The End of IEEE 754

by Shade

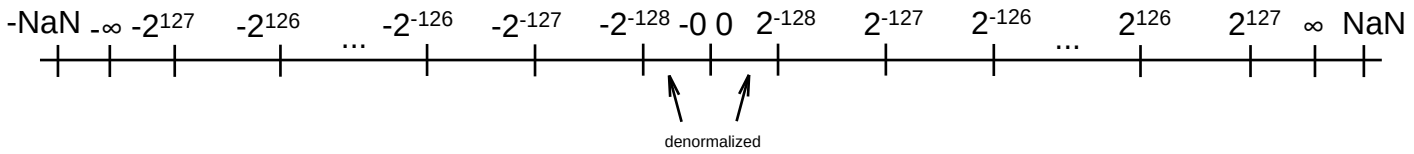
How often does the need for calculations with real numbers in a computer arise? Quite often. In fact, the role of calculations with real numbers cannot be overestimated. Almost all numerical calculations use either the integer data type or the real data type. However, compared to the well-established 2's complement integer format, the most common real number data format is IEEE 754.

But is it the best format?

To begin with, let's remember how the IEEE 754 format itself works at a basic level. Any **bit string**, for example, 32 bits long, is interpreted as:

Sign (most significant bit);
Exponent (next 8 bits);
Mantissa (remaining 23 bits).

It's easy to see that the IEEE 754 numbers are distributed **logarithmically** on the number line:



As you can see, numbers near zero cannot be distributed in the same way as the rest of the numbers. This is due to the fact that on an exponential scale, in order to "go down" to very small values, it's often necessary to "go" much longer than on a linear scale, and it is impossible to get to zero at all - for this we would need to reduce the order of magnitude to negative infinity! However, since orders of magnitude are linearly distributed, we cannot do this. In order to still reach zero, at the minimum magnitude, we will have to abandon the exponential scale and introduce denormalized numbers.

However, the exponential scale is not suitable for all purposes. For example, in games or simulations, a linear scale is often much more useful. In addition, a linear distribution can be easily converted to an exponential distribution and vice versa by using the logarithm and exponential functions, respectively! Thus, in a fixed point number system, you can get all the advantages of floating point numbers, except that addition and subtraction become much more difficult, but multiplication/division is replaced by addition/subtraction, and exponential and logarithm are respectively replaced by multiplication and division!

Moreover, in problems with a wide range of values, where the exponential scale is most suitable, the operations of addition and subtraction by themselves are very rare, if ever! Thus, the exponential scale is, in fact, **a bad option**[1].

Now a small digression. Remembering that any number stored in memory is in fact nothing more than a **bit pattern**, we come to the conclusion that the "meaning" of the bit pattern is given by the functions themselves on them. Recall also that a function is nothing more than a mapping source patterns to result patterns. Thus, bit patterns of length L and functions over it form some **isomorphism** with mathematical numbers. We expect that if we translate real numbers into bit patterns, then apply a function to them and translate the resulting bit pattern back into a real number, we will get the correct value.

However, due to the fact that the quantity of all real numbers is uncountably infinite, and the quantity of bit patterns of length L is finite and equal to 2^L, in **almost all** cases we will have **some error**.

Sorry, John Gustafson.

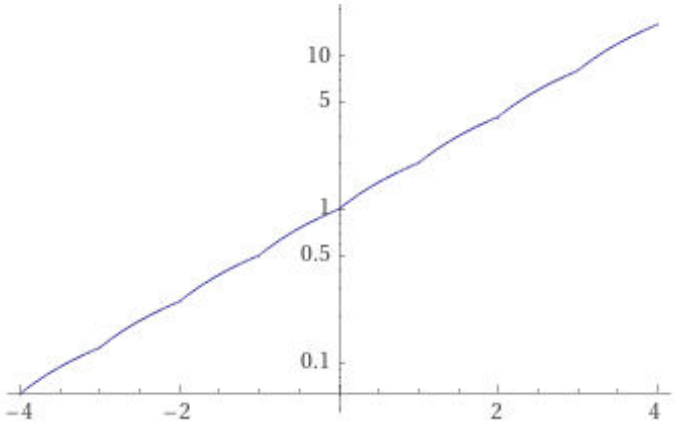
[1]Sometimes you can hear a complaint about the very fact that the errors of calculations with large orders are themselves very large (comparable to these orders), but is it legitimate to complain if the very structure of the exponential scale leads to this? (In fact, as already mentioned, for **some**, but not all, types of problems, the exponential scale fits perfectly!).

