NaN gates and flip FLOPS

Dr. Tom Murphy VII Ph.D.*

1 April 2019

Abstract

Yes, this paper contains many layers of abstraction.

Introduction

Mathematics is fundamental to computer science, and the foundation of mathematics is the real numbers; this is obvious from the name. One of computing's dirtiest secrets, however, is that computers themselves are not based on real numbers—rather, they are based on so-called "ones" and "zeroes" combined with "logic gates" simulated with transistors. While this suffices for most practical purposes, it is unsatisfying from a theoretical perspective.

Recently, some progress has been made by human geniuses on completely replacing integer calculations with calculations on real numbers[4]. While this removes many of the hacks present in modern software, there are still many components of the computer (e.g. RAM, registers, the *scroll lock* LED, a tiny USB-powered fan that can cool you on hot summer days or during particularly strenuous programming sessions) that are not integer-based, and thus cannot be replaced with real numbers via this techique.

In this paper I give a new foundation for computing based solely on real numbers. I begin with a brief reminder of the definition of real numbers, although the reader is expected to be familiar as these are pretty fundamental to everything. The approach of the paper is then to identify a pair of real numbers that have nice properties (Section 1.1), and then to give mathematical operations on these numbers that parallel the logical operations typically used in the construction of computers (Section 1.3). I then discuss how these operations can be implemented efficiently (Section 3). I conclude with some wild speculation.

1 Real numbers

The real numbers are described by IEEE 754, most recently revised in AD 2008[1]. Every real number has a sign, a mantissa, and an exponent. Actually, this understates the elegance of real numbers, since there are a number of numbers, such as NaN ("not a number") which are not of this form; NaN nas no sign nor mantissa nor exponent. We also have inf and −inf, which do have a sign, but no mantissa nor exponent. These are the infinite numbers that you get if you count very high or very low. Excitingly, we also have both positive and negative versions of 0. Some numbers have multiple representations, and almost all numbers cannot be represented at all.

The real numbers have an equality operation ==. This operation has some very exciting properties which are unusual for an equivalence relation: It is not reflexive (NaN == NaN is false), and does not obey substitution (for $+0$ == -0 is true, but $1/+0$ == $1/-0$ is false).

As a result, real numbers are an absolute joy to work with.

1.1 Choosing some distinguished values

Computing will need at least two different values. We could choose 0.0 and 1.0 as in "binary," but these numbers are extremely arbitrary; why not 1.0 and 2.0? e and $\pi/2$? These numbers are easily confused with one another. It seems better to use distinguished values, making the resulting mathematics more distinguished. One of the most distinguished numbers is NaN (Figure 1). One nice thing about using the number NaN is that it is not comparable to other numbers, e.g. both NaN $<$ 0.0 and 0.0 $<$ NaN are false. Does it really make sense for our fundamental particles to be ordered (e.g. $0 < 1$)? The lack of symmetry is abhorrent.

Figure 1: A distinguished gentleNaN.

The two numbers we choose need to be different; alas they cannot both be NaN, since although NaN is different from NaN (NaN != NaN), it is not possible to tell them apart (except that NaN actually has multiple binary representations—see

Section 2). Another great choice is +inf or −inf. We choose to use +inf in order to break symmetry, and because it will make our scientific contribution more positive.

1.2 IEEEuler's Identity

Moreover, NaN and inf are part of the pantheon of special values, exhibiting exquisite properties, such as IEEEuler's identity:

$$e^{i\pi} + 1^{\mathsf{NaN}\times\mathsf{inf}} = 0$$

because 1^n is 1[1], even for $n = \mathsf{NaN}$.[2] Another nice pair of properties ties these fundamental constants together a different way:

$$\left(e^{i\pi}\right)^{-\mathsf{inf}} = \mathtt{compound}(\mathsf{NaN}, 0)$$

$\mathtt{compound}(x, n)$ is the "compound interest" function $(1+x)^n$, defined in the IEEE 754 standard, but only available in C via floating point extensions [2]. This function is 1 for $n = 0$ and $x = \mathsf{NaN}$.[3] More excitingly, we have $e^{i\pi} = -1$ (Euler) and $-1^{-\mathsf{inf}} = 1$ "because all large positive floating-point values are even integers." [3]

