NaN-Gate Synthesis and Hardware Deceleration

Cassie Jones
Witch of Light
list+sigbovik@witchoflight.com

1 April 2020

Abstract

In recent years there has been interest in the field of "hardware decelerators," which serve primarily to make computation more interesting rather than more efficient. This builds off the work of "NaN-Gates and Flip-FLOPS" [9] to provide a hardware synthesis implementation of real-number computational logic using the Yosys Open Synthesis Suite [1] framework, and evaluates the impacts of different floating point formats.

ACH Reference Format:
Cassie Jones. 2020. NaN-Gate Synthesis and Hardware Deceleration. In *Proceedings of SIGBOVIK 2020*, Pittsburgh, PA, USA, April 1, 2020. (SIGBOVIK '20, ACH).

Introduction

Harware decelerators work on the principle of "stop and smell the roses." There are some qualities that are more important than sheer efficiency, and often these improvements can often only be realized by taking the computer and slowing it down to a more leisurely pace. The largest advancements in the field happen in the emulation space, since it's the most widely accessible. It may be most familiar in the form of video-game computers, building computers out of redstone in Minecraft, Factorio combinators, or the like [7] [6].

> "But of course speed is not what we're after here. We're after just, beautiful computation going on inside the heart of this machine." — Tom7 [10]

The SIGBOVIK 2019 paper "NaN-Gates and Flip-FLOPS" decelerates computers in the name of elegance: it throws away the assumption of binary computers and builds ones based on real numbers, specifically IEEE-754 floating point numbers. It aims towards "reboot computing using the beautiful foundation of real numbers," but it still leaves us with room for improvement in a few areas. It leaves the logic gates in the domain of emulation, which limits the types of hardware that are easy to build, and it limits the elegance that can be achieved. Since it uses an existing CPU as the floating point processor, it's still left with a computer that's based on binary emulating the real number logic.

Here, we attempt to remove this limitation by bringing NaN-gate computation to the domain of native hardware, via a custom Yosys synthesis pass.

1 NaN-Gate Synthesis

The Yosys Open SYnthesis Suite [1] is a free and open source archicture-neutral logic synthesis framework. It can synthesize Verilog into a variety of backend formats using a flexible and pluggable architecture of passes. The Yosys manual has a chapter on how to write extensions [2, Ch. 6], which can be consulted for documentation and examples on how Yosys passes are built. We provide a Yosys extension which synthesizes a circuit down to a network of small floating point units implementing the NaN-gate function. This can be further synthesized to a final target, like a specific FPGA architecture.

1.1 Yosys Synthesis

We will demonstrate all synthesis with the following toggle module, since it's small enough for all stages of synthesis results to be understandable and fit neatly on the page.

```
module toggle(input clk, input en, output out);

always @(posedge clk) begin
    if (en) out <= ~out;
end

endmodule
```

Yosys will take a Verilog module like this and flatten the procedural blocks into circuits with memory elements. Running sythesis gives us a circuit with a flip-flop, a toggle, and a multiplexer that's driven by the enable line.

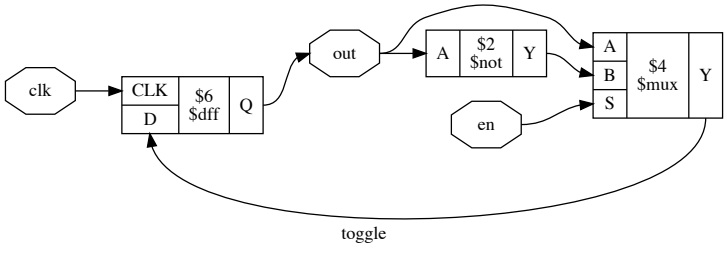

Figure 1: The synthesized toggle circuit.

We can also ask yosys to synthesize this to exclusively NAND and NOT gates with a small synthesis script.

```
read_verilog toggle.v
synth
abc -g NAND
```

This particular design synthesizes to 1 D flip-flop, 3 NAND gates, and 2 NOT gates.

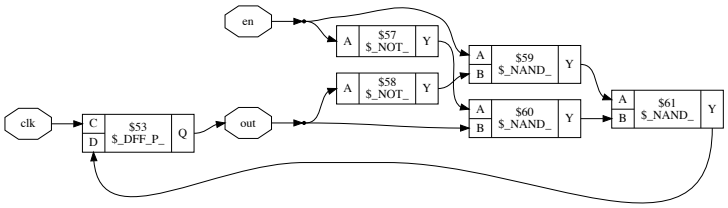

Figure 2: The circuit synthesized down to only NAND, NOT, and memory elements.

