Signed Number Representations
CS101, Mock

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. There are three widely used techniques for doing this: sign/magnitude, complementation, and binary coded decimal.

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:

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. In a minute we will examine another coding scheme that eliminates the duplicates.

Radix Complementation – Two's Complement

Radix complementation is used to represent signed quantities and is based on the ideas of modular arithmetic. In modular arithmetic, there is a value called the Modulus (*M*) which when added to or subtracted from a number, does not change its value.

If we are representing the integer A, where A is composed of n bits, then if A is positive the sign bit, A_{n-1} is zero. The remaining n-1 bits represent the magnitude of the number as in sign magnitude:

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. The remaining n-1 bits are used to represent the negative integers from –1 to -2^{n-1}. Note that we will have the ability to represent one additional negative integer than positive integers, because we're using up one of the patterns starting with 0 to represent 0.

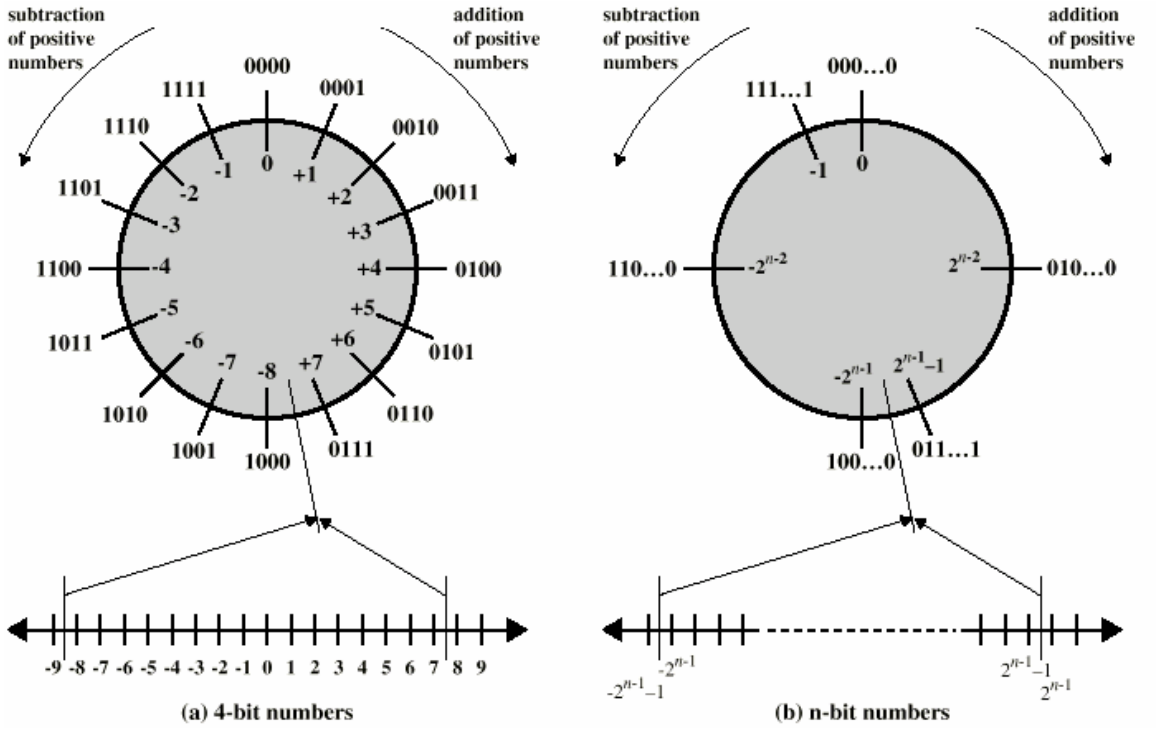
We would like to assign the negative integers to the available bit patterns in a way that facilitates straightforward arithmetic. A method that does this is to use the following formula:

$$Val = -2^{n-1} A_{n-1} + \sum_{i=0}^{n-2} 2^i A_i$$

Consider a positive number. In this case, A_{n-1} is zero. We end up with only the summation of the remaining terms.

Now consider a negative number. The A_{n-1} term is one, so we must add in -2^{n-1}. Now consider if all of the remaining bits are all one's. These will all add up to be $+2^{n-1}$ -1. It will be one smaller than the negative value; e.g. 1111 = yields $-2^3 + 7$, or –8 + 7 = -1. No matter how many bits we have, if they are all ones, we will end up with –1.

Now consider if all of the bits are one's except for the rightmost bit. We have the same case as before, except the positive value will be $+2^{n-1} - 2$ since we just subtracted one from the positive value. When we add this to the negative value, we end up with –2. The end result is we are counting backwards with the negative values, instead of counting forward as with positive values. This is shown in the "circle" 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

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

If you are given a binary value in two's complement and want to know what the value is in decimal, then the process is to:

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 returned the octal number 27, what is this in decimal?

27 octal in binary = 010 111
We have only 5 bits though, so this gives us 10111
This starts with 1, so it is negative. Flip the bits: 01000
Add one to get 01001
This results in 9, so the original number was –9

If the computer returned the hexadecimal number 1D, with m=6, what is this in decimal?

1D in binary is 0001 1101
We have 6 bits, this gives us 011101
It starts with 0, so it is positive.
DON'T FLIP ANY BITS! We just convert this as unsigned binary:
011101 = 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*. The specific uses for this register and the carry bit are discussed later.

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.