## Floating Point Numbers

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

Directly, the smallest is 0 and the largest number is $2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7$. Calculating that sum by hand is kind of painful (unless you remember your geometric partial sum formula!). Alternatively, think about the first number you can't represent: $2^8$. We can represent everything up to that, which is $2^8 - 1 = 256 - 1 = 255$. Thus we can represent 0 to 255.

The advanced student may note that this question is a bit ambiguous. What if one of the bits is used for an exponent? Doing so changes the largest number. Which is more or less the point of this handout.

Suppose you have the following bits in memory.

$$10111111010110001000000000000000$$

How many numbers is this? What are they?

*Remark.* Without knowing how long a number is, we cannot know. In fact, this could be 32 numbers, or 1 number, or any number of numbers in between. The point is: moving to a standard model for computer arithmetic requires a bit more than just going to a binary representation.

Decision number 1 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 a 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?

*Remark.* The first bit is the one that tells you the sign. In almost all systems, 0 = positive and 1 = negative. We conclude this is a negative number. There are also types of numbers which are unsigned, allowing this bit to be used for other purposes.

Then, what about a decimal point?

*Remark.* We will express our floating point numbers like in scientific notation:

$$\pm 1.m_1 m_2 \dots m_M \times 2^p.$$

We assume the first digit is 1 for all numbers where the exponent isn't 0; these are called the *normal* numbers. This gives us the ability to store just a few more numbers without taking up extra space. We'll talk more on the 0 exponent in a bit.

The part containing $m_1 \dots m_M$ is called the *mantissa*. The *exponent* $p$ will also be given in binary as $e_1 e_2 \dots e_N$. The number of bits $M$ and $N$ is fixed by the type. The order of bits is sign-exponent-mantissa. The number of bits by type is summarized in the following table.

| precision | sign | exponent (N) | mantissa (M) |
|---|---|---|---|
| single | 1 | 8 | 23 |
| double | 1 | 11 | 52 |

Another issue though: the exponent needs to be signed as well. We could designate the first bit here to also mean a sign, but we don't. Instead, the IEEE standard floating point representation shifts the exponents so that for an 8-bit exponent, an exponent of 127 actually means 0 (the shift, formally called the *exponent bias*, is $2^7 - 1 = 127$). For double precision the exponent bias is $2^{10} - 1 = 1023$. What this shifting allows is special designation for the smallest and largest exponents - the all 0 and all 1 exponents.

With all this information, and 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

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

This means in base 2 we have $-.110110001000001$, and 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 = 3

  First, translate the exponent: $10000000 = 2^7 = 128$. Then shift: 128-127 = 1. So we have $+1.10 \times 2^1 = (11.00)_2 = 3$.

- (single precision) 1 1000 0011 1111 0100 0000 0000 0000 000 = $\frac{-125}{4} = -31.25$

  First, the exponent: $10000011 = 2^7 + 2^1 + 2^0 = 131$. Shift: $131 - 127 = 4$. So we have $-1.111101 \times 2^4 = (-11111.01)_2 = -1(2^4 + 2^3 + 2^2 + 2^1 + 2^0 + 2^-2) = -31.25$.

- (double precision) 0 0111 1111 111 1110 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 = $\frac{15}{8}$

  Exponent: $01111111111 = 2^{10} - 1 = 1023$, so $1023 - 1023 = 0$. Then we have $+1.1110 \times 2^0 = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} = \frac{15}{8}$.

**Example 2.** How would you represent 10.25? 0?

*Remark.* Start by going to binary: 10.25 is $(1010.01)_2$. Then shift the decimal point: $1.01001 \times 2^3$, and shift the exponent by the bias of the precision: 3+127 = $(130)_{10}$= $(1000\ 0010)_2$. Finally, write out in bits, remembering to remove the leading one: 0 1000 0010 0100 1000 0000 0000 0000 000.

As for 0, this is where the all-0-bit exponent comes in. A number with an all 0 exponent will not have a leading 0 as the normal numbers did; such numbers are called *subnormal.* The subnormal number with all 0's in the mantissa is 0. The exponent bias for subnormal numbers is one less than the normal numbers, that is, 126 for single precision and 1022 for double precision.

**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*)

*Remark.* We begin with representing the number one: $+1.00\ldots00 \times 2^0$. To go up to the next number, we change as little as possible, which means changing the very last bit to 1. Thus the next number after one is $+1.00000000000000000000001 \times 2^0$. The difference between them is $+0.00000000000000000000001 \times 2^0 = 2^{-23}$. This is machine epsilon for single precision; similar reasoning will lead you to machine epsilon equalling $2^{-52}$ for double precision (it's $2^{-M}$). Machine epsilon is sometimes denoted $\epsilon_{mach}$.

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

*Remark.* Let the mantissa be all 0's except the last digit, 1. That digit represents $2^{-23}$. Then the smallest exponent is -126, so multiplying we get $2^{-23} \cdot 2^{-126} = 2^{-149}$.

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 for the all 1's exponent. Infinities are interpreted if the mantissa is all 0's; 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?

*Remark.* $2.5 + 2.5 = 5$. If you round both down, $2 + 2 = 4$, and if you round both down, $3 + 3 = 6$. But $2 + 3 = 5$ is correct.

Now, the following rounding rule is *deterministic*, meaning it will always round 2.5 to the same number, so in fact we would get a "wrong" answer. But with different numbers, say 1.5 and 3.5, the hope that the rounding errors will balance each other out remains.

**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 usually double precision, so if you encounter any situation where the precision is not specified, I suggest you assume double.