Progress Seminar

An automated triple modular redundancy EDA flow for Yosys

Matt Young

25 September 2024

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

Table of contents

1. Background
2. TaMaRa
3. Current status & future
4. Conclusion

Background

Motivation

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

Motivation

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

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

- Must be mitigated to prevent catastrophic failure
- Caused by ionising radiation

Even in terrestrial applications, SEUs can still occur

- Must be mitigated for high reliability applications

Motivation

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

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

- Must be mitigated to prevent catastrophic failure
- Caused by ionising radiation

Even in terrestrial applications, SEUs can still occur

- Must be mitigated for high reliability applications

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

Motivation

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

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

- Must be mitigated to prevent catastrophic failure
- Caused by ionising radiation

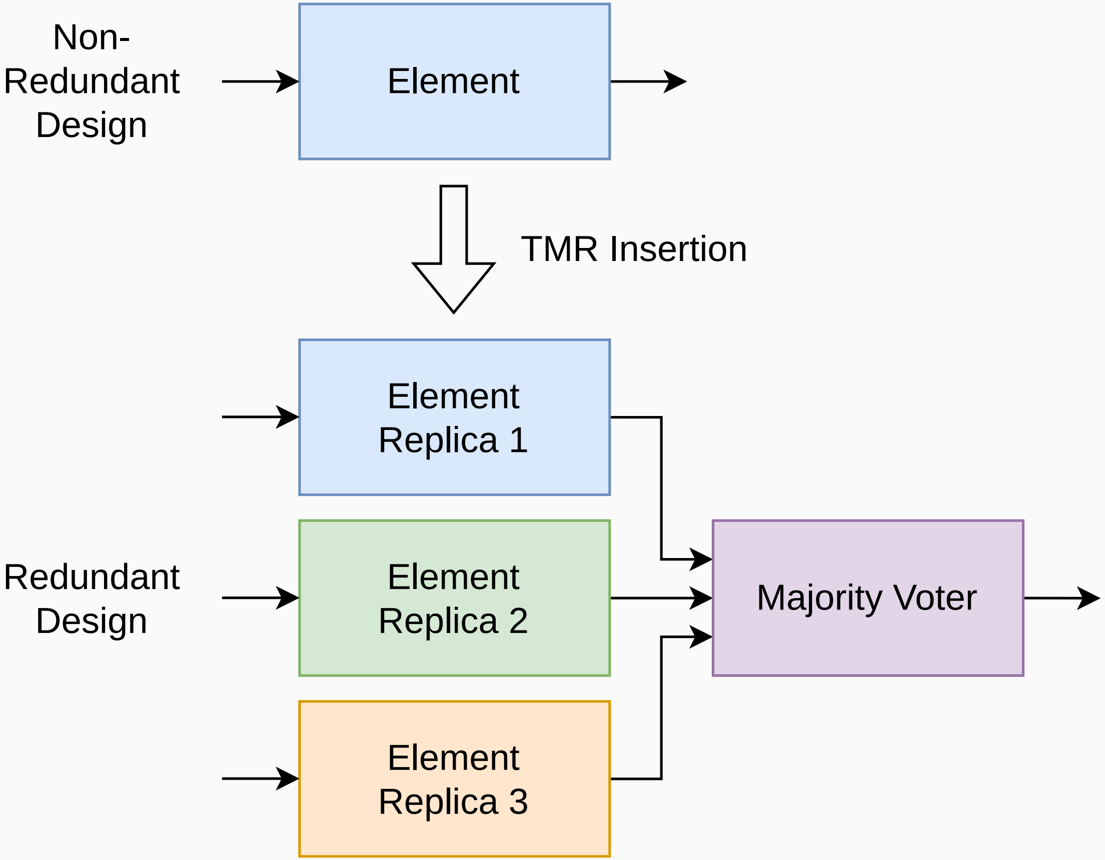
Even in terrestrial applications, SEUs can still occur

- Must be mitigated for high reliability applications

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 [1] (James Webb Space Telescope, Curiosity rover, + many more) is commonly used, but costs **>$200,000 USD** [2]!

