CSCI 120 Introduction to Computation
It's only a matter of representation (cont.) (draft)

Saad Mneimneh
Visiting Professor
Hunter College of CUNY

1 What about negative numbers?

People confronted the problem or representing negative numbers even before the first electronic computer. After all, negative numbers existed long time ago. Before we address this issue, let's see how many (positive) numbers can we represent with n bits. To answer this question, it is enough to determine the largest possible number, since we know that the smallest is zero. Let's transpose this back to the decimal system since we are much more familiar with it. What is the largest number that we can represent with n decimal digits? Well, it is 999...9 (n times). Therefore, the largest number is $10^n - 1$. For instance, with 3 decimal digits, the largest number we can represent is $10^3 - 1 = 999$. The same rule applies for the binary system. The largest number we can represent with n bits is $2^n - 1$. For instance, with 4 bits, the largest number that we can represent is $2^4 - 1 = 15$. Therefore, we have a total of 16 (positive) numbers (including 0).

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

In general, we have a total of 2^n numbers, and they are all positive ranging from 0 to $2^n - 1$. Current machines use $n = 32$ or even $n = 64$ bits (so that's a huge number).

One way of introducing negative numbers is by using one of the bits (usually the leftmost bit) as a sign bit. This seems natural because this is how we represent negative numbers on paper. We simply put a sign to the left of the

number. Therefore, in an n-bit computer, the leftmost bit (also called most significant bit) would be the sign bit followed by the $n-1$ bit number. We end up with the range $[0 \ldots 2^{n-1}-1]$ for positive numbers and $[-(2^{n-1}-1) \ldots 0]$ for negative numbers. This technique is called signed magnitude. Note that zero has two representations now, minus zero and positive zero. For $n = 4$ bits, this is what we get:

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

Basically, each number is interpreted as a sign bit followed by the representation of an $n-1$ bit number. So 1101 is -(101) and 101 in binary is 5, so 1101 is a representation of -5.

Although this seems natural to us, it is not natural to computers at all. Consider for instance when the computer adds two numbers. First it has to look at the signs of both numbers, then decide, based upon the signs, whether to add them or subtract one from the other, and what sign the result will have. Wouldn't it be nice if the computer's arithmetic unit never needed to know if a number is positive or negative and simply added any two numbers and still got the correct result? This turns out to be possible indeed.

2 One's complement

In the one's complement representation of negative numbers, we make the negative of a number its logical negation, i.e. we apply the NOT operator to every bit (everything inverted). As a consequence, the most significant bit is still a sign bit, but the order of the negative numbers is reversed. Here's the example for $n = 4$ bits again:

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

It turns out that we can perform addition without worrying about whether the numbers are positive or negative. This means that we gain subtraction for free. Therefore, we don't need a special circuit for adding and another one for subtracting. The same one will work for both!

Well, except for one tiny detail that is called "end-around carry". That is, whenever the addition results in an outermost carry, we have to bring this carry back and add it. The following is an example. Let's compute $7 - 4$. To perform this operation (a subtraction), we just need to add in binary the two representations of 7 and (-4):

$$\begin{array}{r} 0111 \\ +\ 1011 \\ \hline \mathbf{1}\ 0010 \end{array}$$

If we ignore the outermost carry (after all, we only have 4 bits), then the result is apparently wrong. It is 2 while it should be 3. By bringing this carry around and adding it to the result, we get the correct answer.

As long as we do not have overflow, i.e. the result of our operation is always in the range $[-7 \dots 7]$ (more generally $[-(2^{n-1} - 1) \dots 2^{n-1} - 1]$), this addition process always gives the correct answer. Obviously, we can possibly have overflow only when adding two positives or two negatives (for instance adding 5 and 5 should give 10, but 10 has no representation using 4 bits). It is easy to detect overflow: we get a negative number when adding two positives, or we get a positive number when adding two negatives. So by examining the first bit of each of the two numbers and the result, we can detect overflow.

The one's complement representation has a drawback, namely it did not eliminate the two zeros. We still have a positive zero 0000, and a negative zero 1111. This can create some problems because the computer has to handle special cases; for instance, comparison becomes a bit hard (imagine comparing two zeros). One would think that this is an easy to deal with situation; for instance, why not just use the positive zero all the time? Unfortunately, this cannot be avoided. For instance, adding 5 and -5 (or any number and its negative) gives 1111, the negative zero (now you may see why it is called one's complement). To eliminate the negative zero, the two's complement representation was developed.

