TaMaRa: An automated triple modular redundancy EDA flow for Yosys

With an (attempt at an) introduction to computer engineering

Matt Young

8 October 2024

University of Queensland
School of Electrical Engineering and Computer Science
Supervisor: Assoc. Prof. John Williams

Prepared for Emesent

Table of contents

Prerequisite knowledge

Terminology

For this entire presentation, assume silicon IC == *digital* silicon IC == ASIC

- Analogue ICs are very common, but are significantly different - not much here applies
- Caveat: Most designs are *mixed-signal* (analogue & digital), we consider only the digital part

ASIC = Application Specific Integrated Circuit

- Some examples: NPUs, GPUs, ISPs, display controllers, SuperIO controllers, disk controllers, audio codecs, video encoders, ...

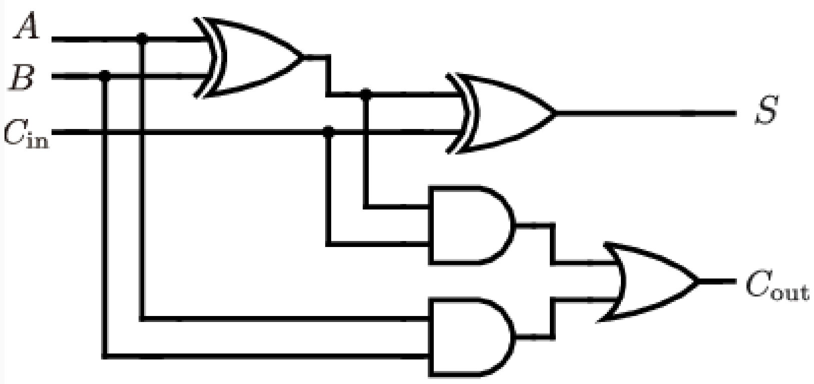
If you can see this presentation, you are using many of the above!

Recall digital logic: fundamentally binary (1/0), using combinatorial logic gates (AND, OR, etc) and sequential gates (D-flip-flops).

Every ASIC is at its core just these fundamental gates.

(Terms and conditions apply, see: mixed signal designs, custom standard cells, optical/MEMS designs, etc.)

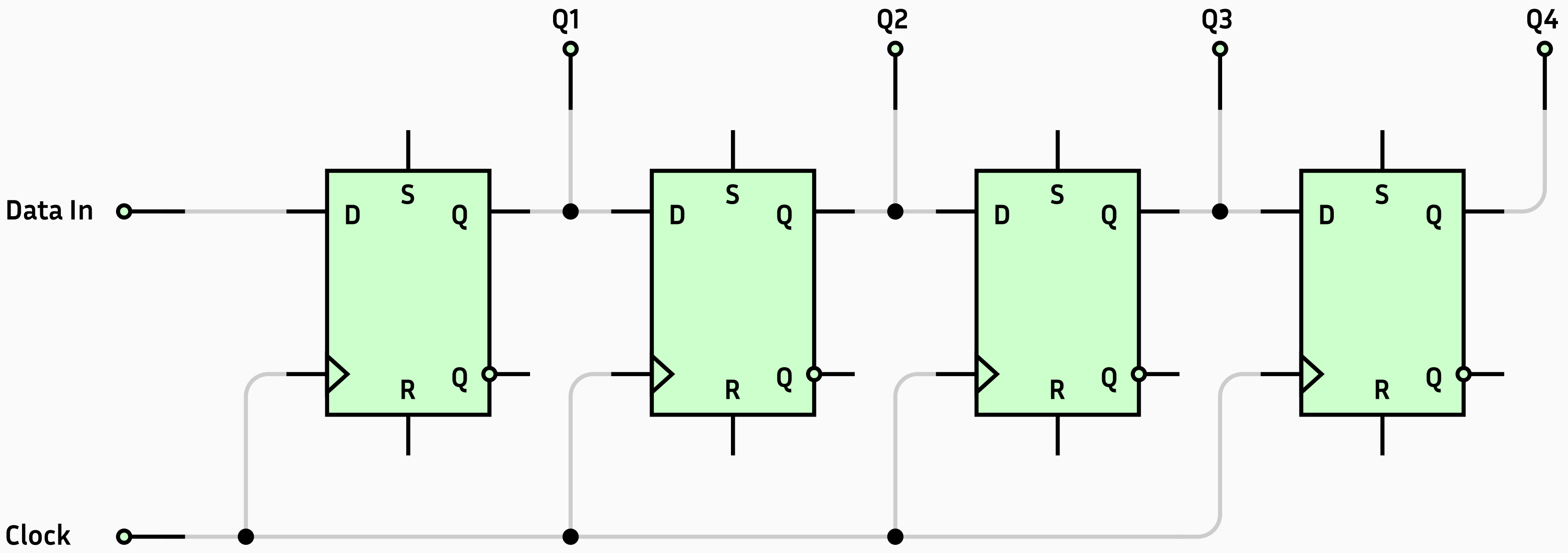
Purely combinatorial example: 1-bit full adder



Inputs			Outputs	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Source: https://www.researchgate.net/figure/Full-adder-circuit-diagram-and-truth-table-where-A-B-and-C-in-are-binary-inputs_fig2_349727409

Sequential example: 4-bit serial in, parallel out (SIPO) shift register



Source: https://commons.wikimedia.org/wiki/File:4-Bit_SIPO_Shift_Register.svg

What even is an integrated circuit???

Digital ICs consist of millions/billions of transistors, etched onto a silicon wafer using *photolithography*.

The photolithography setup forms the *process node*, which in turn forms the transistor size (*gate pitch*).

What even is an integrated circuit???

Digital ICs consist of millions/billions of transistors, etched onto a silicon wafer using *photolithography*.

The photolithography setup forms the *process node*, which in turn forms the transistor size (*gate pitch*).

This is what people talk about with “4 nm” etc - but this is an **advertising term**!

- **Zero** correlation to physical gate size!

What even is an integrated circuit???

Digital ICs consist of millions/billions of transistors, etched onto a silicon wafer using *photolithography*.

The photolithography setup forms the *process node*, which in turn forms the transistor size (*gate pitch*).

This is what people talk about with “4 nm” etc - but this is an **advertising term**!

- **Zero** correlation to physical gate size!

Photolithography techniques include *deep ultraviolet lithography* (DUV) ($\geq$ 14 nm) and *extreme ultraviolet lithography* (EUV) ($\leq$ 7 nm and beyond)

What even is an integrated circuit???

Digital ICs consist of millions/billions of transistors, etched onto a silicon wafer using *photolithography*.

The photolithography setup forms the *process node*, which in turn forms the transistor size (*gate pitch*).

This is what people talk about with “4 nm” etc - but this is an **advertising term**!

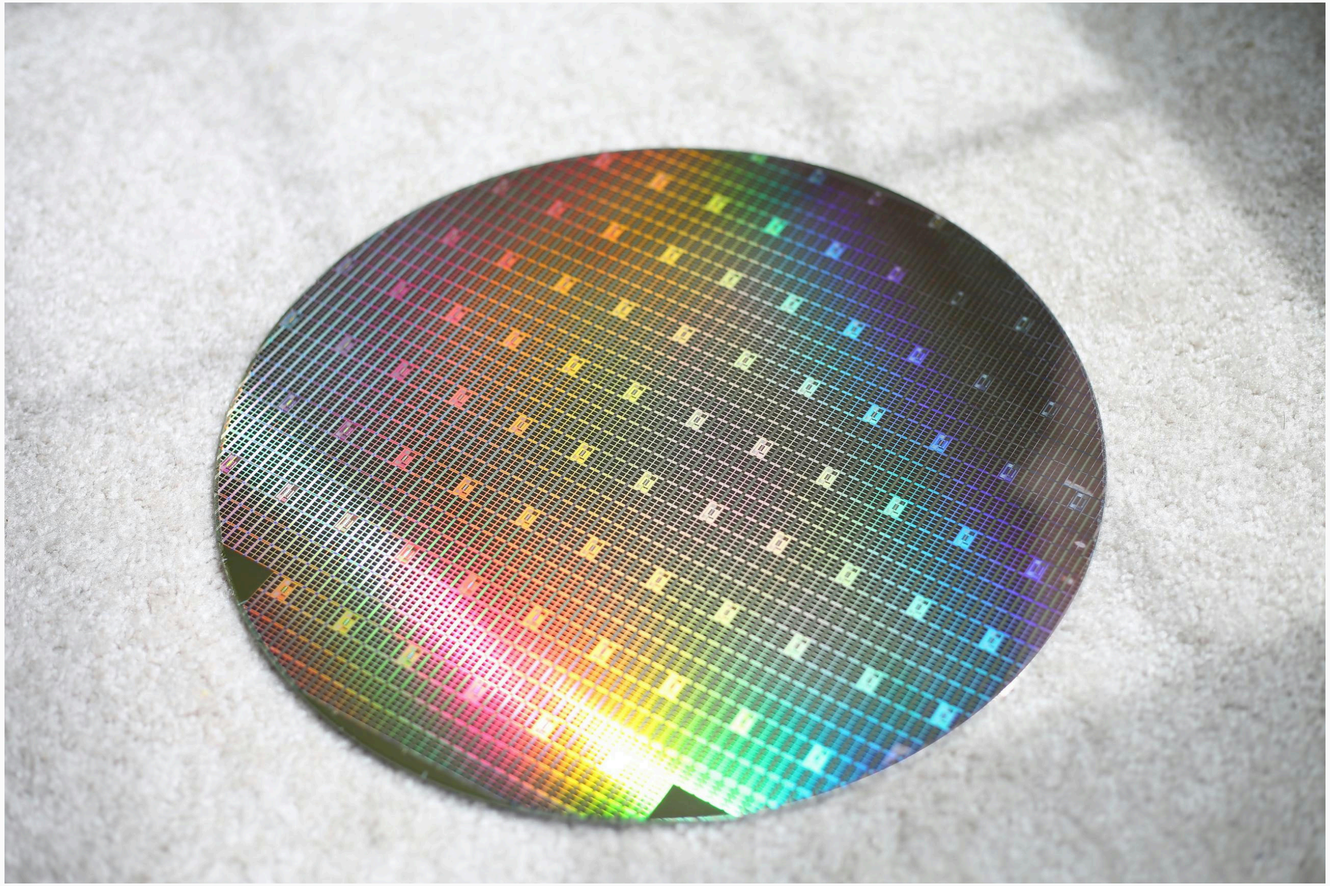
- **Zero** correlation to physical gate size!

Photolithography techniques include *deep ultraviolet lithography* (DUV) ($\geq$ 14 nm) and *extreme ultraviolet lithography* (EUV) ($\leq$ 7 nm and beyond)

ASML EUV machines cost $\geq$ **\$100 million USD** and are considered possibly the most complex machines on Earth.

What even is an integrated circuit???

