An Automated Triple Modular Redundancy EDA Flow for Yosys

by

Matthew Lawrence Young

School of Electrical Engineering and Computer
Science, The University of Queensland.

Submitted for the degree of
Bachelor of Computer Science (Honours)
in June 2025

Matthew Lawrence Young

m.young2@student.uq.edu.au

07 June 2025

Prof. Michael Brünig,

Head of School

School of Electrical Engineering and Computer Science

The University of Queensland

St Lucia QLD 4072

Dear Professor Brünig,

In accordance with the requirements of the Degree of Bachelor of Computer Science (Honours) in the School of Electrical Engineering and Computer Science, I submit the following thesis entitled

"An Automated Triple Modular Redundancy EDA Flow for Yosys"

The thesis was performed under the supervision of Associate Professor John Williams. I declare that the work submitted in the thesis is my own, except as acknowledged in the text and footnotes, and that it has not previously been submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,

Matthew Lawrence Young

cosmic rays n.

Notionally, the cause of bit rot. However, this is a semi-independent usage that may be invoked as a humorous way to handwave away any minor randomness that doesn't seem worth the bother of investigating. "Hey, Eric – I just got a burst of garbage on my tube, where did that come from?" "Cosmic rays, I guess."

Compare **sunspots**, **phase of the moon**.

— The New Hacker's Dictionary, 1991

Acknowledgements

I would like to specially and individually thank these members of the YosysHQ team for their extremely valuable help and guidance throughout the project: N. Engelhardt, George Rennie, Emil J. Tywoniak, Krystine Sherwin, Catherine "whitequark", "Anhijkt", and Jannis Harder. I would also like to extend this gratitude to all of the contributors to Yosys and the YosysHQ team - without their hard work, this thesis would not be possible. I would also like to thank my supervisor Assoc. Prof. John Williams for his excellent support and guidance during this thesis. Additionally, I would like to thank the team at Sandia National Laboratories for their interest in this project and its outcomes.

I would also like to thank my friends. Thank you for listening to me, and for those doing their theses, the best of luck in your own theses too.

Last but not least, I would especially like to thank my family for their immense support along this journey, it is invaluable to me.

Abstract

Safety-critical sectors require Application Specific Integrated Circuit (ASIC) designs and Field Programmable Gate Array (FPGA) gateware to be fault-tolerant. In particular, high-reliability computer systems used in spaceflight computing need to mitigate the effects of Single Event Upsets (SEUs) caused by ionising radiation. One common fault-tolerant design technique is Triple Modular Redundancy (TMR), which mitigates SEUs by triplicating key parts of the design and using voter circuits. Typically, this is manually implemented by designers at the Hardware Description Language (HDL) level, but this is error-prone and time-consuming. Leveraging the power and flexibility of the open-source Yosys Electronic Design Automation (EDA) tool, in this thesis, I present *TaMaRa*: a novel fully automated TMR flow, implemented as a Yosys plugin. I describe the design and implementation of the TaMaRa tool, and present extensive test results using a combination of manual tests, formal verification and RTL fuzzing techniques.

Contents

List of Figures

List of Tables

List of Code Listings

List of Abbreviations and Symbols

Abbreviation	Meaning
ASIC	Application Specific Integrated Circuit
BFS	Breadth-First Search
CI	Continuous Integration
CMOS	Complementary Metal-Oxide Semiconductor
COTS	Commercial Off The Shelf
DFF	D-Flip-Flop
EDA	Electronic Design Automation
FD-SOI	Fully Depleted Silicon-on-Insulator
FF	Flip-Flop
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
IC	Integrated Circuit
LUT	Lookup Table
MBU	Multi-Bit Upset
P&R / PnR	Place and Route
PDK	Process Design Kit
PLL	Phase Locked Loop
PPA	Power, Performance and Area
RTL	Register Transfer Level

Abbreviation	Meaning
RTLIL	RTL Intermediate Language
SEU	Single Event Upset
SRAM	Static Random Access Memory
TMR	Triple Modular Redundancy
YAML	Yet Another Markup Language

Chapter 1
Introduction

For safety-critical sectors such as aerospace, defence, and medicine, both Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Array (FPGA) gateware must be designed to be fault tolerant to prevent catastrophic malfunctions. In the context of digital electronics, *fault tolerant* means that the design is able to gracefully recover and continue operating in the event of a fault, or upset. A Single Event Upset (SEU) occurs when ionising radiation strikes a transistor on a digital circuit, causing it to transition from a 1 to a 0, or vice versa. This type of upset is most common in space, where the Earth's magnetosphere is not present to dissipate the ionising particles [1]. On an unprotected system, an unlucky SEU may corrupt the system's state to such a severe degree that it may cause destruction or loss of life - particularly important given the safety-critical nature of most space-faring systems (satellites, crew capsules, missiles, etc). Thus, fault tolerant computing is widely studied and applied to space-based computing systems.

One common fault-tolerant design technique is Triple Modular Redundancy (TMR), which mitigates SEUs by triplicating key parts of the design and using voter circuits to select a non-corrupted result if an SEU occurs (see Figure 1). Typically, TMR is manually designed at the Hardware Description Language (HDL) level, for example, by manually instantiating three copies of the target module, designing a voter circuit, and linking them all together. However, this approach is an additional time-consuming and potentially error-prone step in the already complex design pipeline.
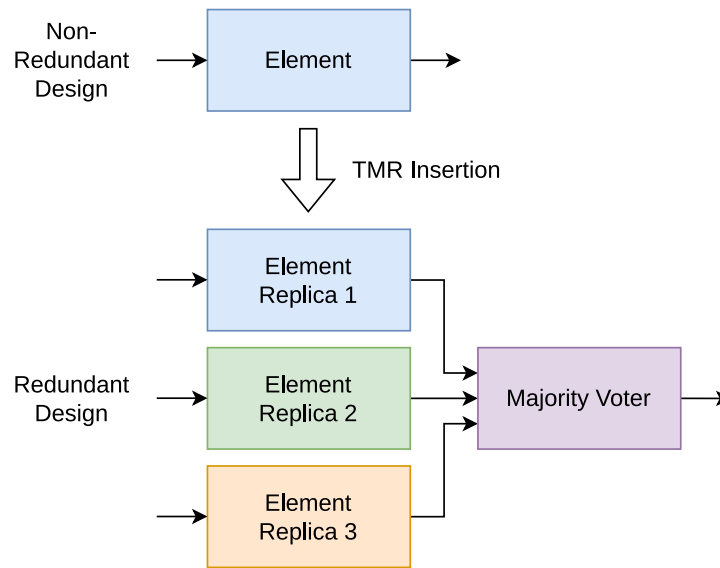
Figure 1: Diagram demonstrating how TMR is inserted into an abstract design

Modern digital ICs and FPGAs are described using Hardware Description Languages (HDLs), such as SystemVerilog or VHDL. The process of transforming this high level description into a photolithography mask (for ICs) or bitstream (for FPGAs) is achieved through the use of Electronic Design Automation (EDA) tools. This generally comprises of the following stages (Figure 2):
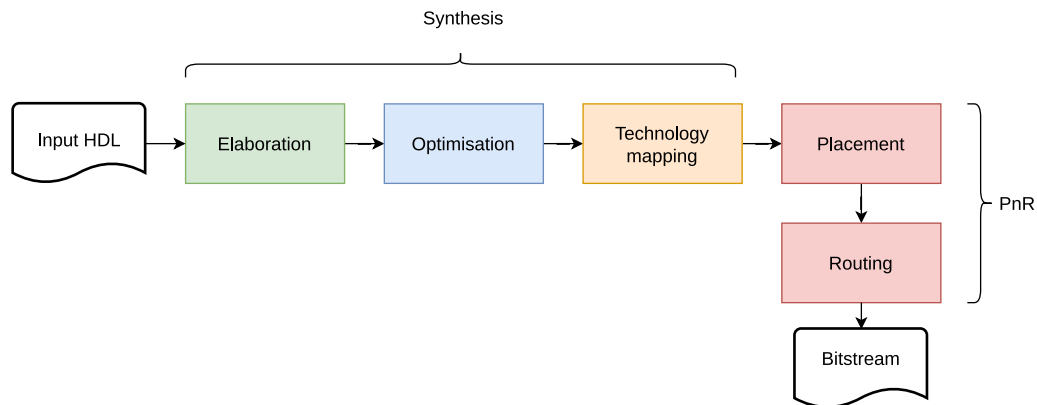
Figure 2: Simplified representation of a typical EDA synthesis flow

- **Synthesis**: The transformation of a high-level textual HDL description into a lower level synthesisable netlist.
  - ▸ **Elaboration:** Includes the instantiation of HDL modules, resolution of generic parameters and constants. Like compilers, synthesis tools are typically split into frontend/backend, and elaboration could be considered a frontend/language parsing task.

- ‣ **Optimisation:** This includes a multitude of tasks, anywhere from small peephole optimisations, to completely re-coding FSMs. In commercial tools, this is typically timing driven.
  - ‣ **Technology mapping:** This involves mapping the technology-independent netlist to the target platform, whether that be FPGA LUTs, or ASIC standard cells.
- **Placement**: The process of optimally placing the netlist onto the target device. For FPGAs, this involves choosing which logic elements to use. For digital ICs, this is much more complex and manual - usually done by dedicated layout engineers who design a *floorplan.*
- **Routing**: The process of optimally connecting all the placed logic elements (FPGAs) or standard cells (ICs).

Due to their enormous complexity and cutting-edge nature, most IC EDA tools are commercial proprietary software sold by the big three vendors: Synopsys, Cadence and Siemens. These are economically infeasible for almost all researchers, and even if they could be licenced, would not be possible to extend to implement custom synthesis passes. The major FPGA vendors, AMD and Intel, also develop their own EDA tools for each of their own devices, which are often cheaper or free. However, these tools are still proprietary software and cannot be modified by researchers. Until recently, there was no freely available, research-grade, open-source EDA tool available for study and improvement. That changed with the introduction of Yosys [2]. Yosys is a capable synthesis tool that can emit optimised netlists for various FPGA families as well as a few silicon process nodes (e.g. Skywater 130nm). When combined with the Nextpnr place and route tool [3], Yosys+Nextpnr forms a fully end-to-end FPGA synthesis flow for Lattice iCE40 and ECP5 devices. Importantly, for this thesis, Yosys can be modified either by changing the source code or by developing modular plugins that can be dynamically loaded at runtime.

This thesis will focus on the design and implementation of *TaMaRa*, an automated Triple Modular Redundancy EDA flow for Yosys. In Chapter 2, I present a comprehensive literature review of prior TMR algorithms by classifying them into dichotomy of either netlist-level or design-level approaches. I evaluate the strengths and weaknesses of each approach, and determine how it will shape the TaMaRa algorithm. In Chapter 3, I introduce the TaMaRa algorithm, and demonstrate how it was implemented as a Yosys plugin. Finally, in Chapter 4, I show the results of the TaMaRa algorithm for a number of real-world circuits and measure its ability to mitigate SEUs.

1.1. Yosys internals

Yosys supports dynamically loading plugins at runtime. These plugins are compiled against the Yosys codebase, and are compiled into Unix shared objects (.so files). This allows users to define and register custom passes and frontends within the main Yosys application, without having to trouble the upstream maintainers with the maintenance of new code. This is precisely why the Yosys authors advised TaMaRa to be implemented

as a Yosys plugin, rather than as an upstream contribution [4]. This plugin system is a unique and powerful part of Yosys, and one of the main advantages of the tool being open-source. End users are free to design and implement their own plugins, under their own choice of licence, to extend Yosys in any way they see fit. Comparatively, proprietary tools are limited to rather simple Tcl scripting, as further modification is locked away behind complex intellectual property (IP) rights, including patents, and non-disclosure agreements (NDAs).

Yosys uses *frontends* to read various Register Transfer Languages (RTLs), such as Verilog or SystemVerilog. Using the typical combination of a lexer and parser, Yosys transforms RTL source code into an Abstract Syntax Tree (AST), and then into an intermediate language called RTL Intermediate Language (RTLIL). RTLIL is, in essence, a model of a given circuit's netlist at various different levels of abstraction. RTLIL can be used anywhere from a near direct 1:1 mapping with the original RTL, complete with processes and annotations, all the way down to very low-level FPGA/ASIC specific primitives. The typical Yosys flow is to use its extensive set of built-in commands and synthesis scripts to process a circuit from this high-level abstraction, down to device-specific low-level primitives; for example, ASIC standard cells for a given process node. This is virtually identical to the flow used by commercial tools such as Synopsys' Design Compiler or Xilinx's Vivado.

TaMaRa operates at the netlist level, which in the context of Yosys means operating on RTLIL circuits. Internally, RTLIL is implemented as a set of C++ classes that describe the general structure of a netlist as wires (`RTLIL::Wire`) and cells (`RTLIL::Cell`). Wires may potentially be multi-bit, which introduces a challenge as TaMaRa voters are single-bit. Groups of wires and cells can be bundled into a module (`RTLIL::Module`). Modules are arranged in a tree structure, where the root of the tree is known as the top module. RTLIL is one of the most important and powerful parts of Yosys, because it allows plugins like TaMaRa to operate on a common intermediate representation of a circuit netlist, irrespective of the user's choice of input RTL language and/or output type. This means that TaMaRa can operate exactly the same for a user using VHDL for a Lattice iCE40 FPGA, and a user using Verilog for a Skywater 130 nm ASIC. RTLIL's importance is further elaborated on by Wolf [2] and Shah et al. [3]. In essence, the unique combination of Yosys being open-source, its runtime plugin system, and its RTL Intermediate Language is what makes Yosys the ideal platform to develop the TaMaRa algorithm for.

Chapter 2
Literature review

2.1. Introduction, methodology and terminology

The automation of triple modular redundancy, as well as associated topics such as general fault-tolerant computing methods and the effects of SEUs on digital hardware, have been studied a fair amount in academic literature. Several authors have invented a number of approaches to automate TMR, at various levels of granularity, and at various points in the FPGA/ASIC synthesis pipeline. This presents an interesting challenge to systematically review and categorise. To address this, I propose that all automated TMR approaches can be fundamentally categorised into the following dichotomy:

- **Design-level approaches** ("thinking in terms of HDL"): These approaches treat the design as *modules*, and introduce TMR by replicating these modules. A module could be anything from an entire CPU, to a register file, to even a single combinational circuit or AND gate. Once the modules are replicated, voters are inserted.
- **Netlist-level approaches** ("thinking in terms of circuits"): These approaches treat the design as a *circuit* or *netlist*, which is internally represented as a graph. TMR is introduced using graph theory algorithms to *cut* the graph in a particular way and insert voters.

