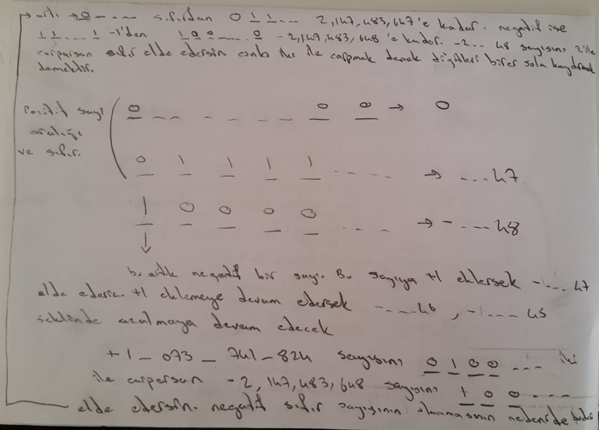
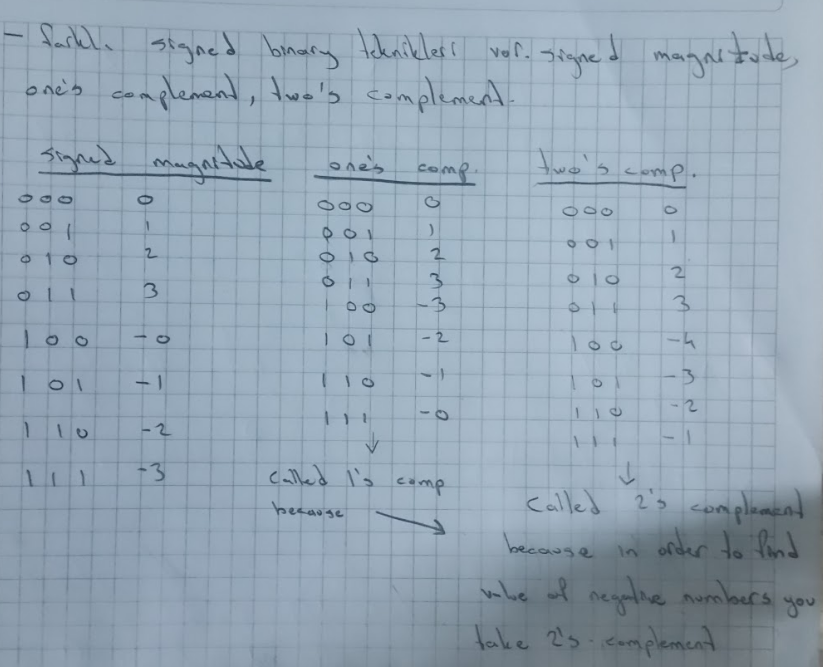
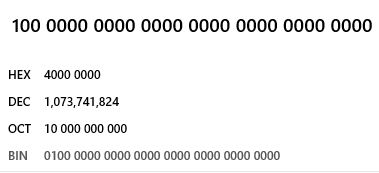
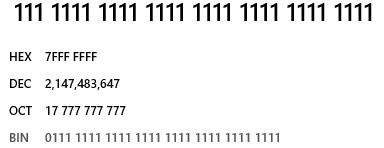
**Binary representation of integer and floating point values**

If we have n bits (n > 0), the unsigned numbers we can represent are from 0 to 2n – 1. The left most bit alone has the value of 2n-1, rest of the bits combined have the value 2n-1 – 1. The signed numbers you can represent with n bits are from -2n-1 to 2n-1 – 1 (including 0 we have 2n numbers).



Correction: One’s complement doesn’t actually work. When you do use the values of one’s complement and add 4 with -2, you get -1. In order for one’s complement to work, you need to add one after taking one’s complement. But two’s complement works fine.