However, it is not at all important that the error is inevitable - what is much more important is how much the error grows in all conceivable cases. If the arithmetic function follows a fairly simple algorithm and in all cases the deviations from the true value approximately correspond to the distribution, then such a system can be considered suitable for practical calculations.

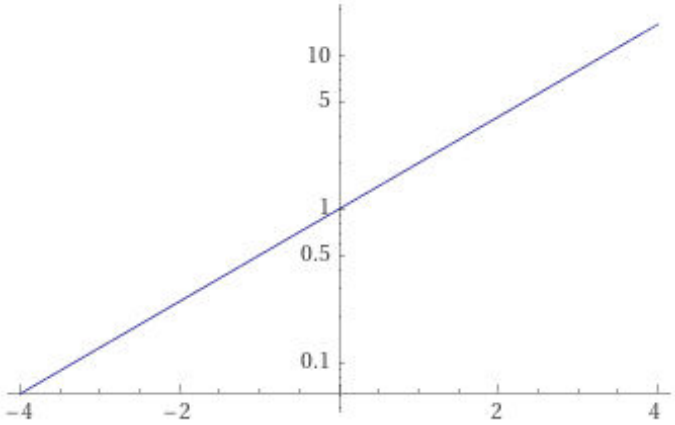
Let's pay attention to the fact that four basic operations are applicable to real numbers: addition, subtraction, multiplication and division. Subtraction and division are not necessary at all - if we can find the opposite number, we can refuse subtraction, and if we can find the reciprocal, we can refuse division. However, for the benefit of not subtracting and dividing to be significant, the negation and reciprocation operations themselves must be fairly simple!

Before looking for such a system, let's discuss in detail the concept of **mapping** numbers and bit patterns. The mapping function maps bit patterns to real numbers. It does not have to be simple - in IEEE 754, orders of magnitude are distributed exponentially, while mantissas are distributed linearly (within one order).

Here is an approximate distribution of IEEE 754 numbers (the scale of ordinate is logarithmic):



As you can see, on each segment with an interval of a power of two, due to the linear distribution of the mantissas, a certain "bump" is formed. The usual function 2^x does not form this and turns into a simple straight line:



Due to the presence of these "bumps", the reciprocation function in IEEE 754 will be quite complex. As a result, it will not be possible to refuse division.

More importantly, IEEE 754 contains features that greatly reduce its performance, such as the presence of two zeros, subnormal numbers, and NaNs. This may already be a reason to look for possible alternatives to the IEEE 754 system. In this article, I will try to sort them out.