1.3 NaN's Not GNU

People who work with real numbers are often taught that the number NaN is propagated through all expressions that use it (e.g. $\mathsf{NaN} - 1 = \mathsf{NaN}$), like some kind of GNU Public Licensed number. This is a misconception. We already saw in the beautiful identities above that some expressions involving NaN do not result in NaN, like $1^{\mathsf{NaN}} = 1$ and $\mathtt{compound}(\mathsf{NaN}, 0) = 0$. But it is also the case that $1^{\mathsf{inf}} = 1$ and $\mathtt{compound}(\mathsf{inf}, 0) = 0$. Are there mathematical functions that distinguish between NaN and inf?

It turns out that there are! For example, the functions `minNum` and `maxNum` ([*IEEE 754-2008*, 5.3.1, p19]) take two arguments and return the min and max, respectively. They have the special, distinguished property that "if exactly one argument is NaN, they return the other. If both are NaN they return NaN."

With functions such as these, we can begin constructing the building blocks of more interesting functions (Figure 2). Unfortunately, $\mathtt{maxNum}(a, b)$ and $a * b$ are not complete on their own; we additionally need at least a function $f(x)$ where

$\mathtt{maxNum}(a,b)$	NaN	inf
NaN	NaN	inf
inf	inf	inf

$a * b$	NaN	inf
NaN	NaN	NaN
inf	NaN	inf

Figure 2: The behavior of some mathematical functions on our distinguished values NaN and inf. `maxNum` returns inf if either of its arguments is inf (some other functions have this property, like `hypot`). $a * b$ is inf only if both of its arguments are inf (there are many other examples, like $a + b$).

$f(\mathsf{NaN}) = \mathsf{inf}$ and $f(\mathsf{inf}) = \mathsf{NaN}$. Does such a function exist? Yes! Several can be built from IEEE 754 primitives:

$$\begin{aligned} f(x) &= \mathtt{minNum}(-x, -1.0) + \mathsf{inf} \\ f(x) &= \mathtt{hypot}(\mathsf{NaN}, \mathtt{maxNum}(1/x, -\mathsf{inf})) \\ f(x) &= \mathsf{inf} - \mathtt{maxNum}(x, 1.0) \\ f(x) &= \mathtt{sqrt}(\mathtt{copysign}(\mathsf{inf}, -x)) \end{aligned}$$

You can try these out in your favorite programming language, and if they don't work, your implementation is not IEEE 754 compliant. Why do these work? Let's take the first one, and compare NaN and inf for x:

$x = \mathsf{NaN}$	$x = \mathsf{inf}$
$\mathtt{minNum}(-x, -1.0) + \mathsf{inf}$	$\mathtt{minNum}(-x, -1.0) + \mathsf{inf}$
$\mathtt{minNum}(-\mathsf{NaN}, -1.0) + \mathsf{inf}$	$\mathtt{minNum}(-\mathsf{inf}, -1.0) + \mathsf{inf}$
$-1.0 + \mathsf{inf}$	$-\mathsf{inf} + \mathsf{inf}$
inf	NaN

Thinking of NaN as `false` and inf as `true`, we now have `AND` (`maxNum`), `OR` ($*$), and `NOT` ($\mathtt{minNum}(-x, -1.0)$). With these we can create arbitrary functions $f(a_1, a_2, \ldots, a_n)$ that return our choice of NaN or inf for the 2^n different combinations of arguments. It is also possible to find more direct expressions that compute simple functions (Figure 3).

$\mathsf{inf} - \mathtt{maxNum}(a+b, -\mathsf{inf})$	NaN	inf
NaN	inf	inf
inf	inf	NaN

$\mathtt{abs}(\mathtt{minNum}(b, -a) + \mathtt{maxNum}(b, -\mathsf{inf}))$	NaN	inf
NaN	NaN	inf
inf	inf	NaN

$-\mathsf{inf}/\mathtt{maxNum}(b, \mathtt{maxNum}(a, -1))$	NaN	inf
NaN	inf	NaN
inf	NaN	NaN

Figure 3: Some interesting functions of two variables. They are isomorphic to the boolean functions `NAND`, `XOR` and `NOR` respectively, but more beautiful.

