CSE 351 Section 3 – Integers and Floating Point

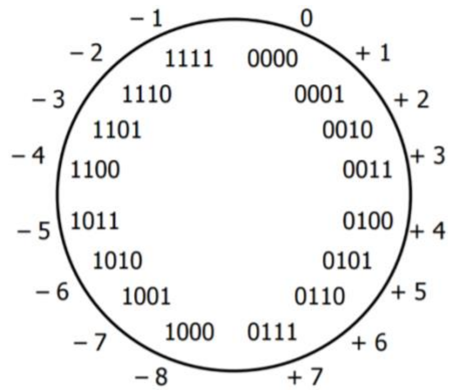
Welcome back to section, we're happy that you're here ☺ .. . .

Signed Integers with Two's Complement

Two's complement is the standard for representing signed integers:

- The most significant bit (MSB) has a negative value; all others have positive values (same as unsigned)
- Binary addition is performed the same way for signed and unsigned
- The bit representation for the negative value (additive inverse) of a Two's Complement number can be found by:

  flipping all the bits and adding 1 (i.e. $-x = \sim x + 1$).



The "number wheel" showing the relationship between 4-bit numerals and their Two's Complement interpretations is shown on the right:

- The largest number is 7 whereas the smallest number is -8
- There is a nice symmetry between numbers and their negative counterparts except for -8

Exercises: (assume 8-bit integers)

1) What is the **largest integer**? The **largest integer + 1**?

Unsigned:	Two's Complement:
2^8 - 1 = 255 largest + 1 = 0 (modular arithmetic) 1111 1111 + 0000 0001 = 0000 0000	2^{8-1} - 1 = 127 largest = 0111 1111 largest + 1 = 1000 0000 = 2^-7 = -128

2) How do you represent (if possible) the following numbers: **39**, **-39**, **127**?

Unsigned:	Two's Complement:
39: 32 + 4 + 2 + 1 = 2^5 + 2^2 + 2^1 + 2^0 = 0010 0111	39: 32 + 4 + 2 + 1 = 0010 0111
-39: not possible	-39: -128 + 64 + 16 + 8 + 1 = 1101 1001
127: 64 + 32 + 16 + 8 + 4 + 2 + 1 = 0111 1111	127: 0111 1111

3) Compute the following sums in binary using your **Two's Complement** answers from above. *Answer in hex.*

use fact that -x = ~x + 1

a. 39 -> 0b 0 0 1 0 0 1 1 1 +(-39) -> 0b 1 1 0 1 1 0 0 1 0x 0 0 <- 0b 0 0 0 0 0 0 0 0	**b.** 127 -> 0b 0 1 1 1 1 1 1 1 + (-39) -> 0b 1 1 0 1 1 0 0 1 0x 58 <- 0b 0 1 0 1 1 0 0 0
c. 39 -> 0b 0 0 1 0 0 1 1 1 - 127 -> 0b 1 0 0 0 0 0 0 1 0x A 8 <- 0b 1 0 1 0 1 0 0 0	**d.** 127 -> 0b 0 1 1 1 1 1 1 1 + 39 -> 0b 0 0 1 0 0 1 1 1 0x A 6 <- 0b 1 0 1 0 0 1 1 0

4) Interpret each of your answers above and indicate whether-or-not overflow has occurred.

a. 39+(-39) Unsigned: Two's Complement:	**b.** 127+(-39) Unsigned: Two's Complement:
c. 39-127 Unsigned: Two's Complement:	**d.** 127+39 Unsigned: Two's Complement:

Goals of Floating Point

Representation should include: [1] a large range of values (both very small and very large numbers), [2] a high amount of precision, and [3] real arithmetic results (*e.g.* ∞ and NaN).

IEEE 754 Floating Point Standard

The value of a real number can be represented in scientific binary notation as:

$$\text{Value} = (-1)^{\text{sign}} \times \text{Mantissa}_2 \times 2^{\text{Exponent}} = (-1)^S \times 1.M_2 \times 2^{E\text{-bias}}$$

The binary representation for floating point values uses three fields:

- **S**: encodes the *sign* of the number (0 for positive, 1 for negative)
- **E**: encodes the *exponent* in **biased notation** with a bias of $2^{w-1}-1$
- **M**: encodes the *mantissa* (or *significand*, or *fraction*) – stores the fractional portion, but does not include the implicit leading 1.