Using these two design paradigms as a guiding point, I analyse the literature on automated TMR, as well as background literature on fault-tolerant computing.

2.2. Fault tolerant computing and redundancy

The application of triple modular redundancy to computer systems was first introduced into academia by Lyons and Vanderkul [5] in 1962. Like much of computer science, however, the authors trace the original concept back to John von Neumann. In addition to introducing the application of TMR to computer systems, the authors also provide a rigorous Monte-Carlo mathematical analysis of the reliability of TMR. One important takeaway from this is that the only way to make a system reliably redundant is to split it into multiple components, each of which is more reliable than the system as a whole. In the modern FPGA concept, this implies applying TMR at an Register Transfer Level (RTL) module level, although as we will soon see, more optimal and finer grained TMR can be applied. Although their Monte Carlo analysis shows that TMR dramatically improves reliability, they importantly show that as the number of modules M in the computer system increases, the computer will eventually become less reliable. This is due to the fact that the voter circuits may not themselves be perfectly reliable, and is important to note for FPGA and ASIC designs which may instantiate hundreds or potentially thousands of modules.

Instead of triple modular redundancy, ASICs can be designed using rad-hardened CMOS process nodes or design techniques. Much has been written about rad-hardened microprocessors, of which many are deployed (and continue to be deployed) in space to this day. One such example is the RAD750 [6], a rad-hardened PowerPC CPU for space applications designed by Berger et al. of BAE Systems. They claim "5-6 orders of magnitude" better SEU performance compared to a stock PowerPC 750 under intense radiation conditions. The processor is manufactured on a six-layer commercial 250 nm process node, using specialty design considerations for the RAM, PLLs, and standard cell libraries. Despite using special design techniques, the process node itself is standard commercial CMOS node and is not inherently rad-hardened. The authors particularly draw attention to the development of a special SEU-hardened RAM cell, although unfortunately they do not elaborate on the exact implementation method used. However, they do mention that these techniques increase the die area from 67 mm² in a standard PowerPC 750, to 120 mm² in the RAD750, a ~1.7x increase. Berger et al. also used an extensive verification methodology, including the formal verification of the gate-level netlist and functional VHDL simulation. The RAD750 has been deployed on numerous high-profile missions including the James Webb Space Telescope and Curiosity Mars rover. Despite its wide utilisation, however, the RAD750 remains extremely expensive - costing over $200,000 USD in 2021 [7]. This makes it well out of the reach of research groups, and possibly even difficult to acquire for space agencies like NASA.

In addition to commercial CMOS process nodes, there are also specialty rad-hardened process nodes designed by several fabs. One such example is Skywater Technologies' 90 nm FD-SOI ("Fully Depleted Silicon-On-Insulator") node. The FD-SOI process, in particular, has been shown to have inherent resistance to SEUs and ionising radiation due to its top thin silicon film and buried insulating oxide layer [8]. Despite this, unfortunately, FD-SOI is an advanced process node that is often expensive to manufacture.

Instead of the above, with a sufficiently reliable TMR technique (that this research ideally would like to help create), it should theoretically be possible to use a commercial-off-the-shelf (COTS) FPGA for mission critical space systems, reducing cost enormously - this is one of the key value propositions of automated TMR research. Of course, TMR is not flawless: its well-known limitations in power, performance and area (PPA) have been documented extensively in the literature, particularly by Johnson [9], [10]. Despite this, TMR does have the advantage of being more general purpose and cost-effective than a specially designed ASIC like the RAD750. TMR can be applied to any design, FPGA or ASIC, at various different levels of granularity and hierarchy, allowing for studies of different trade-offs. For ASICs in particular, unlike the RAD750, TMR as a design technique does not need to be specially ported to new process nodes: an automated TMR approach could be applied to a circuit on a 250 nm or 25 nm process node without any major design changes. Nonetheless, specialty rad-hardened ASICs will likely to see future use in space applications. In fact, it's entirely possible that a rad-hardened FPGA *in combination* with an automated TMR technique is the best way of ensuring reliability.

2.3. Netlist-level approaches

Recognising that prior literature focused mostly around manual or theoretical TMR, and the limitations of a manual approach, Johnson and Wirthlin [9] introduced four algorithms for the automatic insertion of TMR voters in a circuit, with a particular focus on timing and area trade-offs. Together with the thesis this paper was based on [10], these two publications form the seminal works on automated TMR for digital EDA. Johnson's algorithm operates on a post-synthesis netlist before technology mapping. First, he creates three copies of the original circuit, then triplicates component instantiations and wire nets, and finally connects the nets in such a way that the behaviour of the original circuit is preserved. This is described as the "easy part" of TMR - the more complex step is selecting both a valid *and* optimal placement for majority voters. Johnson identifies four main classes of voters. Note that in this section, "SRAM scrubbing" refers to Johnson's method of dynamic runtime reconfiguration of the FPGA configuration SRAM to correct SEUs.

1. **Reducing voters**: Combines the output from three TMR replicas into a single output, in other words, a single majority voter. Used on circuit outputs.
2. **Partitioning voters**: Used to increase reliability within a circuit by partitioning it, and applying TMR separately to each partition. Johnson states that if only reducing voters were used in a circuit, errors would be masked from SRAM scrubbing as long as they only occur in one replica at a time. In addition, multiple SEUs in close proximity can prevent the TMR redundancy from working correctly. Partitioning voters have the benefit of dividing the circuit into independent partitions that can tolerate SEUs independently. One important takeaway that Johnson mentions is that there is an optimal balance between the number of partitions, which increases the likelihood of separate SEUs affecting multiple partitions, and having *too many* partitions which reduces reliability due to the voters being affected. This relates to the early research conducted by Lyons and Vanderkul [5].
3. **Clock domain crossing voters**: These are used due to the special considerations when TMR circuits cross multiple clock domains. In particular, metastability effects are a serious consideration for clock domain crossing voters. Johnson implements this type of voter using a small train of consecutive flip-flops to attempt to reduce the probability of metastable values propagating. However, for TaMaRa, due to the very tight time constraints of an Honours thesis, we will likely not consider multiple clock domains, and thus metastable voters will not be required.
4. **Synchronisation voters**: These are required when SRAM scrubbing is used with TMR that includes sequential logic (i.e. FFs, so most designs). These are meant to restore correct register state after FPGAs are repaired by SRAM scrubbing. Again due to time constraints and the vendor-specific nature of the process, TaMaRa will leave SRAM scrubbing up to the end user, using the provided error signal from the majority voters. Rather than supporting dynamic SRAM scrubbing (as in Bridford et al. [11]), we will suggest users simply reset the device when a fault is detected.

One other consideration that Johnson takes into account is illegal or undesirable voter cuts (a "voter cut" is his terminology for splicing a netlist and inserting a majority voter). He notes that some netlist cuts are illegal, for example, some Xilinx FPGAs do not support configurable routing between different types of primitives, particularly DSP primitives. He also very interestingly notes that there are undesirable, but not strictly illegal cuts that may be performed. These would, for example, break high speed carry chains on Xilinx devices and impact performance. This is a very interesting observation, as it implies the possibility of a placement/routing aware TMR algorithm. This is a fascinating topic for future research, but time does not permit its implementation in TaMaRa. Instead, TaMaRa will likely leave design legalisation to Nextpnr and not strictly consider performance when inserting voters. The majority of Johnson's work, and the complexity he describes, concerns the insertion of synchronisation voters using graph theory algorithms such as Strongly Connected Components (SCC). For TaMaRa, we declared that we do not need synchronisation voters, as we do not perform dynamic SRAM scrubbing, instead fully rebooting the device if we detect a fault. This should mean that TaMaRa is a lot easier to implement. Nonetheless, however, I believe it may be possible to use some of Johnson's SCC algorithm to elegantly decompose a netlist into partitions, and insert partition voters. We will most likely insert reducing voters and partitioning voters, and if time permits, clock domain crossing voters as well.

Whilst they provide an excellent design of TMR insertion algorithms, and a very thorough analysis of their area and timing trade-offs, Johnson and Wirthlin do not have a rigorous analysis of the correctness of these algorithms. They produce experimental results demonstrating the timing and area trade-offs of the TMR algorithms on a real Xilinx Virtix 1000 FPGA, up to the point of P&R, but do not run it on a real device. More importantly, they also do not have any formal verification or simulated SEU scenarios to prove that the algorithms both work as intended, and keep the underlying behaviour of the circuit the same. Finally, in his thesis [10], Johnson states that the benchmark designs were synthesised using a combination of the commercial Synopsys Synplify tool, and the *BYU-LANL Triple Modular Redundancy (BL-TMR) Tool*. This Java-based set of tools ingest EDIF-format netlists, perform TMR on them, and write the processed result to a new EDIF netlist, which can be re-ingested by the synthesis program for place and route. This is quite a complex process, and was also designed before Yosys was introduced in 2013. It would be better if the TMR pass was instead integrated directly into the synthesis tool - which is only possible for Yosys, as Synplify is commercial proprietary software. This is especially important for industry users who often have long and complicated synthesis flows.

Later, Skouson et al. [12] (from the same lab as above) introduced SpyDrNet, a Python-based netlist transformation tool that also implements TMR using the same algorithm as above. SpyDrNet is a great general purpose transformation tool for research purposes, but again is a separate tool that is not integrated *directly* into the synthesis process. I instead aim to make a *production* ready tool, with a focus on ease-of-use, correctness and performance.

Using a similar approach, Benites and Kastensmidt [13], and Benites' thesis [14], introduce an automated TMR approach implemented as a Tcl script for use in Cadence tools. They distinguish between "coarse grained TMR" (which they call "CGTMR"), applied at the RTL module level, and "fine grained TMR" (which they call "FGTMR"), applied at the sub-module (i.e. net) level. Building on that, they develop an approach that replicates both combinational and sequential circuits, which they call "fine grain distributed TMR" or "FGDTMR". They split their TMR pipeline into three stages: implementation ("TMRi"), optimisation ("TMRo"), and verification ("TMRv"). The implementation stage works by creating a new design to house the TMR design (which I'll call the "container design"), and instantiating copies of the original circuit in the container design. Depending on which mode the user selects, the authors state that either each "sequential gate" will be replaced by three copies and a voter, or "triplicated voters" will be inserted. What happens in the optimisation stage is not clear as Benites does not elaborate at all, but he does state it's only relevant for ASICs and involves "gate sizing". For verification, Benites uses a combination of fault-injection simulation (where SEUs are intentionally injected into the simulation), and formal verification through equivalence checking. Equivalence checking involves the use of Boolean satisfiability solvers ("SAT solvers") to mathematically prove one circuit is equivalent to another. Benites' key verification contribution is identifying a more optimal way to use equivalence checking to verify fine-grained TMR. He identified that each combinational logic path will be composed of a path identical to the original logic, plus one or more voters. This way, he only has to prove that each "logic cone" as he describes it is equivalent to the original circuit. Later on, he also uses a more broad-phase equivalence checking system to prove that the circuits pre and post-TMR have the same behaviours.

One of the most important takeaways from these works are related to clock synchronisation. Benites interestingly chooses to not replicate clocks or asynchronous reset lines, which he states is due to clock skew and challenges with multiple clock domains created by the redundancy. Due to the clear challenges involved, ignoring clocks and asynchronous resets is a reasonable limitation introduced by the authors, and potentially reasonable for us to introduce as well. Nonetheless, it is a limitation I would like to address in TaMaRa if possible, since leaving these elements unprotected creates a serious hole that would likely preclude its real-world usage[1]. Arguably, the most important takeaway from Benites' work is the use of equivalence checking in the TMR verification stage. This is especially important since Johnson [9] did not formally verify his approach. Benites' usage of formal verification, in particular, equivalence checking, is an excellent starting point to design the verification methodology for TaMaRa.

Xilinx, the largest designer of FPGAs, also has a netlist-level TMR software package known as TMRTool [15]. This implements a Xilinx proprietary algorithm known as

---

[1]My view is essentially that an unprotected circuit remains unprotected, regardless of how difficult it is to correct clock skewing. In other words, simply saying that the clock skew exists doesn't magically resolve the issue. In Honours, we are severely time limited, but it's still my goal to address this limitation if possible.

XTMR, which differs from traditional TMR approaches in that it also aims to correct faults introduced into the circuit by SEUs (rather than just masking their existence). Xilinx also aims to address single-event transients ("SETs"), where ionising radiation causes voltage spikes on the FPGA routing fabric. TMRTool follows a similar approach to the other netlist-level algorithms described above, with some small improvements and Xilinx-specific features. The flow first triplicates all inputs, combinational logic and routing. Then, it inserts voters downstream in the circuit, particularly on finite state machine (FSM) feedback paths. One important difference is that, at this point in the flow, Xilinx also decides to triplicate the voters themselves. This means there is no single point of failure (which improves redundancy), although it has a higher area cost than approaches that do not triplicate voters. In addition, TMRTool is designed to be used with configuration scrubbing. Xilinx has much further research on FPGA configuration scrubbing [11]. The two main approaches are either a full reboot, or a partial runtime reconfiguration. Since the FPGA configuration is stored in an SRAM that's read at boot-up, a full reboot will naturally reconfigure the device, and thus correct any logic/ routing issues caused by SEUs. However, a more efficient solution is only re-flashing the sectors of the FPGA that are known to be affected by SEUs. This is known as partial runtime reconfiguration. Unfortunately, as noted in the Xilinx documentation, this partial reconfiguration is a vendor-specific process. It would not be possible to design a multi-vendor runtime reconfiguration approach, and worse still, much of this specification is still undocumented and proprietary, precluding its integration with Yosys or Nextpnr. Despite this, we could provide end-users with an error signal from the majority voter, which could be used to one form of reconfiguration if desired. The two most relevant components of TMRTool to the TaMaRa algorithm are its consideration of feedback paths for FSMs, and its consideration of redundant clock domains. Both of these considerations are mentioned in the other netlist-level approaches, but it seems to occupy a considerable amount of engineering time and effort for Xilinx, and thus can be expected to be a significant issue for TaMaRa as well. TMRTool's FSM feedback is important to ensure the synchronisation of triplicated redundant FSMs, but unfortunately requires manual verification in some cases to ensure Xilinx's synthesis has not introduced problems to the design. Finally, TMRTool also has a very flexible architecture. The implementation strategy can be customised to various different approaches. Most are Xilinx-specific, but two relevant ones to TaMaRa are "Standard" and "Don't Touch". "Standard" works by triplicating the underlying FPGA primitives and inserting voters, as usual. "Don't Touch", however, is important to be added to FPGA primitives that cannot be replicated, and avoids TMR entirely. This would be very beneficial to add as an option to the TaMaRa algorithm.