I found these functions through computer search,[4] using a

[1] This paper uses both exponents and footnotes extensively; please be careful of the difference.

[2] [*IEEE 754-2008*, 9.2.1, p44]

[3] [*IEEE 754-2008*, 9.2.1, p44]

[4] Source code is available at `https://sourceforge.net/p/tom7misc/svn/HEAD/tree/trunk/nand/`

parameter	**binary4**	binary16	binary32	binary64	binary128	binary$_k$
k, storage in bits	**4**	16	32	64	128	multiple of 32
p, precision in bits	**2**	11	24	53	113	k - round(4 * $\log_2$(k)) + 13
$emax$, maximum exponent e	**1**	15	127	1023	16383	2^{k-p-1} - 1
$bias = E$ - e	**1**	15	127	1023	16383	emax
signbits	**1**	1	1	1	1	1
w, exponent width	**2**	5	8	11	15	round(4 * $\log_2$(k)) - 13
t, trailing significand width	**1**	10	23	52	112	k - w - 1
k, storage width	**4**	16	32	64	128	1 + w + t

Figure 4: Parameters for the newly-introduced **binary4** encoding for IEEE 754, compared to the standard widths (see Table 3.5 in the standard[1]).

technique like bottom-up logic programming [5]. I start with a small set of constants, including arguments a and b, and then compute all of the expressions that can be made by applying a single mathematical function (e.g. abs(x), $-x$) or binary mathematical function ($x + y$, x/y, maxNum(x, y)) to existing expressions. The expression is actually a collection of values taken on for each possible substitution (in {NaN, inf}) to arguments a and b (i.e., it represents a function). If the expression has the correct values for each possible assignment to the arguments, then we are done. We only need to keep one (the smallest) expression that represents a distinct function, but note that we have to consider intermediate expressions that compute values other than NaN and inf. Also note that we need one of minNum, maxNum or copySign in order to compute the NOT function; we could think of these functions as therefore essential to mathematical completeness.

Particularly nice is inf $-$ maxNum$(a + b, -$inf$)$, which returns inf if either of its arguments is NaN. We will call this the "NAN gate", for "Not NaN". The NAN gate is exciting because it can be used on its own to construct all other boolean functions! We can use NaN, inf, and this function to construct any computer and any computable function. Beautiful!

To program with numbers on computers, the real numbers are represented as strings of bits. Next we'll talk about efficient representations that allow us to manipulate NaN and inf with NAN gates.

2 The binary4 representation

IEEE 754 natively defines several bit widths for floating-point values, such as the 32-bit binary32 (aka "single-precision float") and 64-bit binary64 (aka "double-precision float"). The specification is parameterized to allow other bit widths; for example, half-precision 16-bit floats are common in GPU code for machine learning applications [7]. Smaller floats sacrifice precision, but require less space and allow faster calculations. For our purposes in this paper, since we only need to represent the two numbers NaN and inf, we are interested in the smallest possible representation.

This section describes the binary4 representation, a four-bit floating point number that is clearly allowed by the IEEE 754 standard.

The representation of any floating-point number has a single sign bit, some number w of exponent bits, and some number t of mantissa bits. For binary32, $w = 8$ and $t = 23$; and with the sign bit we have $23 + 8 + 1 = 32$ bits as expected. We need at least a sign bit, but what are the smallest permissible values of w and t?

