -11, is outside of that range. Since we only have 4 bits, our result is 0100 (4), which is not the same as –11, so overflow has occurred. If you're wondering why it's not called "underflow" since the number is too small for us to represent (whereas the example above was above our range), that's because "underflow" is used to describe a floating-point number where the precision is too small to represent. We use "overflow" to describe the case where an integer, be it positive or negative, cannot be correctly represented with the given number of bits.