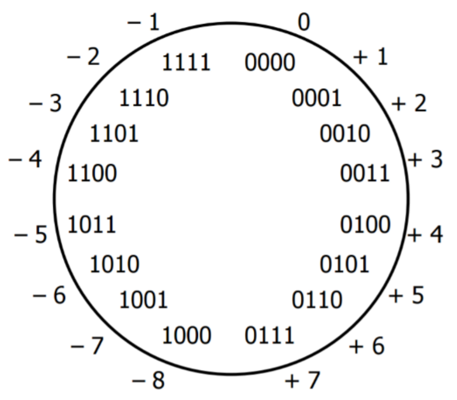
CSE 351 Section 3 – Integers, FP, and Assembly

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

Unsigned:	Two's Complement:
39: 0010 0111	39: 0010 0111
-39: Impossible	-39: 1101 1001
127: 0111 1111	127: 0111 1111

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

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 **5 8** <- 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**

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

a. 39 + (-39) Unsigned: 0 overflow Two's Complement: 0 no overflow	**b.** 127 + (-39) Unsigned: 88 overflow Two's Complement: 88 no overflow
c. 39 + (-127) Unsigned: 168 overflow Two's Complement: -88 no overflow	**d.** 127 + 39 Unsigned: 166 no overflow Two's Complement: -90 overflow

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:

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

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)

Exercises:

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

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

5) What are the decimal values of the following floats?

0x80000000	0xFF94BEEF	0x41180000
–0	NaN	+9.5

0x41180000 = 0b 0|100 0001 0|001 1000 0…0.
S = 0, E = 128+2 = 130 → Exponent = E – bias = 3, Mantissa = 1.0011_2
$1.0011_2 \times 2^3 = 1001.1_2 = 8 + 1 + 0.5 = 9.5$

Floating Point Mathematical Properties

- Not associative: $(2 + 2^{50}) - 2^{50}$!= $2 + (2^{50} - 2^{50})$
- Not distributive: $100 \times (0.1 + 0.2)$!= $100 \times 0.1 + 100 \times 0.2$
- Not cumulative: $2^{25} + 1 + 1 + 1 + 1$!= $2^{25} + 4$

Exercises:

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

Associative: Only 23 bits of mantissa, so $2 + 2^{50} = 2^{50}$ (2 gets rounded off). So LHS = 0, RHS = 2.

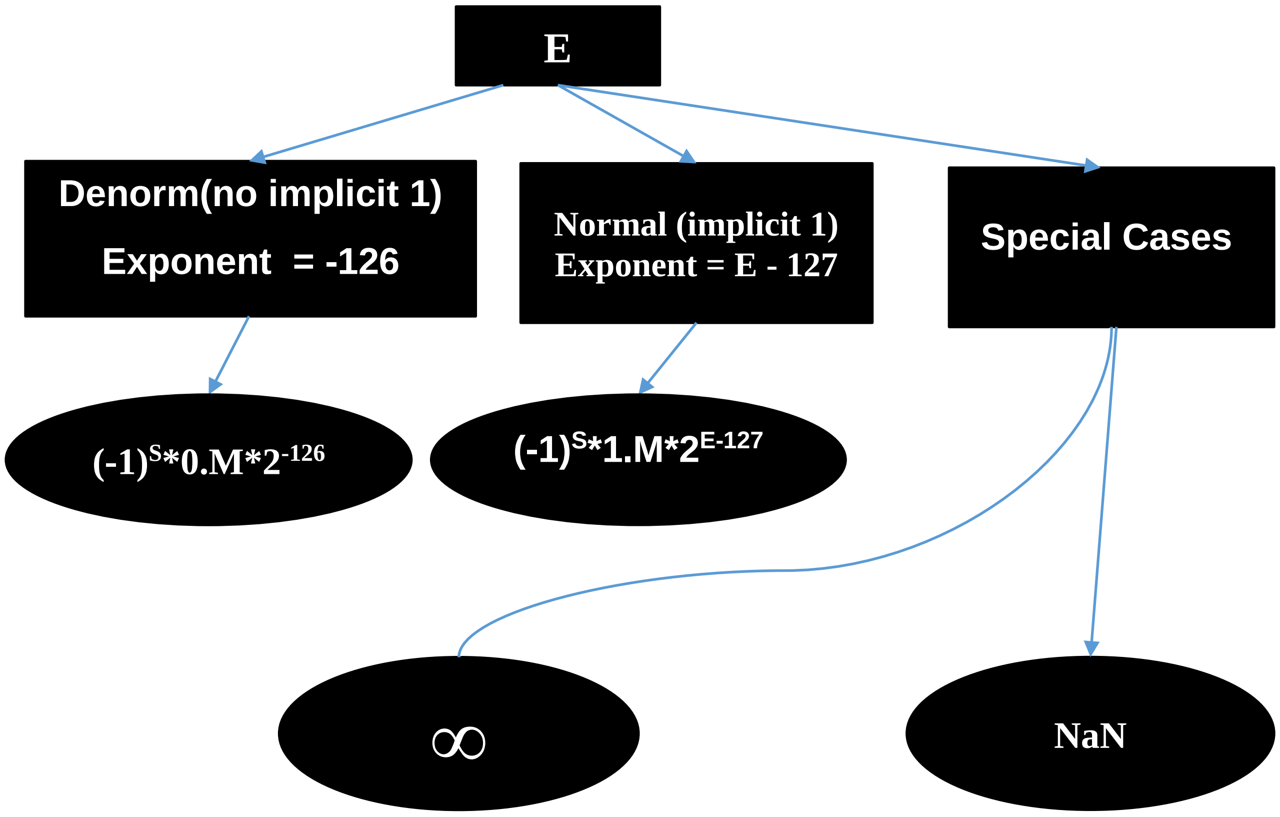
Distributive: 0.1 and 0.2 have infinite representations in binary point (0.2 = $0.\underline{0011}_2$), so the LHS and RHS suffer from different amounts of rounding (try it!).

