Signed Numbers

So far we have discussed unsigned number representations. In particular, we have looked at the binary number system and shorthand methods in representing binary codes. With *m* binary digits, we can represent the 2^m unique patterns, from 000….0 to 111….1. When we try to represent signed quantities in the same m digits, we still only have 2^m patterns to work with. Unless we increase the number of digits available (i.e. make *m* larger), the representation of signed numbers will involve dividing up these 2^m patterns into positive and negative portions. Sign/Magnitude is the simplest way to represent signed numbers, but the most common is a modification to sign/magnitude called two's complement.

Sign/Magnitude Notation

Sign/magnitude notation is the simplest and one of the most obvious methods of encoding positive and negative numbers. Assign the leftmost (most significant) bit to be the **sign bit**. If the sign bit is 0, this means the number is positive. If the sign bit is 1, then the number is negative. The remaining m-1 bits are used to represent the magnitude of the binary number in the unsigned binary notation.

Example using 4 bits:

Binary	Value
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	-0
1001	-1
1010	-2
1011	-3
1100	-4
1101	-5
1110	-6
1111	-7

Looking at the list you should notice an immediate peculiarity; there are two representations for zero! There is positive zero, and negative zero. This can cause complications for computers checking numbers for equality. Normally one can just compare all the bits between two numbers to see if they are the same. But now we will need a special case for zero, to check for the two different representations. More significant though, is if we perform addition on numbers in this representation, we don't get the correct answer (e.g. +1 + -1 = 0001 + 1001 = 1010 which is -2, not 0).

Radix Complementation – Two's Complement

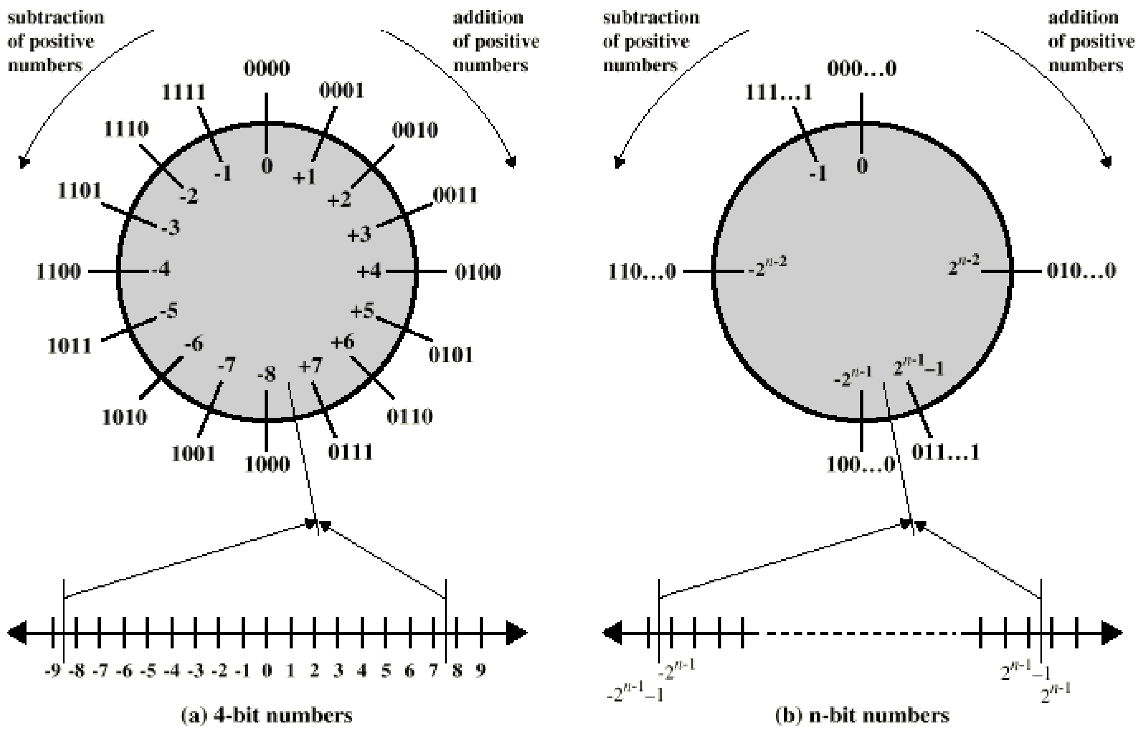
In two's complement, positive integers are represented the same was as sign magnitude (or regular unsigned binary) where the leftmost bit is 0:

Binary	Two's Complement Value
0000	0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7

Using four bits, the largest positive number we can represent is +7 since the first bit must be a 0 to denote positive.

