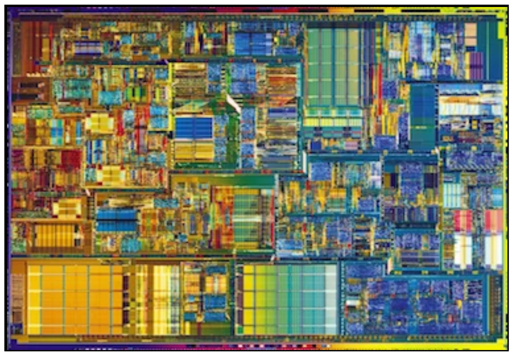
August 29, 2017

# FLOATING POINT VISUALLY EXPLAINED



While I was writing a book about Wolfenstein 3D[1], I wanted to vividly demonstrate how much of a handicap it was to work without floating points. My personal attempts at understanding floating points using canonical[2] articles[3] were met with significant resistance from my brain.

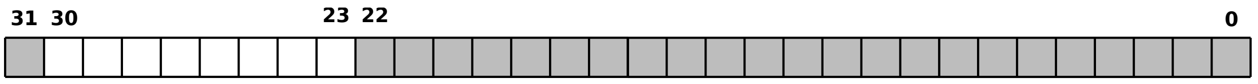
I tried to find a different way. Something far from $(-1)^S * 1.M * 2^{(E-127)}$ and its mysterious exponent/mantissa. Possibly a drawing since they seem to flow through my brain effortlessly.

I ended up with what follows and I decided to include it in the book. I am not claiming this is my invention but I have never seen floating points explained this way so far. I hope it will helps a few people like me who are a bit allergic to mathematic notations.

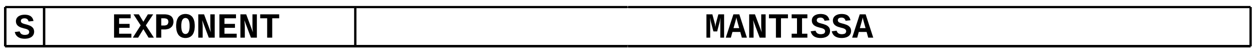
## HOW FLOATING POINT ARE USUALLY EXPLAINED

In the C language, floats are 32-bit container following the IEEE 754 standard. Their purpose is to store and allow operations on approximation of real numbers. The way I have seen them explained so far is as follow. The 32 bits are divided in three sections:

- 1 bit S for the sign
- 8 bits E for the exponent
- 23 bits for the mantissa



*Floating Point internals.*



*The three sections of a floating Point number.*

So far, so good. Now, how numbers are interpreted is usually explained with the formula:

$$(-1)^S * 1.M * 2^{(E-127)}$$

*How everybody hates floating point to be explained to them.*

This is usually where I flip the table. Maybe I am allergic to mathematic notation but something just doesn't click in my brain when I read this. It feels like learning to **draw a owl**.

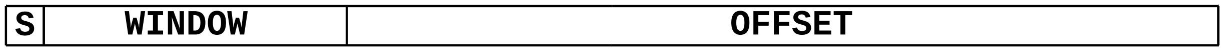
*Floating-point arithmetic is considered an esoteric subject by many people.*

*- David Goldberg*

## A DIFFERENT WAY TO EXPLAIN...

Although correct, this way of explaining floating point usually leaves some of us completely clueless. I blame this dreadful notation for discouraging legions of programmers, scaring them to the point where they never looked back to understand how floating point actually works.

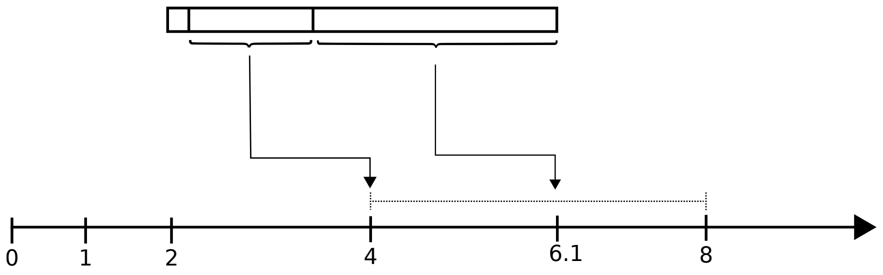
Fortunately, there is a different way to explain it. Instead of Exponent, think of a Window between two consecutive power of two integers. Instead of a Mantissa, think of an Offset within that window.