Cumulative: 1 is 25 powers of 2 away from 2^{25}, so $2^{25} + 1 = 2^{25}$, but 4 is 23 powers of 2 away from 2^{25}, so it doesn't get rounded off.

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

(1) Rounding error: like what is seen in the examples above.
(2) Overflow: if x and y are large enough, then x+2*y may result in infinity when x+y does not.

IEEE 754 Float (32 bit) Flowchart



x86_64 Assembly

- These are the actual instructions which are running on your computer
- In practice, it's another language you get to learn and become familiar with

- Recall:
  - Registers hold 64 bits, and are names things like `rax, rbx, rcx, rdx, rdi, rsi, rbp, rsp`. In x86-64 assembly, they are prefixed with "%".
  - Instructions consist of an opcode, and then multiple operands, e.g.
    - `movq %rdi, %rbp`
    - This moves (copies) the 64 bit contents of register `rdi` into register `rbp`
  - Destination registers are always on the **Right Hand Side** (opposite of assignment statements in C, Java, Python, Ruby, C++, C#, Bash, Scala, and most any other language you can think of)
  - In order to follow (dereference) a pointer value contained in a register, use parentheses, e.g.
    - `movq %rdi, (%rbp)`
    - This moves (copies) the 64 bit contents of `rdi` into the memory location which `rbp` points to
  - You may only use one set of parens per instruction (an instruction is either a load or a store, but not both)
  - Immediate (constant) operands are denoted with "$" (e.g. `movq $13, %rax`)

The Swap Example (again)

In class, we discussed a series of instructions to swap two locations in memory. Below are two similar sequences, except each has at least one bug. However, buggy assembly doesn't mean the program crashes; it just means it does something different from what we want. Explain what each sequence of instructions actually does, and fix it to swap the two intended locations. It may help you to draw a picture.

1.
```
movq  %rdx, (%rdi)
movq  %rax, (%rsi)
movq  (%rdi), %rax
movq  (%rsi), %rdx
```

2.
```
movq  (%rdi), %rax
movq  %rdx, (%rdi)
movq  (%rsi), %rdx
movq  %rax, (%rsi)
```

1. This clobbers memory at (%rdi) and (%rsi) first with the contents of %rdx and %rax. Do the first two instructions last, and the last two first

2. This is almost there, but stores the original contents of %rdx to (%rdi) instead of what was loaded from (%rsi). Switch instructions 2 and 3 to get the right answer.