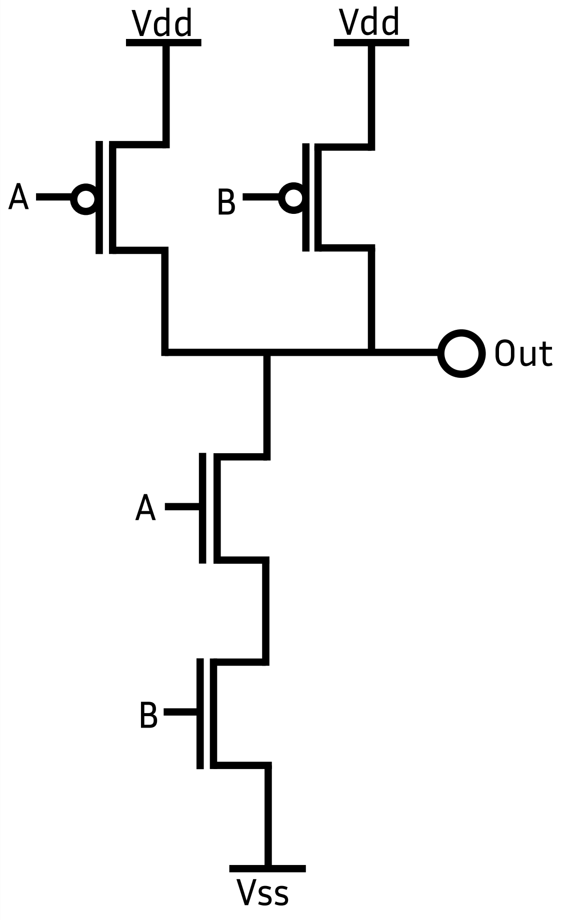
Source: https://electronics.stackexchange.com/questions/518573/can-somebody-identify-this-12-silicon-wafer



Modern ICs are built using complementary metal oxide semiconductor (CMOS) transistors.

Combination of NMOS and PMOS transistors.

Significantly better static power (leakage current) NMOS/PMOS, and faster switching times, at the cost of higher area.



Source: https://en.wikipedia.org/wiki/File:CMOS_NAND.svg

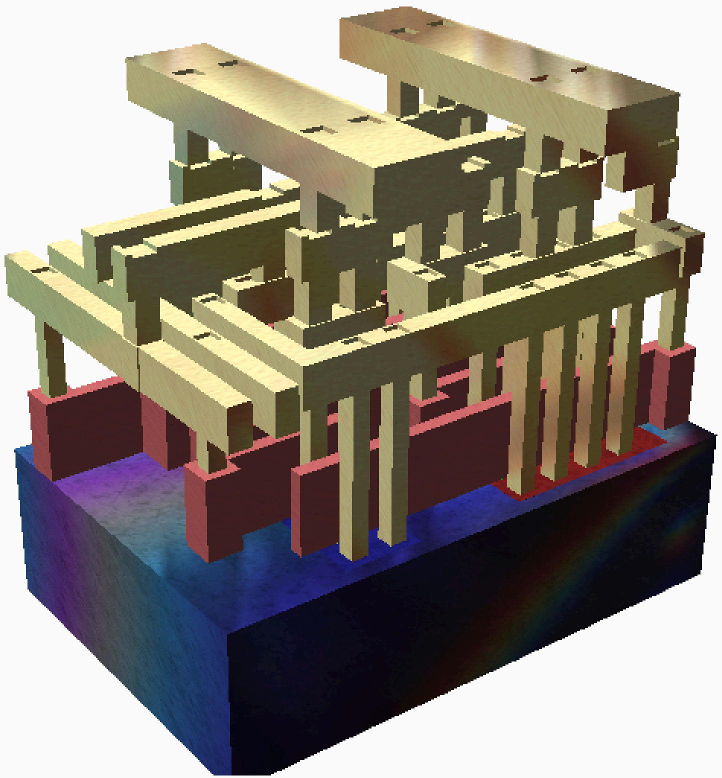
What even is an integrated circuit???

Standard cells: Building blocks of modern digital ASICs. Unique to a particular foundry and process node.

Delivered as part of the Process Design Kit (PDK) after signing 9 million NDAs.

Rather than each company re-designing standard logic cells like AND, NAND, SRAM memory, etc: the foundry does it.

Saves a lot of designer time, makes digital IC design much easier. Improves efficiency of PnR tools. All round good idea!



Source: https://commons.wikimedia.org/wiki/File:Silicon_chip_3d.png

What even is an FPGA???

Manufacturing silicon ICs is *extraordinarily* expensive, and totally uneconomic for low-volume runs.

But people still need digital circuits in many low-volume industries!

Field Programmable Gate Arrays (FPGAs) allow for many of the benefits of silicon ICs at a fraction of the cost.

What even is an FPGA???

Manufacturing silicon ICs is *extraordinarily* expensive, and totally uneconomic for low-volume runs.

But people still need digital circuits in many low-volume industries!

Field Programmable Gate Arrays (FPGAs) allow for many of the benefits of silicon ICs at a fraction of the cost.

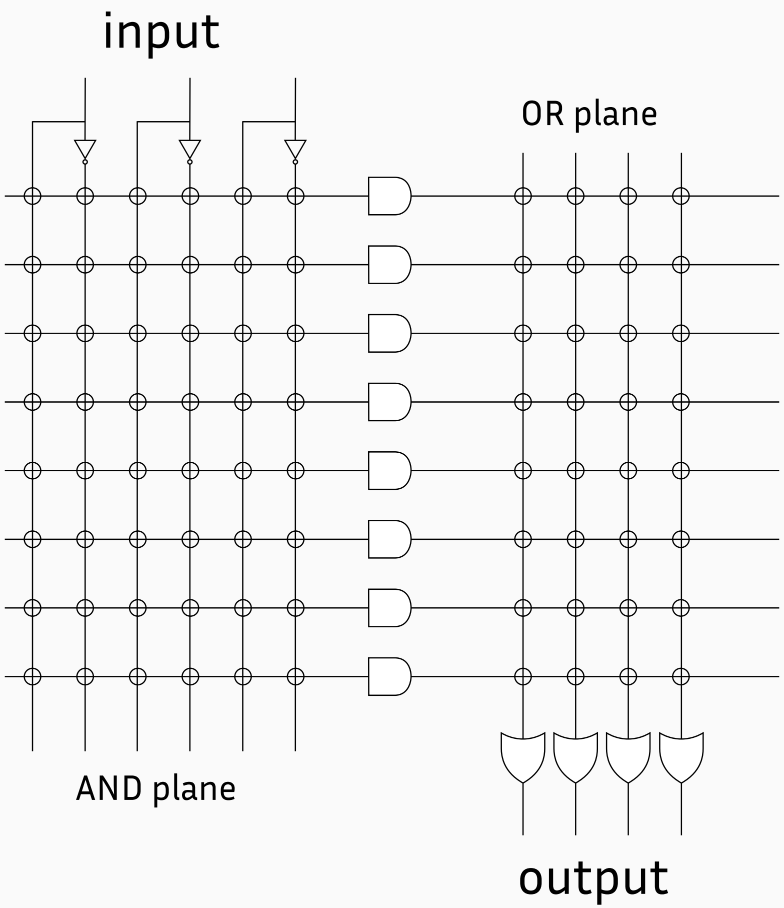
But to understand what an FPGA is, we first need to talk about PALs...

PALs

Programmable Array Logic: The precursor to FPGAs (c. 1978).

Designer implements Boolean logic manually using sum of products on a programmable AND/OR plane.

Recall *sum of products*: canonical representation of Boolean truth table (e.g. $A.B + \overline{B}.C + ...$)



Source: https://commons.wikimedia.org/wiki/File:Programmable_Logic_Device.svg

FPGAs

Eventually PALs turned into CPLDs, and finally CPLDs turned into... FPGAs!

Now we have 100,000+ *logic cells* (terminology depends on vendor), that can be chained together to implement any digital logic. Super flexible!

FPGAs

Eventually PALs turned into CPLDs, and finally CPLDs turned into... FPGAs!

Now we have 100,000+ *logic cells* (terminology depends on vendor), that can be chained together to implement any digital logic. Super flexible!

Configuration (*bitstream*) is written to FPGA SRAM on boot, so cheap & easy to flash.

Eventually PALs turned into CPLDs, and finally CPLDs turned into... FPGAs!

Now we have 100,000+ *logic cells* (terminology depends on vendor), that can be chained together to implement any digital logic. Super flexible!

Configuration (*bitstream*) is written to FPGA SRAM on boot, so cheap & easy to flash.

Hardened functional blocks for better performance/power (typically multipliers, RAM, IO, etc, nowadays even CPUs, NPUs).

Tricky mixed signal components also hardened (PLLs, SERDES, etc).

FPGAs

Eventually PALs turned into CPLDs, and finally CPLDs turned into... FPGAs!

Now we have 100,000+ *logic cells* (terminology depends on vendor), that can be chained together to implement any digital logic. Super flexible!

Configuration (*bitstream*) is written to FPGA SRAM on boot, so cheap & easy to flash.

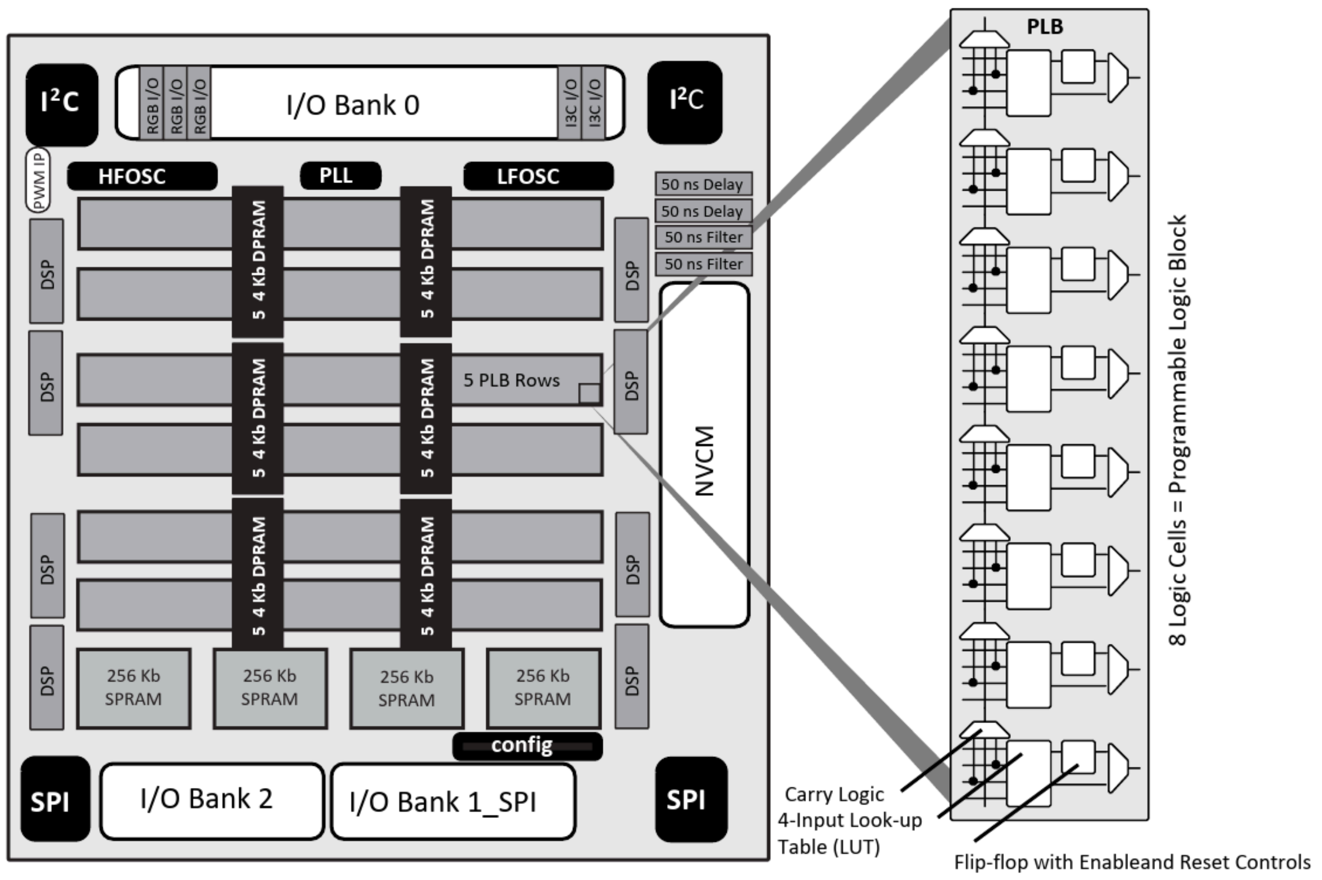
Hardened functional blocks for better performance/power (typically multipliers, RAM, IO, etc, nowadays even CPUs, NPUs).