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 [3] 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 [3] 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:

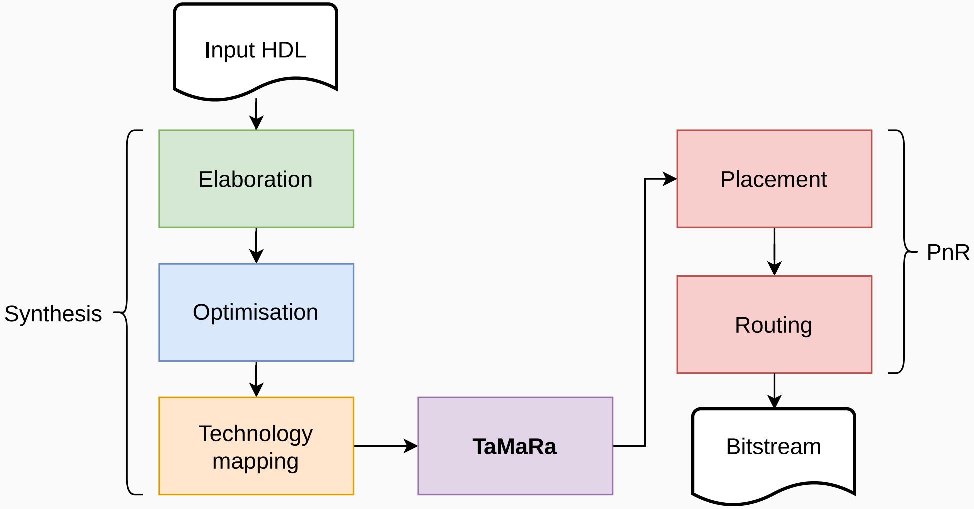
- **Design-level approaches** (“thinking in terms of HDL”)
  - Kulis [4], Lee [5]
- **Netlist-level approaches** (“thinking in terms of circuits”)
  - Johnson [6], Benites [7], Skouson [8]

The TaMaRa algorithm

TaMaRa is mainly netlist-driven. Voter insertion is inspired by Benites [7] “logic cones” concept, and parts of Johnson [6].

Also aim to propagate a `(* triplicate *)` HDL annotation to select TMR granularity (similar to Kulis [4]).

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



The TaMaRa algorithm

Why netlist driven with the `(* triplicate *)` annotation?

The TaMaRa algorithm

Why netlist driven with the `(* triplicate *)` annotation?

- Removes the possibility of Yosys optimisation eliminating redundant TMR logic
- Removes the necessity of complex blackboxing logic and trickery to bypass the normal design flow
- Cell type shouldn't matter, TaMaRa targets FPGAs and ASICs
- Still allows selecting TMR granularity - **best of both worlds**

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 [9], Boolector [10], etc)

Formal verification

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

Formal verification

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.

Formal verification

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 SEUs (single bit only)

Formal verification

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 SEUs (single bit only)

- Ensures TaMaRa does its job!

Formal verification

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 SEUs (single bit only)

- Ensures TaMaRa does its job!

Also considering Beltrame's verification tool [11], and other literature on TMR formal verification.

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 [12] 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 [12] 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

Fuzzing

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

Idea:

1. Use Verismith [12] 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, how would we generate that?
- Under active research in academia (may not be possible at the moment)

Simulation

We want to simulate an SEU environment.

- UQ doesn’t have the capability to expose FPGAs to real radiation
- Physical verification is challenging (how do you measure it?)

Simulation

We want to simulate an SEU environment.

- UQ doesn’t have the capability to expose FPGAs to real radiation
- Physical verification is challenging (how do you measure it?)

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

- Each simulator has different trade-offs
- Currently considering picorv32 or Hazard3 RISC-V CPUs as the Device Under Test (DUT)

Simulation

We want to simulate an SEU environment.

- UQ doesn’t have the capability to expose FPGAs to real radiation
- Physical verification is challenging (how do you measure it?)

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

- Each simulator has different trade-offs
- Currently considering picorv32 or Hazard3 RISC-V CPUs as the Device Under Test (DUT)

Concept:

- Iterate over the netlist, randomly consider flipping a bit every cycle
- Write a self-checking testbench and ensure that the DUT responds correctly (e.g. RISC-V CoreMark)

Technical implementation

Implemented in C++20, using CMake.

Load into Yosys: `plugin -i libtamara.so`

TMR is implemented as two separate commands: `tamara_propagate` and `tamara_tmr`

Technical implementation

Implemented in C++20, using CMake.