1.2 The Yosys `synth_nan` Pass

Our `synth_nan` pass is implemented as a Yosys extension. For convenience, we'll describe the behavior in terms of the 3-bit float synthesis. It converts a module to NaN-gate logic. It summarizes itself as running:

```
synth
abc -g NAND
nand_to_nan
share_nan
dff_nan
simplify_nan
clean
techmap_nan
```

The first two steps there are standard synthesis. The `synth` pass will convert a module into coarsly optimized circuits, and `abc -g NAND` will remap the entire thing into optimized NAND logic.

One complexity we have to deal with is external interfaces. Despite the wonderful realms of pure real-number computation we want to interact with, when interacting with fixed hardware component interfaces, we have to convert between flots and bits. In order to handle this, the NaN gate tech library has modules like `fp3_to_bit` and `bit_to_fp3` which perform this boundary conversion. In order to deal with the chaotic diversity of real circuits, for robustness, the `nand_to_nan` pass converts each NAND gate to a `bit_to_fp3 -> NaN -> fp3_to_bit` chain. Don't worry, these conversions will later be removed everywhere they don't turn out to be necessary.

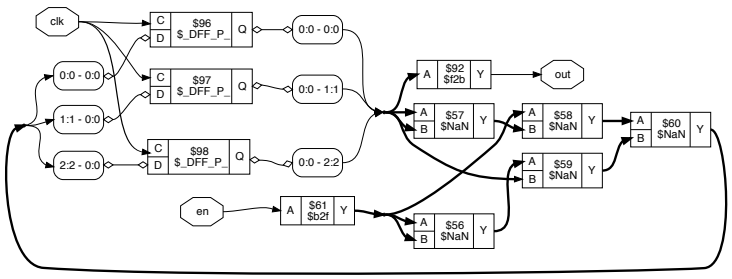

Figure 3: The toggle circuits synthesized to NaN gates. Note that the external logic ports have floating point conversion modules, but the clock line doesn't.

The `share_nan` pass reduces the number of conversions by sharing ones that have the same inputs. Then, the `dff_nan` pass can expand the flip-flops in the circuit into a set of enough flip-flops to store the floating point values.

The `simplify_nan` pass converts any instance of `fp3_to_bit -> bit_to_fp3` to just a wire that passes the floats straight through.

We do `clean` to remove dead wires and useless buffers, and then finally the `techmap_nan` pass replaces the opaque NaN-gate modules with actual modules so that further synthesis can properly make them realizable on real hardware.

1.3 Module Ports

If you want your circuit to support external floating-point based interfaces, you can use the floating point conversion modules yourself.

```
module toggle(
    input clk, input [2:0] en, output [2:0] out);

wire en_b;
reg out_b;
fp3_to_bit en_cvt(en, en_b);
bit_to_fp3 out_cvt(out_b, out);
always @(posedge clk) begin
    if (en_b) out_b = ~out_b;
end

endmodule
```

The NaN synthesis will end up erasing the floating point conversions on either side of the interface since they connect to floating point units. Future work could include automatically expanding ports using something like a `(* nan_port *)` attribute.

2 Floating Point Formats

While tom7's work asserts that a **binary4** floating point format is "clearly allowed by the IEEE-754 standard," this doesn't seem to hold up under a close examination. Brought to my attention by Steve Canon [8], there are two cases where these floating point formats fall down. First, and most importantly in the case of **binary4**, you need to encode both quiet- and signaling-NaNs. Section 3.3 of IEEE-754 says [5]:

> "Within each format, the following floating-point data shall be represented: [...] Two NaNs, qNaN (quiet) and sNaN (signaling)."

While **binary4** does have two separate NaN values (a positive and a negative), they are distinguished only by their sign bit, which isn't allowed to distinguish the two types of NaNs, as we can see in 6.2.1:

> "When encoded, all NaNs have a sign bit and a pattern of bits necessary to identify the encoding as a NaN and which determines its kind (sNaN vs. qNaN)."