Tricky mixed signal components also hardened (PLLs, SERDES, etc).

Still: Worse power, performance and area (PPA) than an actual silicon ASIC (hence why ASICs are still designed!)

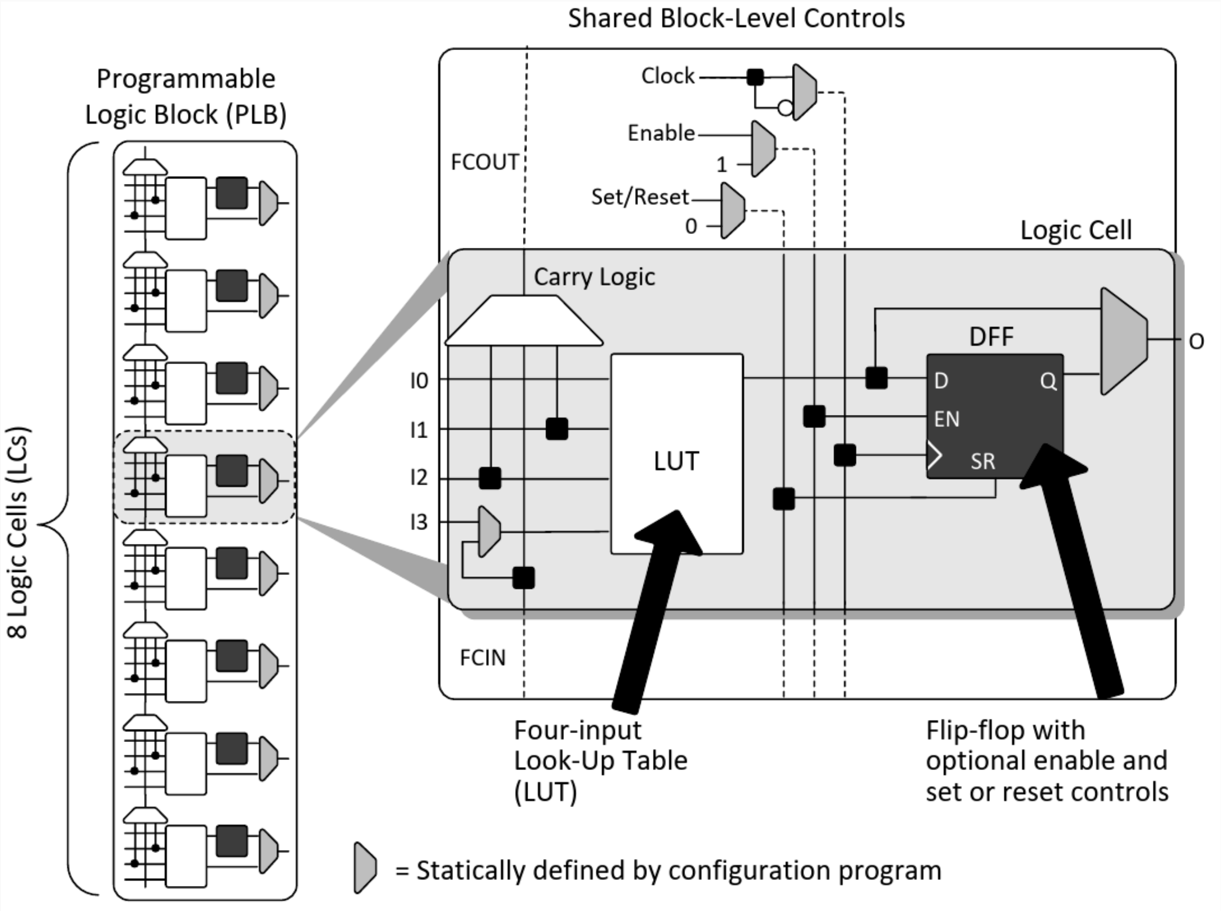
FPGAs

Source: Lattice iCE40 UltraPlus Family Data Sheet. © 2021 Lattice Semiconductor Corp.



FPGAs

Source: Lattice iCE40 UltraPlus Family Data Sheet. © 2021 Lattice Semiconductor Corp.



FPGAs are used in everything, everywhere. Anywhere you need fast, low-power, application specific processing.

Big sectors include aerospace/space, defence, science, high frequency trading, DSP, RF, machine learning, video processing.

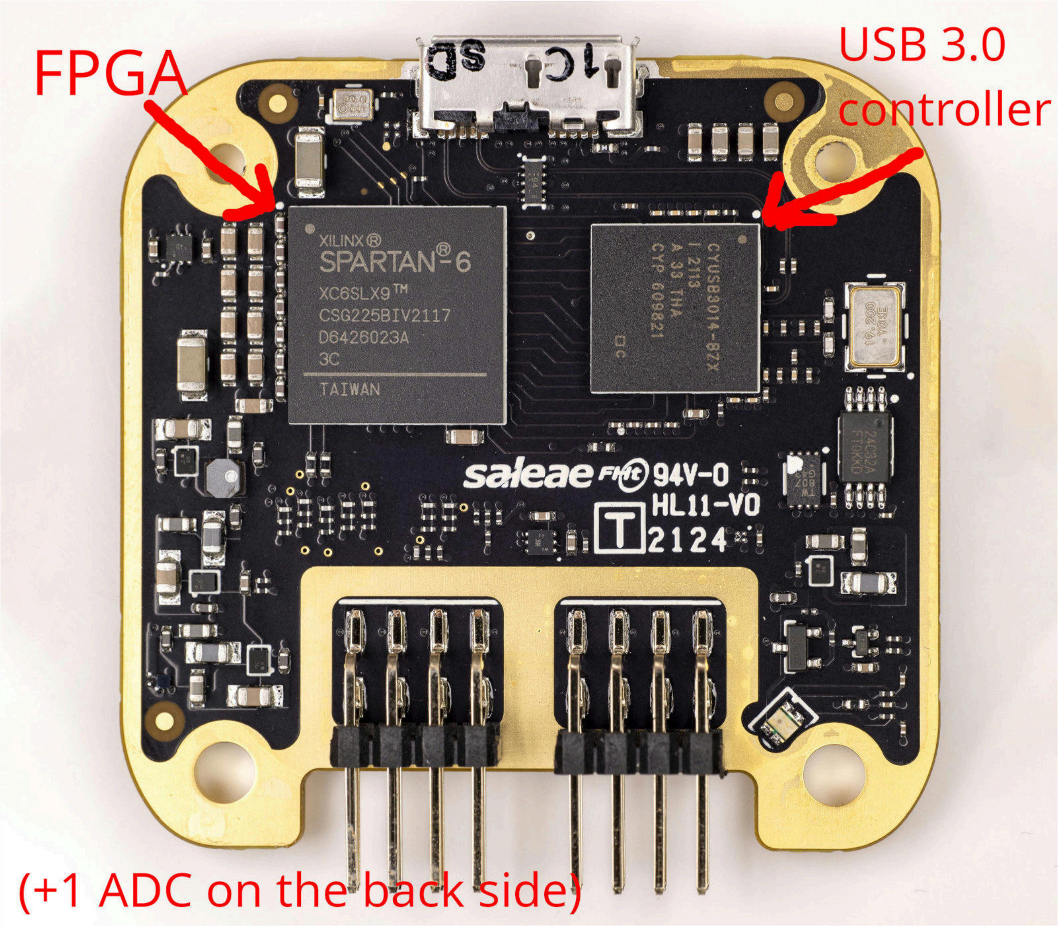
LiDARS! Every LiDAR Emesent uses has *at least* one FPGA.

- "Vendor A": 1x Altera Cyclone V (ancient).
- "Vendor B": 2x Xilinx Artix-7
- "Vendor C": They actually use a custom ASIC (!), but also likely ≥ 1 FPGA.

Rule of thumb: You'll be surprised how often an FPGA shows up when you pull apart something.

Case study: Saleae Logic 8 logic analyser

Source: https://twitter.com/timonsku/status/1497725434888437762



In ye olden days, circuits were *manually* designed using pencil and paper (including first Intel CPUs!)

Lithography masks were manually drawn by hand, hence the term “tape out”.

Nowadays, ICs consist of billions of transistors. Manual design has not been an option since the late 80s.

Instead, Electronic Design Automation (EDA) tools are used.

Electronic Design Automation (EDA)

Verilog/SystemVerilog/VHDL: Hardware description languages (HDLs), the “source code” of FPGAs and ICs.

In the semiconductor industry, we call this code Register Transfer Language (RTL).

Electronic Design Automation (EDA)

Verilog/SystemVerilog/VHDL: Hardware description languages (HDLs), the “source code” of FPGAs and ICs.

In the semiconductor industry, we call this code Register Transfer Language (RTL).

Used for both synthesis (what is on the actual chip) and simulation. *(Which is totally not confusing and has never caused any bugs ever...)*

Verilog/SystemVerilog/VHDL: Hardware description languages (HDLs), the “source code” of FPGAs and ICs.

In the semiconductor industry, we call this code Register Transfer Language (RTL).

Used for both synthesis (what is on the actual chip) and simulation. *(Which is totally not confusing and has never caused any bugs ever...)*

Describe circuits and simulation testbenches using “simple” text-based constructs.

Similar to software code... *but be careful*! Hardware and software are *very* different. HDLs are *not* the same as code!

Electronic Design Automation (EDA)

EDA tools: the “compilers” of the semiconductor industry.

Take HDL code and produce a bitstream (for FPGAs), or a photolithography mask (for ASICs).

Electronic Design Automation (EDA)

EDA tools: the “compilers” of the semiconductor industry.

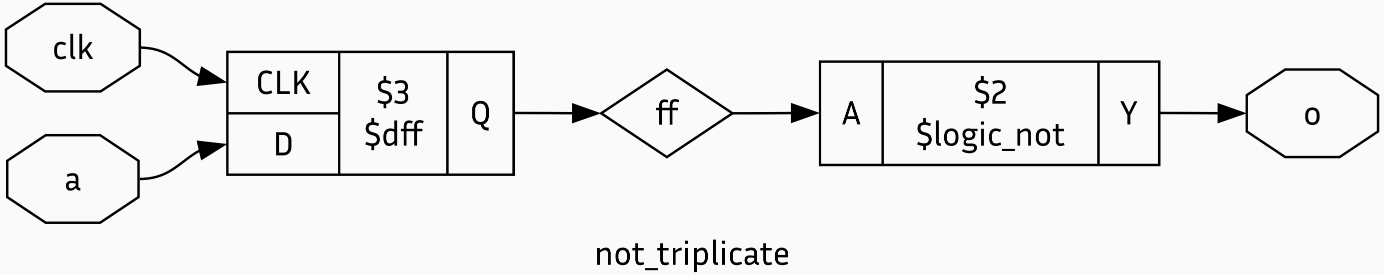
Take HDL code and produce a bitstream (for FPGAs), or a photolithography mask (for ASICs).

A bitstream/photolith mask is a bit like machine code/object files in the software world.

Again, be warned: These are similar in principle, but *very very* different from compilers.