For a negative value for A, the sign bit, A_{n-1} is one instead of zero. We would like to assign the negative integers to the available bit patterns in a way that facilitates straightforward arithmetic. A method that does this is to "count backwards" where 1111 represents -1, 1110 represents -2, etc.

This can be visualized in the figure below:



This representation has the benefit that if we start at any number on the circle, we can add positive k (or subtract negative k) from that number by moving k position clockwise or counterclockwise. If an arithmetic operation results in traversal of the point where the endpoints are joined, an incorrect answer is given. However, we are guaranteed that if we add a positive and a negative value together, we will result in a value that is possible to represent using the number of bits available.

A complete binary table for four bits is shown below:

Binary	Two's Comp Value	Binary	Two' Comp
0000	0	1111	-1
0001	+1	1110	-2
0010	+2	1101	-3
0011	+3	1100	-4
0100	+4	1011	-5
0101	+5	1010	-6
0110	+6	1001	-7
0111	+7	1000	-8

Note that we no longer have two representations for zero; we can represent down to -8 and up to +7.

To summarize, two's complement lets us have only one representation for zero and allows us to easily perform arithmetic operations without special cases for sign bits.

Shortcut for two's complement

If we are given a decimal value, A, that we want to represent in two's complement, there is an easy way to do it:

1. If A is positive, represent it using the sign-magnitude representation. The leftmost bit must be 0, and the remaining bits are the binary for the integer. Be careful there are enough bits available to represent the number.
2. If A is negative, first represent in binary +A.
    a. Flip all the 1's to 0's and the 0's to 1's
    b. Add 1 to the result using unsigned binary notation

Given a binary value in two's complement to compute its value in decimal:

1. If the leftmost bit is 0, the number is positive. Compute the magnitude as an unsigned binary number.
2. If the leftmost bit is 1, the number is negative.
    a. Flip all the 1's to 0's and the 0's to 1's
    b. Add 1 to the result using unsigned binary notation
    c. Compute the value as if it were an unsigned binary value, say it is B. This is the magnitude of the negative number.
    d. The actual value is -B

Examples:

Assume that m = 5, i.e. we have 5 bits available to represent our values.

What is –5 (decimal) in two's complement?
    +5 in unsigned binary is 00101
    Flip the bits to get 11010
    Now add 1: 11011
    The answer is 11011

What is –7 (decimal) in two's complement?
    +7 in unsigned binary is 00111
    Flip the bits to get 11000
    Now add 1: 11001

Note that as *m* changes, we get different bit strings for negative numbers.

What is the decimal value of the two's complement binary value 11100?
    Flip bits: 00011
    Add one: 00100
    This is 4, so the answer is –4

If the computer displays the hexadecimal number 1D, with m=8, what is this in decimal?
    1D in binary is 0001 1101
    It starts with 0, so it is positive.
    DON'T FLIP ANY BITS! We just convert this as unsigned binary:
        00011101 = 29 in decimal

Arithmetic on Two's Complement Values

Learning the rules of binary arithmetic is much easier than learning the rules of decimal arithmetic. Instead of memorizing 10x10 addition and subtraction tables, you only need to learn a 2x2 table:

+	**0**	**1**
0	0	1
1	1	0*

* includes a carry to the next column

A piece of hardware called an adder performs the task. It takes two binary numbers A and B, adds them bit by bit, and computes carries.

Example:

Add 00101 + 00110 (+5 and +6)

```
     00101
+    00110
     -------
     01011  = 11 (decimal)
```

Example:

Add +7 and –2 : +7 = 00111
+2 = 00010, so flip the bits: 11101 and add 1: 11110

```
     00111
+    11110
     -------
    100101         Discard the extra carry to give 00101 = 5
```

How can we discard the extra 1 and be sure we have the right result? The explanation is that the 1 in the column to the left of the high order bit is simply the value 2^m or M, the modulus. Remember that adding or subtracting the modulus will not change the value of a number. Discarding the extra bit is the same as subtracting the modulus, and is a perfectly legal operation. Nevertheless, there are times when we want to know about this carry, particularly when we overflow when adding two numbers using unsigned binary. For this reason, instead of actually discarding the carry bit, it is usually stored in a special location called the *carry register*.

Example:

Add –5 + -4
5 = 00101. Flip the bits to get 11010, and add 1 to get 11011
4 = 00100. Flip the bits to get 11011, and add 1 to get 11100

```
     11011
+    11100
     -------
    110111         = 10111 when we discard the carry
```

10111 is negative, as indicated by the leading 1.
Flip the bits to get 01000. Add 1 to get 01001. The result is 9. Since it is negative, we really have –9.