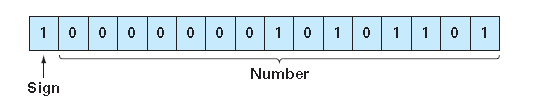
 

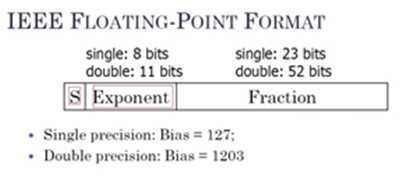


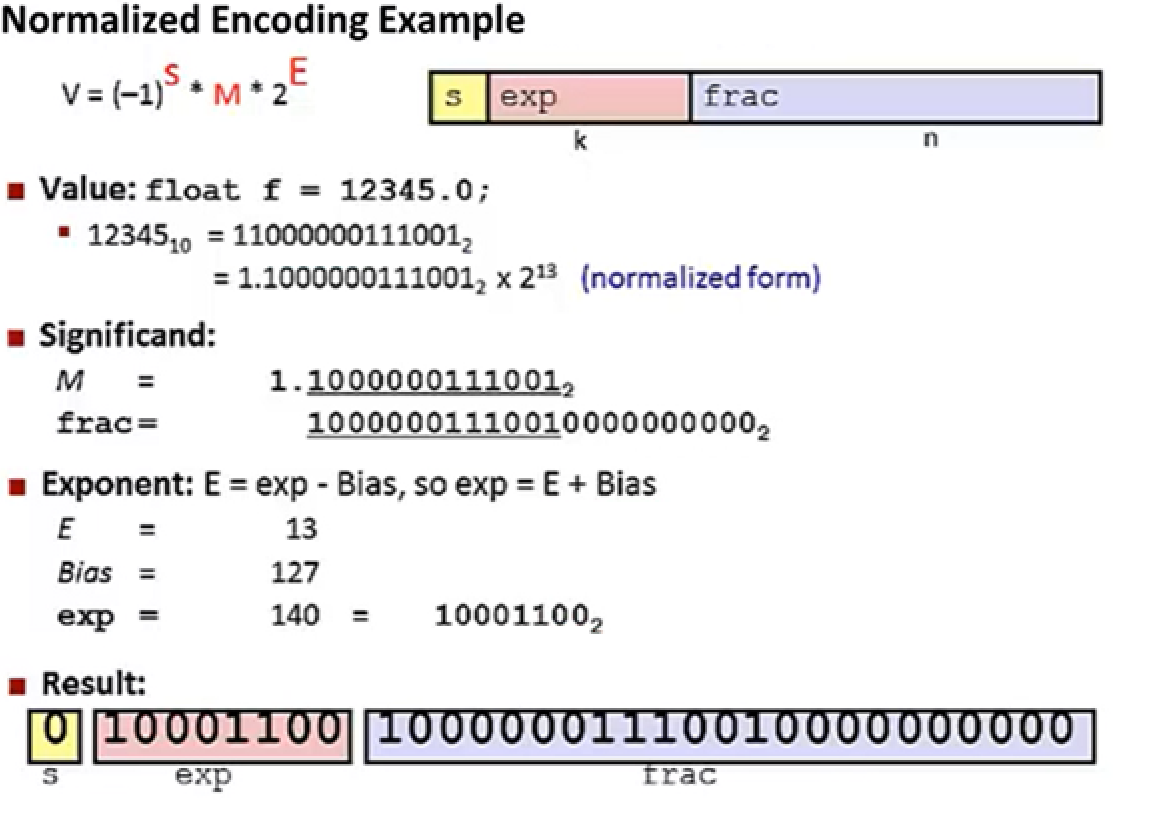
Left most bit is sign bit. 0 for positive 1 for negative. When we have 1000...0 we get -2....48. Numbers are two’s complement in Java. So when you take two’s complement of a negative number (in binary) you can see its value. For example 1111 is -1. Two’s complement of 1111 is 0001. In two’s complement you look for the first 1 from right and flip the bits left of that 1. Or you can just flip all bits and then add one.

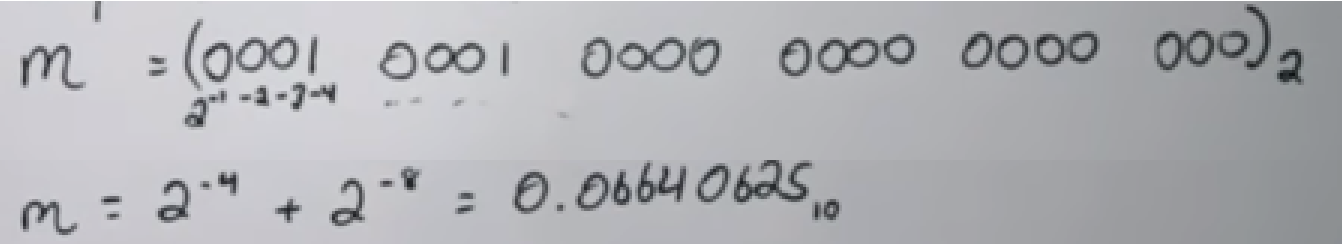
This shortcut is based on the observation that the sum of a number and its inverted representation must be 111 . . . 111two, which represents -1. Since  therefore   (We use the notation  to mean invert every bit in x from 0 to 1 and vice versa. one’s complement)   
 Two’s complement gets its name from the rule that the unsigned sum of an *n*-bit number and its *n*-bit negative is 2*n*; hence, the complement of a number *x* is 2*n* - *x*, or its “two’s complement.” 1101 + 0011 = 1 000 which is 24.  
 In signed magnitude, you use the same system for positive and negative numbers. But positive numbers have left most bit as 0 and negative numbers have left most bit as 1.