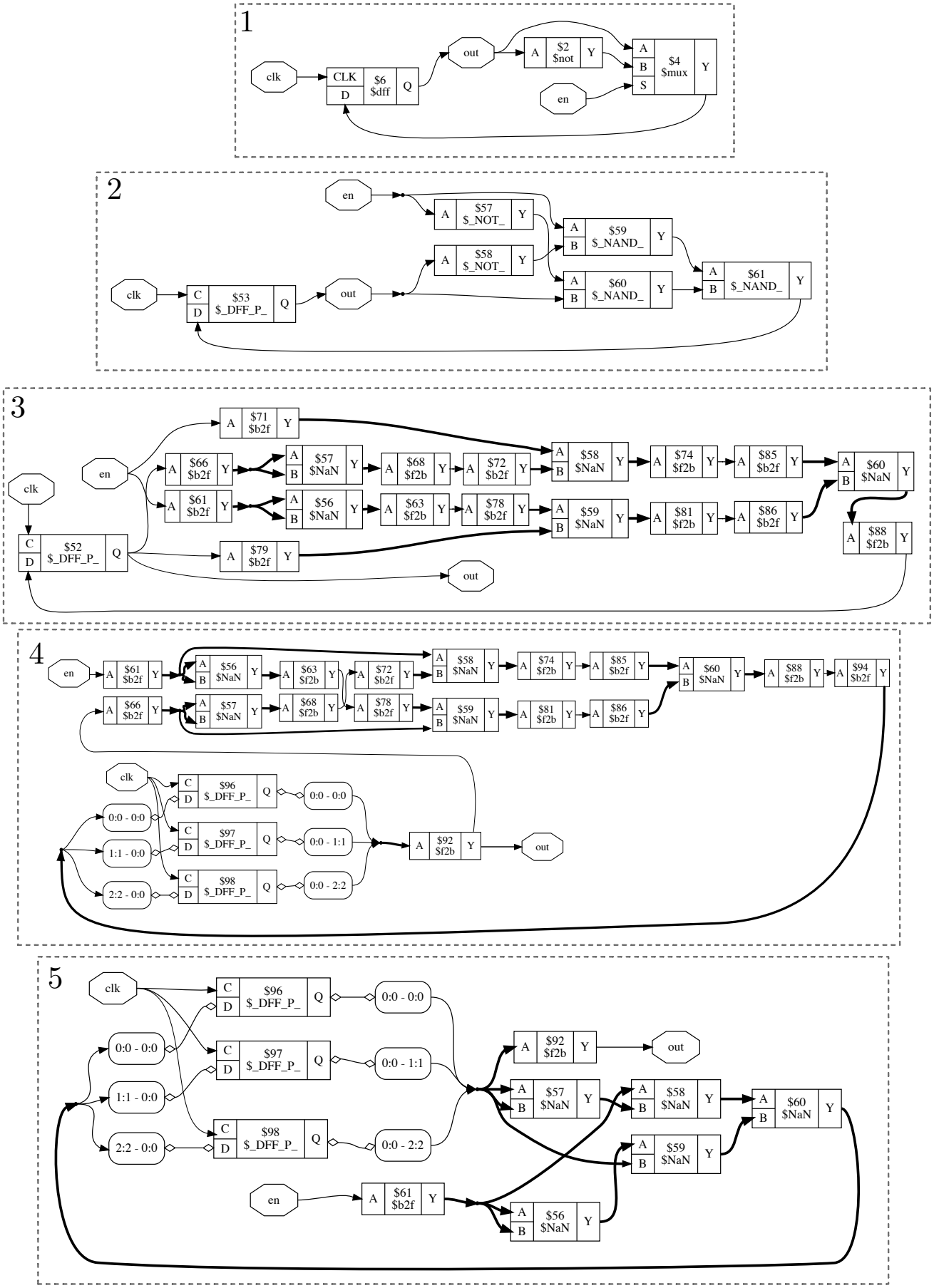

Figure 4: The full NaN-gate synthesis process for the toggle module. In step 1 we have the logical circuit after coarse synthesis. In step 2 it's synthesized to NAND and NOT gates. Step 3 converts the gates to NaN gates and adds conversion chains. Step 4 expands the flip-flops to store floats. Step 5 collapses redundant conversion chains to give the final NaN-synthesized module.

This means that we need at least two bits in the mantissa in order to represent the infinities (stored as a 0 mantissa with the maximum exponent) and the NaN values (stored with two distinct non-zero mantissas).