$s\ E\ T$	value
0 0 0 0	$+0$
0 0 0 1	subnormal: $2^0 * 2^{1-2} * 1 = 1 * 1/2 * 1 = 0.5$
0 0 1 0	normal: $2^0 * (1 + 1/2 * 0) = 1$
0 0 1 1	$2^0 * (1 + 1/2 * 1) = 1.5$
0 1 0 0	$2^1 * (1 + 1/2 * 0) = 2$
0 1 0 1	$2^1 * (1 + 1/2 * 1) = 3$
0 1 1 0	$+$inf
0 1 1 1	NaN
1 0 0 0	-0
1 0 0 1	-0.5
1 0 1 0	-1
1 0 1 1	-1.5
1 1 0 0	-2
1 1 0 1	-3
1 1 1 0	$-$inf
1 1 1 1	NaN

Figure 5: All 16 values representable in binary4 floating-point. The format works reasonably well even at this very low precision, although note how many of the values are not finite.

The most stringent constraint on w comes in [*IEEE 754-2008*, 3.4, p9], which states

> The range of the encoding's biased exponent E shall include:
>
> — every integer between 1 and $2^w - 2$, inclusive, to encode normal numbers
>
> — the reserved value 0 to encode ± 0 and subnormal numbers
>
> — the reserved value $2^w - 1$ to encode $\pm\infty$ and NaNs.

E is the binary number encoded by w. It must include at least the two special values consisting of all zeroes and all ones (second and third clause). A conservative reading of "every integer between 1 and $2^w - 2$" seems to require that $1 \leq 2^w - 2$ (otherwise how could the interval be inclusive of its endpoints?), which would imply that w is at least 2. (However, see Section 2.1 for the hypothesized case where $w = 1$.)

The representation of NaN and inf are distinguished by the value of t when E is all ones. We certainly need to distinguish

these, so $t = 1$ is the minimal size.

We have one sign bit, two exponent bits, and one mantissa bit, for a total of four. Since "single precision" is 32 bits, "half precision" is 16, 4 bits is "eighth precision." Given how nicely all this works out, shouldn't there be an `eighth` base type in most modern programming languages and GPUs? Since there are so few values representable, it would be practical for all the standard operations to be done in constant time via table lookups. All 16 possible values are given in Figure 4.

Four bits is not many, but is it possible to represent these two values more efficiently?

2.1 The hypothesized binary3 format

$s\,E\,T$	value
0 0 0	$+0$
0 0 1	subnormal: $2^1 * 2^{1-2} * 1 = 2 * 1/2 * 1 = 1$
0 1 0	+inf
0 1 1	NaN
1 0 0	-0
1 0 1	-1
1 1 0	$-$inf
1 1 1	NaN

Figure 6: All 8 values of the hypothetical binary3 representation. There are no normal values; the only finite values are the positive and negative zero and a single subnormal which denotes 1 (or -1).

The IEEE 754 representation clearly requires a sign bit, and for this purpose we need at least one bit for the mantissa in order to distinguish NaN and inf. It is perhaps a stretch of the wording, but arguably the spec permits a 1-bit exponent ($w = 1$). To rationalize this we need to interpret the phrase "every integer between 1 and $2^w - 2$ inclusive" (that is, between 1 and 0 inclusive) as denoting the empty set. This seems reasonable.

With one bit for sign, exponent, and mantissa, we can represent just 8 different values. Here *emax* is 0, and the standard clearly requires $emin = 1 - emax$, so $emin = 1$. Certainly fishy for *emin* to be larger than *emax*, but we can just not stress out about it; the representable values are all reasonable-looking (Figure 6).

3 A hardware math accelerator

So now we know that we can build arbitrary computers with the `NAN` gate, representing the interconnects between the gates efficiently with binary3-coded real numbers. All that remains is an efficient implementation of the `NAN` gate itself. We could emulate such a thing in software, but software is much slower than hardware; we would also like to maximize the number of times that we can flip between states of the gate (the flip FLOPS) per second.