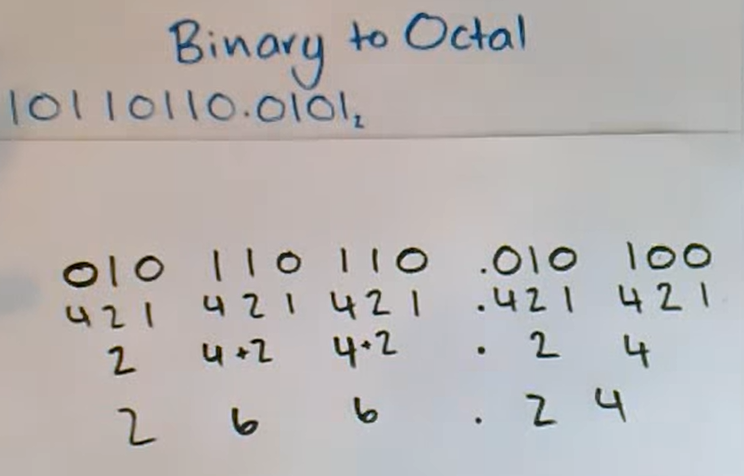
[Why is two's complement better than signed magnitude](https://stackoverflow.com/questions/1125304/why-prefer-twos-complement-over-sign-and-magnitude-for-signed-numbers)

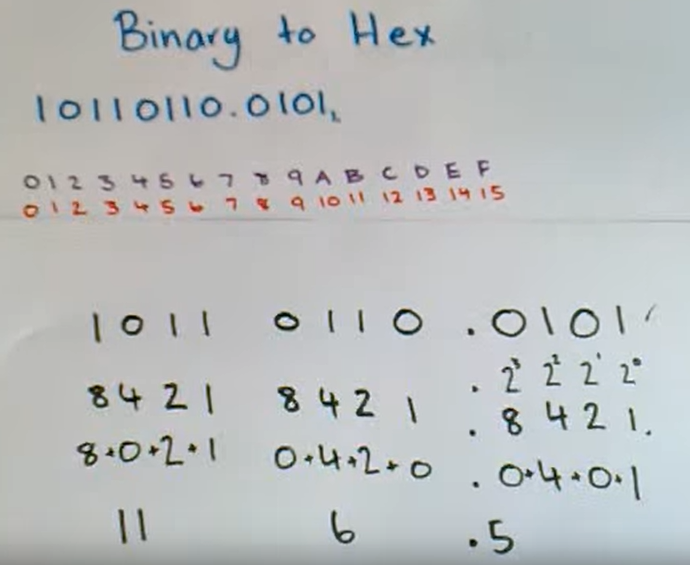
**

Bit pattern 0...0 represents zero. Exp = 1...0 and frac = 0...0 represents infinity. Exp = 1...1 and frac != 0...0 represents NaN (Not a Number)  
  
  
IEEE 754 floating point format



  
  
[Floating point to binary](https://www.youtube.com/watch?v=tx-M_rqhuUA)

-Converting between hexadecimal <-> binary <-> octal: It takes 3 bits to write the maximum digit of octal (7) and 4 bits for hexadecimal (15). So it would take 3 bits to code 8 different digits and it would take 4 bits to code 16 different digits. So if you see the number 7FFF EFB0, that means it would take 8 x 4 = 32 bits to encode that hexadecimal value. Also this is the reasoning behind the conversion we do in logic circuit design (Base 8 to 2 to 16). When you see a binary number, you group the bits in groups of 3 from right to make it octal. Group bits in groups of 4 to make it hexadecimal. Add 0s to the left if you need to fill the last group. But for floating point part, add 0s to the right, not left.  
  




When it comes to octal or hex values, you dont make the octal/hex value itself negative. Octal/Hex is just a way of showing binary. When you see a octal/hex, just convert it to binary to see the actual value.  
[Negative octal/hex](https://www.youtube.com/watch?v=zC9cd9w75Nc)

-Converting between decimal and binary: There are two ways to convert a decimal into binary. The second method works because it first checks whether the number is odd or even. This determines whether the right most bit is 0 or 1. Then we divide (integer division) the number by 2 (right shifting by 1). Now we check whether second bit from right is 0 or 1.