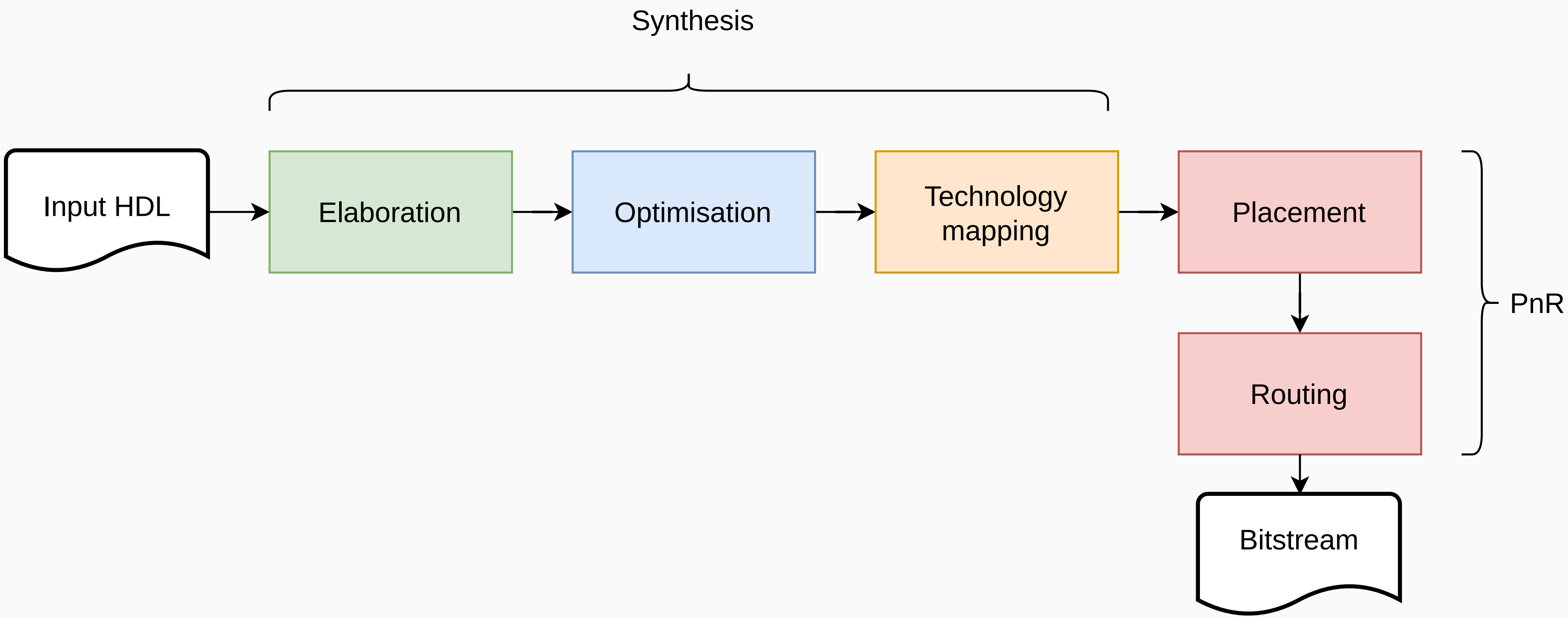
Yes, they have a frontend that lexes/parses Verilog, but the backend consists of *multiple* NP-complete placing/routing problems. Large ASICs can take weeks to “compile”.

SystemVerilog HDL example




```
module not_triplicate(
    input logic a,
    input logic clk,
    output logic o
);

logic ff;

always_ff @(posedge clk) begin
    ff <= a;
end

assign o = !ff;

endmodule
```

EDA pipeline



Yosys

If EDA tools are the “compilers” of the semiconductor industry, then **Yosys** [1] is GCC/Clang.

If EDA tools are the "compilers" of the semiconductor industry, then **Yosys** [1] is GCC/Clang.

Context: Semiconductor industry is very privatised and *very* expensive. Until last decade, open-source did not exist. Everything is IP'd/patented to hell and back. FPGA vendors very hostile to open-source.

So not only are ASICs expensive to manufacture, but just the tools to design them can set you back $\geq$ **$1 million**.

This sucks unless you're Intel/AMD/whoever. Good luck if you're a researcher/startup.

Yosys+nextpnr

Yosys is a free, open-source EDA synthesis tool, with an accompanying PnR tool nextpnr [2] that is high quality, research grade and production ready. Managed by YosysHQ GmbH.

Yosys+nextpnr support various FPGAs: Lattice iCE40/ECP5, Gowin, and a few others. Built using very complex bitstream reverse engineering.

Yosys+nextpnr

Yosys is a free, open-source EDA synthesis tool, with an accompanying PnR tool nextpnr [2] that is high quality, research grade and production ready. Managed by YosysHQ GmbH.

Yosys+nextpnr support various FPGAs: Lattice iCE40/ECP5, Gowin, and a few others. Built using very complex bitstream reverse engineering.

Also supports interesting *formal verification* flows like equivalence checking and mutation coverage.

Yosys+nextpnr

Yosys is a free, open-source EDA synthesis tool, with an accompanying PnR tool nextpnr [2] that is high quality, research grade and production ready. Managed by YosysHQ GmbH.

Yosys+nextpnr support various FPGAs: Lattice iCE40/ECP5, Gowin, and a few others. Built using very complex bitstream reverse engineering.

Also supports interesting *formal verification* flows like equivalence checking and mutation coverage.

The holy grail of open-source EDA. (Wouldn't be possible without Berkeley's abc [3] tool!)

Yosys+nextpnr

Yosys is a free, open-source EDA synthesis tool, with an accompanying PnR tool nextpnr [2] that is high quality, research grade and production ready. Managed by YosysHQ GmbH.

Yosys+nextpnr support various FPGAs: Lattice iCE40/ECP5, Gowin, and a few others. Built using very complex bitstream reverse engineering.

Also supports interesting *formal verification* flows like equivalence checking and mutation coverage.

The holy grail of open-source EDA. (Wouldn't be possible without Berkeley's abc [3] tool!)

State of the art: We can actually design 130 nm ASICs end-to-end (Verilog to GDSII mask) using fully open-source tools, thanks to the efforts of OpenLane [4], OpenSTA, Skywater Technologies [5], Google and Yosys. *Wow!*

Further reading

S. Harris, D. Harris, *Digital Design and Computer Architecture, RISC-V Edition.* Morgan Kaufmann, 2021.

Probably the only textbook in the world actually worth buying :)

A large majority of this info I learned from this book.

Thesis background

Single Event Upsets

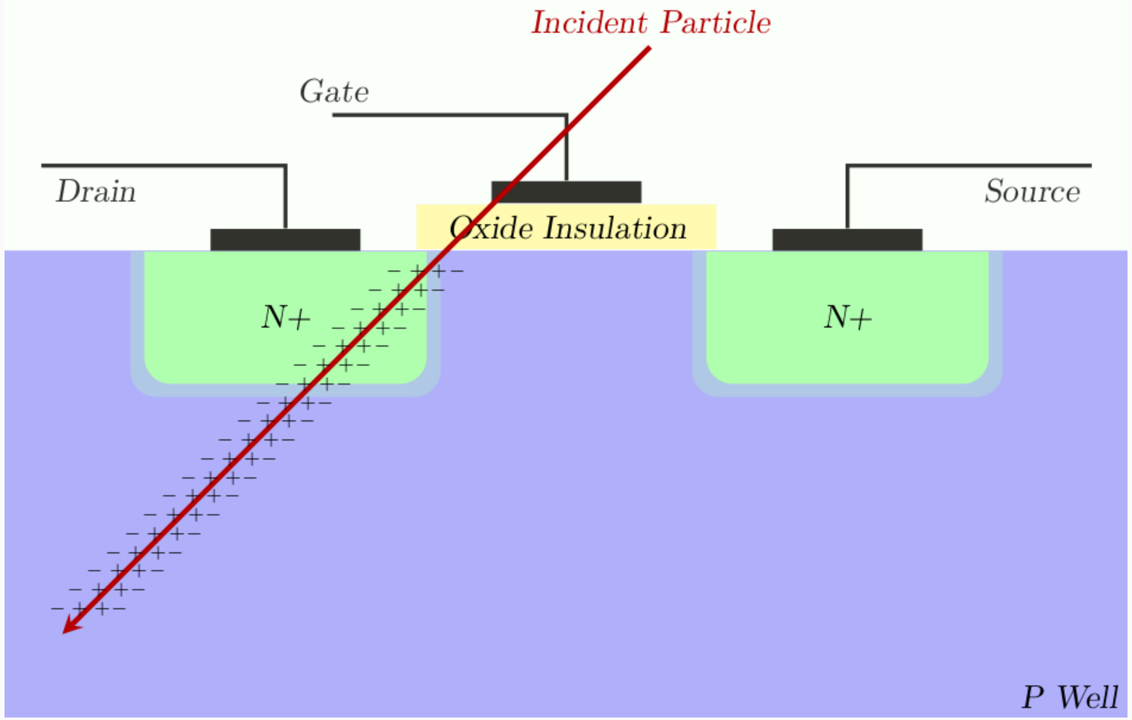
Fault tolerant computing is important for safety critical sectors (aerospace, defence, medicine, etc.)

For space-based applications, Single Event Upsets (SEUs) are very common

- Bit flips caused by ionising radiation
- Must be mitigated to prevent catastrophic failures

Even in terrestrial applications, SEUs can still occur

- Must be mitigated for high reliability applications



Source: https://www.cogenda.com/article/SEE

Motivation

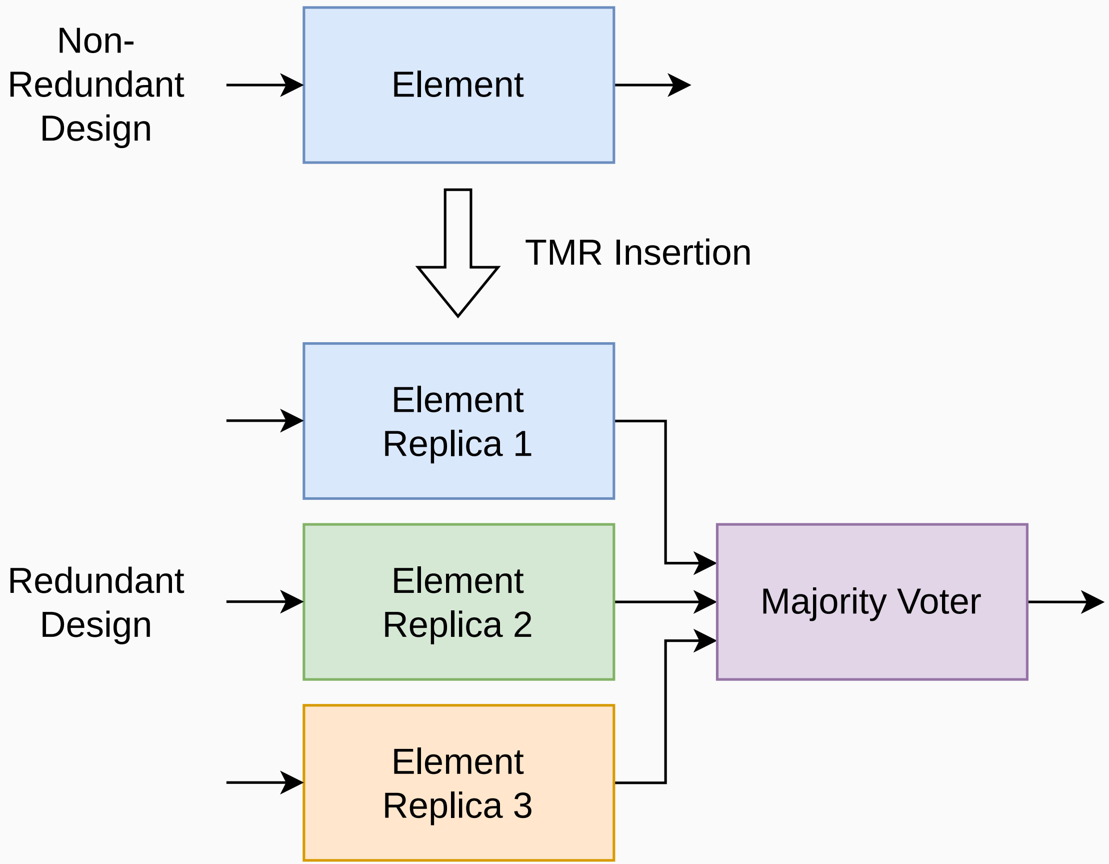
Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs) commonly deployed in space (and on Earth)...