3 Two's complement

The two's complement representation eliminates the presence of a negative zero by slightly changing the one's complement representation. Since -0 is not needed, the representation for -0 becomes -1, that of -1 becomes -2, etc... Here the example with $n = 4$ bits in two's complement representation:

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111

The name two's complement comes from the fact that a number and its negative add to 10000 (when $n = 4$), as compared to 1111 in one's complement. With this, we introduce two other drawbacks. First, there is an asymmetry in the positive and the negative ranges. For instance, with $n = 4$ bits, we have a -8 but we do not have an 8. Second, the negative of a number is not its logical negation anymore. Therefore, we have to have two separate circuits for inverting and negating.

On the other hand, we do not have to worry about the end-around carry. If we get an outermost carry, we can simply ignore it. This always gives the correct answer, provided that we do not have overflow. Overflow can be detected in the same way as before. Here's the same example of computing $7 - 4$ that we considered before (note that -4 is now 1100):

$$\begin{array}{r} 0111 \\ +\ 1100 \\ \hline \mathbf{1}\ 0011 \end{array}$$

Note that we can simply ignore the outermost carry. The answer is correctly 0011, which is the representation of 3.

While the negative of a number could be obtained by simply inverting every bit in the one's complement representation, a similar, but slightly more complicated process can be done in two's complement: scan the bits from right to left until you hit the first 1, copy all that part, but starting from the next bit on, invert everything. Let's try it on an example. Take for instance 6, which has the following representation: 0110. Scanning from right to left until we hit the first 1 produces 10. Now we continue scanning while inverting every bit. We get 10. Putting these together we have 1010, which is the correct representation of -6. If you try it the other way around, it should also work. Starting from 1010

and applying the same process will give us 0110.

Here's a table summarizing the features of the different representation systems:

	Binary	signed magnitue	1's complement	2's complement
Add	ignore left-most carry	not easy	add left-most carry	ignore left-most carry
Subtract	not easy	not easy	same as Add	same as Add
Negate	N/A	flip left-most bit	flip all bits	flip all bits and add 1
zeros	one zero	two zeros	two zeros	one zero

4 Excess representation

Excess representation makes comparison easy. For instance, the ALU would like to interpret that 0011 is smaller than 1001, regardless of what they represent (imagine a dictionary order with the alphabet 0 1). Therefore, in excess representation we take the first pattern of bits, **in alphabetical order**, and assign it to the smallest number we would like to represent. Then we go forward from that point. Here's the example of $n = 4$ bits in excess representation.

-8	0000
-7	0001
-6	0010
-5	0011
-4	0100
-3	0101
-2	0110
-1	0111
0	1000
1	1001
2	1010
3	1011
4	1100
5	1101
6	1110
7	1111

Excess representation is used for representing the exponent part of fractional number, as discussed below. One last word about representation of integers: With today's advanced computers and sophisticated circuitry, any alternative is a good one, despite its drawbacks. But two's complement seems to have dominated.

5 Fractions

As in the decimal system, to represent fractions in binary we use a radix point (it plays the exact same role as a decimal point). The bits to the left of the point represent the integer part (the whole part that is), and the bits to the right of the point represent the fractional part. The bits to the right of the point as interpreted in a manner similar to the other bits, except that their positions are assigned the fractional quantities 1/2, 1/4, 1/8, 1/16, 1/32, ... These are basically the negative powers of 2, i.e. 2^{-1}, 2^{-2}, 2^{-3}, 2^{-4}, 2^{-5}, ... This is

merely a continuation of the rule stated previously. Each position is assigned a quantity twice the size of the one to its right.

...	□	□	□	□	□	□	.	□	□	□	□	□	...
...	32	16	8	4	2	1		1/2	1/4	1/8	1/16	1/32	...

For instance, 101.101 is $1 \times 4 + 0 \times 2 + 1 \times 1 + 1 \times 1/2 + 0 \times 1/4 + 1 \times 1/8 = 4 + 1 + 1/2 + 1/8 = 5\frac{5}{8}$. Note it is not 5.5.

				1	0	1		1	0	1			
...	□	□	□	□	□	□	.	□	□	□	□	□	...
...	32	16	8	4	2	1		1/2	1/4	1/8	1/16	1/32	...

When dealing with fractions such as 101.101 above, we need not only store the integer part, but also the fractional part that comes after the radix point. One simple way of doing it is to visualize an imaginary point and split the bits in half to represent the two parts (one before the point and one after the point). However, this is not very flexible. For instance, what if we want to represent 101.101011010010011. It seems to be a waste of bits to represent both parts using the same number of bits (the fractional part requires a lot of bits). It would be more efficient if, given the bits we have, we can assign just three bits for the representation of 101, and the rest of the bits to accommodate the fractional part. Such a representation, is called the *floating-point* representation (that's why sometimes fractional numbers are called *floats*. Basically, the position of the radix point is not fixed.