Load into Yosys: `plugin -i libtamara.so`

TMR is implemented as two separate commands: `tamara_propagate` and `tamara_tmr`

Run `tamara_propagate` after `read_verilog` to propagate the `(* tamara_triplicate *)` annotations.

Technical implementation

Implemented in C++20, using CMake.

Load into Yosys: `plugin -i libtamara.so`

TMR is implemented as two separate commands: `tamara_propagate` and `tamara_tmr`

Run `tamara_propagate` after `read_verilog` to propagate the `(* tamara_triplicate *)` annotations.

Run `tamara_tmr` after techmapping to perform triplication and voter insertion (add TMR).

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, have 700 lines and growing. 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, have 700 lines and growing. 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, have 700 lines and growing. 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. Proved that TaMaRa generated voter is equivalent to manual SV design, and that a simple circuit is identical after manual voter insertion.

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, have 700 lines and growing. Using modern C++20 features like `shared_ptr` and `std::variant` meta-programming.

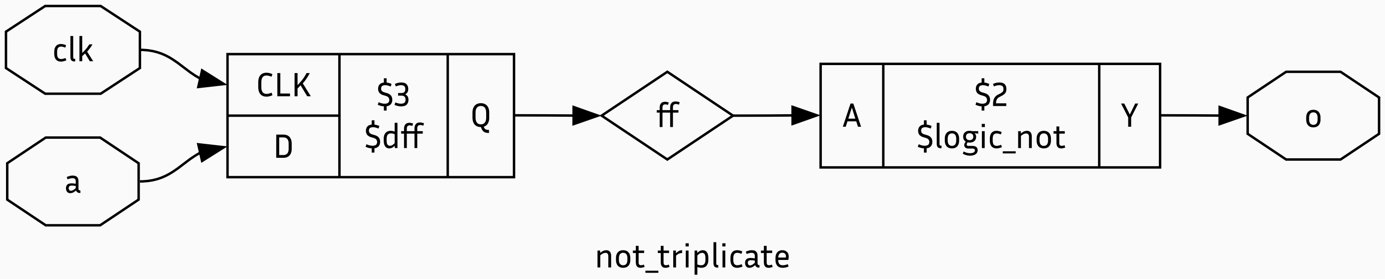
Designed majority voters and other simple circuits in Logisim and translated to SystemVerilog HDL.

Started on formal equivalence checking. Proved that TaMaRa generated voter is equivalent to manual SV design, and that a simple circuit is identical after manual voter insertion.

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

Progress: Automatically triplicating a NOT gate and inserting a voter

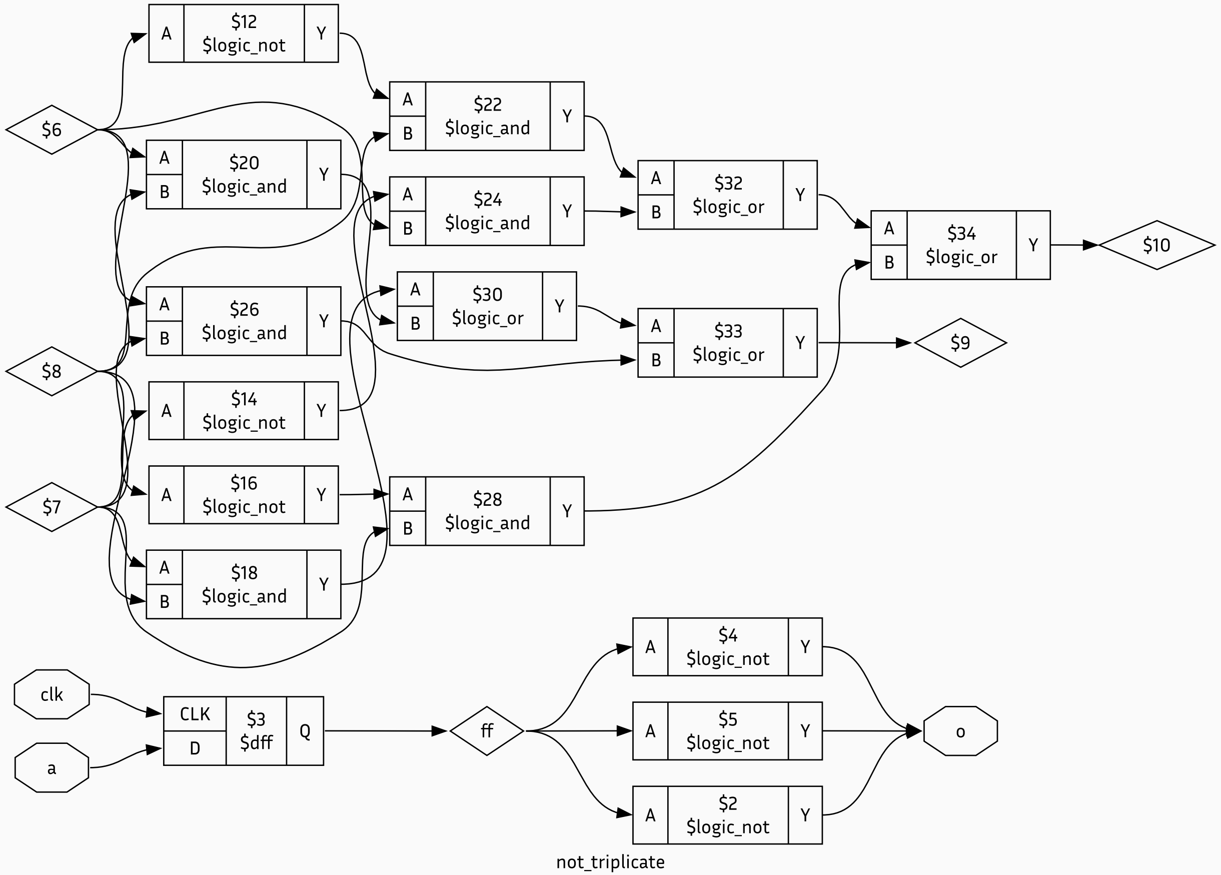
Original circuit:



```
(* 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 notTriplicate`:



Progress: Automatically triplicating a NOT gate and inserting a voter

Results:

- NOT circuit identified in `tamara::LogicGraph`
- RTLIL primitives replicated correctly
- Voter inserted using `tamara::VoterBuilder`
- Voter *not* yet wired up to main design
- Replicated components *not* yet re-wired

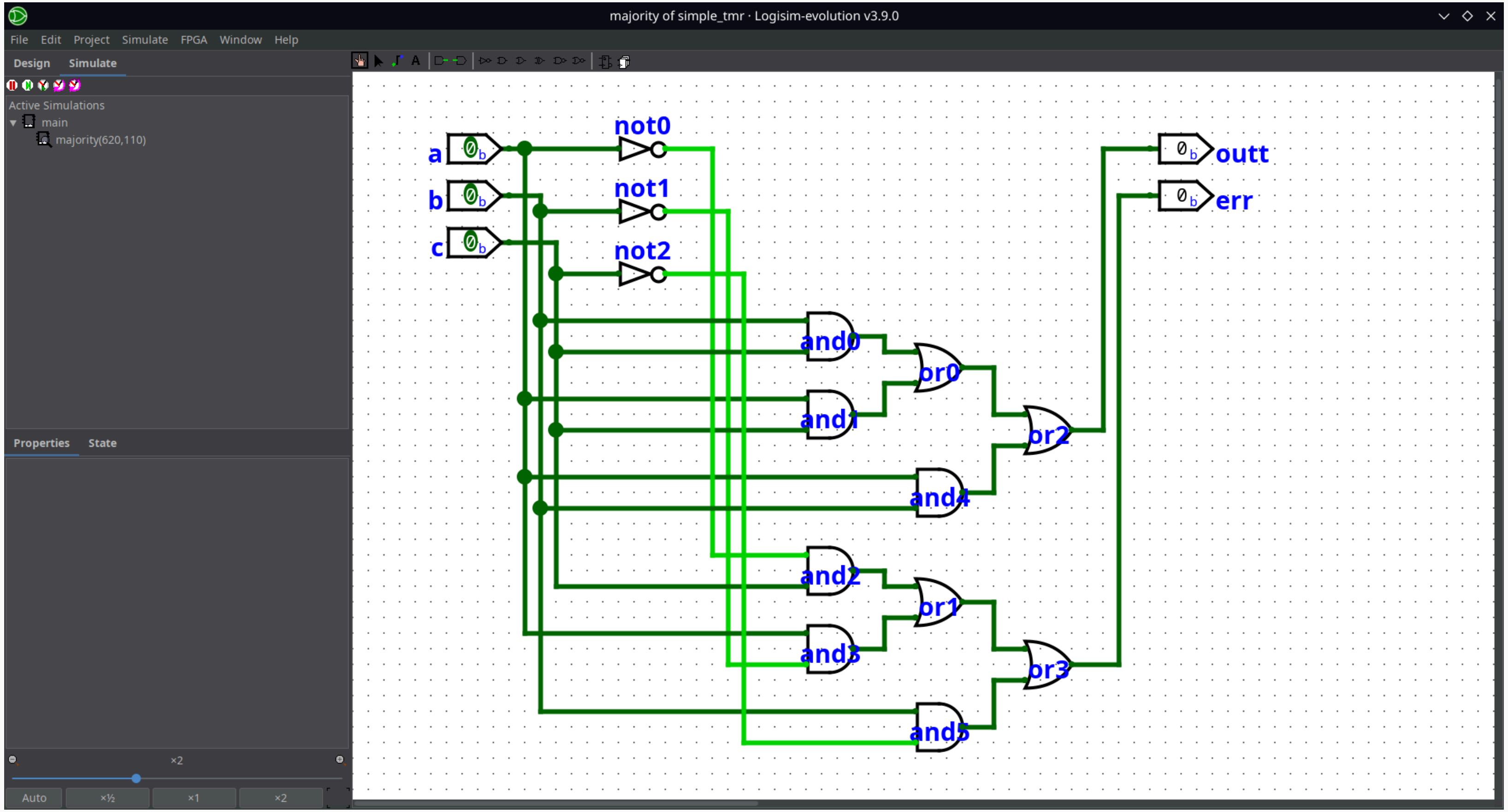
Progress: Equivalence checking

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
```

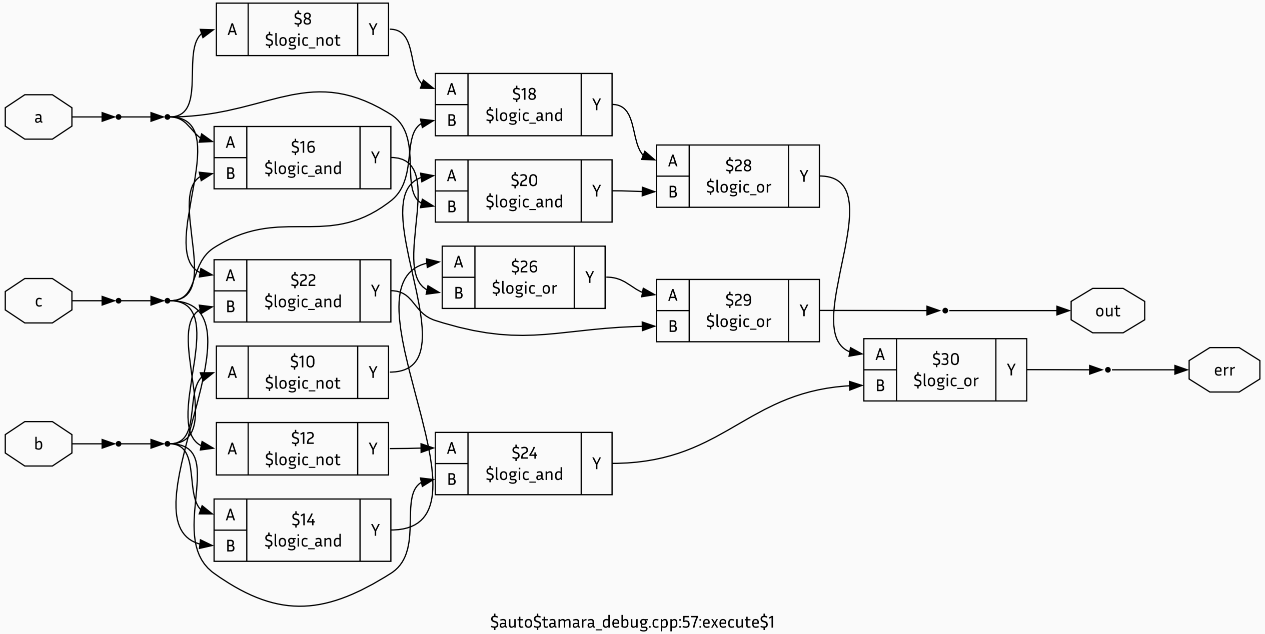
Progress: Equivalence checking

Manual design in Logisim:



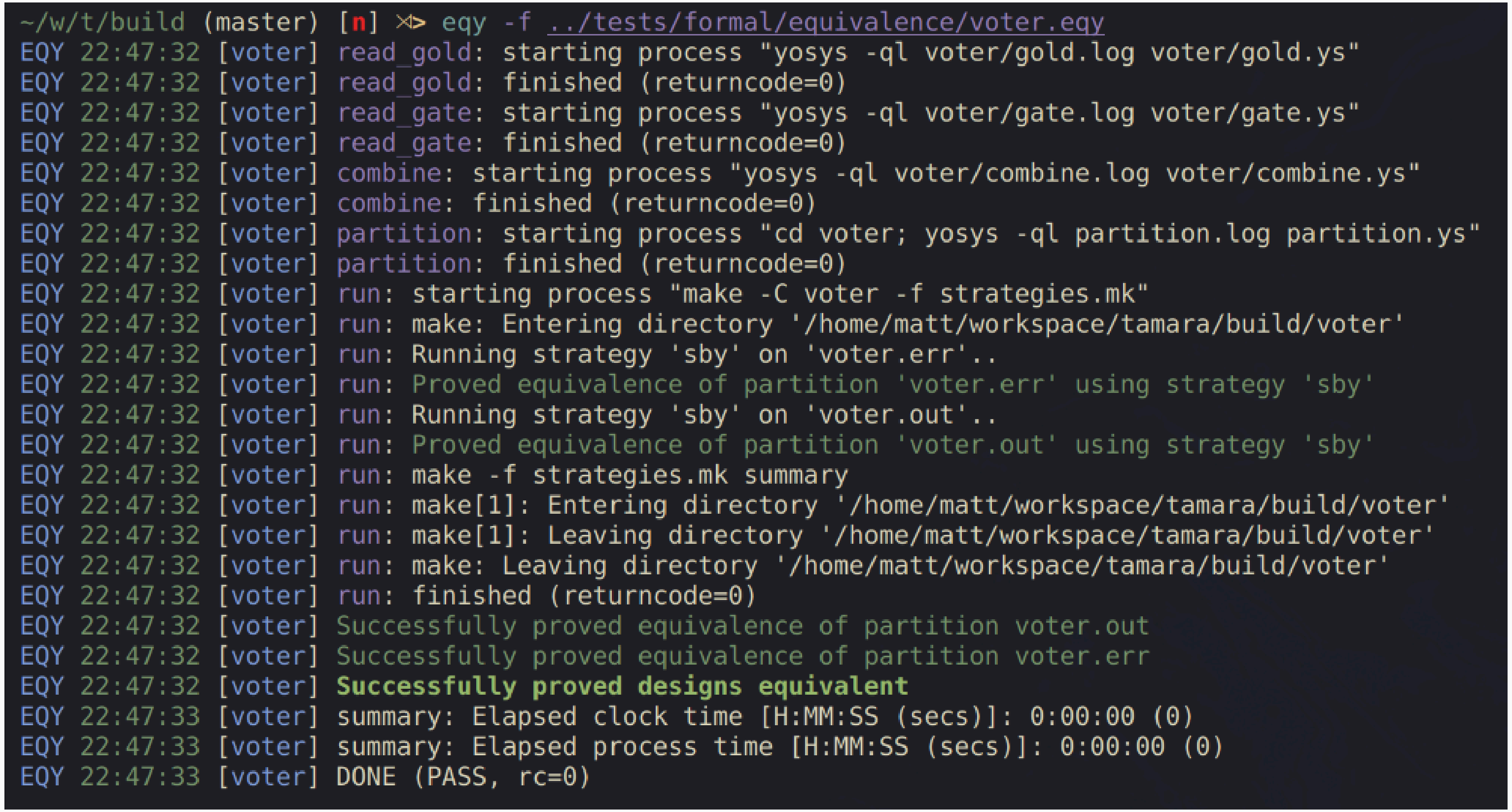
Progress: Equivalence checking

```
    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)
```

The future

I’m aiming to produce at least one academic publication from TaMaRa research.

The future

I'm aiming to produce at least one academic publication from TaMaRa research.

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 academic publication from TaMaRa research.

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
- Netlist-driven algorithm based on Johnson's work [6] (TODO NOT TRUE)
- **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] 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.

[2] 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/

[3] 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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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?