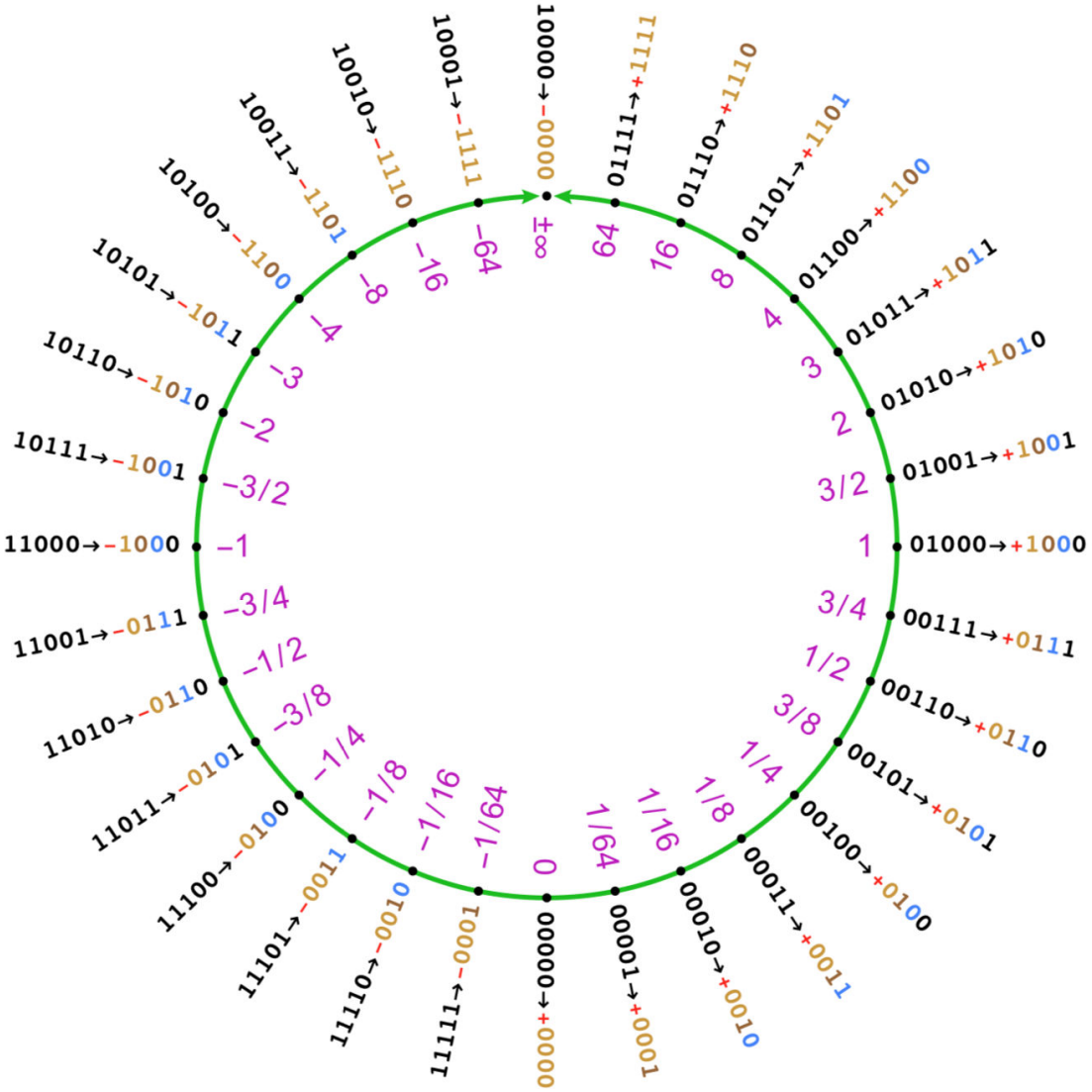
I have already noted above that algorithms involving IEEE 754 use addition and subtraction rarely, if ever, but it is interesting that zero and negative numbers are also almost never used! This feature led to the creation of a **logarithmic number system**. Although there is a sign of the number (but no zero) and addition and subtraction operations, multiplication and division are replaced by addition and subtraction, thus greatly reducing the number of calculations in areas where the logarithmic system is "natural".

Moreover, if you omit the sign, addition and subtraction, then for such a number system you don’t even need any special circuitry - adders are enough! The only non-trivial operation is taking the power and the logarithm to convert numbers to and from the logarithmic system (however, this can also be done using addition and shift).

But what about non-logarithmic mapping of numbers? The first thing that comes to mind is a simple **fixed-point** representation. Addition and subtraction at a fixed point don't require any special schemes and practically don't differ from the usual ones, multiplication and division are somewhat more complicated - you have to "watch the point", but otherwise everything is the same.