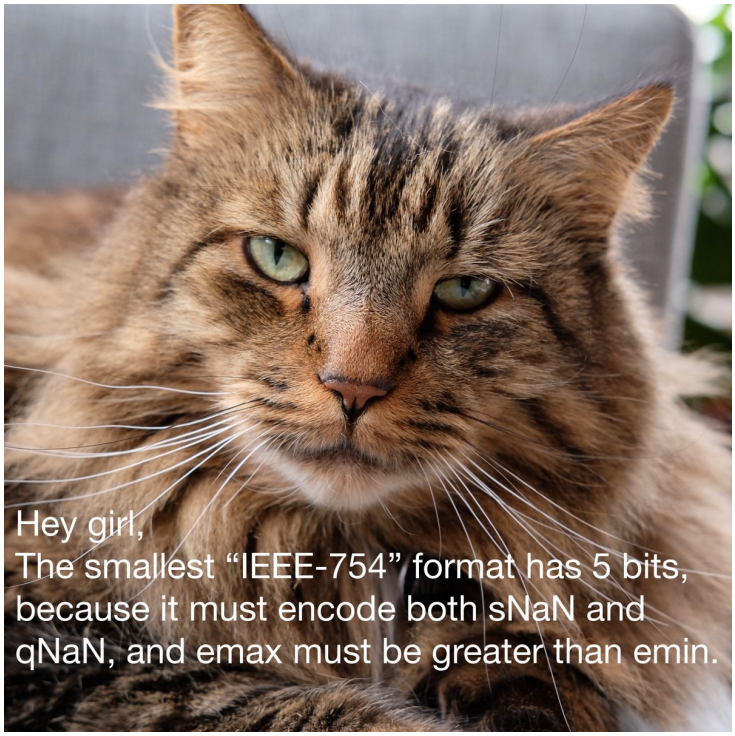

Figure 5: From Steve Canon's tweet [8]. As the cat says, since IEEE-754 requires *emax* to be greater than *emin*, there must be two exponent bits, and since the mantissa must be used to distinguish sNaN, qNaN, and infinity, so it also needs at least two bits, leading to a minimum of 5-bit floats.

The **binary3** format is further disrupted in section 3.3, which rules-out the idea of having an empty range of *emin* and *emax*, since they're used in an inequality and $emin \leq emax$, and is elsewhere forced to be strictly less by other constraints:

> "q is any integer $emin \leq q + p - 1 \leq emax$" 3.3

Still, the **binary3** format is very useful for efficient implementation of NaN gates, and is worth including in synthesis for people who aren't bothered by standards compliance. For completeness, the `synth_nan` implementation supports synthesis to **binary3**, **binary4**, and the definitely-IEEE-754-compliant **binary5** format NaN-gates. Furthermore, the architecture would support easy extensions to larger, more conventional floating point formats like **binary8**, or even larger, by simply loading your own libary of modules named `nan_fpN`, `bit_to_fpN`, and `fpN_to_bit`, for any value of `N` you want to synthesize with.

2.1 The binary5 Representation

Here we document the representation in the **binary5** format, the smallest legal IEEE-754 compliant binary floating-point format. It has a sign bit, a two bit exponent, and a two bit mantissa. We include a table of all of the positive values here:

s	E	T	value
0	00	00	+0.0
0	00	01	+0.25
0	00	10	+0.5
0	00	11	+0.75
0	01	00	+1.0
0	01	01	+1.25
0	01	10	+1.5
0	01	11	+1.75
0	10	00	+2.0
0	10	01	+2.5
0	10	10	+3.0
0	10	11	+3.5
0	11	00	+inf
0	11	01	sNaN
0	11	10	qNaN
0	11	11	qNaN

The positive values representable in the **binary5** format. Note that infinity, sNaN, and qNaN are all distinguished by the mantissa value when the exponent is all ones, so this is the smallest possible floating point format. The negative values for each are the same bit patterns but with a 1 in the sign bit.

2.2 Evaluation

We compare the size (in both logic and memory elements) and clock speed of modules synthesized with the different floating points. For the benchmark, we use a pipelined 32-bit multiplier, and the PicoRV32 processor [3] synthesized for the ECP5 architecture, and placed and routed using nextpnr [4]. The numbers given for clock frequency are the best result of 10 runs of placement and routing.

The "NAND" variant are synthesized to NAND gates before architecture-specific optimization, in order to obscure some of the higher-level modules that are present in the original design and prevent optimizations that won't be available to the NaN-gate synthesis. This gives a clearer baseline for comparison with the NaN gates, and so this is used as the basis for relative numbers. Times marked **DNP** are those that did not successfully place and route for timing analysis, so no frequency can be reported.