On the lower level side, Hindman et al. [16] introduce an ASIC standard-cell based automated TMR approach. When digital circuits are synthesised into ASICs, they are technology mapped onto standard cells provided by the foundry as part of their Process Design Kit (PDK). For example, SkyWater Technology provides an open-source 130 nm ASIC PDK, which contains standard cells for NAND gates, muxes and more [17]. The

authors design a TMR flip-flop cell, known as a "Triple Redundant Self Correcting Master-Slave Flip-Flop" (TRSCMSFF), that mitigates SEUs at the implementation level. Since this is so low level and operates below the entire synthesis/place and route pipeline, their approach has the advantage that *any* design - including proprietary encrypted IP cores that are (unfortunately) common in industry - can be made redundant. Very importantly, the original design need not be aware of the TMR implementation, so this approach fulfills my goal of making TMR available seamlessly to designers. The authors demonstrate that the TRSCMSFF cell adds minimal overhead to logic speed and power consumption, and even perform a real-life radiation test under a high energy ion beam. Overall, this is an excellent approach for ASICs. However, this approach, being standard-cell specific, cannot be applied to FPGA designs. Rather, the FPGA manufacturers themselves would need to apply this method to make a series of specially rad-hardened devices. It would also appear that designers would have to port the TRSCMSFF cell to each fab and process node they intend to target. While TaMaRa will have worse power, performance and area (PPA) trade-offs on ASICs than this method, it is also more general in that it can target FPGAs *and* ASICs due to being integrated directly into Yosys. Nevertheless, it would appear that for the specific case of targeting the best PPA trade-offs for TMR on ASICs, the approach described in [16] is the most optimal one available.

2.4. Design-level approaches

Several authors have investigated applying TMR directly to HDL source code. One of the most notable examples was introduced by Kulis [18], through a tool he calls "TMRG". TMRG operates on Verilog RTL by implementing the majority of a Verilog parser and elaborator from scratch. It takes as input Verilog RTL, as well as a selection of Verilog source comments that act as annotations to guide the tool on its behaviour. In turn, the tool modifies the design code and outputs processed Verilog RTL that implements TMR, as well as Synopsys Design Compiler design constraints. Like the goal of TaMaRa, the TMRG approach is designed to target both FPGAs and ASICs, and for FPGAs, Kulis correctly identifies the issue that not all FPGA blocks can be replicated. For example, a design that instantiates a PLL clock divider on an FPGA that only contains one PLL could not be replicated. Kulis also correctly identifies that optimisation-driven synthesis tools such as Yosys and Synopsys DC will eliminate TMR logic as part of the synthesis pipeline, as the redundancy is, by nature, redundant and subject to removal. In Yosys, this occurs specifically in the `opt_share` and `opt_clean` passes according to specific advice from the development team [4]. However, unlike Synopsys DC, Yosys is not constraint driven, which means that Kulis' constraint-based approach to preserving TMR logic through optimisation would not work in this case. Finally, since TMRG re-implements the majority of a synthesis tool's frontend (including the parser and elaborator), it is limited to only supporting Verilog. Yosys natively supports Verilog and some SystemVerilog, with plugins [19] providing more complete SV and VHDL support. Since TaMaRa uses Yosys' existing frontend, it should be more reliable and useable with many more HDLs.

Lee et al. [20] present "TLegUp", an extension to the prior "LegUp" High Level Synthesis (HLS) tool. As stated earlier in this document, modern FPGAs and ASICs are

typically designed using Hardware Description Languages (HDLs). HLS is an alternative approach that aims to synthesise FPGA/ASIC designs from high-level software source code, typically the C or C++ programming languages. On the background of TMR in FPGAs in general, the authors identify the necessity of "configuration scrubbing", that is, the FPGA reconfiguring itself when one of the TMR voters detects a fault. Neither their TLegUp nor our TaMaRa will address this directly, instead, it's usually best left up to the FPGA vendors themselves (additionally, TaMaRa targets ASICs which cannot be runtime reconfigured). Using voter insertion algorithms inspired by Johnson [9], the authors operate on LLVM Intermediate Representation (IR) code generated by the Clang compiler. By inserting voters before both the HLS and RTL synthesis processes have been run, cleverly the LegUp HLS system will account for the critical path delays introduced by the TMR process. This means that, in addition to performance benefits, pipelined TMR voters can be inserted. The authors also identify four major locations to insert voters: feedback paths from the datapath, FSMs, memory output signals and output ports on the top module. Although TaMaRa isn't HLS-based, Yosys does have the ability to detect abstract features like FSMs, so we could potentially follow this methodology as well. The authors also perform functional simulation using Xilinx ISE, and a real-world simulation by using a Microblaze soft core to inject faults into various designs. They state TLegUp reduces the error rate by 9x, which could be further improved by using better placement algorithms. Despite the productivity gains, and in this case the benefits of pipelined voters, HLS does not come without its own issues. Lahti et al. [21] note that the quality of HLS results continues to be worse than those designed with RTL, and HLS generally does not see widespread industry usage in production designs. One other key limitation that Lee et al. do not fully address is the synthesis optimisation process incorrectly removing the redundant TMR logic. Their workaround is to disable "resource sharing options in Xilinx ISE to prevent sub-expression elimination", but ideally we would not have to disable such a critical optimisation step just to implement TMR. TaMaRa aims to address this limitation by working with Yosys directly.

2.5. TMR verification

While Benites [13], [14] discusses verification of the automated TMR process, and other authors [6], [16], [20] also use various different verification/testing methodologies, there is also some literature that exclusively focuses on the verification aspect. Verification is one of the most important parts of this process due to the safety-critical nature of the devices TMR is typically deployed to. Additionally, there are interesting trade-offs between different verification methodologies, particularly fault injection vs. formal verification.

Beltrame [22] uses a divide and conquer approach for TMR netlist verification. Specifically, identifying limitations with prior fault-injection simulation and formal verification techniques, he presents an approach described as fault injection combined with formal verification: instead of simulating the entire netlist with timing accurate simulation, he uses a behavioural timeless simulation of small submodules ("logic cones") extracted by automatic analysis. The algorithm then has three main phases:

1. Triplet identification: Determine all the FF (flip-flop) triplets present in each logic cone.
2. TMR structure analysis: Perform exhaustive fault injection on valid configurations.
3. Clock and reset tree verification: Assure that no FF triplets have common clock or set/reset lines.

This seems to be an effective and rigorous approach, as Beltrame mentions he was able to find TMR bugs in the radiation-hardened netlist of a LEON3 processor. Importantly, the code for the tool appears is available on GitHub as *InFault*. It would be highly worthwhile investigating the use of this tool for verification, as it has already been proven in prior research and may overall save time. That being said, a quick analysis of the code appears to reveal it to be "research quality" (i.e. zero documentation and seems partially unfinished). Another problematic issue is that the code does not seem to readily compile under a modern version of GCC or Clang, and would require manual fixing in order to do so. Finally, the *InFault* tool implements a custom Verilog frontend for reading designs. This has the exact same problem as Kulis' [18] custom Verilog frontend: it's not clear to what standard this is implemented. We may have to write a custom RTLIL or EDIF frontend to ingest Yosys netlists. The main question is whether resolving these issues would take more time than implementing formal verification ourselves in Yosys. One other important limitation not yet mentioned in Beltrame's approach is the presence of false positives. Beltrame's "splitting algorithm" requires a tunable threshold which, if set too low, may cause false positive detections of invalid TMR FFs. These false positives require manual inspection of the netlist graph in order to understand. This is extremely problematic for large designs, as it would seem to require many laborious hours from an engineer familiar with the *InFault* algorithm to determine if any given detection was a false positive or not. It's also not immediately clear what the range of suitable thresholds for this value are that would prevent or possibly eliminate false positives.

Even if we do not end up using Beltrame's [22] approach in its entirety (for example, if the false positives are a significant issue or if it's too much work to read Yosys designs), we may nonetheless be able to repurpose parts of his work for TaMaRa. One aspect that would work particularly well inside of Yosys itself is step 3 from the algorithm, clock and reset tree verification. Yosys already has tools to identify clock and reset lines, so it should not be too much extra work to build a pass that verifies the clock and resets in the netlist are suitable for TMR. In addition, parts of Beltrame's algorithm may be implementable using other Yosys formal verification tools, particularly SymbiYosys. His terminology as well, particularly the use of "logic cones", will likely be critical in the development of TaMaRa.

2.6. Formal verification

Formal verification is increasingly being pursued in the development of FPGAs and ASICs as part of a comprehensive design verification methodology. The foundations for the formal verification of digital circuits extend back to traditional Boolean algebra and set theory in discrete mathematics. Building on these foundations, digital circuit

verification can be represented as a Boolean satisfiability ("SAT") problem. The SAT problem asks us to prove whether or not there is a consistent assignment of input variables to a circuit to make the circuit evaluate to *true*. This forms the basic primitive, the underlying problem to solve, for nearly all circuit formal verification tasks; including both formal property verification and equivalence checking. Despite SAT's usefulness, Karp [23] proved via the Cook-Levin theorem that it is an NP-complete problem (i.e. there is likely no polynomial time solution). Despite this, there exist a number of fast-enough SAT solvers [24], [25], that make the verification of Boolean circuits using SAT a tractable problem.

However, on large and complex designs, using SAT solvers directly on multi-bit buses can be slow. Instead, Satisfiability Modulo Theories (SMT) solvers can be used instead. SMT is a generalisation of SAT that introduces richer types such as bit vectors, integers, and reals [26]. Solving satisfiability modulo theories is still at least NP-complete, sometimes undecidable. Most SMT solvers either depend on or "call out" to an underlying SAT solver. One such SMT solver that uses this approach is Bitwuzla [27]. Others, however, such as Z3 [28] include their own SAT logic and other methods for computing solutions. The speed of SMT solvers is very important when performing formal verification of digital circuits, and there is a yearly SMT solving competition to encourage the development and analysis of high-performance SMT solvers [29].

Formal equivalence checking uses formal techniques to verify that two circuits are equivalent in functionality. Typically, circuits are partitioned into the *gate* circuit, which is the circuit to be verified, and the *gold* circuit, which is the reference model to be proved against. Equivalence is proved by constructing a Miter circuit, which is a type of circuit that uses the exclusive-OR (XOR) on the output of the gate and gold circuits. If the outputs of these XOR cells can be proven to be true, then the circuits are not equivalent [30]. In Yosys, this is achieved by using the `miter` command to automatically construct the equivalence circuit and insert formal assertions, which are then proved or disproved using an internal SAT solver via the `sat` command.

2.7. RTL fuzzing

In the software world, "fuzzing" refers to a process of randomly generating inputs designed to induce problematic behaviour in programs. Typically, fuzzing is started by referencing an initial corpus, and the program under test is then instrumented to add control flow tracking code. The goal of the fuzzer is to generate inputs such that the program reaches 100% instrumented branch coverage once the fuzzing process is completed.

While fuzzing is typically started from an initial corpus, there has also been interest in fuzzing languages directly without any initial examples, using information from the language's grammar. One example is Holler's LangFuzz [31], which uses a tree formed by the JavaScript grammar to generate random, but valid, JavaScript code. Mozilla developers have used LangFuzz successfully to find numerous bugs in their SpiderMonkey JavaScript engine. Generating code from the grammar directly also has the advantage

of making the fuzzing process significantly more efficient, as the fuzzer tool has the *a priori* knowledge necessary to "understand" the language. Compared to using a general purpose random fuzzer that typically generates and mutates test cases on a byte-by-byte basis [32], grammar fuzzers should be able to get significantly higher coverage of a target program much more efficiently.

Although these techniques are typically used for software projects, they can also be useful for hardware, given that Verilog is more or less just another programming language. RTL fuzzing is an emerging technique that can be used to generate large-scale coverage of Verilog design files for EDA tools. Herklotz [33] describes "Verismith", a tool capable of generating random and correct Verilog RTL. This is useful for TaMaRa verification, because it allows us to investigate *en masse* whether the tool changes the behaviour of the underlying circuit. It also allows us to quickly find, reproduce, minimise and fix challenging designs, which should hopefully lead to a more reliable algorithm with better coverage of industry-standard designs.

Chapter 3
Methodology

3.1. Concept

In the previous Chapter 2, I presented a comprehensive literature review of existing automated TMR approaches. One of the main limitations that these algorithms have is that none are specifically integrated into the Yosys synthesis tool. I envision TaMaRa as a platform that provides a baseline TMR implementation that other researchers can extend upon, and that industry users can experiment with, all the while supporting both FPGAs/ASICs and being fully integrated as part of a widely used open-source EDA synthesis tool. Operating directly on Yosys' RTLIL intermediate representation ensures that any future optimisations Yosys gains, or any languages it supports in future, can be immediately also supported by TaMaRa.

To design the TaMaRa algorithm, I synthesise existing approaches from the literature review to form a novel approach suitable for implementation in Yosys. Specifically, I synthesise the voter insertion algorithms of Johnson [9], the RTL annotation-driven approach of Kulis [18], and parts of the verification methodology of Benites [13] and Beltrame [22], to form the TaMaRa algorithm and verification methodology. Based on the dichotomy identified in Chapter 2.1, TaMaRa will be classified as a *netlist-level* approach, as the algorithms are designed by treating the design as a circuit (rather than HDL).

I propose a modification to the synthesis flow that inserts TaMaRa before technology mapping. This means that the circuit can be processed at a low level, with fewer concerns about optimisation removing the redundant TMR logic, as has been observed in other approaches [20] and through conversation with the Yosys developers [4]. However, some Yosys synthesis scripts do perform additional optimisation *after* technology mapping, which again risks the removal of the TMR logic. Yet, we also cannot operate after technology mapping, since TaMaRa voter circuits are described using relatively high level circuit primitives (AND gates, NOT gates, etc) instead of vendor-specific FPGA primitives like LUTs. The best solution to this involves an upstream modification to Yosys that allows for certain optimisation passes to be selectively skipped; this is further discussed in Chapter 5.1. Whilst this would effectively resolve the problem, it would require collaboration with the upstream Yosys maintainers.

Whilst TaMaRa aims to be compatible with all existing designs with minimal changes, some preconditions are necessary for the algorithm to process the circuit correctly.