*The three sections of a floating Point number.*

The window tells within which two consecutive power-of-two the number will be: [0.5,1], [1,2], [2,4], [4,8] and so on (up to [$2^{127}$,$2^{128}$]). The offset divides the window in $2^{23} = 8388608$ buckets. With the window and the offset you can approximate a number. The window is an excellent mechanism to protect from overflowing. Once you have reached the maximum in a window (e.g [2,4]), you can "float" it right and represent the number within the next window (e.g [4,8]). It only costs a little bit of precision since the window becomes twice as large.

The next figure illustrates how the number 6.1 would be encoded. The window must start at 4 and span to next power of two, 8. The offset is about half way down the window.



*Value 6.1 approximated with floating point.*

## PRECISION

How much precision is lost when the window covers a wider range? Let's take an example with window [1,2] where the 8388608 offsets cover a range of 1 which gives a precision of $\frac{(2-1)}{8388608} = 0.00000011920929$. In the window [2048,4096] the 8388608 offsets cover a range of $(4096 - 2048) = 2048$ which gives a precision $\frac{(4096-2048)}{8388608} = 0.0002$.

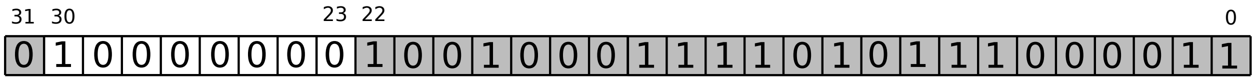
## AN OTHER EXAMPLE

Let's take an other example with the detailed calculations of the floating point representation of a number we all know well: 3.14.

- The number 3.14 is positive $\rightarrow S = 0$.
- The number 3.14 is between the power of two 2 and 4 so the floating window must start at $2^1 \rightarrow E = 128$ (see formula where window is $2^{(E-127)}$).
- Finally there are $2^{23}$ offsets available to express where 3.14 falls within the interval [2-4]. It is at $\frac{3.14-2}{4-2} = 0.57$ within the interval which makes the offset $M = 2^{23} * 0.57 = 4781507$

Which in binary translates to:

- S = 0 = 0b
- E = 128 = 10000000b
- M = 4781507 = 10010001111010111000011b

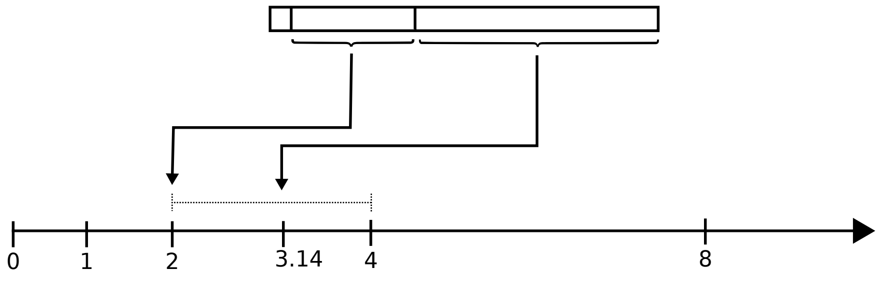


*3.14 floating point binary representation.*

The value 3.14 is therefore approximated to 3.1400001049041748046875. The corresponding value with the ugly formula:

$$3.14 = (-1)^0 * 1.57 * 2^{(128-127)}$$

And finally the graphic representation with window and offset:



*3.14 window and offset.*

I hope that helped :) !

## BLAST FROM THE PAST

Since floating point units were so slow, why did the C language end up with float and double types ? After all, the machine used to invent the language (PDP-11) did not have a floating point unit! The manufacturer (DEC) had promised to Dennis Ritchie and Ken Thompson the next model would have one. Being astronomy enthusiasts they decided to add those two types to their language.

**Trivia:** People who really wanted a hardware floating point unit in 1991 could buy one. The only people who could possibly want one back then would have been scientists (as per Intel understanding of the market). They were marketed as "Math CoProcessor". Performance were average and price was outrageous (200 USD in 1993 equivalent to 350 USD in 2016.). As a result, sales were mediocre.

## REFERENCES

^ [1] Source : **Game Engine Black Book: Wolfenstein 3D**
^ [2] Source : **Wikipedia, Floating-point arithmetic**
^ [3] Source : **What Every Computer Scientist Should Know About Floating-Point Arithmetic**

*