To represent a moving point, we have to say where the point is. So a number of bits must be reserved to encode this information. These bits are known as the *exponent*. For instance, to represent 101.101011010010011, we would need the representation 101101011010010011, and another number saying that the point comes after the third bit.

Let's illustrate the idea with an example using just one byte, i.e. 8 bits (of course a real computer would use 32 or 64 bits).

We use the leftmost bit (the most significant bit) as a sign bit (remember we still need to represent negative fractions). Again 0 means that the number is positive, and 1 means that it is negative. We are left with 7 bits. We divide them into two parts: (1) the exponent, to encode the position of the radix point, and (2) the actual bit pattern that represents the number (positive, because we are using a sign bit), this is often called the *mantissa*. For illustration, let's designate three bits for the exponent and four bits for the mantissa.

$\underbrace{\square}$	$\underbrace{\square\square\square}$	$\underbrace{\square\square\square\square}$
sign	exponent	mantissa

Since the floating point can move either left or right, the exponent can have either a negative or a positive value (for instance -1 moves left one position). Usually, this value is encoded using excess representation, where the pattern consisting of 1 followed by all zeros is considered to be the **0** (see Section 4). Moreover, since the floating point will move either left or right (as dictated by the exponent), we must, by convention, assume that it has an initial position. We assume that initially, there is an *implicit* **1.** to the left of the mantissa.

Now suppose the byte consists of the following: 11010110. The first bit is 1; therefore, this number is negative. Now the next three bits represent the exponent. The exponent is 101, which, in excess representation of 3 bits, is +1. The

last four bits are the mantissa. The mantissa is 0110. Therefore, it represents the pattern 1.0110. Putting it all together, this byte is a representation of the fractional number (after moving the floating point one position to the right):

$$-10.11$$

which is $-2\frac{3}{4}$.

Let's try to encode $\frac{5}{8}$. First we express it in binary 0.101. Therefore, we know that our mantissa has to be 0100 (because of the implitict **1.** to the left of the mantissa). The mantissa now represents 1.0100. Therefore, we need to move the floating point one position to the left. So the exponent must be -1, i.e. 011 in excess representation. Finally, since the number is positive, the sign bit is 0. We end up with 00110100.

Interesting question: If an implitict **1.** is assumed to exist to the left of the mantissa, how de we represent for instance the number zero (there are no ones in zero)? This is usually handled by a special case: when the exponent is 000 (the smallest value), there is an implicit **0.** to the left of the mantissa. This, however, creates non-unique representations for small numbers. For instance 0 is 0 000 0000 (or 1 000 0000), and 0.000001 is 0 000 0100 or 0 001 0010 or 0 010 0001.

6 Text

Text must also be represented with bits. Therefore, each symbol in the text must have a unique bit pattern. The text is then represented as a long string of bits in which the successive patterns represent the successive symbols in the text.

Many ways to code text into binary are possible. In fact in the 1940s and the 1950s, many such codes were designed, depending on the equipment being used. This caused a lot of incompatibilities in communicating information.

To remedy this problem, the American National Standards Institute (ANSI), adopted the American Standard Code for Information Interchange **ASCII**. The ASCII code is a 7 bit code to represent:

- Upper and lower case letters of the alphabet (52 patterns)
- Digits from 0 to 9 (10 patterns)
- Control information such as Tabs, Line feeds, etc...
- Some extra symbols like punctuations

Since 7 bits can represent 128 different symbols, the ASCII code contains other symbols in addition to letters and digits and punctuations. However, today, the ASCII code is extended to 8 bits, by adding a 0 as the most significant bit (so that it conforms to a byte), giving room for 128 extra symbols (by setting the first bit to 1).

The following 6 bytes represent the text "Hello.":

01001000	01100101	01101100	01101100	01101111	00101110
H	e	l	l	o	.

Although ASCII has been dominant for many years, other code have been developed to represent documents in a variety of languages. One of these codes is **Unicode**. Unicode was developed through a collaboration of several leading companies of hardware and software. Unicode uses 16 bits (instead of 8), and hence is capable or representing 63335 different bit patterns, enough to represent Chinese, Japanese, Arabic, etc...

The International Organization for Standardization (also known as ISO) also developed a code using 32 bit patterns. This can represent billions of different symbols.