Since the algorithm is intended to work with all possible circuits, it cannot predict what the end user wants to do with the voter error signal (if anything). As discussed in the literature review, the typical use case for the error signal is to perform configuration scrubbing when upsets are detected. This, however, is a highly vendor-specific process for FPGAs, and is not at all possible on ASICs. To solve this problem, TaMaRa does not aim to provide configuration scrubbing directly, instead leaving this for the end user. Instead, the end user can attach an HDL annotation to indicate an output port on a module that TaMaRa should wire a global voter error signal to. In SystemVerilog, this uses the `(* tamara_error_sink *)` annotation, as shown in Listing 1:

```
module my_module(
    input logic a,
    (* tamara_error_sink *)
    output logic err
);
```

Listing 1: Usage of the `(* tamara_error_sink *)` annotation

End users are then free to implement configuration scrubbing using the tool and methodology appropriate to their platform.

Additionally, while TaMaRa aims to require minimal or no changes to the circuit itself, there are changes necessary to the synthesis pipeline. Unlike in normal Yosys synthesis scripts, the design cannot be lowered directly to FPGA/ASIC primitives (LUTs, standard cells, etc). It first needs to be lowered to abstract logic primitives (AND gates, NOT gates, etc) that TaMaRa can process, particularly, that it can generate voter circuits in. Then, TaMaRa can be run, after which the design can be lowered to FPGA primitives or ASIC standard cells. TaMaRa currently also requires the user to run the `splitcells` and `splitnets` commands before it is invoked to split apart multi-bit buses and cells, which are not yet directly supported. These additional requirements may introduce a slight area overhead, but it would be possible to eliminate them in the future through better algorithm design.

3.2. Implementation

Over the course of this thesis, TaMaRa was successfully written from the ground up as a Yosys plugin. This plugin consists of around 2,300 lines of C++20, and introduces one new command to Yosys: `tamara_tmr`.

TaMaRa is currently designed to only operate on one module, that being the top module. This is typical of space applications. For example, consider a Verilog top module called `cpu_top` that contains a 32-bit RISC-V CPU, along with its register file, ALU, memory and instruction decoder. To ensure full rad-hard reliability in space, the whole `cpu_top` module needs to be triplicated. However, in the future, it would be a nice feature to be able to have finer grained control over the parts of the design are triplicated. This does unfortunately introduce some significant problems that will be elaborated on later.
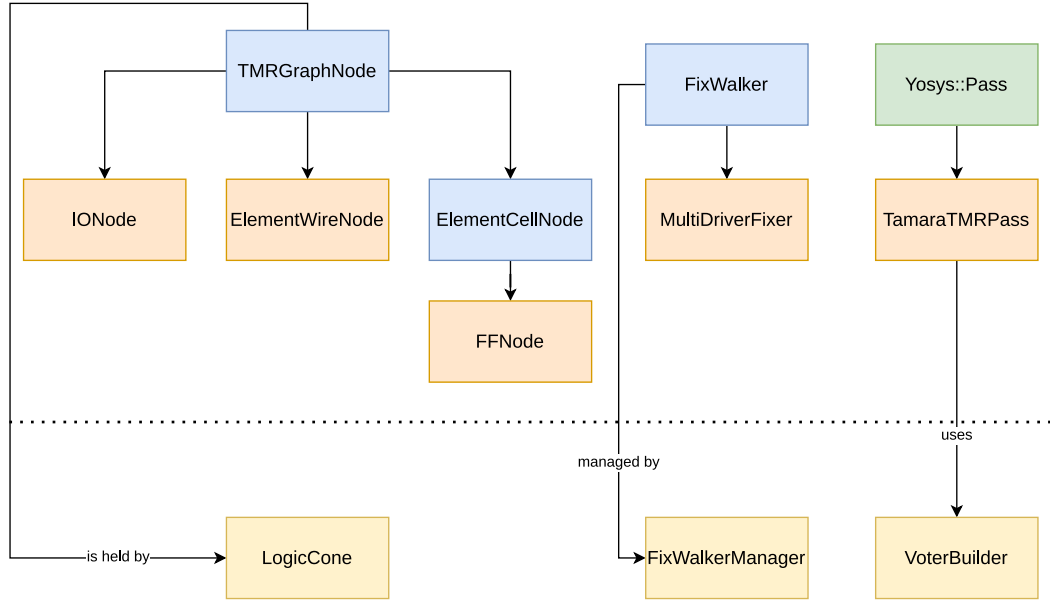
Figure 3: Class diagram of the TaMaRa codebase

TaMaRa consists of multiple C++ classes (Figure 3). Broadly speaking, these classes combine together to form the following algorithm. This is also shown in Figure 11.

3.2.1. Voter design

Voters are one of the most important parts of a TMR circuit, and so I believed it was extremely important to design them and verify them with a high degree of assurance. In the very beginning, the voter circuit was designed manually; first by sketching the truth table by hand, then automatically converting this to a logic schematic using Logisim Evolution [34]. The Logisim circuit was then transformed manually into a series of C++ macros that build an equivalent circuit in RTLIL. A formal equivalence check was performed between this RTLIL design and the original truth table sketched by hand, which was correct.

The voter consists of three input signals: *a*, *b* and *c*, which are respectively the 1-bit inputs from each of the triplicated elements. The voter then has two output signals: *out* and *err*. The *out* signal is result of the majority voter, so it should be equivalent to the input signal minus any SEUs. The *err* signal is set to '1' if and only if a fault was detected. This could be used for diagnostics, or to perform a configuration reset if possible on FPGAs, and reboot on ASICs.

Given these constraints, the truth table for a majority voter can be described as follows (Table 1):

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

Table 1: Truth table for a single-bit majority voter

Using techniques such as Karnaugh mapping [35], this truth table can be optimally mapped to a combinational circuit. In this case, Logisim Evolution [34] was used, which produced the following result as shown in Figure 4. This circuit consists of 3 NOT gates, 6 AND gates, and 4 OR gates. These are all two-input gates.
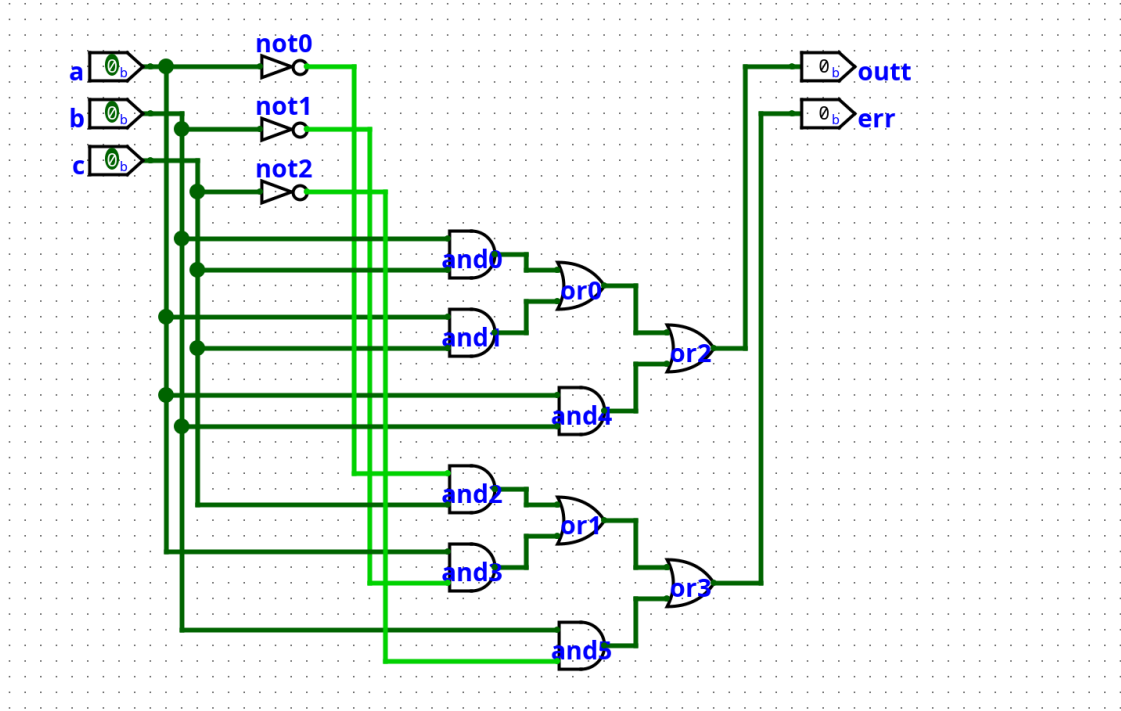
Figure 4: Logisim Evolution schematic for voter circuit

Using a series of macros, this can be translated into C++ code that generates RTLIL circuits. First, we define a series of macros like WIRE, NOT, AND and OR that add RTLIL objects to the current module with the correct tamara_voter annotation, which is expected to be applied to all voter logic elements. Then, I performed a direct, manual translation as shown in Listing 2:

```
// NOT
// a -> not0 -> and2
WIRE(not0, and2);
NOT(0, a, not0_and2_wire);

// b -> not1 -> and3
WIRE(not1, and3);
NOT(1, b, not1_and3_wire);

// c -> not2 -> and5
WIRE(not2, and5);
NOT(2, c, not2_and5_wire);

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

// a, c -> and1 -> or0
WIRE(and1, or0);
AND(1, a, c, and1_or0_wire);

// and so on, and so on...
```

Listing 2: Partial listing of C++ macros to generate a voter

When applied in Yosys, a schematic similar to the Logisim circuit is generated, as shown in Figure 5:

$auto$tamara_debug.cpp:57:execute$1
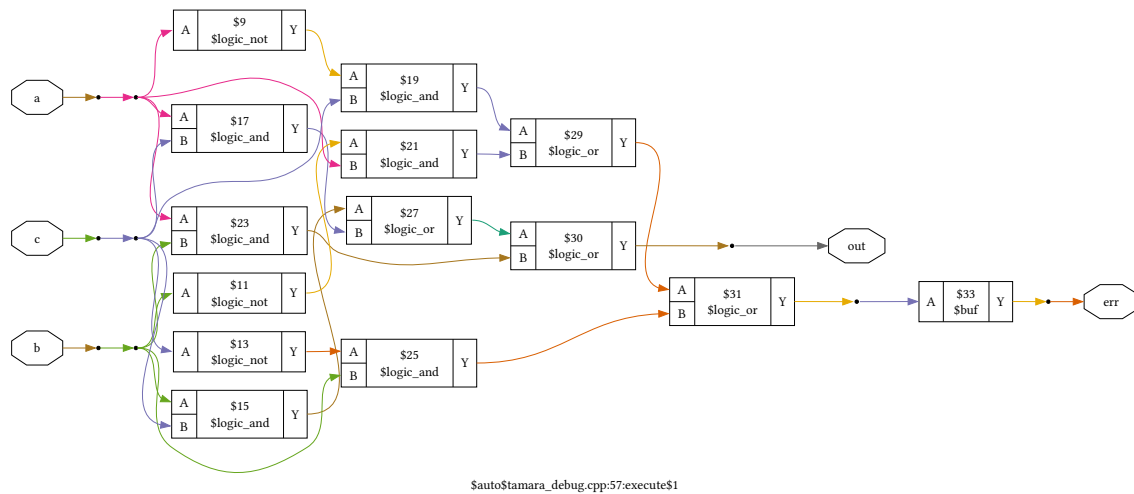
Figure 5: Voter schematic generated by Yosys

Using a SystemVerilog translation of the Boolean function implementing a voter (Listing 3), the RTLIL generated voter was formally verified to be correctly implemented. Further details on the formal verification procedure are presented later in Chapter 3.3.2.

20

```
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
```

Listing 3: SystemVerilog implementation of 1-bit majority voter

3.2.2. RTLIL netlist analysis

In Yosys, although RTLIL is used to model the netlist, the connections between cells and wires are not immediately available for use in TaMaRa. Instead, I first perform a topological analysis of all the cells and wires in the netlist. I consider output and input ports for cells, and also uniquely consider wires as well. The aim is to construct a `tamara::RTLILWireConnections` object, which is a mapping between the name of a wire or cell (which is guaranteed to be unique in an RTLIL design), and the set of wires or cells it may be connected to on a backwards traversal. The last element is important, because this data structure also acts as an efficient cache to use when searching the circuit on a backwards-BFS. During this step, I also construct other similar data structures that are used to lookup `RTLIL::SigSpec` objects, which are unique in RTLIL and can be used to represent RTL concepts like constants and wires. An example of this construction is shown in Figure 6.
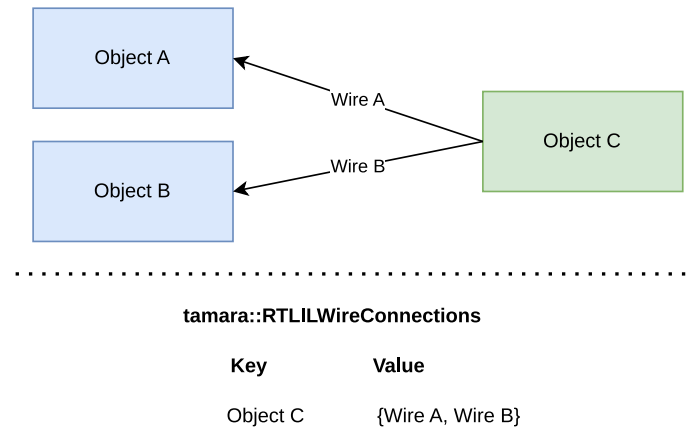

Figure 6: Demonstration of RTLILWireConnections construction

3.2.3. Backwards breadth-first search

The key step of the TaMaRa algorithm is mapping out and tracking the combinational logic primitives that are located in between sequential logic primitives in a given design. This enables us to correctly replicate the design, without introducing sequential delays that would invalidate the circuit's design. In order to achieve this, I perform a breadth-first search (BFS) search, operating backwards *from* the output of the circuit *towards* the

input of the circuit. A backwards search is used under the assumption that the *outputs* of a circuit naturally depend on both the combinational and sequential path through the circuit; so, by working from outputs backwards to inputs, we naturally cover only the essential circuit elements and guarantee we won't miss anything. This is the same approach used by Beltrame [22].

On the backwards BFS, when we reach a flip-flop or an IO node (i.e. an input to the circuit), we wrap up the search[2] and declare the current collected RTLIL primitives as part of a single *logic cone.*

TaMaRa's definition of a logic cone is shown in Figure 7. The first combinational logic cone is shown in blue, the second in green; both of these would be discovered separately by the backwards BFS.
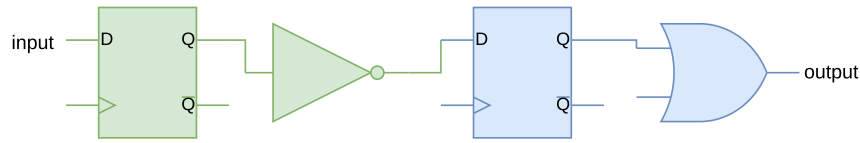
Figure 7: Description of TaMaRa's definition of logic cones

3.2.4. Combinational replication

Once we have formed a logic cone, we are able to replicate all of the components inside it. This is a relatively trivial operation and is simply a matter of using the Yosys API to instantiate two replicas for each original node. These replicas are also marked with special TaMaRa annotations to indicate that they are replicas, and what logic cone they belong to.

3.2.5. Voter insertion

With the combinational primitives in the circuit replicated, the next step is to generate and insert majority voters to vote on the redundant logic, and thereby actually implement TMR.

TaMaRa voters are always single-bit. Handling multi-bit signals is a two-stage process. Firstly, before TaMaRa is run, the user is required to run the `splitcells` and `splitnets` commands, which break multi-bit cells and multi-bit wires respectively into multiple single-bit instances. Whilst this handles most of the internals of the circuit, the inputs/outputs to the circuit will still be multi-bit. For example, consider a module with an `input logic [3:0] a`; the port `a` will still be 4-bits wide. To work with this, the voter generator is able to split apart these multi-bit signals and attach a unique voter for each bit. When these chains are built, the voter builder dynamically inserts a Yosys `$reduce_or` cell to OR together all the error signals from all voters. The voter builder is able to detect when this cell is necessary or not, and if it's not necessary, emits a `$buf` cell to improve PPA.

---

[2]This is not quite the same as terminating the search immediately; it's important that we consider remaining items in the BFS queue before instantly terminating the search.

For multi-cone designs, the voter builder is also capable of building a tree structure of OR gates to bubble up the individual voter error signals to a global error signal. It is worth noting that this tree-like structure will significantly increase the combinational critical path delay of the circuit, and it would be better replaced with more optimal structures in future work.

On any given logic cone, we define the "voter cut point" to be the location in the logic cone netlist where we should splice the circuit and insert the majority voter. Currently, TaMaRa determines the voter cut point during the backwards BFS. It is set to be the first node encountered on the backwards BFS that fulfills the following criteria:

- It is not the very first node in the entire backwards BFS (i.e. this would be the output cell); and
- It is not a *terminal node* (i.e. not an `IONode` or an `FFNode`); and
- It is not an `ElementWireNode`

The voter cut point is set once and only once per node. Once it is set, it's not immediately used by the backwards BFS, but rather passed onto the wiring stage for later use.

3.2.6. Wiring

The most complex (and error prone) element of the TaMaRa algorithm is, by far, the wiring logic. As a generalised representation of RTL at various stages of synthesis, RTLIL is extremely complex. Handling complex, recurrent, multi-bit circuits with elements like bit-selects, slicing and splicing is very challenging. TaMaRa approaches this on a "best effort" basis, but is currently unable to handle all wiring types present in Yosys. In essence, the wiring logic is similar to performing "surgery" on the circuit; cutting and splicing complicated RTLIL primitives that can be challenging to re-attach correctly. The wiring logic remains the single biggest limitation in the TaMaRa algorithm, and is often the sole reason that more complex circuits cannot yet be processed.

The first step of the wiring process is to insert the majority voter, which was covered separately in the prior section Chapter 3.2.5. Specifically, the voter cut point is determined from the earlier backwards BFS stage, and a procedure[3] is used to extract the correct wires from replicated elements in the circuit to use as the A, B, C, OUT and ERR ports of the voter. This is then passed to the voter inserter to insert directly into the circuit.

Once the voter has been inserted, and holding a reference to the RTLIL output wire of the circuit, the set of RTLIL `SigSpecs` attached to the output wire are located. This is mainly used to determine if multiple wires, bits, chunks or other similarly complex RTLIL primitives are attached to the wire. If there is only one single `SigSpec` attached, then wiring logic "stitches together" the voter output port directly to the original circuit. If there are multiple `SigSpecss`, then the wiring logic finds the `SigSpecs` that are attached to the *output* of the voter on the *original* circuit before modification. Then, it stitches together this original `SigSpec` and the rest of the circuit. This is a very large (and poor)

---

[3]It should be noted that this procedure is one of the largest causes of errors and other crashes in circuit processing, and should likely be overhauled in future work.

assumption, so we also warn the user if our assumptions do not hold; for example, if there are *multiple* SigSpecs originally attached to the voter output wire, which is currently not handled.

3.2.7. Wiring fix-up

TaMaRa's wiring logic currently cannot handle the entire circuit in a single pass. Hence, TaMaRa wiring cannot simply be done in a single stage. Instead, a multi-stage process was developed that uses a second pass to detect and "fix-up" cases of invalid wiring. This is achieved by sub-classing a tamara::FixWalker interface, which is in turn processed by a tamara::FixWalkerManager. This utility walks the RTLIL circuit using the visitor pattern [36], and invokes methods on the tamara::FixWalker accordingly.

At present, the main problem to fix up is situations in which cells can have multiple drivers. This occurs when replicating the wires connecting the cells. A second pass is necessary in order to correctly connect the replicated wires to the replicated cells. This special case is detected by a tamara::FixWalker instance that specifically looks for cells:

- Which are driven by 3 wires; and
- Which drive 3 wires; and
- Where each of the driven wires have a (* tamara_cone *) annotation (i.e. they have been replicated by TaMaRa); and
- Where each of the driver wires have a (* tamara_cone *) annotation

Figure 8 shows the schematic of a circuit before the multi-driver fixer described above was run. From the schematic, it is clear that the ff wire has multiple drivers, and drives multiple cells.
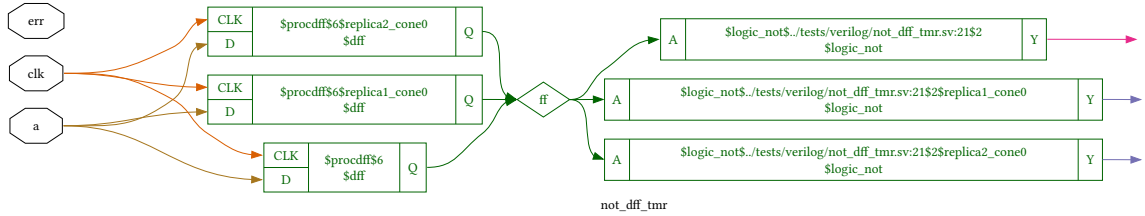

Figure 8: Schematic of circuit before running the multi-driver fixer

After running the multi-driver fixer, Figure 9 shows that the multi-driver cell has been detected and repaired successfully.
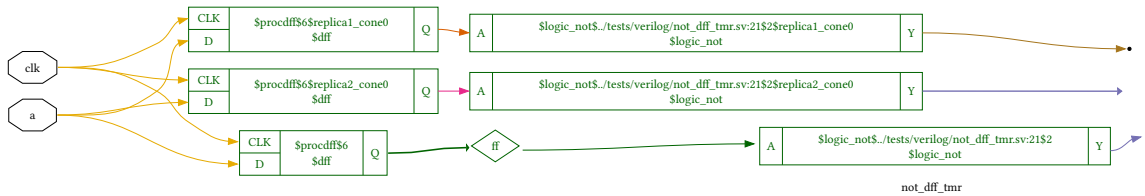

Figure 9: Schematic of circuit after running the multi-driver fixer

3.2.8. Search continuation

For multi-cone circuits with more than one logic cone between the inputs and outputs, the algorithm needs to continue analysing the circuit. During the search stage as

described in Chapter 3.2.3, there is an additional step where the TaMaRa algorithm checks for successor logic cones and therefore if a continued search is required.

During the search, if the input node of a logic cone is connected to other cells, and those cells have not yet been explored as part of a previous search, then the input node is added to a queue of successor nodes. These successor nodes are searched using the exact same process, essentially taking the algorithm back to Chapter 3.2.3, and repeating until no more successor nodes remain (i.e. the input of the circuit is reached). This is further illustrated in Figure 10.

A global set of successor nodes that have been already explored is maintained, so that recurrent circuits do not produce infinite searching loops.
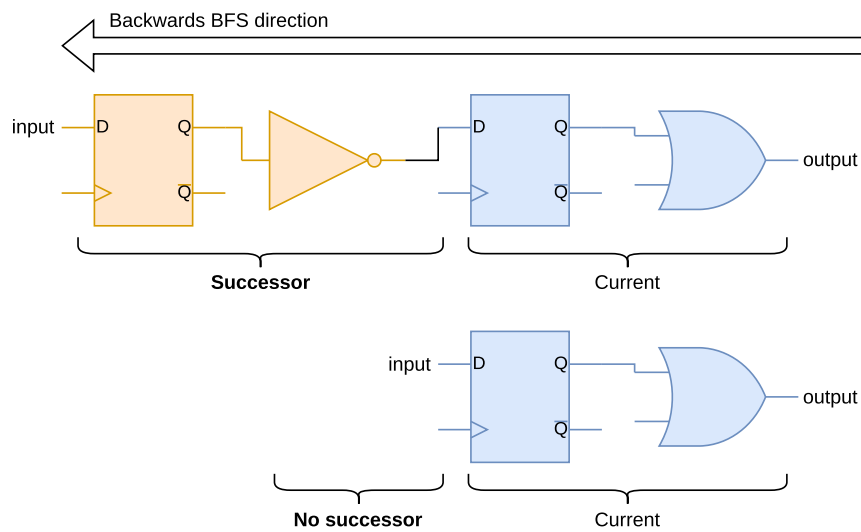

Figure 10: Demonstration of search continuation for two circuits

3.2.9. Summary

In summary, the algorithm can be briefly described as follows:

1. Analyse the RTLIL netlist to generate `tamara::RTLILWireConnections` mapping; which is a mapping between an RTLIL Cell or Wire and the other Cells or Wires it may be connected to
2. For each output port in the top module:
    1. Perform a backwards breadth-first search through the RTLIL netlist to form a logic cone
    2. Replicate all combinational RTLIL primitives inside the logic cone
    3. Generate and insert the necessary voter(s) for each bit
    4. Wire up the newly formed netlist, including connected the voters
3. Perform any necessary fixes to the wiring, if required
4. With the initial search complete, compute any follow on/successor logic cones from the initial terminals

5. Repeat step 2 but for each successor logic cone
6. Continue until no more successors remain
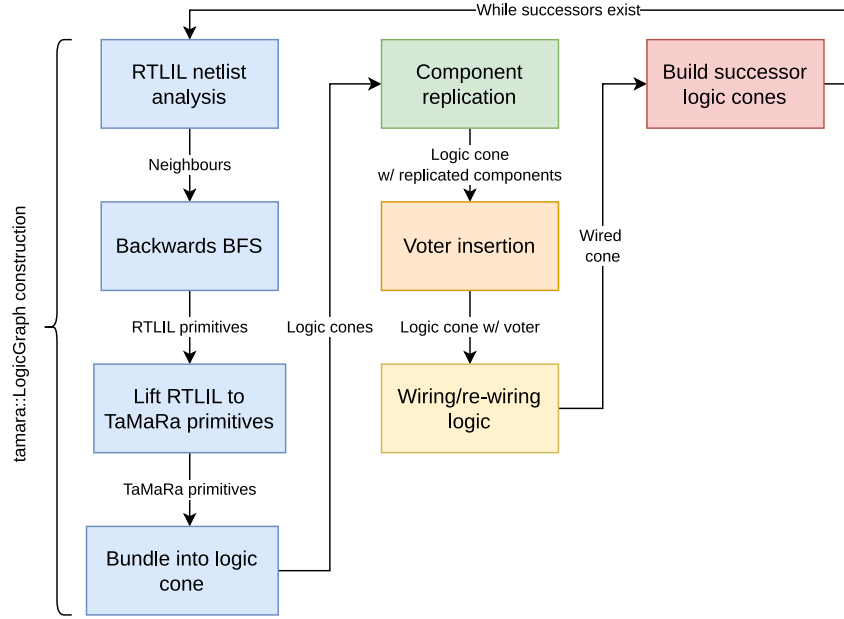
Figure 11: Logic flow of the TaMaRa TMR algorithm

In general, the TaMaRa code is designed to be robust to any and all user inputs, and easy to debug when the algorithm does not work as expected. This is achieved by a combination of detailed, friendly error reporting and copious `assert` statements available in debug builds. For example, if a user specifies an error port marked (`* tamara_error_sink *`) that is multi-bit, which is not supported, TaMaRa will print an error explaining this in detail. The algorithm also performs self-checking using `assert` statements throughout the process to catch internal errors that may occur.

The friendly error reporting is designed as a "first line of defence" for the most common user errors, and the addition of `asserts` plays an important role in debugging end user crashes. Ideally, TaMaRa will rather crash then generate an impossible design. All of this combines together to hopefully make a tool that users can be confident deploying in rad-hardened, safety critical scenarios.

3.3. Verification

Due to its potential for use in safety critical sectors like aerospace and defence, comprehensive verification and testing of the TaMaRa flow is extremely important in this thesis.

TaMaRa is a highly complex project that, during the course of this one year thesis, developed into a substantial and complicated codebase. Not only is the TaMaRa algorithm itself complex, but it is also dependent on top of the very complex Yosys codebase. In addition, the very process of EDA synthesis is highly non-trivial; in some ways, akin to writing a compiler. This means that an extensive verification methodology is required not just as a once-off, but throughout development.

26

To ensure this verification could be achieved, I implemented a regression test suite, which is common in large-scale software projects. The regression test script is written in Python, and reads the list of tests to run from a YAML document. This script is capable of running both Yosys script tests as well as formal equivalence checking tests using the `eqy` tool. The script also keeps track of prior results and recently failed tests, so that regressions can be easily detected. This tool was an essential part of the TaMaRa development process, as it allowed major refactors to be performed without the worry of breaking any prior tests.

3.3.1. Manual verification

The design and use of RTL testbenches continues to be important when designing FPGA and ASIC projects. Likewise, RTL testbenches are very important when designing EDA tools. Compared to FPGA/ASIC design, when working on EDA tools, having a representative sample of a large number of projects is the most important aspect. For TaMaRa, I sourced a number of representative small open-source Verilog projects with acceptable licences for inclusion in the `test` directory. These designs include:

- Various cyclic redundancy check (CRC) calculators of varying bit-depths
  - ▸ Tests TaMaRa's handling of combinational circuits
- Small RISC-V CPUs: picorv32, femtorv32, minimax, Browndeer Technologies' rv8
  - ▸ CPUs are highly representative of large Verilog projects, and include complex combinational and sequential circuits

In addition, I also wrote a number of much smaller testbenches to target specific bugs or specific features in TaMaRa. These were very important in the initial development and verification of the algorithm, as their tiny size allowed for visual debugging using Yosys' `show` command. These circuits are documented in full in Chapter 4.1.

3.3.2. Formal verification

For TaMaRa specifically, formal verification is abstracted through the use of Yosys' `miter` and `sat` commands. Yosys' built-in SAT solver is based on MiniSAT [24], which is a widely-used and powerful SAT solver, although is not as powerful as SMT solvers. Nonetheless, it suffices for these small test circuits. Yosys also features a `mutate` command that was originally written to verify the correctness of self-checking testbenches. Essentially, it statically injects various types of faults, including SEU equivalents, into the design's netlist, which is then usually used to prove that the testbench fails when the circuit is modified. This is repurposed for a similar use-case in TaMaRa, to instead prove that faults are mitigated when the circuit is processed through the algorithm.

Formal equivalence checking is used in the TaMaRa verification flow to formally prove (for specific circuits, at least) that the tool holds up two of its key guarantees: that it does not change the underlying behaviour of the circuit during processing, and that it actually protects the circuit from SEUs. We could also check this using testbenches, or for simple combinational circuits by comparing the truth table manually, but SAT-based formal equivalence is actually easier to implement, and provides significantly stronger proofs of correctness. If the formal equivalence check passes, we can be absolutely certain that

the behaviour of the circuit has not changed, for all possible inputs; and for sequential circuits, for all possible inputs *and* all possible states the circuit may be in.[4]

The first guarantee, that the TaMaRa algorithm does not change the behaviour of the circuit, can be verified directly by using the formal equivalence checker and SAT solver. However, verifying the second guarantee - that TaMaRa actually protects against SEUs - is more challenging. To do this, I devised a method that combines the `mutate` command with the equivalence checker. The *gold* circuit (the circuit to be verified against) is set to be the original design, with no faults and no TaMaRa replicas. The *gate* circuit (the circuit that is being verified) is set to the original circuit, with TaMaRa replicas, and then with a certain number of mutations. This methodology operates under the realisation that if the *gate* correctly masks faults using its voter, then should then have the same behaviour as the *gold* circuit, and thus be equivalent. This is shown below in Figure 12.
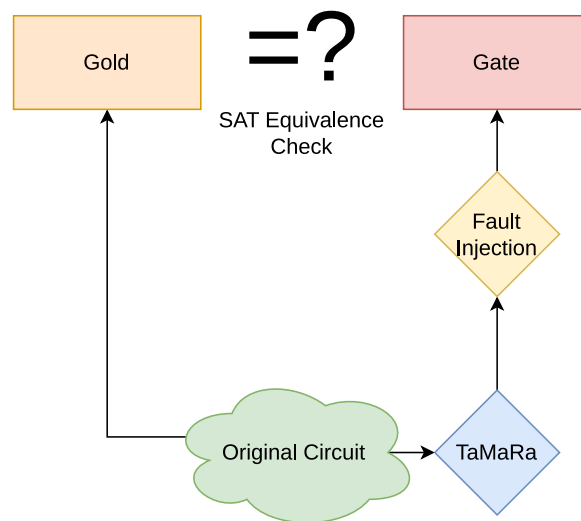

Figure 12: Diagram of TaMaRa verification methodology

3.3.3. RTL fuzzing techniques

As mentioned in Chapter 2.7, RTL fuzzing is an emerging technique for generating large-scale coverage of Verilog design files for EDA tools. Hence, part of the TaMaRa verification flow involves using Verismith [33] to generate small random Verilog designs, and running TaMaRa end-to-end on these designs. Initially, I was looking for crashes, assert failures and memory errors using AddressSanitizer, but later also used Yosys' eqy tool to prove that the designs stay the same before and after TaMaRa processing. Using the GNU Parallel tool, this work can be trivially distributed across multiple cores. Running TaMaRa with the Verismith fuzzer on 8192 small designs takes around 5 minutes on an AMD Ryzen 9 5950X workstation.

---

[4]Although, it should be noted that there are specific constraints with Yosys' `sat` solver in regards to the number of clock cycles it considers when proving sequential circuits.

To simplify the above, a shell script was developed to run this process end-to-end, and automatically clean up tests that did not fail; leaving only failed tests left over. With automatic timestamped directories, this is potentially a tool that could run as part of a continuous integration (CI) workflow for automatic validation.

Chapter 4
Results and Discussion

4.1. Testbench suite

In order to test the TaMaRa algorithm, a number of SystemVerilog testbenches implementing various different types of circuits were designed. This section will detail the testbench suite in full. In each of the selections below, the attached table shows the circuit name, the SystemVerilog RTL describing the circuit, and its schematic after running the following Yosys script:

```
read_verilog -sv $name
prep
splitcells
splitnets
show -colors 420 -format svg -prefix $output
```

This list approaches circuits in their order of complexity: first starting with simple, single-bit combinational circuits, and then progressing up to advanced, multi-bit, multi-cone, recurrent, sequential circuits. This was also the order in which the TaMaRa algorithm was verified: starting with small, simple circuits, and progressing to complex ones, verifying incrementally along the way.

4.1.1. Combinational circuits

The simplest type of digital circuit is a single-bit, purely combinational one. Table 2 lists these single-bit combinational circuits. These were critical for initial, early implementation of the TaMaRa algorithm, as their small size allowed for visual debugging using the Yosys show tool.

Name	SystemVerilog RTL	Synthesised schematic
not	<code>module inverter(input logic a, output logic o, (* tamara_error_sink *) output logic err); assign o = !a; endmodule</code>	
not_slice	<code>module not_slice(input logic a, input logic b, output logic[1:0] out, (* tamara_error_sink *) output logic err);</code>	
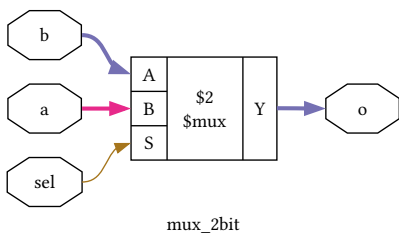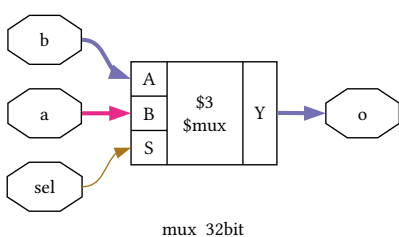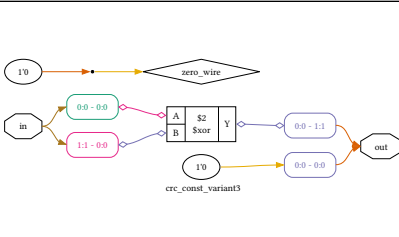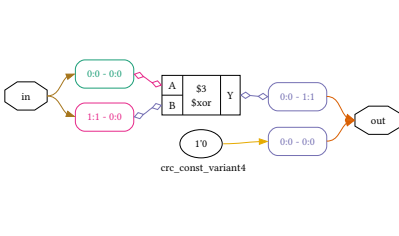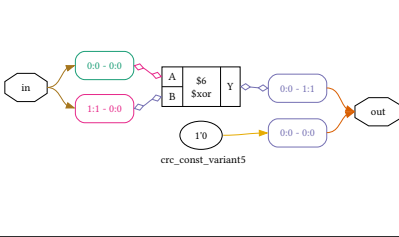
| | ```
    assign out = { !a, !b };
 endmodule
``` | |
| not_swizzle_low | ```
module not_swizzle_low(
    input logic a,
    output logic[1:0] out,
    (* tamara_error_sink *)
    output logic err
);
    assign out = { !a, 1'd0 };
endmodule
``` | <br>not_swizzle_low |
| not_swizzle_high | ```
module not_swizzle_high(
    input logic a,
    output logic[1:0] out,
    (* tamara_error_sink *)
    output logic err
);
    assign out = { 1'd0, !a };
endmodule
``` | <br>not_swizzle_high |
| mux_1bit | ```
module mux_1bit(
    input logic a,
    input logic b,
    input logic sel,
    output logic o
);
    assign o = sel ? a : b;
endmodule
``` | <br>mux_1bit |

Table 2: Table of single-bit combinational circuit designs

4.1.2. Multi-bit combinational circuits

Once the single-bit combinational circuits were confirmed to be working, the next set of tests involved multi-bit combinational circuits, as shown in Table 3. This is useful to ensure that the voter builder, as described in Chapter 3.2.5, correctly inserts a voter for each bit in the bus; and also that the wiring code can handle multi-bit edge signals.

Name	SystemVerilog RTL	Synthesised schematic
| not_2bit | ```
module inverter(
    input logic[1:0] a,
    output logic[1:0] o,
    (* tamara_error_sink *)
    output logic err
);
assign o = ~a;
endmodule
``` | <br>inverter |
| not_32bit | ```
module inverter(
    input logic[31:0] a,
    output logic[31:0] o,
    (* tamara_error_sink *)
    output logic err
);
assign o = ~a;
endmodule
``` | <br>inverter |

| mux_2bit | ```systemverilog
module mux_2bit(
    input logic[1:0] a,
    input logic[1:0] b,
    input logic sel,
    output logic[1:0] o
);
    assign o = sel ? a : b;
endmodule
``` |  mux_2bit |
| mux_32bit | ```systemverilog
module mux_32bit(
    input logic[31:0] a,
    input logic[31:0] b,
    input logic sel,
    output logic[31:0] o
);
    assign o = sel ? a : b;
endmodule
``` |  mux_32bit |
| crc_const_variant3 | ```systemverilog
module crc_const_variant3(
    input logic[1:0] in,
    output logic[1:0] out
);
    wire zero_wire = 1'b0;
    assign out = {in[0] ^
in[1], zero_wire};
endmodule
``` |  crc_const_variant3 |
| crc_const_variant4 | ```systemverilog
module crc_const_variant4(input
logic[1:0] in, output
logic[1:0] out);
    // (1'b1 - 1'b1) evaluates
to 0
    assign out = {in[0] ^
in[1], (1'b1 - 1'b1)};
endmodule
``` |  crc_const_variant4 |
| crc_const_variant5 | ```systemverilog
module crc_const_variant5(input
logic [1:0] in, output logic
[1:0] out);
    // (in[1] & ~in[1]) always
equals 0 regardless of in[1]
    assign out = {in[0] ^
in[1], (in[1] & ~in[1])};
endmodule
``` |  crc_const_variant5 |

Table 3: Table of multi-bit combinational circuit designs

4.1.3. Sequential circuits

In the previous sections, the testbenches were all combinational circuits, and did not use sequential elements such as D-flip-flops. Combinational circuits are easy to design and especially easy to verify, but are not at all representative of the majority of complex SystemVerilog designs. Both Yosys' formal verification tools, and the underlying Yices [37] SMT solver support sequential circuits, and they form a valuable part of the testbench suite. Additionally, sequential circuits involve a clock signal, where the TaMaRa algorithm must be careful to connect the clock signal correctly to the TMR replicas. These testbenches are shown in Table 4.

Name	SystemVerilog RTL	Synthesised schematic
| not_dff_tmr | ```
module not_dff_tmr(
    input logic a,
    input logic clk,
    output logic o,
    output logic err
);
logic ff = 0;

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

assign o = !ff;

endmodule
``` |  |

Table 4: Table of sequential circuit designs

4.1.4. Multi-cone circuits

Whilst sequential circuits more accurately represent complex industry designs than combinational circuits, they are still not adequate test-cases for even the simplest industry designs. These designs typically connect together multiple sequential circuits in a pipeline. These circuits challenge TaMaRa's multi-cone capabilities, and additionally its multi-voter insertion methodology. In the most complex case, it is possible to have multi-cone, multi-voter, multi-bit circuits. These multi-cone designs are shown in Table 5.

Name	SystemVerilog RTL	Synthesised schematic
| cones | ```
module cones(
    input logic a,
    input logic clk,
    output logic out
);
    logic stage1;
    logic stage2;

    always_ff @(posedge clk) begin
        stage1 <= !a;
    end

    always_ff @(posedge clk) begin
        stage2 <= !stage1;
    end

    assign out = stage2;
endmodule
``` |  |
| cones_min | ```
module cones_min(
    input logic a,
    input logic clk,
    output logic out
);
    logic stage1;

    always_ff @(posedge clk) begin
``` |  |

| | ```systemverilog
        stage1 <= !a;
    end

    assign out = !stage1;
endmodule
``` | |
| cones_2bit | ```systemverilog
module cones_2bit(
    input logic[1:0] a,
    input logic clk,
    output logic[1:0] out
);
    logic[1:0] stage1;
    logic[1:0] stage2;

    always_ff @(posedge clk) begin
        stage1 <= ~a;
    end

    always_ff @(posedge clk) begin
        stage2 <= ~stage1;
    end

    assign out = stage2;
endmodule
``` |  |

Table 5: Table of multi-cone circuits

4.1.5. Feedback circuits

Feedback circuits are arguably the most complicated challenge for any algorithm that performs automated TMR insertion. These designs consist of sequential circuits with logic loops that form a logical path from the output of the circuit to the input. Without proper handling, these circuits could easily break the logic cone identification system used in TaMaRa and other TMR-insertion algorithms. A simple feedback circuit design is shown in Table 6.

Name	SystemVerilog RTL	Synthesised schematic
| recurrent_dff | ```systemverilog
module recurrent_dff (
    input  logic clk,
    output logic q
);
    always_ff @(posedge clk) begin
        q <= ~q;
    end
endmodule
``` |  |

Table 6: Table of feedback circuits

4.2. Processed circuits

In this section, we demonstrate the result of TaMaRa algorithm on the netlist of a simple, but representative circuit. Figure 13 shows the schematic for the `not_2bit` circuit, a 2-bit multiplexer, before TMR.

Figure 13: Schematic for the `not_2bit` circuit before TMR

[Figure 14](#) shows the schematic for the `not_2bit` circuit after TMR.

Figure 14: Schematic for the `not_2bit` circuit after TMR

There are several important things to note in this schematic. Firstly, and most obviously, note the large increase in area[5] - TMR is a very high-area technique that more than triples the area utilisation. Observe that the `$not` cell has been correctly triplicated, fanned out from the input `a` wire. Following that, the pink, green, and purple cells show the unpacking of this 2-bit bus into multiple 1-bit voters, which are then stitched back together again on the right-hand side. You will see that part of these signals go directly to the output `o` - which is now the corrected and voted upon output. Some signals continue on to a `$reduce_or` cell, which performs a bitwise-OR upon all error signals, and routes this to the output `err` signal. Importantly, observe in [Figure 13](#) that in the original circuit, this `err` signal was completely unconnected - TaMaRa has both generated the error signal logic, and connected it up successfully.

Although, as will be covered later in this section, there is some trouble formally verifying circuits that use DFFs, it is still possible to process them successfully with the TaMaRa

---

[5]Or at least, what *would* become ASIC/FPGA area, as we are operating on a netlist.

35

algorithm. Figure 15 shows the schematic of a simple circuit using a DFF and a NOT gate, before TMR, and Figure 16 shows the same circuit after TMR processing. In this example, note the triplication of *both* the DFF and the NOT gate, and the successful connection between the two.

Figure 15: Schematic for the `not_dff_tmr` circuit before TMR

Figure 16: Schematic for the `not_dff_tmr` circuit after TMR

4.3. Formal verification

4.3.1. Equivalence checking

To ensure the reliability of the algorithm, an attempt was made to perform formal equivalence checking on all of the combinational circuits. As of writing, there are 20 equivalence checks performed in total.

The follow circuits passed the equivalence check:

`not_2bit, not_32bit, not_tmr, voter, crc_min, crc_const_variant3, crc_const_variant4, crc_const_variant5, mux_1bit, mux_2bit, bug7, not_swizzle_low, not_swizzle_high`

The following circuits failed the equivalence check:

`crc2, crc4, crc6, crc7, crc8, crc16, not_slice`

The reasons for these failures are covered later in the thesis, but in short are known faults in the current implementation of the TaMaRa algorithm. For more info, see Table 11.

4.4. Fault injection

I propose three main classes of fault injection studies. In the first class of tests, known as "Protected voter tests", the voter circuit itself is ignored and not subject to faults. Whilst this is unrepresentative of real-world faults, it enables us to explore the validity of the voter circuit on its own. Particularly, it enables us to formally verify that the voter is able to protect a given circuit against a wide variety of faults, and to understand how many

faults this is before the test fails. In the second class of tests, known as "Unprotected voter tests", we inject faults into the entire circuit, including potentially into the voter. In the third and final class of tests, known as "Unmitigated tests", we do not apply any TMR at all, which is useful as a baseline control to compare against.

Figure 17 demonstrates the effects of fault injection, showing the circuit `not_tmr` with a single fault injected after TMR. The fault is highlighted in red.

Figure 17: Result of applying a mutation (SEU) to the `not_tmr` circuit

In this case, the `mutate` command has randomly chosen to insert an additional inverter after the triplicated `$not` gate in the original circuit, but before the voter. This is a formal simulation of an SEU that could then be corrected by the voter circuit. The `mutate` command has many different faults it can inject to simulate SEUs, including "stuck-at-0" or "stuck-at-1" situations, in addition to inverters.

Due to the stochastic nature of this process, multiple samples of each fault are taken and the mean mitigation rate is calculated. In this case, I perform 100 samples for each number (i.e. 100x 1 fault, 100x 2 faults, 100x 3 faults, etc).

4.4.1. Protected voter

Table 7 presents the results for this protected voter fault-injection study. As the fault injection process is stochastic, it uses a sample of 100 runs per fault. The circuits listed in this table correspond to the suite of circuits presented in Table 2, after they have been processed end-to-end correctly by the TaMaRa algorithm.

Circuit name	Fault injection results
not_tmr	

not_2bit	
not_32bit	
mux_1bit	
mux_2bit	
mux_32bit	

not_swizzle_low	
not_swizzle_high	
crc_const_variant3	
crc_const_variant4	
crc_const_variant5	

Table 7: Protected voter fault injection study results

Figure 18 shows the combined results of all combinational circuits under protected voter fault injection tests.

Figure 18: Comparison of all results for protected voter combinational circuits

Generally, all circuits follow roughly the same inverse logarithmic curve, and are within a few percentage points of each other. All tested circuits are able to mitigate 100% of injected faults when one fault is injected, which is a very positive sign for the protected voter. With two injected faults, the effectiveness ranges between roughly 40% and 75%, with `not_swizzle_low` performing the worst, and `mux_2bit` and `not_2bit` performing the best. Interestingly, the only outliers here are the two 32-bit circuits: `not_32bit` and `not_2bit`, which do not have an inverse logarithmic curve, rather an almost linear curve that's significantly better than all of the other circuits. To investigate further, I performed a sweep of fault-injection tests with a 1-bit, 2-bit, 4-bit, 8-bit, 16-bit, 24-bit and 32-bit multiplexer respectively, which is shown in Figure 19.

Figure 19: Sweep of fault-injection tests on differing-width multiplexers, protected voters

4.4.2. Unprotected voter

While the protected voter study in the prior section is useful for verifying the correctness of the voter circuit itself, it is not representative of real-world fault scenarios. In the unprotected voter studies, we subject the entire netlist to faults, including the voter circuit.

The Yosys script used to run these tests was the same, except that the statements to select only voters were removed. Results for this set of tests are shown in Table 8. The process remains stochastic, and the same parameters as in the protected voter experiments were used (100 samples per fault).

Circuit name	Fault injection results
not_tmr	
not_2bit	
not_32bit	
mux_1bit	
mux_2bit	

mux_32bit	Fault injection study on 'mux_32bit', unprotected voter
not_swizzle_low	Fault injection study on 'not_swizzle_low', unprotected voter
not_swizzle_high	Fault injection study on 'not_swizzle_high', unprotected voter
crc_const_variant3	Fault injection study on 'crc_const_variant3', unprotected voter
crc_const_variant4	Fault injection study on 'crc_const_variant4', unprotected voter

crc_const_variant5	

Table 8: Unprotected voter fault injection study results

Figure 20 shows the combined results of all combinational circuits under unprotected voter fault injection tests.

Figure 20: Comparison of all results for unprotected voter combinational circuits

In this case, rather than being an inverse logarithmic curve, all tests have a very sharp fall-off in the percentage of mitigated faults, even between one and two faults. Note that, even in the case of one fault, only between 50% and 60% of faults were mitigated, and this declines sharply to between roughly 5% and 15% at two faults. Although this is an unfortunate result, as will be covered in Chapter 4.4.6, the voter takes up the vast majority of the circuit area in these tests, meaning it's much more likely to be the target of faults, so this result does make sense on this test suite. Nonetheless, there's clear room for methodological improvement here.

Also interesting to note is that, unlike in Figure 18 with the protected voters, these results are all largely the same across all circuits. We do not see see any differences between 32-bit and 1-bit circuits in their effectiveness at mitigating faults. To investigate this further,

I performed the same multiplexer sweep as before, but using unprotected voters, which is shown below in Figure 21.

Figure 21: Sweep of fault-injection tests on differing-width multiplexers, unprotected voters

4.4.3. Multi-TMR

As we have seen in the prior sections, a serious issue with TMR setups is that the voters themselves can be subject to SEUs. As long as the voter occupies area on the circuit die, it will always be a target of SEUs. Nevertheless, it is possible (in theory) to reduce the probability that a bit strikes and disables a voter, by triplicating the voter itself. This is commonly done in highly safety-critical scenarios such as spaceflight computing, as it increases area cost immensely but does help to protect the circuit.

As a general purpose algorithm, TaMaRa does support this - quite simply in fact - by running the `tamara_tmr` command twice in a row. This will triplicate the circuit, and then triplicate it once more. To demonstrate this, Figure 22 shows the `mux_1bit` circuit with one round of triplication, and Figure 23 shows the result of triplicating it twice. Observe the very large area increase, as each voter is voted on itself by another voter.

Figure 22: Application of TaMaRa TMR once to the mux_1bit circuit

Figure 23: Application of TaMaRa TMR twice to the mux_1bit circuit

Next, to determine if this actually has any effectiveness on the reliability of the circuit, the previous fault-injection tests were run on the `mux_1bit` circuit. Figure 24 compares the reliability of the multi-TMR circuit to a regular TMR circuit.

Figure 24: Multi-TMR vs. regular TMR for mux_1bit, unprotected voter

Whilst there is some notable improvement, it is definitely not significant enough to sacrifice the considerable increase in area caused by triplicated voters. The exact reason *why* triplicating the voters doesn't improve reliability is probably an artefact of the test situation. As in many other tests, the test circuits are too simple (in this case, a single 1-bit multiplexer), and the voter ends up taking significantly more area than the circuit itself. This is covered in Chapter 4.4.6.

4.4.4. Unmitigated circuits

To compare against a baseline, Figure 25 shows the results of fault injection on all combinational circuits with no mitigation (i.e. no TMR) whatsoever.

Figure 25: Comparison of all results for unmitigated combinational circuits

As expected, these largely result in 0% mitigation rate across the board. However, there's an interesting spike up to 10% mitigated for some circuits at exactly two faults. My hypothesis here is that two faults being injected into the circuit can, on occasion, cancel each other out.

4.4.5. Error signal verification

As covered in the methodology chapter, TaMaRa adds voters with an "error" signal that is intended to be set high whenever an SEU is detected at any point in the entire circuit. This is an important signal to ensure is correct, as it could be used to reset the device when faults occur, which is critical to ensuring SEUs don't propagate and "pile up" on physical devices. Yet, as this error signal itself is subject to SEUs, it is possible that it could be incorrect if it is struck by an SEU.

With a small modification to the testing script used in the prior sections, it is possible to investigate this using formal methods. Rather than proving a miter circuit, we can ask the SAT solver to prove whether or not the error signal is set to '1', given that we know we are injecting faults into the circuit.

Figure 26 shows the result of this experiment on the `not_tmr` circuit with unprotected voters.

Figure 26: Testing for error signal validity on `not_tmr` circuit

There are a few notable things with this result. Firstly, unlike the other results in this chapter, there's not an obvious trend here. The percentage of correctly set error signals does not obviously linearly, logarithmically or exponentially increase, and there's either some noise or bizarre behaviour when between five and seven faults are injected. Also interesting to note is that, at best, only around 55% of the time is the error signal set correctly, and this is with eight faults. More realistically, in the case of a single fault, the error signal is set correctly only 30% of the time. However, as more faults are injected, the correctness of the error signal improves. This does seem to make some statistical sense, as the increased number of faults injected into the circuit seems more likely to trip the combinational path that sets the error signal to '1'. Comparatively, injecting a fault in just the right place to cause the error signal to be stuck at '0' would be statistically less likely.

4.4.6. Analysis

In many of the unprotected voter tests, the results are significantly worse than with the protected voter. This is because the voter takes up the majority of the gate-area of the circuit in a number of these cases, meaning the likelihood of the fault applying to the voter and hence invalidating it is much higher. In Table 9, I calculate the percentage area that the voter accounts for by summing the wires and cells reported by the Yosys `stat` command before and after voter insertion. The real effectiveness of the algorithm in representative real-world tests is by comparing unprotected voter tests vs. unmitigated voter.

Circuit name	Area taken by voter
not_2bit	90.4%
not_32bit	99.3%
not_tmr	90.9%

mux_1bit	63.2%
not_dff_tmr	64.3%

Table 9: Voter area for different representative circuits

This raises an interesting question for further research. There is an implication here that the area a voter takes up is a very important characteristic that impacts how fault-tolerant a TMR circuit is. This, in turn, suggests that area driven (or, alternatively, placement-driven) TMR approaches would be a valuable field to investigate in future research. Regardless, since TaMaRa operates entirely in the synthesis phase without any knowledge or consideration of the final area the voter takes, it makes sense that high voter areas as per Table 9 correspond with significantly reduced fault-tolerance in unprotected voter scenarios.

Although the protected voter tests are useful for proving the fundamental correctness of the algorithm, they are not representative of real-world fault-injection scenarios. In the real world, radiation can (and will) strike voter circuits. A sampling of unprotected vs. unmitigated circuit results are shown in Table 10.

Table 10: Sample of unprotected vs. unmitigated circuit results

Comparing unprotected vs. unmitigated circuits for a variety of circuits shows that the TaMaRa algorithm *is* effective, compared to a control, against mitigating between one

and three SEUs. As mentioned earlier, there is inverse logarithmic shaped curve as the number of faults increases, showing an exponential decay in the effectiveness of the algorithm until it eventually is no longer effective at mitigating more than eight upsets. This was hypothesised to happen in the early stages of drafting the algorithm, so it's a positive sign to see it in reality. One of the biggest takeaways from this data is that single-voter TMR is not enough to mitigate *all* SEUs, and is certainly not enough to mitigate multi-bit upsets (MBUs). For space-faring applications, SEUs landing in the voter circuitry remain a serious issue, especially for smaller circuits. Based on the results available, it does appear that smaller circuits (such as the ones tested in this chapter) suffer more adversely from SEUs. It is likely that this issue becomes less serious on more complex circuits such as CPUs, but the TaMaRa algorithm would need improvements in order to handle these more complicated circuits.

Chapter 5
Conclusion

5.1. Issues with the current implementation

Throughout the duration of this thesis, there have been a number of improvements and areas for future research identified.

Firstly, in future work, it would be very important to address a number of limitations in the TaMaRa algorithm. At this point, the algorithm is unfortunately unable to handle a number of critical circuits, particularly circuits types that are commonly used in industry designs. This includes many (but not all) circuits which use sequential elements such as DFFs, multi-cone circuits, recurrent circuits, and a small number of combinational circuits with very complex bit swizzling. The critical flaw that causes this is the lack of robustness of the wiring stage (Chapter 3.2.6 and Chapter 3.2.7). As an abstraction over any and all circuits at various levels of the design process, RTLIL is extremely complex, and hence splicing an RTLIL netlist to insert majority voters in all of these cases is very challenging. This is particularly the case for both multi-cone circuits, circuits with complex bit manipulation, and especially when both are combined, as can often happen in many industry designs.

One other area for improvement is the fact that TaMaRa currently does not handle memories. In its current implementation, the tool adds a `(* tamara_ignore *)` annotation to all memory cells, and warns the user that memories are not handled. Memory cells do not behave like other cells in Yosys, and it makes replicating them a challenge, from both a design and verification perspective. On FPGAs, memory cells would likely require special handling to be able to replicate them while respecting the FPGA's SRAM limitations. This could be similar, although less of an issue, on ASICs as well. Likewise, verification of circuits with memory cells in them is a challenge in Yosys. Memories cannot yet be directly proved by Yosys' built-in SAT solver, so would have to be transformed into an array of flip-flops before they could be proved equivalent, which can be done with Yosys' `memory` command.

In total, Table 11 shows a list of all bugs I am aware of in the algorithm:

Bug	Description	Overall impact
Multi-cone circuits are not handled correctly	The wire rip-up phase fails for multi-cone circuits, dropping the intermediate wire connection between the two cones, and breaking the circuit.	Severe. Multi-cone circuits would be very common in any typical industry design.
Multiple voters in the same cone cause multiple drivers on output	A combination of failures in the voter insertion code and the wiring code can cause illegal multi-driver situations on the output signal of an RTL module. This is likely caused by a poor implementation of voter cut point selection.	Severe. This, along with the first bug, means that multi-cone circuits cannot be handled.
MultiDriverFixer can crash on some designs	The wiring code in MultiDriverFixer is not robust enough to handle re-wiring certain circuits, such as the picorv32 CPU. This causes an assert failure.	Moderate. We would like to process picorv32 if possible.
Fuzzer cases can cause TaMaRa to crash	Certain fuzzer-generated code can incorrectly convince TaMaRa that regular wiring is required (when there is only one attached SigChunk), when in fact special wiring is required. This causes an assert failure.	Minor. Should not be common in real designs, but a bug nonetheless.

Table 11: List of known TaMaRa bugs

It is worth mentioning that not all of the causes of these failures are technical: there is a large human aspect as well. In other words, a lot of these failures were caused by oversights or errors on my part. Although I knew the wiring stage of the algorithm was going to be complex, I failed to consider *just how* complex it would be. Particularly, I failed to account for the challenges of working with RTLIL in general, not just dealing with complicated circuits. This includes seemingly ordinary tasks like querying the neighbours of a particular cell that end up being very complicated and require manual work. Whilst I understood that recurrent circuits would pose an issue from the beginning of the project and planned around this, I failed to realise that multi-bit buses would also be an issue. This caused a loss of time mid-way through the project to understand this issue and determine the best way to resolve it, which is something that should have been planned out from the start. Related to this, I failed to properly and deeply understand Yosys' internals, including RTLIL, from an early stage. This was because I dived into programming TaMaRa too quickly, and didn't spend enough time reading Yosys code and the documentation available to understand its structure and approach. A good example of this is that I implemented a union type `tamara::RTLILAnyPtr` as a `std::variant<RTLIL::Wire *, RTLIL::Cell *>`, as I believed Yosys did not have this type. However, as it turns out, this is identical to the Yosys `RTLIL::SigSpec` type definition.

Unfortunately, while Yosys has good documentation for its end-user commands, it does not have very good internal documentation at all, which makes it easy to miss concepts like this. Nevertheless, I *should* have spent more time reading the Yosys codebase much more carefully, particularly existing passes that perform similar operations to TaMaRa. My belief is that a large amount of the headaches caused working with Yosys were my lack of understanding how RTLIL is designed, and "working against the grain" as it were with the tool.

Likewise, there are some limitations in the verification methodology that need to be addressed. Whilst I do believe that the verification proofs are strong for the circuits we were able to prove, there are a number of circuits that I was unable to prove, mainly those that use sequential logic. The main issue is an unexpected result when injecting numerous faults into the circuit. As shown in [Figure 27](#), after a certain number of faults, even an unmitigated circuit (without any TMR at all!) is apparently able to mitigate 100% of the injected faults. This is a result that should not be possible, and is likely a methodological error caused by the sheer number of faults being introduced into a small circuit cancelling each other out. Likewise, however, we unfortunately cannot process more complex circuits due to the fundamental algorithmic issues described above. This puts us in a difficult situation where, for this thesis, I was unfortunately able to prove sequential circuits at all. I can say that from a detailed visual analysis, certain sequential circuits appear correct, but without the SAT proofs (or an equivalently rigorous testbench), we cannot say this for sure.

Figure 27: Erroneous test results from the `not_dff_tmr` circuit

One other issue identified in the methodology ([Chapter 3](#)) was an issue regarding the potential for the TMR logic to be optimised as Yosys. As we covered in the literature review, one major problem in all prior TMR approaches is that they have poor approaches to handling the synthesis tool incorrectly detecting TMR logic as redundant and removing it. I planned to address this in TaMaRa by running the algorithm post-synthesis, pre-

techmapping, ideally after all optimisation had been run. Unfortunately, in practice, this line is not so clear cut. The technology mapping happens in stages, during which certain optimisation passes are run. Importantly, after the final technology mapping pass, a final full optimisation step is run, which in principle would remove TaMaRa's redundant TMR logic. There are a number of solutions to this, but likely the best involves a modification to the upstream Yosys tool. Ideally, each optimisation pass would respect a scratchpad variable to optionally disable it. For example, `opt_share`, which is the main pass that would remove redundant logic, would respect `opt.share.disable = true` as a scratchpad variable and prevent itself from running. This approach was not implemented in this thesis, as it would require discussion with Yosys maintainers and a pull request to be merged, but could perhaps be useful future work.

5.2. Future work

During the course of this thesis, I was also able to uncover some upstream Yosys bugs[6][7] using the Verismith fuzzer tool. This does raise the question about what further bugs are lurking within Yosys. In my opinion, a valuable research effort would be a large-scale fuzzing of Yosys, not just to uncover crashes, but to formally prove equivalence between optimisation passes. This would be a relatively straightforward process of generating Verilog with Verismith, and then performing a Miter-SAT equivalence check between the optimised and unoptimised passes. I have drafted up a script that does this, and have found what I believe to be a few issues (i.e. non-equivalent results) with the `split_cells` pass, which I plan to report in the future. The main issue with this is being able to triage results into a minimised and useful form to report to maintainers, which is quite a non-trivial task.

One other verification technique that would be interesting to explore is that of *bitstream fault-injection.* This was planned earlier in development with a tool known as the `ecp5_shotgun`, which was going to be used to inject faults into the FPGA bitstream on a Lattice ECP5 FPGA, and test its behaviour in the real world. For FPGAs in particular, one of the biggest problems with their usage in space is that their configuration SRAM ("CRAM") takes up significantly more area than the actual LUTs themselves, thus making it more susceptible to SEUs. When a fault occurs, it has the effect of actually *changing* the circuit entirely, which could be significantly harder to correct. This is an area that would benefit from further research.

Finally, in addition to all the above improvements, there are also some new research directions I have identified to explore. Interestingly, in many of these test cases, the voter circuit occupies a significantly larger area compared to the main circuit. This raises the possibility of an area-driven, or even a placement-driven approach to TMR. To elaborate on that concept, in addition to performing TMR at the synthesis level, we would modify the EDA placer to attempt to place ASIC/FPGA cells in such a way as to mitigate multi-bit upsets, and potentially single-event upsets, between replicas of the

---

[6]https://github.com/YosysHQ/yosys/issues/4599
[7]https://github.com/YosysHQ/yosys/issues/4909

TMR. In essence, this would involve placing the ASIC/FPGA to maximise frequency while minimising the probability of a particle striking both replicas in a single TMR block. Potentially, this could also be used to address some of the performance issues introduced by TMR. Likewise, although I have performed formal equivalence-based fault injection studies in this thesis, if the TaMaRa algorithm was powerful enough to handle industry-standard circuits such as CPUs, it would be very interesting to fabricate a chip or design an FPGA using the TaMaRa algorithm and subject it to real radiation. Similarly, it would be interesting to compare the performance of non-TMR techniques such as Error Correcting Codes (ECCs) in real-world radiation scenarios. Potentially, Hamming or Bose–Chaudhuri–Hocquenghem (BCH) codes could provide similar or greater SEU-mitigation performance at the cost of significantly less area, particularly in microprocessor designs. This could also be combined with techniques such as rolling back and re-issuing instructions when faults occur, or issuing each instruction three times and comparing the result.

Performance wise, in the current algorithm, there is a very long combinational path introduced by the error signal propagating all the way through a module that is likely to significantly worsen PPA on real designs. Whilst we cannot fully avoid the area costs introduced by TMR, using the placement-driven approach above, we might be able to create some sort of PPA-driven TMR approach that helps to lessen the slow-down caused by this very long combinational path. Alternatively, it may be possible to pipeline the error signal to a configurable degree. On the software side itself, there are numerous cases where lookups in the TaMaRa algorithm have time complexity $O(n^2)$ or even $O(n^3)$, which could relatively easily be fixed by restructuring the way the hash table is referenced.

All in all, I think there are a number of interesting avenues to pursue in radiation-hardening research for integrated circuits, and I do intend to pursue these through a PhD.

5.3. Summary

In this thesis, I have presented *TaMaRa*: an automated Triple Modular Redundancy EDA flow for Yosys. In Chapter 1, I introduced the background and the concept of the TaMaRa algorithm. In Chapter 2, I performed an extensive literature review of automated-TMR over many years of academic literature, and divided the literature into two main approaches: design-level and netlist-level. In Chapter 3, I described the TaMaRa algorithm in detail, and explained how it uses a backwards-BFS logic-cone based approach to find and replicate TMR elements without changing circuit behaviour. I also introduced the extensive verification methodology I used. In Chapter 4, I presented a number of test circuits, as well as comprehensive formally-verified fault-injection studies. Finally, in the prior section of this chapter, I presented a detailed analysis of future work and improvements for the TaMaRa algorithm. While it may be early days for TaMaRa algorithm, my hope is that this algorithm and future research that I perform in this space will be beneficial to many.

Thank you for reading.

∎

Appendix A Source code and licensing

Source code for the TaMaRa algorithm will be made available free of charge on GitHub: https://github.com/mattyoung101/tamara. This code will be made available under the Mozilla Public License v2.0 open-source licence.

This thesis and all associated documentation is hereby released under the CC-BY 4.0 (Creative Commons Attribution) licence.

Bibliography

[1] M. O'Bryan, "Single Event Effects." Accessed: Jul. 29, 2024. [Online]. Available: https://radhome.gsfc.nasa.gov/radhome/see.htm

[2] 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

[3] 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

[4] N. Engelhardt, "Personal communication." May 2024.

[5] R. Lyons and W. Vanderkul, "The Use of Triple-Modular Redundancy to Improve Computer Reliability," *IBM Journal of Research and Development*, vol. 6, pp. 200–209, 1962.

[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] K. Zhao, Z. Liu, and F. Yu, "The avenue of FDSOI radiation hardening," in *2014 12th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, 2014, pp. 1–3. doi: 10.1109/ICSICT.2014.7021436.

[9] 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.

[10] J. Johnson, "Synchronization Voter Insertion Algorithms for FPGA Designs Using Triple Modular Redundancy," 2010. [Online]. Available: https://scholarsarchive.byu.edu/etd/2068/

[11] B. Bridford, C. Carmichael, and C. Wei Tseng, "Single-Event Upset Mitigation Selection Guide," Mar. 2008. [Online]. Available: https://docs.amd.com/v/u/en-US/xapp987

[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] 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.

[14] L. A. C. Benites, "Automated Design Flow for Applying Triple Modular Redundancy in Complex Semi-Custom Digital Integrated Circuits," 2018.

[15] "Xilinx TMRTool User Guide (TMRTool Software Version 13.2)," 2017. [Online]. Available: https://docs.amd.com/v/u/en-US/ug156-tmrtool

[16] N. D. Hindman, L. T. Clark, D. W. Patterson, and K. E. Holbert, "Fully Automated, Testable Design of Fine-Grained Triple Mode Redundant Logic," *IEEE Transactions on Nuclear Science*, vol. 58, no. 6, pp. 3046–3052, Dec. 2011, doi: 10.1109/tns.2011.2169280.

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

[18] S. Kulis, "Single Event Effects mitigation with TMRG tool," *Journal of Instrumentation*, vol. 12, no. 1, p. C1082–C1082, Jan. 2017, doi: 10.1088/1748-0221/12/01/c01082.

[19] "Synlig - SystemVerilog support for Yosys." Accessed: Aug. 04, 2024. [Online]. Available: https://github.com/chipsalliance/synlig

[20] 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.

[21] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are We There Yet? A Study on the State of High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, 2019, doi: 10.1109/TCAD.2018.2834439.

[22] 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.

[23] R. M. Karp, "Reducibility among Combinatorial Problems," in *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*, R. E. Miller, J. W. Thatcher, and J. D. Bohlinger, Eds., Boston, MA: Springer US, 1972, pp. 85–103. doi: 10.1007/978-1-4684-2001-2_9.

[24] N. Sorensson and N. Een, "Minisat v1.13 - A SAT solver with conflict-clause minimization," 2005. Accessed: Jan. 26, 2025. [Online]. Available: http://minisat.se/downloads/MiniSat_v1.13_short.pdf

[25] G. Audemard and L. Simon, "On the Glucose SAT solver," *International Journal on Artificial Intelligence Tools*, vol. 27, no. 1, p. 1840001, 2018.

[26] C. Barrett and C. Tinelli, "Satisfiability Modulo Theories," in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds., Cham: Springer International Publishing, 2018, pp. 305–343. doi: 10.1007/978-3-319-10575-8_11.

[27] A. Niemetz and M. Preiner, "Bitwuzla," in *Computer Aided Verification*, C. Enea and A. Lal, Eds., Cham: Springer Nature Switzerland, 2023, pp. 3–17.

[28] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[29] T. Weber, S. Conchon, D. Déharbe, M. Heizmann, A. Niemetz, and G. Reger, "The SMT Competition 2015–2018," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 11, pp. 221–259, 2019, doi: 10.3233/SAT190123.

[30] A. Biere, M. J. Heule, M. Järvisalo, and N. Manthey, "Equivalence checking of HWMCC 2012 Circuits," in *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, Department of Computer Science Series of Publications B, University of Helsinki, 2013. [Online]. Available: https://fmv.jku.at/papers/Biere-SAT-Competition-2013-hwmcc12-miters.pdf

[31] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with Code Fragments," in *21st USENIX Security Symposium (USENIX Security 12)*, Bellevue, WA: USENIX Association, Aug. 2012, pp. 445–458. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[32] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, in WOOT'20. USA: USENIX Association, 2020.

[33] 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.

[34] C. Burch *et al.*, "Logisim-evolution." [Online]. Available: https://github.com/logisim-evolution/logisim-evolution/releases

[35] M. Karnaugh, "The map method for synthesis of combinational logic circuits," *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, vol. 72, no. 5, pp. 593–599, 1953, doi: 10.1109/TCE.1953.6371932.

[36] A. Wesley, Ed., "Visitor," in *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.

[37] B. Dutertre, "Yices 2.2," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds., Cham: Springer International Publishing, 2014, pp. 737–744.