Example: What is the two's complement of 0?

00000 flip the bits = 11111. Add 1 and we get 100000. Since we ignore the carry bit, we end up with just 00000. That is, 0 and –0 are represented the same way in our system (yay!)

Example: What is 5 + 14, using 5 bits?

5 = 00101, 14 = 01110

```
    00101
+   01110
    -------
    10011
```

If we didn't have the convention that the first bit indicates the sign, this would be 19 in unsigned binary. But we are using two's complement, so this number would be:

Flip bits: 01100, Add 1: 01101 = 13 giving –13

Obviously, 5 plus 14 is not –13. What is wrong?

We have encountered what is called an *overflow condition*. We must be warned that this condition has occurred so that we do not improperly try to use the result that is produced. All computers have an *overflow register* that is turned on if the previous arithmetic operation resulted in an overflow condition.

Fortunately, it is easy to check for the overflow condition. We simply check to see if we are adding two positive numbers. If the result is a negative number, then there was an overflow. You cannot generate an overflow when adding a positive and negative quantity. In a similar fashion, if we add two negative numbers and end up with a positive number, we have also encountered overflow.

To summarize, the results of an addition A + B are:

- A + B with any carries discarded
- Carry register = 0 or 1 if there was a carry
- Overflow register = 0 or 1 if there was an overflow

Performing Subtraction

In order to do subtraction, hardware designers do not like to require extra hardware. Instead of separate circuits for subtraction, subtraction is performed by using addition with a negative number. That is:

D = Y – X

Is computed by:

D = -X + Y

To compute –X, we simply perform the process of flipping the bits on X and then adding 1, giving us the negative of X. We then perform the addition routine which is identical to what we previously discussed.

Storing Fractions

So far we have discussed signed and unsigned number representations. But how do we represent fractions? For example, we also need a way to represent a number like 409.331. This is done based on scientific notation and is called floating point notation.

Converting from Decimal to Binary

Let's say that we want to convert 252.390625 into binary. The first task is to convert the number 252 into binary. We already know how to do this, we just divide 252 by 2 and keep the remainders, repeating the process with the non-fractional part. 252 = 11111100

The next step is to convert 0.390625 into binary. To do this, **instead of dividing by 2, we multiply by 2**. Each time we multiply, record whatever is to the left of the decimal place after the operation. The first number becomes the leftmost bit, and the final number will be the rightmost bit. We then repeat this process using whatever is to the right of the decimal place.

0.390625 * 2 =	0.78125	0 as leftmost bit
0.78125 * 2 =	1.5625	1 as the next bit
0.5625 * 2 =	1.125	1
0.125 * 2 =	0.25	0
0.25 * 2 =	0.5	0
0.5 * 2 =	1.0	1
0		

Upon hitting 0, we're finished. The binary representation of this number is then:

11111100.011001

Exercise:

What is 3.625 in binary?
What is 0.1 in binary?

Note that with the last example, we could continue forever. In practice, we continue the process until we reach the precision desired to represent the number. Also note that if we wanted to use this process on something other than base 2, we would just multiply by whatever base we were interested in (e.g., base 16).

Converting Binary to Decimal

Given a floating point number in binary like 1100.011001, how do we convert this back to decimal? The process is almost identical to the process for unsigned binary. The stuff to the left of the decimal point is the same:

$$\begin{aligned} 1100 &= \\ & 0 * 2^0 + 0 * 2^1 + 1*2^2 + 1*2^3 \\ &= 4+8 \\ &= 12 \end{aligned}$$

For the fractional part, .011001 we multiply and sum each bit, but starting with 2^{-1} power and continuing up to 2^{-2}, 2^{-3}, etc.

.	0	1	1	0	0	1
	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}

Recall that 2^{-1} is just 1/2, 2^{-2} is 1/4, 2^{-3} is 1/8, etc.

Summing this up gives us:

$$0 * 2^{-1} + 1*2^{-2} + 1*2^{-3} + 0*2^{-4} +0*2^{-5} +1*2^{-6}$$

$$\begin{aligned} &= 1/4 + 1/8 + 1/64 \\ &= .25 + .125 + .015625 \\ &= .390625 \end{aligned}$$

Putting these together gives us 12.390625.

Exercise: What is 0100.1111 in decimal?

Scientific Notation

What we've described is conceptually how to convert fractions from decimal to binary, but typically the data isn't stored in the computer in the same format. First we need to normalize the data, using scientific notation.

