

Floating point representation

FP is useful for representing a number in a wide range: **very small** to **very large**. It is widely used in the scientific world. Consider, the following FP representation of a number

Exponent E significand F (also called *mantissa*)

+/-	x x x x	y y y y y y y y y y y y
------------	---------	-------------------------

In **decimal** it means (+/-) d. yyyyyyyyyyyy × 10^{xxxx}

In **binary**, it means (+/-) 1. yyyyyyyyyyyy × 2^{xxxx}

Overflow and underflow in FP

An overflow occurs when the number is **too large to fit** in the frame. An underflow occurs when the number **is too small to fit** in the given frame.

How do we represent zero?

IEEE standards committee solve this by making zero a special case: **if every bit is zero** (the sign bit being irrelevant), **then the number is considered zero.**

Then how do we represent 1.0?

It should have been 1.0×2^0 (same as 0)! The way out of this is that the interpretation of the exponent bits is not straightforward. The exponent of a single-precision float is "shift -127" encoded (biased representation), meaning that the actual exponent is (xxxxxxx minus 127). So thankfully, we can get an exponent of zero by storing 127. One consequence is that we forego something at the extreme low end of the spectrum of representable magnitudes, which should be 2^{-127} . Due to shift-127, the lowest possible exponent is actually -126 (1 - 127). It seems wise, to give up the smallest exponent instead of giving up the ability to represent 1 or zero!

More special cases

Zero is not the only "special case" of float. There are also representations for positive and negative infinity, and for a not-a-number (NaN) value, for results that do not make sense (for example, non-real numbers, or the result of an operation like infinity times zero). How do these work? A number is infinite if every bit of the exponent is set (yes, we lose another one), and is NaN if every bit of the exponent is set plus any mantissa bits are set. The sign bit still distinguishes +/-inf and +/-NaN. Here are a few sample floating point representations:

Exponent	Mantissa	Object
0	0	Zero
0	Nonzero	Denormalized number*
1-254	Anything	+/- FP number
255	0	+ / - infinity
255	Nonzero	NaN

* Any non-zero number that is smaller than the smallest normal number is a denormalized number. The production of a denormal is sometimes called gradual underflow because it allows a calculation to lose precision slowly when the result is small.

Floating Point Addition

Example using decimal

$$A = 9.999 \times 10^1, B = 1.610 \times 10^{-1}, A+B = ?$$

Step 1. Align the smaller exponent with the larger one.

$$B = 0.0161 \times 10^1 = 0.016 \times 10^1 \text{ (round off)}$$

Step 2. Add significands

$$9.999 + 0.016 = 10.015, \text{ so } A+B = 10.015 \times 10^1$$

Step 3. Normalize

$$A+B = 1.0015 \times 10^2$$

Step 4. Round off

$$A+B = 1.002 \times 10^2$$

Now, try to add 0.5 and -0.4375 in binary.

Floating Point Multiplication

Example using decimal

$$A = 1.110 \times 10^{10}, B = 9.200 \times 10^{-5} \quad A \times B = ?$$

Step 1. Exponent of $A \times B = 10 + (-5) = -5$

Step 2. Multiply significands

$$1.110 \times 9.200 = 10.212000$$

Step 3. Normalize the product

$$10.212 \times 10^{-5} = 1.0212 \times 10^{-5}$$

Step 4. Round off

$$A \times B = 1.021 \times 10^{-5}$$

Step 5. Decide the sign of $A \times B$ ($+ \times + = +$)

$$\text{So, } A \times B = + 1.021 \times 10^{-5}$$

Division

The restoring division algorithm follows the simple idea from the elementary school days. It involves subtraction and shift. This can be implemented by hardware or software. Here is the scheme:

