CSE 351 Section 3 – Floating Point and x86-64 Assembly

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

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:

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

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:

Bias Notation

1) Suppose that instead of 8 bits, E was only designated 5 bits. What is the bias in this case? _____

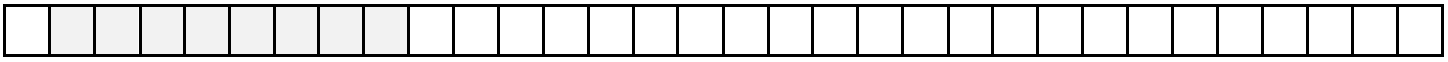
2) Compare these two representations of E for the following values:

Exponent	E (5 bits)	E (8 bits)
1		
0		
-1		

Notice any patterns?

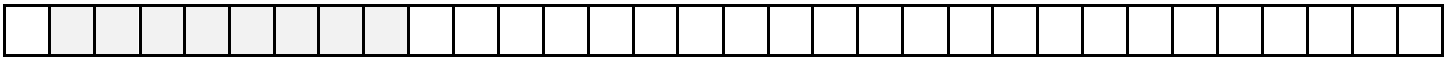
Floating Point / Decimal Conversions

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



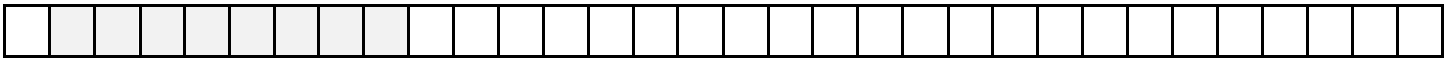
4) Convert the decimal number -7.375 into single precision floating point representation:

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



6) Let's say that we want to represent the number 3145728.125 (broken down as $2^{21} + 2^{20} + 2^{-3}$)

   a. Convert this number to into single precision floating point representation:



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

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

   0x80000000 0xFF94BEEF 0x41180000

Floating Point Mathematical Properties

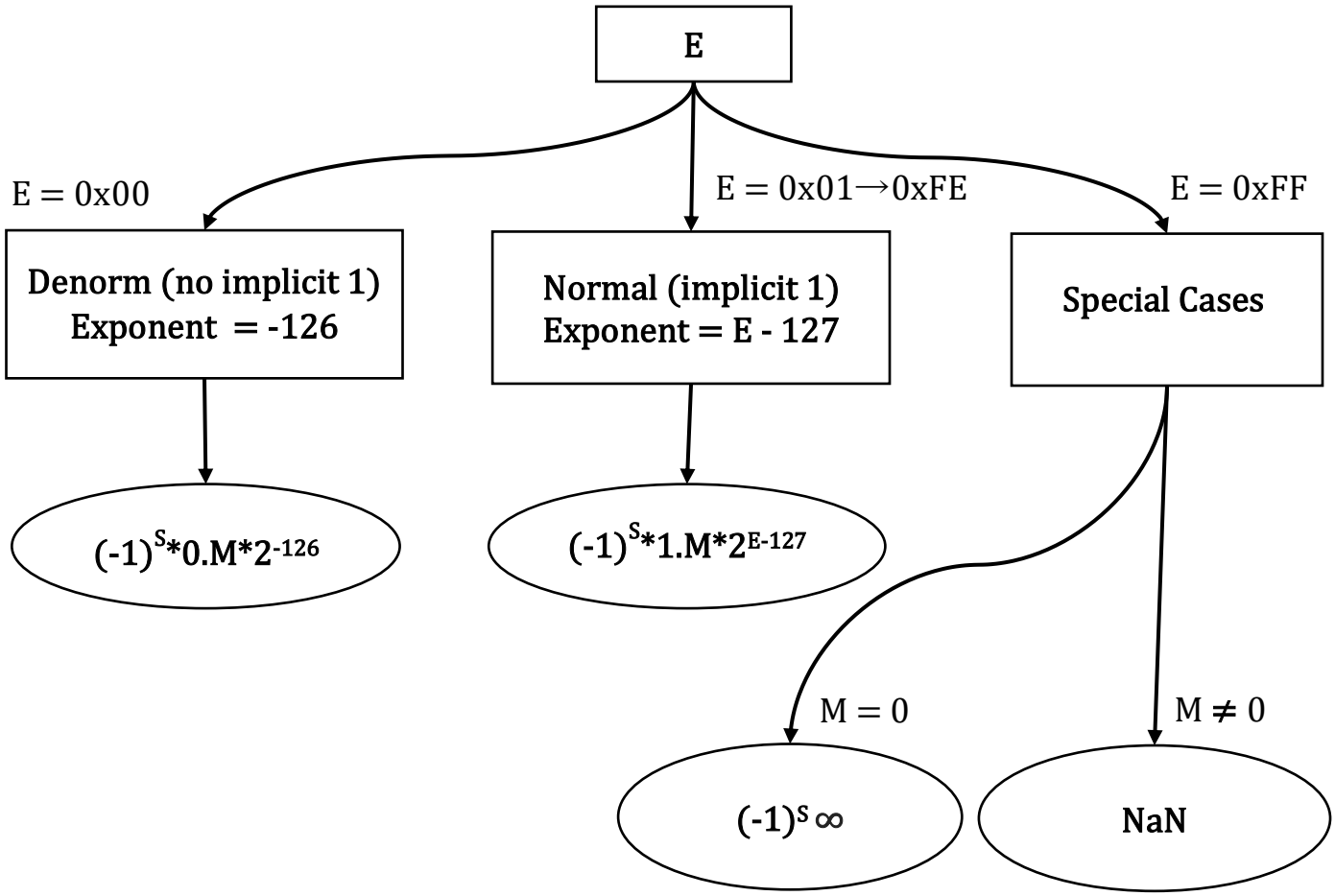
- Not <u>associative</u>: $(2 + 2^{50}) - 2^{50} \neq 2 + (2^{50} - 2^{50})$
- Not <u>distributive</u>: $100 \times (0.1 + 0.2) \neq 100 \times 0.1 + 100 \times 0.2$
- Not <u>cumulative</u>: $2^{25} + 1 + 1 + 1 + 1 \neq 2^{25} + 4$