Design	Variant	Cells	Cell%	DFFs	DFF%	(MHz)
PicoRV	Direct	1884	57%	888	48%	103.55
	NAND	3328	100%	1848	100%	47.30
	fp3	43739	1314%	5544	299%	**DNP**
	fp4	32853	987%	7392	400%	**DNP**
	fp5	65511	1968%	9240	500%	**DNP**
Mult32	Direct	2879	104%	628	100%	143.04
	NAND	2773	100%	628	100%	154.37
	fp3	25349	880%	1884	300%	25.26
	fp4	19026	661%	2520	400%	21.88
	fp5	38001	1320%	3140	500%	21.18

It's interesting that fp4 is the smallest of the floating point variants in logic, rather than fp3. It seems likely that this is because the ECP5 architecture is based on "LUT4" cells—4-input lookup tables—which means individual NaN-gates might happen to synthesize more efficiently with 4-bit inputs.

2.3 Flattening

For this benchmark, we synthesize designs without flattening post NaN-gate synthesis, because the optimizer is too effective and eliminates most of the floating point logic. When they are flattened, the optimzer can consider the logic involved in the individual NaN gates and re-combine them and erase constant-value flip-flops. Designs that are flattened before optimizing have no flip-flop overhead, and have on the order of 5% overhead in logic elements vs the reference NAND-gate versions.

While synthesizing with post-NaN flattening substantially undermines the floating point logic and mostly demonstrates the impressive quality of Yosys's optimizations, it suggests as an option a sort of "homeopathic floating-point logic." For users that require efficiency but still want *some* of the elegance benefits, they can flatten it and optimize it away, keeping some peace-of-mind in knowing that their final circuit is derived from an elegant real-number system, regardless of how it currently behaves.

3 Future Work

Floating point synthesis still has many avenues for improvement and future work.

The current synthesis approach used by `synth_nan` remains fragile in the face of flattening and pass-ordering. It should be possible to make it harder to accidentally flatten the designs away into nothing, but they still do need to be eventually flattened since the `nextpnr` place-and-route flow is still not fully reliable in the presence of un-flattened designs. Currently the `synth_nan` pass must be run before any device-specific passes, which can be fine but it prevents the utilization of resources such as distributed RAMs.

Float synthesis tools should make it easier to define module ports that should be expanded to accomodate floating-point based signals, so that designs can operate fully in the glorious domain of the real numbers, without having to flatten all designs.

More work could be done into ensuring that the individual gates are properly optimized for different architectures, since it seems unreasonable for fp4 to remain more efficient than fp3. The system could also benefit from implementing a larger set of primitive gates, to avoid the blowup of using NAND gates to emulate everything, since they should be implementable in similar amounts of elementary logic.

With **binary5** and larger, there looks like there could be potential in attempting to explore designs that work *purely* on NaN values, exploring the flexibility in the handling of signaling and quiet NaN values.

The NaN-gate synthesis plugin for Yosys can be found at https://git.witchoflight.com/nan-gate. This paper and the examples materials in it can be found at https://git.witchoflight.com/sigbovik-nan-2020.

References

[1] Claire Wolf. The Yosys Open SYnthesis Suite. Online. http://www.clifford.at/yosys/

[2] Claire Wolf. The Yosys Manual. Online. http://www.clifford.at/yosys/files/yosys_manual.pdf

[3] Claire Wolf. PicoRV32 - A Size Optimized RISC-V CPU. Online. https://github.com/cliffordwolf/picorv32

[4] Claire Wolf, David Shah, Dan Gisselquist, Serge Bazanski, Miodrag Milanovic, and Eddie Hung. NextPNR Portable FPGA Place and Route Tool. 2018. Online. https://github.com/YosysHQ/nextpnr

[5] IEEE Standard for Floating-Point Artithmetic. 2019. In *IEEE Std 754-2019 (Revision of IEEE 754-2008).* Pages 1–84. DOI 10.1109/IEEESTD.2019.8766229.

[6] justarandomgeek. 2017. justarandomgeek's Combinator Computer Mk5. Online. https://github.com/justarandomgeek/factorio-computer

[7] legomasta99. 2018. Minecraft Computer Engineering - Redstone Computers. Online. https://www.youtube.com/watch?v=aj6IUuwLyOE&list=PLwdt_GQ3o0Xe6pUS1vdzy0ZqXrApB2MNu

[8] Stephen Canon. 2017. A Tweet About IEEE-754. Tweet by @stephentyrone on April 1, 2017. https://twitter.com/stephentyrone/status/848172687268687873

[9] Tom Murphy VII. 2019. NaN Gates and Flip FLOPS. In *Proceedings of SIGBOVIK 2019*, Pittsburgh, PA, USA, April 1, 2019. (SIGBOVIK '19, ACH). Pages 98–102.

[10] Tom Murphy VII. 2019. NaN Gates and Flip FLOPS. Online. http://tom7.org/nand/