Motivation

Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs) commonly deployed in space (and on Earth)... but protection from SEUs remains expensive!

RAD750 CPU [6] (James Webb Space Telescope, Curiosity rover, + many more) is commonly used, but costs **>$200,000 USD** [7]!

Triple Modular Redundancy



Triple Modular Redundancy

TMR can be added manually...

but this is **time consuming** and **error prone**.

Can we automate it?

TaMaRa

Concept

Implement TMR as a pass in an EDA synthesis tool.

- Integrated with the rest of the flow
- Easy to use
- Fully automated

Concept

Implement TMR as a pass in an EDA synthesis tool.

- Integrated with the rest of the flow
- Easy to use
- Fully automated

Goal: Pick any design, of any complexity, “press a button” and have it be rad-hardened.

Concept

Implement TMR as a pass in an EDA synthesis tool.

- Integrated with the rest of the flow
- Easy to use
- Fully automated

Goal: Pick any design, of any complexity, “press a button” and have it be rad-hardened.

Yosys [1] is the best (and the only) open-source, research grade EDA synthesis tool.

Concept

Implement TMR as a pass in an EDA synthesis tool.

- Integrated with the rest of the flow
- Easy to use
- Fully automated

Goal: Pick any design, of any complexity, “press a button” and have it be rad-hardened.

Yosys [1] is the best (and the only) open-source, research grade EDA synthesis tool.

- Proprietary vendor tools (Synopsys, Cadence, Xilinx, etc) immediately discarded
- Can’t be extended to add custom passes

Existing works

Two main paradigms:

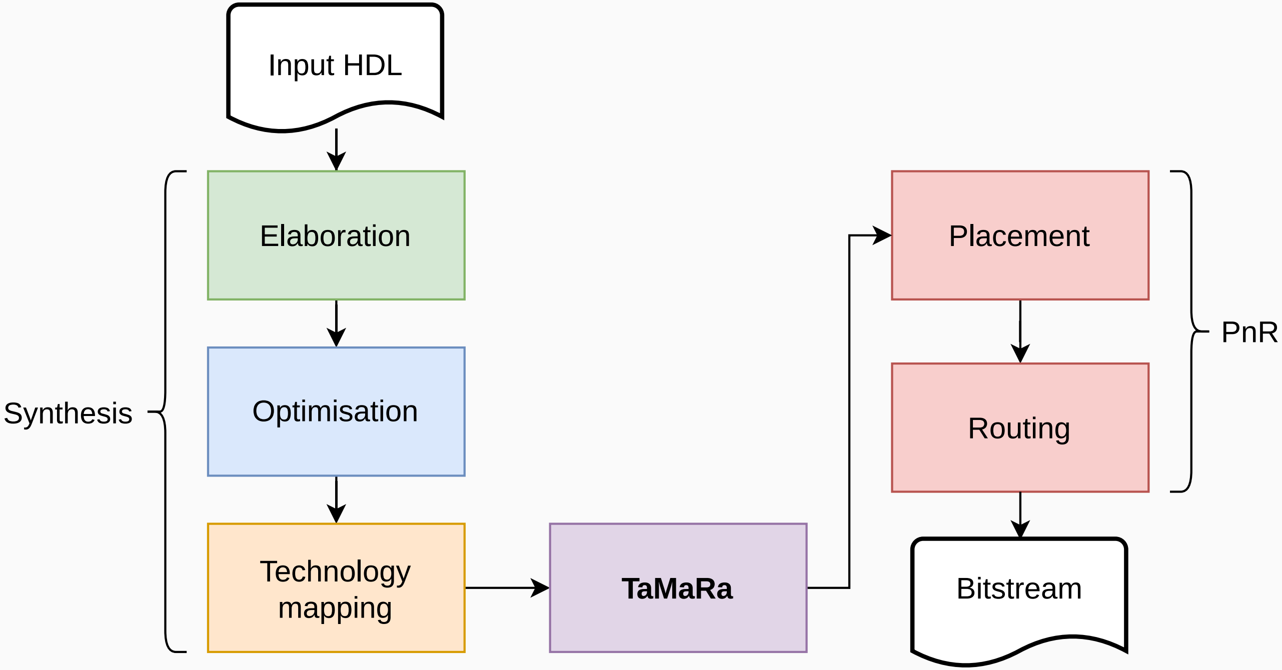
- **Design-level approaches** (“thinking in terms of HDL”)
  - Kulis [8], Lee [9]
- **Netlist-level approaches** (“thinking in terms of circuits”)
  - Johnson [10], Benites [11], Skouson [12]

The TaMaRa algorithm

TaMaRa is mainly netlist-driven. Voter insertion is inspired by Benites [11] "logic cones" concept, and parts of Johnson [10].

Also propagate a Verilog annotation to select TMR granularity (like Kulis [8]).

Runs after techmapping (i.e. after abc in Yosys)



TaMaRa algorithm: Logic cones

Source: [13]

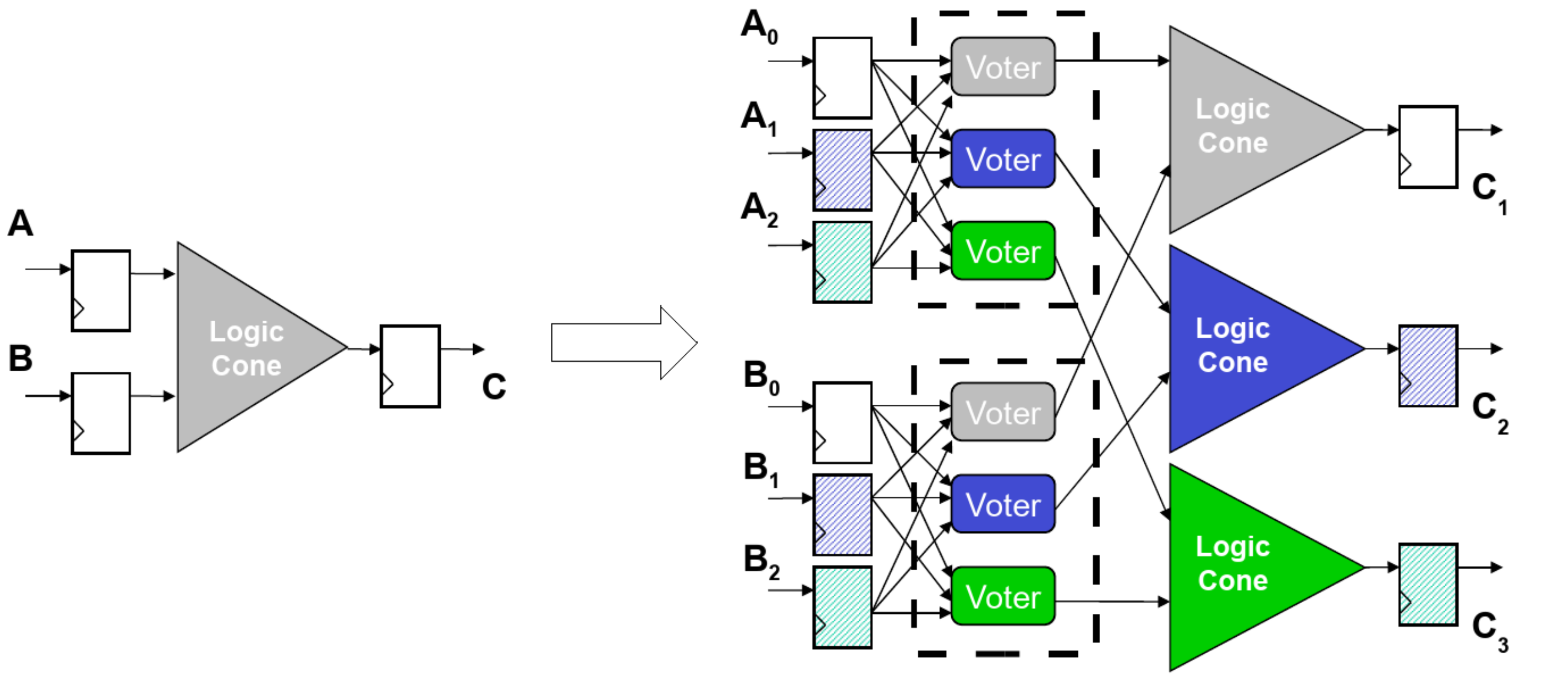


Figure 1 A logic cone is a set of logic bounded by FFs and I/O. When TMR is applied, each logic cone contains part of the voting logic.

TaMaRa algorithm: In depth

- Construct TaMaRa logic graph and logic cones
  - Analyse Yosys RTLIL netlist
  - Perform backwards BFS from IOs to FFs (or other IOs) to collect combinatorial RTLIL primitives
  - Convert RTLIL primitives into TaMaRa primitives
  - Bundle into logic cone
- Replicate RTLIL primitives inside logic cones
- Insert voters into logic cones
- Wiring
  - Wire voter up to replicated primitives
  - Wire replicated primitive IOs to the rest of the circuit
- Build successor logic cones
- Repeat until no more successors

Verification

Comprehensive verification procedure using formal methods, simulation and fuzzing.

Driven by SymbiYosys tools *eqy* and *mcy*

- In turn driven by Satisfiability Modulo Theorem (SMT) solvers (Yices [14], Boolector [15], etc)

Verification

Comprehensive verification procedure using formal methods, simulation and fuzzing.

Driven by SymbiYosys tools *eqy* and *mcy*

- In turn driven by Satisfiability Modulo Theorem (SMT) solvers (Yices [14], Boolector [15], etc)

Equivalence checking: Formally verify that the circuit is functionally equivalent before and after the TaMaRa pass.

- Ensures TaMaRa does not change the underlying behaviour of the circuit.

Verification

Comprehensive verification procedure using formal methods, simulation and fuzzing.

Driven by SymbiYosys tools *eqy* and *mcy*

- In turn driven by Satisfiability Modulo Theorem (SMT) solvers (Yices [14], Boolector [15], etc)

Equivalence checking: Formally verify that the circuit is functionally equivalent before and after the TaMaRa pass.

- Ensures TaMaRa does not change the underlying behaviour of the circuit.

Mutation: Formally verify that TaMaRa-processed circuits correct injected faults in a testbench

- Ensures TaMaRa does its job!

Fuzzing

TaMaRa must work for *all* input circuits, so we need to test at scale.

Fuzzing

TaMaRa must work for *all* input circuits, so we need to test at scale.

Idea:

1. Use Verismith [16] to generate random Verilog RTL.
2. Run TaMaRa synthesis end-to-end.
3. Use formal equivalence checking to verify the random circuits behave the same before/after TMR.