<u>Exercises</u>:

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

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

IEEE 754 Float (32 bit) Flowchart



x86-64 Assembly Language

Assembly language is a human-readable representation of machine code instructions (generally a one-to-one correspondence). Assembly is machine-specific because the computer architecture and hardware are designed to execute a particular machine code instruction set.

x86-64 is the primary 64-bit instruction set architecture (ISA) used by modern personal computers. It was developed by Intel and AMD and its 32-bit predecessor is called IA32. x86-64 is designed for complex instruction set computing (CISC), generally meaning it contains a larger set of more versatile and more complex instructions.

For this course, we will utilize only a small subset of x86-64's instruction set and omit floating point instructions.

x86-64 Instructions

The subset of x86-64 instructions that we will use in this course take either one or two operands, usually in the form: `instruction operand1, operand2`. There are three options for operands:

- Immediate: constant integer data (*e.g.* `$0x400`, `$-533`) or an address/label (*e.g.* `Loop`, `main`)
- Register: use the data stored in one of the 16 general purpose registers or subsets (*e.g.* `%rax`, `%edi`)
- Memory: use the data at the memory address specified by the addressing mode `D(Rb,Ri,S)`

The operation determines the effect of the operands on the processor state and has a suffix ("`b`" for byte, "`w`" for word, "`l`" for long, "`q`" for quad word) that determines the bit width of the operation. Sometimes the operation size can be inferred from the operands, so the suffix is omitted for brevity.

x86 instruction	English equivalent
`movq $351, %rax`	Move the number 351 into 8-byte (quad) register "rax"
`addq %rdi, %rsi`	
`movq (%rdi), %r8`	
`leaq (%rax,%rax,8), %rax`	

Exercises:

1. [CSE351 Au14 Midterm] Symbolically, what does the following code return?

```
movl  (%rdi), %eax          # %rdi -> x
leal  (%eax,%eax,2), %eax   # %rax -> r
addl  %eax, %eax
andl  %esi, %eax            # %rsi -> y
subl  %esi, %eax
ret
```

2. Log on to Gradescope and start the "GDB Tutorial (optional)" assignment.
   This includes the basic workflow on how to use GDB, and should prove very useful for Lab 2 and beyond (Q4 even includes a walkthrough of Lab 2 Phase 1).