Fortunately, there are several pieces of hardware that implement IEEE 754 real numbers. I found a moderately-priced micrprocessor ($6.48/ea.), the STM32F303RDT6. This is a 32-bit ARM Cortex M4F processor with hardware floating-point running at 72MHz [6]. In the rather-difficult-to-solder

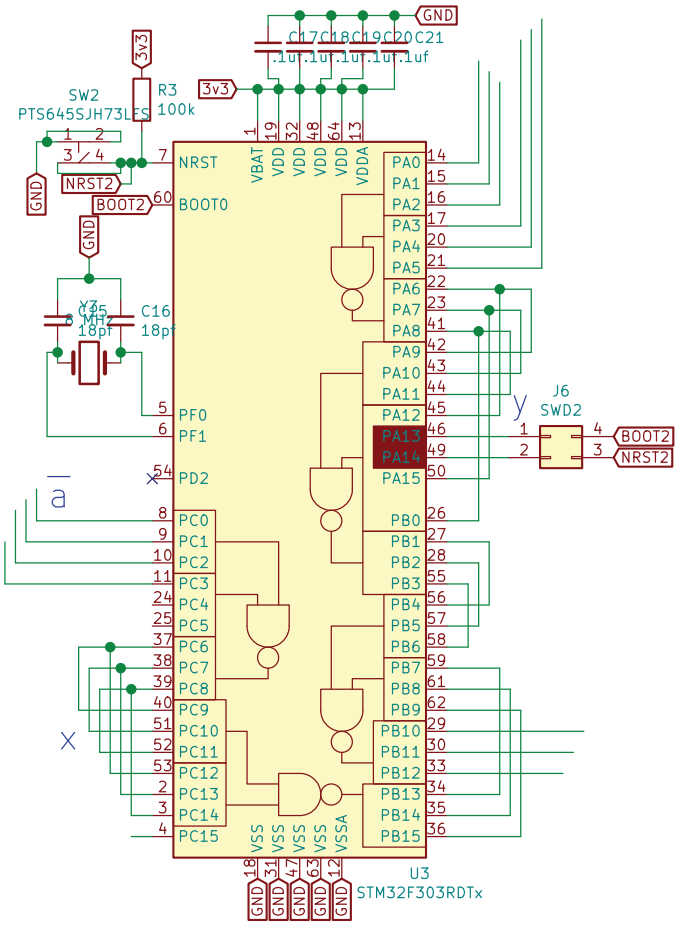

Figure 7: The STM32F303RDT6 wired up as 5 `NAN` gates, shown here *in situ*. This is a portion of a larger schematic. Also show is some support hardware needed for each microprocessor: A programming header, 5 filter capacitors, a crystal oscillator circuit, and a reset switch with external pull-up.

10mm surface-mount LQFP64 package, it has 64 pins, 51 of which can be used for general-purpose IO. Since a `NAN` gate using the binary3 representation needs 9 pins (3×2 for the inputs, 3 for the outputs), it is feasible to implement five `NAN` gates on the same chip with a few pins left over for jiggery pokery (Figure 7).

The hardware math accelerator itself can be thought of as a floating point unit (FPU), but one that is streamlined to run only a single instruction, the universal function inf$-$`maxNum`$(a+b, -$inf$)$. This is a function taking two binary3 real numbers and outputting a single binary3 number. Since there are only $2^6 = 64$ possible inputs, it can be straightforwardly implemented with table lookup, but this would require dozens of microprocessors, which might exceed our power budget. In fact there is significant structure to the function; for one thing, it can only return NaN or inf (even if arguments like -1.0 or 0.0 are given), and the binary3 representation of these only differ in one bit. Equivalent logic to determine that bit is as follows:

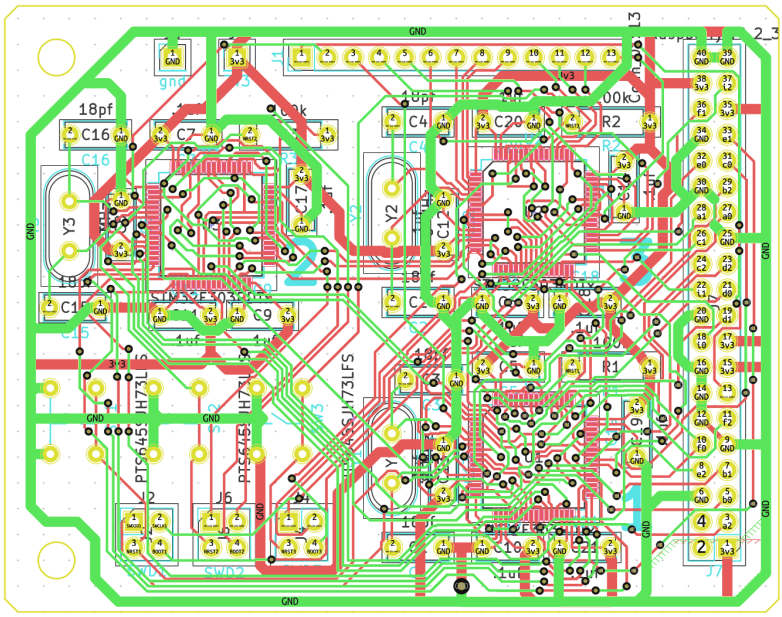

Figure 8: A beautiful hand-routed circuit board implementing a universal math accelerator, using only the universal `NAN` gate implemented with native floating point hardware.

```
if isfinite(a) && isfinite(b)
then
  // inf - (a + b) = inf
  0
else
  // a + b is nan when a is nan, b is nan, or a and b are
  // infinites with different signs. If they are both
  // -inf, then we have max(-inf, -inf) anyway, which is
  // the same as max(nan, -inf). So we have:
  if a == inf && b == inf
    // both positive infinities
    // inf - inf = nan
    1
  else
    // inf - -inf = inf
    0
```

So ultimately this function only returns 1 in the case that both inputs are exactly +inf, the pattern `0 1 0`.

If the inputs are a0 a1 a2, b0, b1, b2, and outputs are c0, c1, c2, then:

```
c0 = 0
c1 = 1
c2 = !a0 && !a2 && !b0 && !b2 && a1 && b1
```

So we can hardwire the outputs c0 and c1, and use the microprocessor-based `NAN` gates to compute c2 as a small boolean function.

Of course, each 0 or 1 above is actually itself a binary3-coded NaN or inf. Thus on the physical circuit board, this math accelerator has $2 \times 3 \times 3$ input pins and $1 \times 3 \times 3$ output pins. This is just shy of the total number of IO pins on the Raspberry Pi, so we use such a computer to drive the math accelerator. Given the large number of traces and small footprint of the microprocessors, routing the board gets somewhat involved (Figure 8).

As of the SIGBOVIK 2019 deadline, such a circuit board has been manufactured in China and is in possession of the author (actually the minimum order quantity of 10), but he is somewhat nervous about his ability to hand-solder these 0.1mm surface-mount leads, so we'll see how it goes! Please see `http://tom7.org/nan` for project updates or an embarrassing `404 Not Found` if I fail to reboot computing using the beautiful foundation of real numbers.

References

[1] 754–2008 IEEE standard for floating-point arithmetic. Technical Report 754–2008, IEEE Computer Society, August 2008.

[2] Floating-point extensions for C—part 4: Supplementary functions. Technical Report TS 18661-4:2015, ISO/IEC, 2015.

[3] JTC1-SC22-WG14. Rationale for international standard—programming languages—C. Technical Report Revision 5.10, ISO/IEC 9899, April 2003. `http://www.open-std.org/jtc1/sc22/wg14/www/C99RationaleV5.10.pdf`.

[4] Jim McCann and Tom Murphy, VII. The fluint8 software integer library. In *A Record of the Proceedings of SIGBOVIK 2018*, pages 125–128. ACH, April 2018. `sigbovik.org/2018`.

[5] Frank Pfenning. Bottom-up logic programming, November 2006. Course notes for 15–819K: Logic Programming.

[6] STMicroelectronics. STM32F303xD STM32F303xE. ARM®Cortex®-M4 32b MCU+FPU, up to 512kb flash, 80kb SRAM, FSMC, 4 ADCs, 2 DAC ch., 7 comp, 4 op-amp, 2.0–3.6 V, October 2016. Revision 5.

[7] Wikipedia. Half-precision floating-point format, 2019. `https://en.wikipedia.org/wiki/Half-precision_floating-point_format`.