Fuzzing

TaMaRa must work for *all* input circuits, so we need to test at scale.

Idea:

1. Use Verismith [16] to generate random Verilog RTL.
2. Run TaMaRa synthesis end-to-end.
3. Use formal equivalence checking to verify the random circuits behave the same before/after TMR.

Problem: Mutation

- We need valid testbenches for these random circuits
- Requires automatic test pattern generation (ATPG), highly non-trivial
- Future topic of further research

Simulation

We want to simulate an SEU environment.

- UQ doesn't have the capability to expose FPGAs to real radiation
- Physical verification is challenging (particularly measurement)

Simulation

We want to simulate an SEU environment.

- UQ doesn't have the capability to expose FPGAs to real radiation
- Physical verification is challenging (particularly measurement)

Use one of Verilator or Yosys' own cxxrtl to simulate a full design.

- Each simulator has different trade-offs
- Currently considering picorv32 RISC-V CPU as the Device Under Test (DUT)
- Simpler DUTs will be tested as well

Simulation

We want to simulate an SEU environment.

- UQ doesn't have the capability to expose FPGAs to real radiation
- Physical verification is challenging (particularly measurement)

Use one of Verilator or Yosys' own cxxrtl to simulate a full design.

- Each simulator has different trade-offs
- Currently considering picorv32 RISC-V CPU as the Device Under Test (DUT)
- Simpler DUTs will be tested as well

Concept:

- Iterate over the netlist, randomly consider flipping a bit every cycle
  - May be non-trivial depending on simulator
- Self-checking testbench that ensures the DUT responds correctly (e.g. RISC-V CoreMark)

Current status & future

Current status

Algorithm design and planning essentially complete. Yosys internals (particularly RTLIL) understood to a satisfactory level (still learning as I go).

Current status

Algorithm design and planning essentially complete. Yosys internals (particularly RTLIL) understood to a satisfactory level (still learning as I go).

C++ development well under way, approaching 1000 lines across 8 files. Using modern C++20 features like `shared_ptr` and `std::variant` meta-programming.

Current status

Algorithm design and planning essentially complete. Yosys internals (particularly RTLIL) understood to a satisfactory level (still learning as I go).

C++ development well under way, approaching 1000 lines across 8 files. Using modern C++20 features like `shared_ptr` and `std::variant` meta-programming.

Designed majority voters and other simple circuits in Logisim and translated to SystemVerilog HDL.

Current status

Algorithm design and planning essentially complete. Yosys internals (particularly RTLIL) understood to a satisfactory level (still learning as I go).

C++ development well under way, approaching 1000 lines across 8 files. Using modern C++20 features like `shared_ptr` and `std::variant` meta-programming.

Designed majority voters and other simple circuits in Logisim and translated to SystemVerilog HDL.

Started on formal equivalence checking for TaMaRa voters and simple manually-designed combinatorial circuits.

Current status

Algorithm design and planning essentially complete. Yosys internals (particularly RTLIL) understood to a satisfactory level (still learning as I go).

C++ development well under way, approaching 1000 lines across 8 files. Using modern C++20 features like `shared_ptr` and `std::variant` meta-programming.

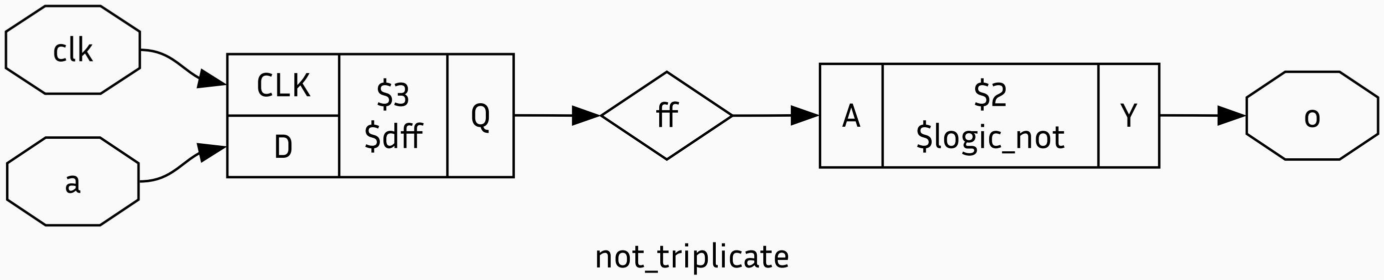
Designed majority voters and other simple circuits in Logisim and translated to SystemVerilog HDL.

Started on formal equivalence checking for TaMaRa voters and simple manually-designed combinatorial circuits.

Programming hopefully finished *around* February 2025, verification by April 2025.

Progress: Automatically triplicating a NOT gate and inserting a voter

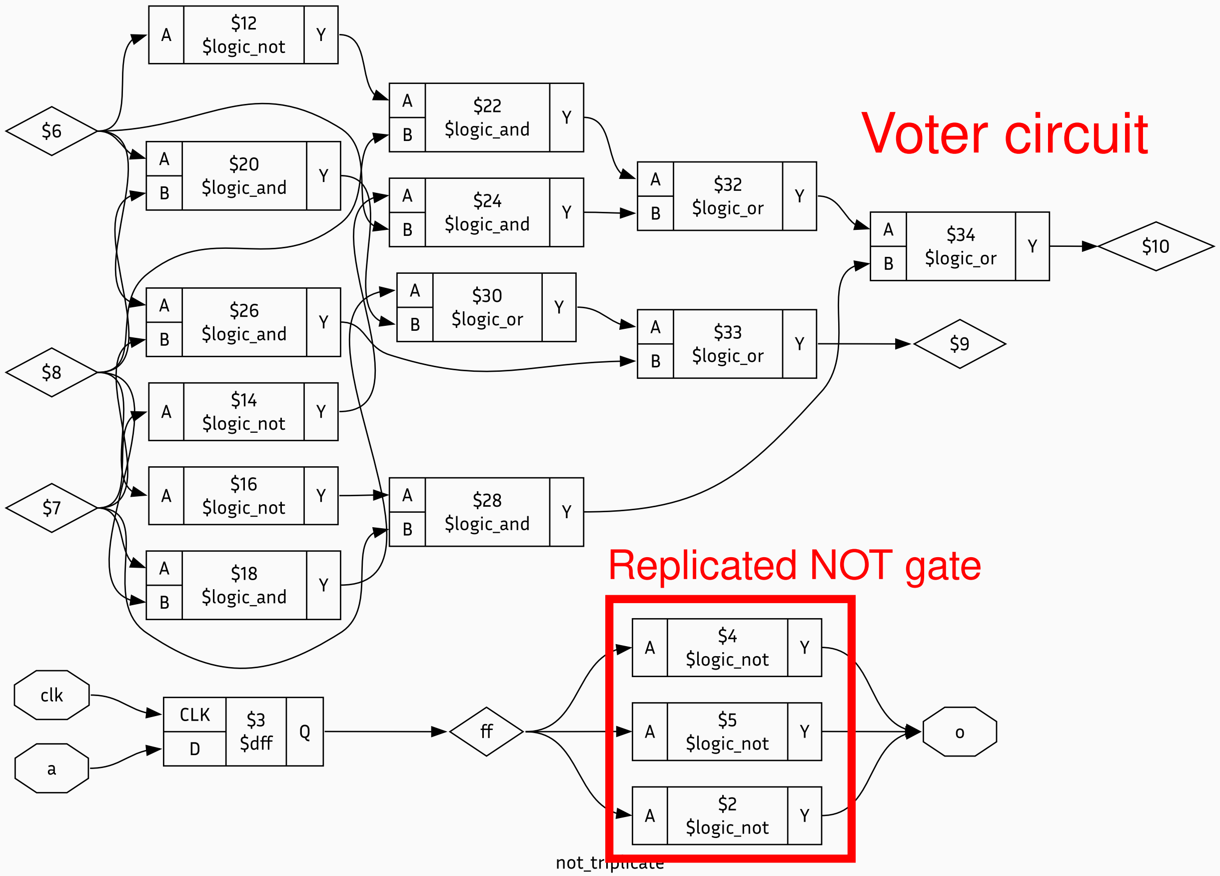
Original circuit:




```
(* tamara_triplicate *)
module not_triplicate(
    input logic a,
    input logic clk,
    output logic o
);

logic ff;

always_ff @(posedge clk) begin
    ff <= a;
end

assign o = !ff;

endmodule
```

Progress: Automatically triplicating a NOT gate and inserting a voter

After `tamara_debug replicateNot`:

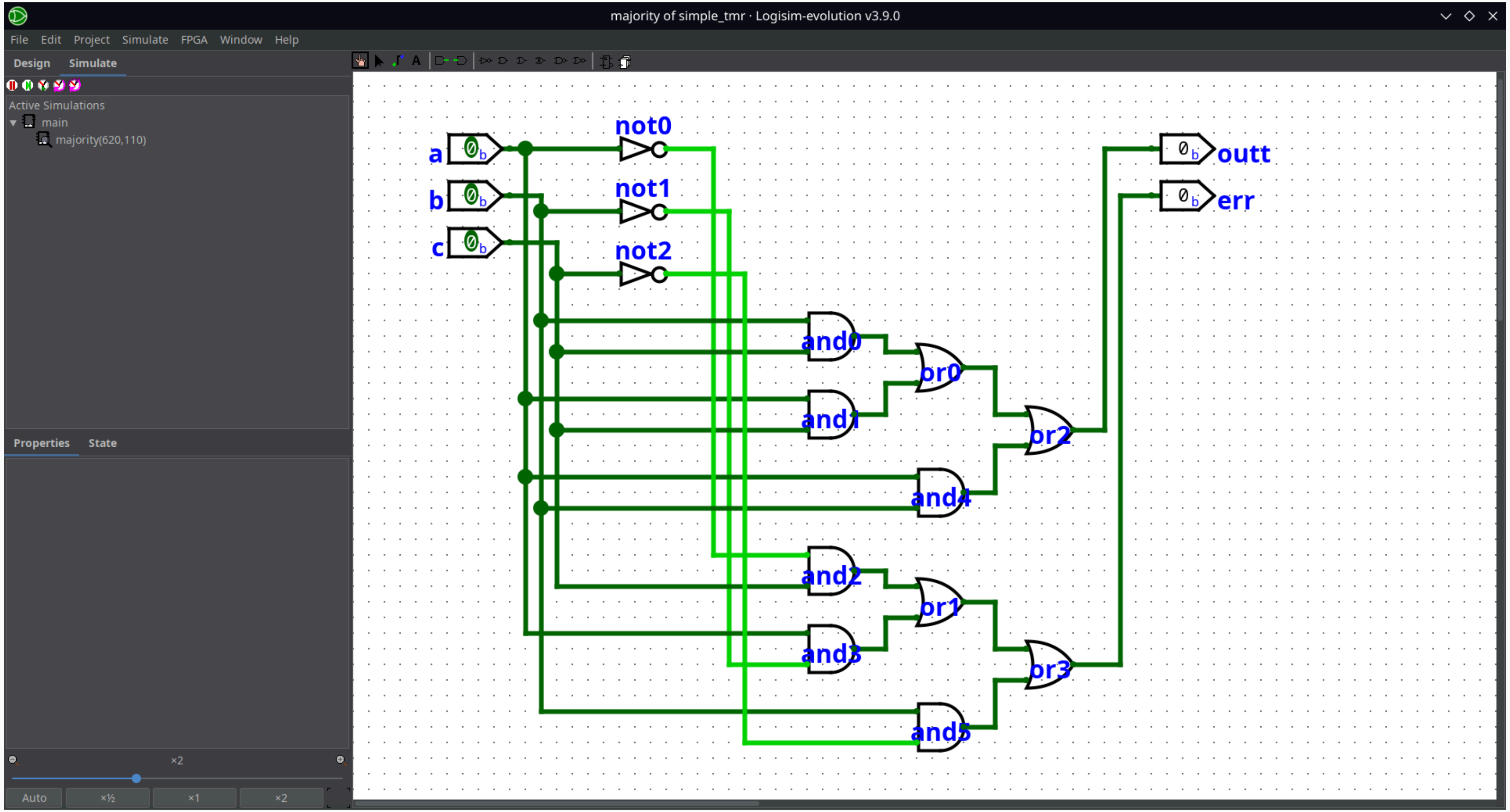


Voter circuit:

a	b	c	out	err
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	1
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	0