	S	E	M
`float`	1 bit	8 bits	23 bits
`double`	1 bit	11 bits	52 bits

How a `float` is interpreted depends on the values in the exponent and mantissa fields:

E	M	Meaning
0	anything	denormalized number (denorm)
1-254	anything	normalized number
255	zero	infinity (∞)
255	nonzero	not-a-number (NaN)

with denormalized numbers, we drop the leading implicit zero on M so we can represent numbers close to zero

Exercises:

Bias Notation

5) Suppose that instead of 8 bits, E was only designated 5 bits. What is the bias in this case? 2^{5-1} -1 = 15 ______________

6) Compare these two representations of E for the following values:

Exponent E - Bias	E (5 bits) bias = 15					E (8 bits) bias = 127							
1	1	0	0	0	0	1	0	0	0	0	0	0	0
0	0	1	1	1	1	0	1	1	1	1	1	1	1
-1	0	1	1	1	0	0	1	1	1	1	1	1	1

Notice any patterns?

1.25 = 1.25 * 2^0 --> E - Bias = 0, M = 0.25, Bias = 127

Floating Point / Decimal Conversions

7) Convert the decimal number 1.25 into single precision floating point representation:

0	0	1	1	1	1	1	1	1 0	1						0																

-7.375 = -1 * (4 + 2 + 1 + 0.25 + 0.125) -1 * 2^2 * (1 + 1/2 + 1/4 + 1/16 + 1/32)

8) Convert the decimal number -7.375 into single precision floating point representation:

E - 127 = 2 --> E = 129
M = 1/2 + 1/.4 + 1/16 + 1/32

1	1	0	0	0	0	0	0	1	1	1	0	1	1	0																	

9) Add the previous two floats from exercise 7 and 8 together.
Convert that number into single precision floating point representation:

10) Let's say that we want to represent the number 3145728.125 (2^21 + 2^20 + 2^-3) = 2^21 * (1 + 1/2 + 2^{-24})

a. Convert this number to into single precision floating point representation: E - 127 = 21 --> E = 148

1	1	0	0	1	0	1	0	0	1	0	0																				

b. How does this number highlight a limitation of floating point representation?

2^{-24} is not representable in M as we only have 23 bits allocated to it.

11) What are the decimal values of the following `float`s?

0x80000000

1000 0000 0000 … = -0

0xFF94BEEF

1111 1111 1001 0100 1011 1110 1110 1111 =

Nan

0x41180000

0100 0001 0001 1000 0000 ...

1 * 2^{130-127} * (1 + 1/8 + 1/16)
the one in the right sum is the implicit one from 1.M

= 9.5

Floating Point Mathematical Properties

- Not <u>associative</u>: $(2 + 2^{50}) - 2^{50} \neq 2 + (2^{50} - 2^{50})$
- Not <u>distributive</u>: $100 \times (0.1 + 0.2) \neq 100 \times 0.1 + 100 \times 0.2$
- Not <u>cumulative</u>: $2^{25} + 1 + 1 + 1 + 1 \neq 2^{25} + 4$

<u>Exercises</u>:

12) Based on floating point representation, explain why each of the three statements above occurs.

Trunctation occurs when adding 2 + 2^50 (same reason as in problem 10) because the numbers are too far apart. Hence 2+2^50 = 2^50.
This does not occur on the RHS when we add the two 2^50 together first. So we get 0 =/= 2

The representations for 0.1 and 0.2 are not exact in the single precision 32 bits.

Again, 2^25 + 1 = 2^25 as the difference in powers is too large for M to completely represent the sum. But the difference between 2^25 + 4 is just small enough that the sum can be represented.

13) If `x` and `y` are variable type `float`, give two *different* reasons why `(x+2*y)-y==x+y` might evaluate to false.

Lack of associativity: maybe (x+2y) - y =/= x + (2y-y)
Overflow: if x,y are too large, x+2y may be infinity
(overflow) while x+y is not

1EEE 754 Float (32 bit) Flowchart