Floating Point Numbers

Warm-up: If you have 8 binary bits, what are the largest and smallest numbers you can represent?

Suppose you have the following bits in memory.

10111111010110001000000000000000

How many numbers is this? What are they?

The first requirement in developing a standard model for computer arithmetic is that numbers of a particular type need to take up the same number of bits. The most common are *single* and *double* precision numbers, which get 32 and 64 bits respectively.

You might then conclude that this is one single precision number, and start to translate the binary. However, there are still some open questions regarding our storage format. How do you store a negative number? What about a decimal point?

Assuming this is one single precision number, the bits are divided up this way:

$$1 \mid 01111110 \mid 10110001011111100000000$$

which means the exponent of the number is $2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 = 126$ minus the bias of 127, resulting in -1. So the floating point representation is

$$-1.10110001000001000000000 \times 2^{-1}.$$

Converting to decimal, we get $-1 \cdot (\frac{1}{2} + \frac{1}{4} + \frac{1}{16} + \frac{1}{32} + \frac{1}{2^9} + \frac{1}{2^{15}}) = -\frac{1239}{1465} = .845733642578125$.

Example 1. Convert the following floating point numbers to decimal.

- (single precision) 0 1000 0000 1000 0000 0000 0000 0000 000
- (single precision) 1 1000 0011 1111 0100 0000 0000 0000 000
- (double precision) 0 0111 1111 111 1110 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Example 2. How would you represent 10.25? 0?

Example 3. What's the next floating point number after 1 in single precision? What's the difference between them? (The difference is *machine epsilon*)

Example 4. What's the smallest positive number representable exactly in single precision?

The terms *overflow* and *underflow* refer to the condition of arithmetic operations when the result cannot be stored. Overflow is when the result is too big (exponent beyond p), and underflow is when it is too small (smaller than the above-mentioned smallest positive subnormal). Usually an underflow result is just rounded to 0. Most will convert an overflow result to plus or minus infinity, or NAN (not a number), both of which are also representable using the floating point representation: they use the all 1's exponent. If the mantissa is all 0's, the number is an infinity; otherwise it is NAN.

Rounding

When learning to round numbers to the nearest integer, you probably learned something very close to what we are doing now. When storing numbers, we round to the closest floating point representation. But what about when it's exactly halfway in between?

Unlike what you were probably taught, we try NOT to always round in the same direction.

Example 5. Suppose you're working with two standard decimal numbers, where you round to the nearest and then do your arithmetic. Can you come up with an example where rounding the same direction gives you a wrong answer, and rounding in the opposite direction gives you a correct answer?

Theorem 6 (Round to Nearest Rule)**.** *For double precision, if the 53rd bit to the right of the binary point is 0, then round down (truncate). If the 53rd bit is 1, then round up (add 1 to the 52nd bit), UNLESS all known bits to the right of the 1 are 0's, in which case 1 is added to the 52nd bit if and only if bit 52 is 1.*

One last note: I used single precision for easier demonstration in this handout. But the default is almost always double precision, so if you encounter any situation where the precision is not specified, assume double.