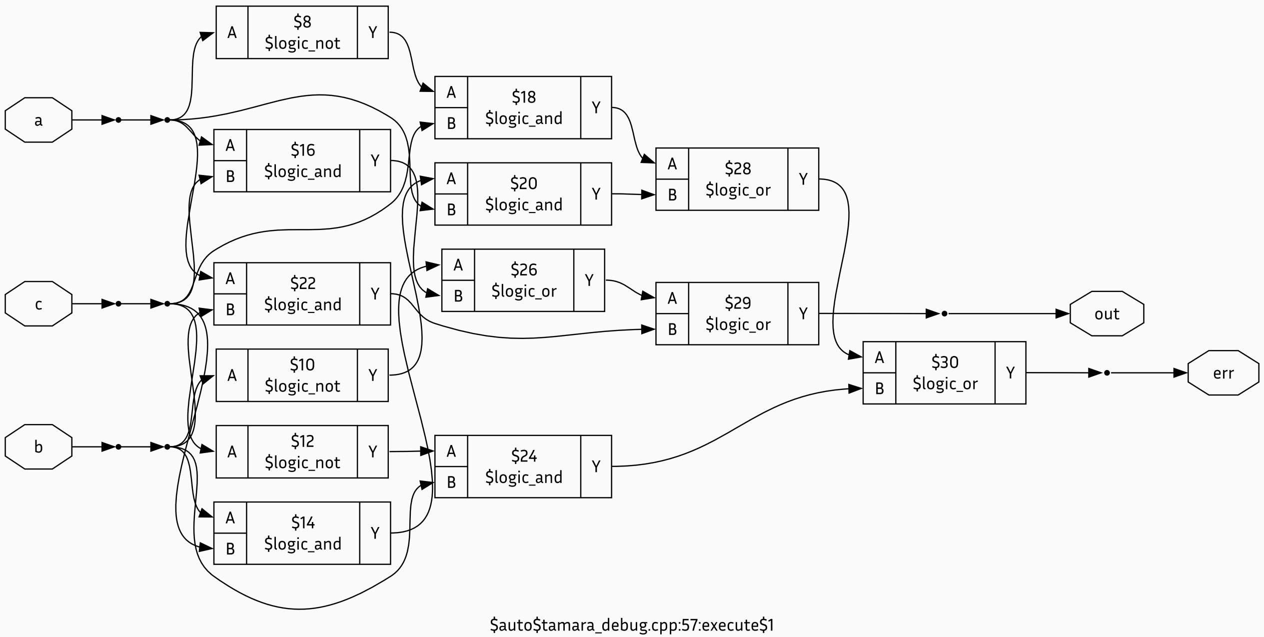
```
module voter(
    input logic a,
    input logic b,
    input logic c,
    output logic out,
    output logic err
);
    assign out = (a && b) || (b && c) || (a && c);
    assign err = (!a && c) || (a && !b) || (b && !c);
endmodule
```

Manual design in Logisim:



Progress: Equivalence checking

```
    Voter
tamara::VoterBuilder::build(RTLIL::Module
*module) {
        // NOT
        // a -> not0 -> and2
        WIRE(not0, and2);
        NOT(0, a, not0_and2_wire);
        ...

        // AND
        // b, c -> and0 -> or0
        WIRE(and0, or0);
        AND(0, b, c, and0_or0_wire);
        ...

        // OR
        // and0, and1 -> or0 -> or2
        WIRE(or0, or2);
        OR(0, and0_or0_wire,
and1_or0_wire, or0_or2_wire);
        ...

        return ...;
    }
```



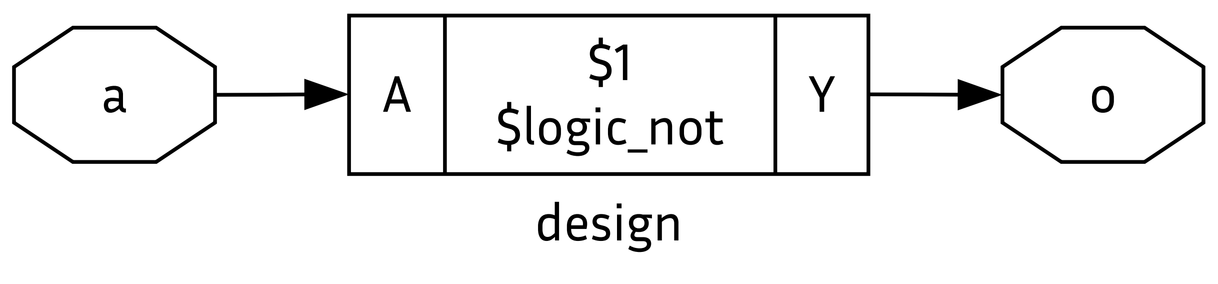
Progress: Equivalence checking

Marked equivalent by eqy in conjunction with Yices!

```
~/w/t/build (master) [n] ⤫> eqy -f ../tests/formal/equivalence/voter.eqy
EQY 22:47:32 [voter] read_gold: starting process "yosys -ql voter/gold.log voter/gold.ys"
EQY 22:47:32 [voter] read_gold: finished (returncode=0)
EQY 22:47:32 [voter] read_gate: starting process "yosys -ql voter/gate.log voter/gate.ys"
EQY 22:47:32 [voter] read_gate: finished (returncode=0)
EQY 22:47:32 [voter] combine: starting process "yosys -ql voter/combine.log voter/combine.ys"
EQY 22:47:32 [voter] combine: finished (returncode=0)
EQY 22:47:32 [voter] partition: starting process "cd voter; yosys -ql partition.log partition.ys"
EQY 22:47:32 [voter] partition: finished (returncode=0)
EQY 22:47:32 [voter] run: starting process "make -C voter -f strategies.mk"
EQY 22:47:32 [voter] run: make: Entering directory '/home/matt/workspace/tamara/build/voter'
EQY 22:47:32 [voter] run: Running strategy 'sby' on 'voter.err'..
EQY 22:47:32 [voter] run: Proved equivalence of partition 'voter.err' using strategy 'sby'
EQY 22:47:32 [voter] run: Running strategy 'sby' on 'voter.out'..
EQY 22:47:32 [voter] run: Proved equivalence of partition 'voter.out' using strategy 'sby'
EQY 22:47:32 [voter] run: make -f strategies.mk summary
EQY 22:47:32 [voter] run: make[1]: Entering directory '/home/matt/workspace/tamara/build/voter'
EQY 22:47:32 [voter] run: make[1]: Leaving directory '/home/matt/workspace/tamara/build/voter'
EQY 22:47:32 [voter] run: make: Leaving directory '/home/matt/workspace/tamara/build/voter'
EQY 22:47:32 [voter] run: finished (returncode=0)
EQY 22:47:32 [voter] Successfully proved equivalence of partition voter.out
EQY 22:47:32 [voter] Successfully proved equivalence of partition voter.err
EQY 22:47:32 [voter] Successfully proved designs equivalent
EQY 22:47:33 [voter] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
EQY 22:47:33 [voter] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)
EQY 22:47:33 [voter] DONE (PASS, rc=0)
```

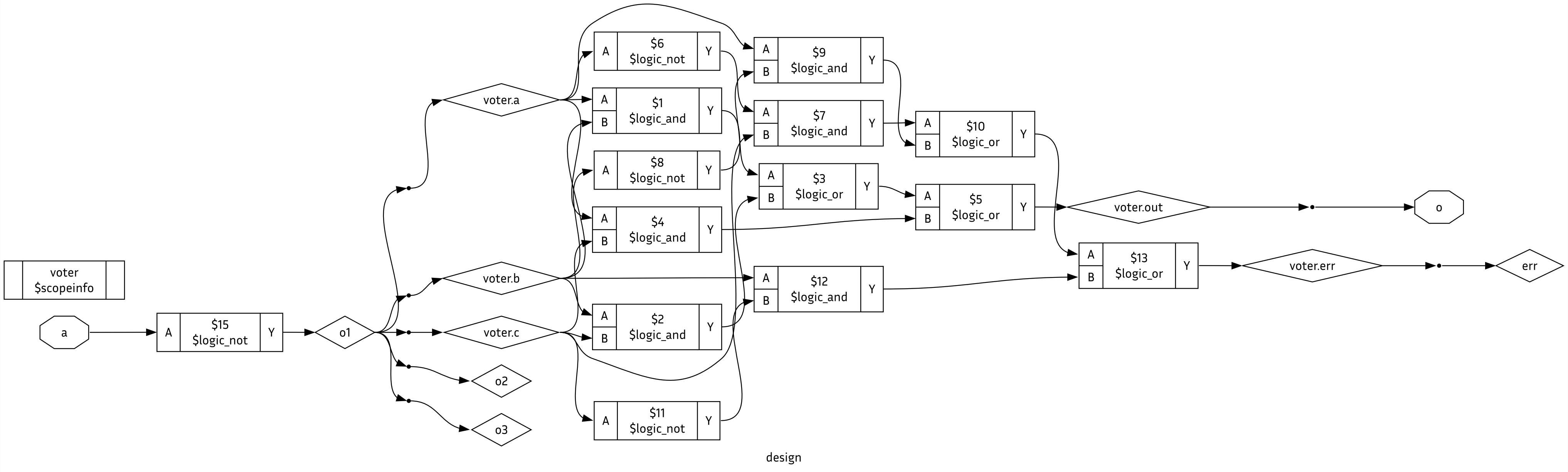
Progress: Equivalence checking (Voter insertion)

Original, very simple circuit:



Progress: Equivalence checking (Voter insertion)

After manual voter insertion (using SystemVerilog):



Progress: Equivalence checking (Voter insertion)

Are they equivalent? Yes! (Thankfully)

```
~/w/t/build (master) [n] ⋊> eqy -f ../tests/formal/equivalence/not_voter.eqy
EQY 22:10:20 [not_voter] read_gold: starting process "yosys -ql not_voter/gold.log not_voter/gold.ys"
EQY 22:10:20 [not_voter] read_gold: finished (returncode=0)
EQY 22:10:20 [not_voter] read_gate: starting process "yosys -ql not_voter/gate.log not_voter/gate.ys"
EQY 22:10:20 [not_voter] read_gate: finished (returncode=0)
EQY 22:10:20 [not_voter] combine: starting process "yosys -ql not_voter/combine.log not_voter/combine.ys"
EQY 22:10:20 [not_voter] combine: finished (returncode=0)
EQY 22:10:20 [not_voter] partition: starting process "cd not_voter; yosys -ql partition.log partition.ys"
EQY 22:10:20 [not_voter] partition: finished (returncode=0)
EQY 22:10:20 [not_voter] run: starting process "make -C not_voter -f strategies.mk"
EQY 22:10:20 [not_voter] run: make: Entering directory '/home/matt/workspace/tamara/build/not_voter'
EQY 22:10:20 [not_voter] run: Running strategy 'sby' on 'design.o'..
EQY 22:10:20 [not_voter] run: Proved equivalence of partition 'design.o' using strategy 'sby'
EQY 22:10:20 [not_voter] run: make -f strategies.mk summary
EQY 22:10:20 [not_voter] run: make[1]: Entering directory '/home/matt/workspace/tamara/build/not_voter'
EQY 22:10:20 [not_voter] run: make[1]: Leaving directory '/home/matt/workspace/tamara/build/not_voter'
EQY 22:10:20 [not_voter] run: make: Leaving directory '/home/matt/workspace/tamara/build/not_voter'
EQY 22:10:20 [not_voter] run: finished (returncode=0)
EQY 22:10:20 [not_voter] Successfully proved equivalence of partition design.o
EQY 22:10:20 [not_voter] Successfully proved designs equivalent
EQY 22:10:20 [not_voter] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
EQY 22:10:20 [not_voter] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)
EQY 22:10:20 [not_voter] DONE (PASS, rc=0)
```