However, above I talked about the possibility of doing without the subtraction and division functions if the negation and reciprocation are simple (otherwise they just have to be added to the circuitry instead of subtraction and division). One such system, according to its author John Gustafson, is the **Posit** system. It uses a rather interesting coding system and is worth a closer look. However, if the negation in it is exact, then the reciprocation is often yield an error: for example, the reciprocal to the value 3 is 3/8, which is greater than the real result (1/3).

Can we try to improve Gustafson's system? Well, let's try this. First, note that Gustafson's main idea is to map the entire real line with some precision, like in this picture:



This uses a fairly complex coding system reminiscent of Elias coding. But is there a better approach? Let's try to find a mapping function for such a system. To begin with, we note that all possible bit patterns of length L can be considered as "rolled into a ring" natural numbers from 0 to 2^L-1. Our task is to find a mapping function that maps each bit pattern to a real number.

First, recall that in the desired system, inverting all bits and adding 1 is equivalent to negating a number, while reciprocation corresponds to inverting all **except the most significant bit** and adding 1 (or, which is the same, inverting all bits and adding 1000000...001). This brings us to four necessary properties of the desired function:

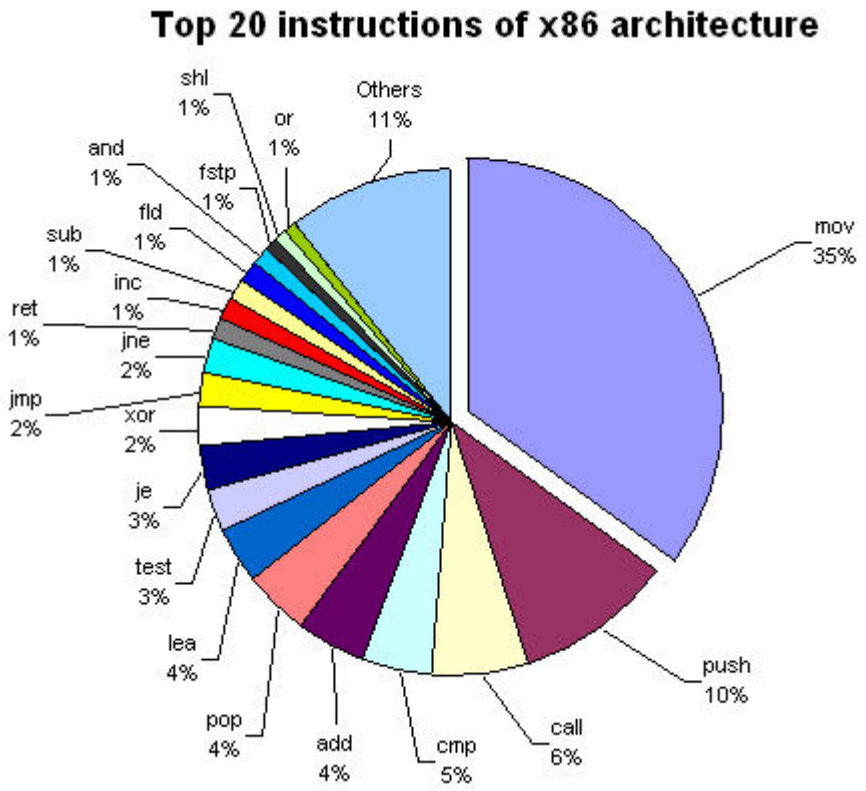
1. f(0) = 0;
2. f(x) = f(x+P) for some period P;
3. f(x) = -f(-x);
4. f(x) = 1/-f(x+P/2).

And such a function exists - if we map the bit patterns in order in the range [0;pi) to the **tangent function**, we get exactly what we need! In the resulting system, negation and reciprocation are performed exactly as described above. (In another way, the mapping method we performed is also called **stereographic projection**).

