Note 6: Floating Points

Kevin Moy

Abstract

1 Numbers Review

Remember that N bits can represent exactly 2^N things. We can see this range accounted for in the various number representations using N bits:

- **Unsigned**: Range $[0, 2^N - 1]$
- **Two's Complement**: $[-2^{N-1}, 2^{N-1} - 1]$
- **Biased**: $[-B, 2^N - 1 - B]$

Calculating the full range (upper - lower + 1) for any of these representations will give 2^N.

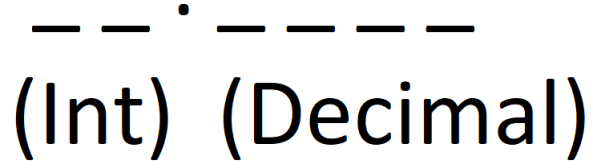
2 Fractional Representation

What if we want to represent a number like 100.25? We certainly know how to represent 100 with binary, but what about 0.25?

Well, if you think about it, we can represent pretty much any fractional component of a number with **negative** powers of 2! Notice that 0.25 is simply 2^{-2}, so we can represent it with 0.01_2.

In general, just like for decimal numbers, the **binary decimal point** separates integer and fraction. Just like for standard unsigned integers, having a **fixed** number of bits (N bits for integer and M bits for fractional) also yields

a range. For example, if $N = 2$ and $M = 4$, then we know that all numbers with our constraints can be represented by:



The maximum integer from this representation is $2^2 - 1 = 3$, and the maximum fraction from this representation is $2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} = 0.9375$. Thus the range for this fixed binary point representation is [0, 3.9375].

3 Fractional Addition + Multiplication

Fraction binary addition is exactly the same: fractional columns follow the same addition and carrying rules.

Multiplication, on the other hand, is a little trickier. In addition to the stacking and left shifting each multiplication run (like in normal addition), we must also shift the 1000.0101 such that it is aligned with the original multiplicands. It's far easier to look at an example: let's say we want to calculate 4.25 * 8.5:

```
    0100.01
x   1000.10
-----------
    0000.00
   01000.1
  000000.
  000000 .
 000000  .
010001   .
------------
01001000010
```

Now just like in standard fractional multiplication, the number of places the decimal point "jumps" is equal to the sum of the number of places after the

decimal point in the two multiplicands, which in this case is 4. So our final answer is 100100.0010, or 36.125, which is correct.

4 Floating Point

Up until now we've used *fixed* binary point numbers, i.e. ones with m fractional bits allocated. The "floating" in "floating point", however, implies we want to "float" this binary point, i.e. the point can be placed *anywhere* in the number. This will allow far better use of our (limited) bits to represent numbers.

Let's say we wanted to put 0.1640625 into binary. Without a set floating point rep, the computer would just represent this number as `....0000.00101010000...` with leading and trailing zeros to pad all available bits.

This is a massive waste. Instead, what we can do is simply store the important bits, or the **significand**, and keep track of the decimal place (2) bits to the left of the MSB of the significand. These (7) bits are the minimal number of bits we need to exactly represent 0.1640625: any other representation or solution would lose accuracy.

In floating point representation, the "location" of the binary point relative to the significand is represented by a separate number. Of course, this binary point can be outside the stored bits, such that very large AND very small numbers can be represented.

5 Scientific Notation

In decimal representation, we have a **mantissa** and a base-10 **exponent**. When this format is standardized, there is exactly one digit to the left of the decimal point. For example, consider $6.62607004 * 10^{-34}$.

In binary representation, we have the same mantissa but now a base-2 exponent as well as a binary point instead of decimal point. Computers that support binary scientific notation are called **floating point**: the binary point "floats", i.e. it isn't in a fixed spot as is for integers.

6 Floating Point Representation

Here's how we'll represent floats using our same standard 32 bits, also known as **single-precision floats**:

- Bit 31: Used for sign
- Bits 23-30 (8 bits) used for exponent.
- Bits 0-22 (23 bits) used for significand, i.e. the significant digits.

With this representation, we can quickly calculate that our representable range is $[2.0*10^{-38}, 2.0*10^{38}]$. If our number falls outside this range, overflow will occur, since the exponent will be require more than 8 bits to represent. *Underflow* is the specific term for numbers too SMALL to be represented: i.e. way too close to 0.

For normalized numbers, the significand is always some fraction between 0 and 1, as they are bits for negative-base-2.

IEEE 754 uses "biased exponent representation". Designers wanted to make the exponent field larger to be able to represent larger numbers. This is called **biased notation**, and has the same exact bit representation as single-precision floats, just now subtracting a bias from the exponent field:

- For single-precision floats (32 bits), bias = 127.
- For double-precision floats (64 bits), bias = 1023.

To convert a binary single-precision biased FP representation to decimal, we use this formula: $(-1)^{sign} * (1 + significand) * 2^{exp-127}$

6.1 Representing Infinity

How do we represent $\pm\infty$? We want to represent these numbers as the result of dividing by 0. In IEEE 754, infinity is represented by a 0 significand and maximal exponent field ($2^8 - 1$) exponent for single-precision).

6.2 Representing 0

Naturally, we represent 0 by having both the significand and exponent fields zeroed, and then the sign bit does not matter.

6.3 NaN Representation

Sometimes numbers like $\sqrt{-4}$ or $\frac{0}{0}$ simply do not have valid representations. In this case, we reserve `NaN` (for "Not a Number"). We represent `NaN` with exponent 255 and some nonzero significand.

6.4 Representing Denormalized Numbers

We know the smallest possible positive number is $1.0 * 2^{-126}$, or simply 2^{-126}. The SECOND smallest possible positive number is then $1.00...01 * 2^{-126}$, or $2^{-126}+2^{-149}$. How do we represent numbers in the 2^{-149}- sized "gap" between the smallest and second-smallest number?

In searching for a representation, notice still haven't used zeroing the EXPONENT field and having some nonzero significand. For a denormalized number with 0 as an exponent, then, we treat this as an *implicit exponent* of -126. So now, with 23 bits for our significand, we can reach the full range from 2^{-126} to 2^{-149}: we can now represent numbers like 2^{-148} and so on.

The table below summarizes floating point (biased) binary representation:

Exponent (8)	Significand (23)	Object
0	0	0
0	nonzero	denorm
1-254	anything	float
255	0	infinity
255	nonzero	NaN

7 Other FP Reps

Other floating point representations exist, such as quad-precision, oct-precision, and half-precision. All these representations utilize more bits to represent a floating point number to a higher precision. However, the IEEE 754 single-precision standard will be what's important in this class.

8 Conclusion

Floating point numbers, represented as `float` in C, are incredibly important for both representing fractional numbers as well as storing approximations of

very large AND very small numbers. The IEEE 754 FP standard is considered the universal standard: we use 1 bit for sign, 8 bits for exponent, and 23 for significand/mantissa. We also incorporate biased representation only for the exponent field to expand our range of numbers.