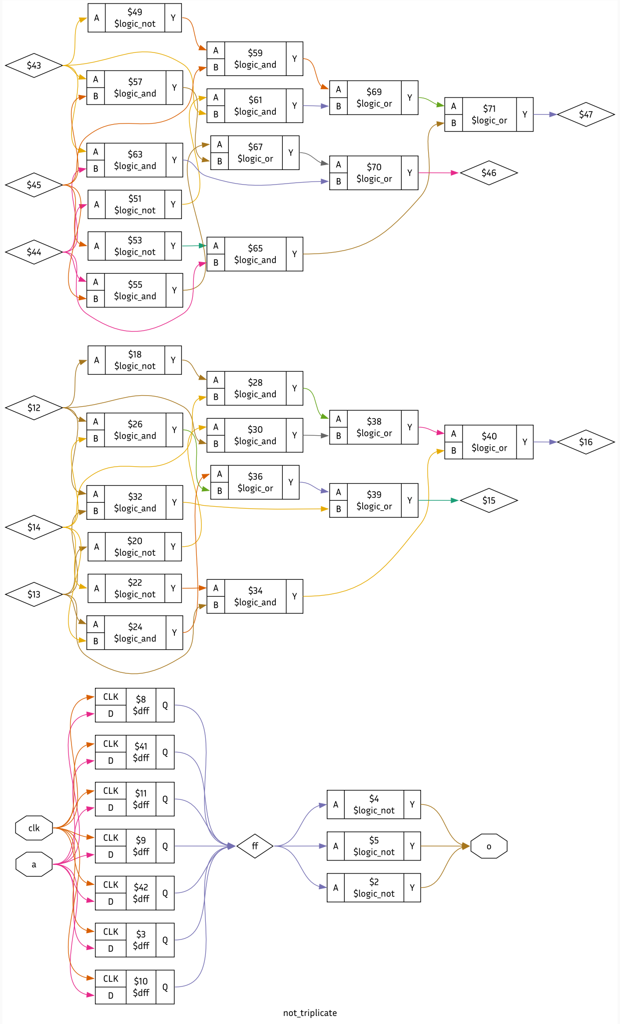
Progress: Equivalence checking (Voter insertion)

Are they equivalent? Yes! (Thankfully)

```
~/w/t/build (master) [n] ⋊> eqy -f ../tests/formal/equivalence/not_voter.eqy
EQY 22:10:20 [not_voter] read_gold: starting process "yosys -ql not_voter/gold.log not_voter/gold.ys"
EQY 22:10:20 [not_voter] read_gold: finished (returncode=0)
EQY 22:10:20 [not_voter] read_gate: starting process "yosys -ql not_voter/gate.log not_voter/gate.ys"
EQY 22:10:20 [not_voter] read_gate: finished (returncode=0)
EQY 22:10:20 [not_voter] combine: starting process "yosys -ql not_voter/combine.log not_voter/combine.ys"
EQY 22:10:20 [not_voter] combine: finished (returncode=0)
EQY 22:10:20 [not_voter] partition: starting process "cd not_voter; yosys -ql partition.log partition.ys"
EQY 22:10:20 [not_voter] partition: finished (returncode=0)
EQY 22:10:20 [not_voter] run: starting process "make -C not_voter -f strategies.mk"
EQY 22:10:20 [not_voter] run: make: Entering directory '/home/matt/workspace/tamara/build/not_voter'
EQY 22:10:20 [not_voter] run: Running strategy 'sby' on 'design.o'..
EQY 22:10:20 [not_voter] run: Proved equivalence of partition 'design.o' using strategy 'sby'
EQY 22:10:20 [not_voter] run: make -f strategies.mk summary
EQY 22:10:20 [not_voter] run: make[1]: Entering directory '/home/matt/workspace/tamara/build/not_voter'
EQY 22:10:20 [not_voter] run: make[1]: Leaving directory '/home/matt/workspace/tamara/build/not_voter'
EQY 22:10:20 [not_voter] run: make: Leaving directory '/home/matt/workspace/tamara/build/not_voter'
EQY 22:10:20 [not_voter] run: finished (returncode=0)
EQY 22:10:20 [not_voter] Successfully proved equivalence of partition design.o
EQY 22:10:20 [not_voter] Successfully proved designs equivalent
EQY 22:10:20 [not_voter] summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
EQY 22:10:20 [not_voter] summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)
EQY 22:10:20 [not_voter] DONE (PASS, rc=0)
```

Caveat: Still need to verify circuits with more complex logic (i.e. DFFs).

Current problem: Duplicate DFFs




```
7.2. Computing logic graph
Module has 1 output ports, 2 selected cells

Searching from output port o
Starting search for cone 0
    ... [snip] ...
Search complete for cone 0, have 3 items

Replicating 3 collected items for logic cone 0
    Replicating ElementCellNode $logic_not$../tests/verilog/
not_triplicate.sv:16$2
    Replicating ElementWireNode ff
    Replicating FFNode $procdff$3
Checking terminals
Input node $procdff$3 is not IONode, replicating it
    Replicating FFNode $procdff$3
Warning: When replicating FFNode $procdff$3 in cone 0: Already
replicated in logic cone 0
Input node o is IONode, it will NOT be replicated

Inserting voter into logic cone 0

... [snip] ...
```

The future

Tasks that remain (more or less):

- Fixing duplicate logic elements when replicating RTLIL primitives
- Wiring voter to logic elements, and wiring replicated logic elements to the rest of the circuit
- Considering wiring for feedback circuits *(expected to be complex/massive time sink!)*
- Global routing of error signal to a net
- Processing complex circuits like picorv32
- Writing a cycle-accurate fault-injection simulator, and associated testbenches
- Formal equivalence checking for complex circuits
- Formal mutation coverage
- Fuzzing *(if time permits)*

The future

I’m aiming to produce at least one proper academic publication from this thesis.

The future

I'm aiming to produce at least one proper academic publication from this thesis.

TaMaRa plugin code and tests will be released open-source under the Mozilla Public Licence 2.0 (used by Firefox, Eigen, etc).

Papers, including thesis and hopefully any future academic publications, will be available under CC-BY.

In short, TaMaRa will be freely available for anyone to use and build on.

The future

I'm aiming to produce at least one proper academic publication from this thesis.

TaMaRa plugin code and tests will be released open-source under the Mozilla Public Licence 2.0 (used by Firefox, Eigen, etc).

Papers, including thesis and hopefully any future academic publications, will be available under CC-BY.

In short, TaMaRa will be freely available for anyone to use and build on.

I have also spoken with the team at YosysHQ GmbH and Sandia National Laboratories, who are very interested in the results of this project and its applications.

Conclusion

Summary

- TaMaRa: Automated triple modular redundancy EDA flow for Yosys
- Fully integrated into Yosys suite
- Takes any circuit, helps to prevent it from experiencing SEUs by adding TMR
- Synthesises netlist-driven approaches [13], [10] with design-level approaches [8]
- **Key goal:** "Click a button" and have any circuit run in space/in high reliability environments!

I'd like to extend my gratitude to N. Engelhardt of YosysHQ, the team at Sandia National Laboratories, and my supervisor Assoc. Prof. John Williams for their support and interest during this thesis so far.

References

[1] C. Wolf and J. Glaser, "Yosys - A Free Verilog Synthesis Suite," in *Proceedings of Austrochip 2013*, 2013. [Online]. Available: http://yosyshq.net/yosys/files/yosys-austrochip2013.pdf

[2] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic, "Yosys+nextpnr: an Open Source Framework from Verilog to Bitstream for Commercial FPGAs," *CoRR*, 2019, [Online]. Available: http://arxiv.org/abs/1903.10407

[3] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*, 2010, pp. 24–40.

[4] M. Shalan and T. Edwards, "Building OpenLANE: A 130nm OpenROAD-based Tapeout- Proven Flow : Invited Paper," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–6.

[5] "SkyWater Open Source PDK." Accessed: Aug. 11, 2024. [Online]. Available: https://github.com/google/skywater-pdk

[6] R. Berger *et al.*, "The RAD750™ - a radiation hardened PowerPC™ processor for high performance spaceborne applications," in *2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542)*, 2001, pp. 2263–2272. doi: 10.1109/AERO.2001.931184.

[7] H. Hagedoorn, "NASA Perseverance rover 200 MHZ CPU costs $200K." Accessed: Aug. 20, 2024. [Online]. Available: https://www.guru3d.com/story/nasa-perseverance-rover-200-mhz-cpu-costs-200k/

[8] S. Kulis, "Single Event Effects mitigation with TMRG tool," *Journal of Instrumentation*, vol. 12, no. 1, p. C01082–C01082, Jan. 2017, doi: 10.1088/1748-0221/12/01/c01082.

[9] G. Lee, D. Agiakatsikas, T. Wu, E. Cetin, and O. Diessel, "TLegUp: A TMR Code Generation Tool for SRAM-Based FPGA Applications Using HLS," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 129–132. doi: 10.1109/FCCM.2017.57.

[10] J. M. Johnson and M. J. Wirthlin, "Voter insertion algorithms for FPGA designs using triple modular redundancy," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, in FPGA '10. ACM, Feb. 2010. doi: 10.1145/1723112.1723154.

[11] L. A. C. Benites and F. L. Kastensmidt, "Automated design flow for applying Triple Modular Redundancy (TMR) in complex digital circuits," in *2018 IEEE 19th Latin-American Test Symposium (LATS)*, 2018, pp. 1–4. doi: 10.1109/LATW.2018.8349668.

[12] D. Skouson, A. Keller, and M. Wirthlin, "Netlist Analysis and Transformations Using SpyDrNet," in *Proceedings of the 19th Python in Science Conference*, M. Agarwal, C. Calloway, D. Niederhut, and D. Shupe, Eds., 2020, pp. 40–47. doi: 10.25080/Majora-342d178e-006.

[13] G. Beltrame, "Triple Modular Redundancy verification via heuristic netlist analysis," *PeerJ Computer Science*, vol. 1, p. e21, Aug. 2015, doi: 10.7717/peerj-cs.21.

References

[14] B. Dutertre, "Yices 2.2," in *International Conference on Computer Aided Verification*, 2014, pp. 737–744.

[15] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, no. 1, pp. 53–58, 2014.

[16] Y. Herklotz and J. Wickerson, "Finding and Understanding Bugs in FPGA Synthesis Tools," in *ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, in FPGA '20. Seaside, CA, USA: ACM, 2020. doi: 10.1145/3373087.3375310.

Thank you! Any questions?