However, there is another area where all of the above systems are less suitable than the one that I will now introduce. In the representation of **angles**, floating point or logarithmic system are absolutely unsuitable, fixed point, posit or stereographic projection can in principle be used, but will give worse accuracy than **linear mapping closed on itself**. Such a system would look like a circle with a zero at the bottom and a full rotation, increasing linearly to 1, or **tau** (2pi), when measured in radians rather than revolutions (however, the 1 or tau itself would lie in the same place as zero). You can also think of it as taking the real remainder when a real number is divided by 1 (or tau).

There may be other situations for which representations of numbers are most suitable, which I have not listed. But now let's pay attention to one detail: all representations of numbers in a computer are primarily constituted by functions that work with these representations, because the bit pattern as such has no type (except, perhaps, for length). However, many complex functions can be implemented in software (for example, multiplication of integers by the "shift and add" method, trigonometric functions of fixed-point numbers by the CORDIC method, which is also an "shift and add" algorithm).

Recall also that certain functions occur more often in programs than others - if some complex functions are rare, it is quite possible to get by with simple hardware functions without a significant increase in the required time and memory. But what do the stats say? Take, for example, the statistics given by Peter Kankowski. 3 programs were analyzed: 7-Zip archiver, LAME encoder and NSIS installer:



It turned out that the MOV instruction is used, which is expected, most often.

We conditionally divide all instructions into 3 types - data manipulation, computations and control flow. Computation instructions that got on the diagram - ADD, XOR, INC, SUB, AND, SHL, OR (I also referred the shift instruction to computational ones). But, as we know, it's perfectly possible to perform subtraction with 2's complement addition, so we'll only be left with addition, left/right shift, and boolean functions (including NOT, which didn't make it onto the diagram).

Let's also refer to the article by five authors "x86-64 Instruction Usage among C/C++ Applications". Apparently, the authors did not include some instructions in their diagram, but as we can see, the differences from Kankowski's result are quite small. The most popular instruction is still MOV, and addition is even more common!

But what about complex arithmetic instructions like integer multiplication and division, floating-point arithmetic and trigonometric functions? As it turns out, complex integer instructions are extremely rare - the most common of them, multiplication of two's complement integers, occurs in only **0.13%** of cases! Other functions are even rarer: integer division in two's complement occurs in only **0.04%** of cases, unsigned operations are almost almost never found in code - unsigned multiplication and division combined are **0.02%**! Even string instructions are used **0.32%** of the time - that's over 1.5 times as common as all integer multiplications and divisions!

Floating-point instructions are somewhat more common, although also rare. As expected by me, FMUL is much more common than FADD - **0.71%** vs. **0.27%** - more than 2.5 times more common! But even they, taken together, **don't even reach 1%** of all instructions.

Of course, counting instructions in code doesn't tell the whole story - for a more accurate analysis, one should count the use of instructions during program execution, which I currently can't do. However, already such a primitive analysis shows that ordinary integer addition occurs an order of magnitude more often than any other instructions on numbers combined.

Thus, **addition instruction is enough**.

To summarize:

Different methods of encoding numbers are suitable for different purposes.

IEEE 754 **is a bad system**.

For purposes such as representing coordinates, it is much better to use **fixed point** numbers.

Where a logarithmic scale is needed, the **logarithmic number system** is much better than IEEE 754.

There are interesting and little studied methods for representing real numbers, such as Gustafson's **Posit** and the **stereographic projection** method using the tangent function.

To represent **angles**, the most suitable method is to take the real remainder of division by 1, where [0;1) are the values that the angle can take.

All functions for working with these systems can be replaced by algorithms using addition and shift. Thus, **addition, logical operations and bit manipulations are enough**.

References:

1. John L. Gustafson, Isaac Yonemoto, 2017. Beating Floating Point at its Own Game: Posit Arithmetic
2. Peter Kankowski, 2008. x86 Machine Code Statistics
3. Amogh Akshintala, 2019. x86-64 Instruction Usage among C/C++ Applications