Consider the decimal number 0201.0900 . First we need to determine which digits are significant. The formal rules for significant digits is:

1. A nonzero digit is always significant
2. The digit 0 is never significant when it precedes nonzero digits

Using these rules, we can discard the initial 0 leaving us with 201.0900. Notice that we kept the trailing zeros; they may or may not be significant. Without more information we can't tell if we want precisely 201.0900, or if perhaps 201.09 is all we knew and the extra zeros are padding.

To represent this in scientific notation, we move the decimal point to the position immediately to the right of the leftmost significant digit, and multiply by the correct factor of 10 to get back the original value:

$$2.010900 * 10^2$$

In this case, we moved the decimal point two places to the left, so we multiply by 10^2. If we had a fraction we would do the opposite and multiply by a fraction of 10:

$$0.0020109 = 2.0109 * 10^{-3}$$

We can apply the same process to binary fractions, but use powers of 2 instead of powers of 10. Say that we have the binary value 100.1011. Converting this to scientific notation results in:

$$1.001011 * 2^2$$

Similarly, 0.00100 converted to scientific notation results in:

$$1.00 * 2^{-3}$$

Exercise: What is 101011.101 in scientific notation?

Floating Point Representation

Let's describe a simple format for storing floating point numbers. In practice, a different (but similar in principle) format called IEEE 754 is used in which 32 bits store a floating point value. In our example, we'll use 16 bits.

To represent a floating point number, first convert it to binary scientific notation. For example, let's say that we end up with 1.001011 * 2^3. In general terms, this value is: (Sign) * (Mantissa) * $2^{(\text{exponent})}$

In storing this value, by default, we will assume that the power is 2. To get this value back, we will need to store the "1.001011", the sign using a sign bit, and then the exponent 3.

The pieces that we must store are the:

- Sign
- Exponent (the 3)
- Mantissa (the 1.001011 part)

The format we will use for our 16 bit format is:

S	EEEEE	MMMMMMMMMM

1 5 10

1 bit is allocated to the sign field (the leftmost bit).
5 bits are allocated to store the exponent field.
10 bits are allocated to store the mantissa field.

Sign Field

This is just a sign bit, like we used with signed binary numbers. It is either 0 or 1. 0 indicates a positive number, and 1 indicates a negative number. For our example number of 1.001011* 2^3, this is positive so the sign bit would hold a 0.

Exponent Field

The exponent section is five bits long. In our case let's just store an integer using two's complement. In the IEEE format, a different notation (biased or excess notation) is used. For our example number of 1.001011 * 2^3 the exponent is 3, so we store 00011 for the exponent field.

Mantissa Field

The mantissa section stores the rest of the floating point number. Since the power of 2 is implicit, all that is left to store are the significant digits of the number to represent. In the case of our example of $1.001011 * 2^3$ this corresponds to the 1.001011 part.

Putting everything together we have:

Sign = 0
Exponent = 00011
Mantissa = 0001001011

Or: 0 00011 0001001011

Grouping in 4 bits each we can express this succinctly as a hex number:

0000 1100 0100 1011 = 0D4B

Precision

Precision refers to the number of bits that we can store in the mantissa. The more bits we have, the more precise the value we can represent. One way we can increase the precision slightly is by dropping the leftmost 1 of the mantissa. We can drop this bit because it will always be there in scientific notation. In calculating the floating point value we would have to insert the 1 back in. In our example we won't use this "hidden bit" but it is used in practice.

Let's look at the range of values we can store in our floating point representation. With 5 bits for an exponent, we can represent exponents using our two's complement notation from
10000 = -16 to
01111 = +15

So the smallest value we can store is $1.0 * 2^{-16}$ (pretty small!) and the biggest value we can store is $1.111111111 * 2^{15}$. If we expand this by moving the decimal point 15 places to the right then we get:

1111111111000000 = 65472

What's the second largest value we could store?
$1.111111110 * 2^{15}$ = 1111111110000000 = 65408

This means that our floating point number format can't exactly store any value between 65408 and 65472. This problem is more significant with larger numbers than with smaller numbers. We can alleviate the problem by using more bits for the mantissa, but ultimately all floating point number formats suffer from this problem.

In contrast, consider a 16 bit unsigned integer. It can store values from 0 to 2^{16} or 65536. There are also exactly 65536 different bit patterns. So using 16 bits for just integers can exactly represent all the values between 65408 and 65472. Of course, it can’t represent floating point values like ½ or 2.75. In the unsigned integer format, every one of the 65536 bit patterns is used to represent an integer. In the floating point format, there are still only 65536 bit patterns available. Some of those bit patterns are used to represent floating point numbers, so we lose the ability to represent some integers. It turns out we lose more large integers than small ones.