THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

An Automated Triple Modular Redundancy EDA Flow for Yosys

by

Matthew Lawrence Young

School of Electrical Engineering and Computer Science, 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

26 January 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

To everyone who believed in me.

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, 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.

Abstract

Safety-critical sectors require Application Specific Integrated Circuit (ASIC) designs and Field Programmable Gate Array (FPGA) gateware to be fault-tolerant. In particular, space-fairing computers 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 Abbreviations and Symbols

Abbreviation	Meaning
FPGA	Field Programmable Gate Array
ASIC	Application Specific Integrated Circuit
EDA	Electronic Design Automation
TMR	Triple Modular Redundancy
SEU	Single Event Upset
HDL	Hardware Description Language
SystemVerilog / SV	A specific hardware description language
VHDL	A specific hardware description language
LUT	Lookup Table, combinatorial logic primitives of FPGAs
IC	Integrated Circuit
RTL	Register Transfer Level
CMOS	Complementary metal-oxide semiconductor
PPA	Power, Performance and Area
COTS	Commercial Off The Shelf
PLL	Phase Locked Loop
FF	D-Flip-Flop
P&R / PnR	Place and Route
RTLIL	RTL Intermediate Language

Abbreviation	Meaning
Fab	Fabrication facility used to produce ICs
PDK	Process Design Kit

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-fairing systems (satellites, crew capsules, missiles, etc). Thus, fault tolerant computing is widely studied and applied for 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.

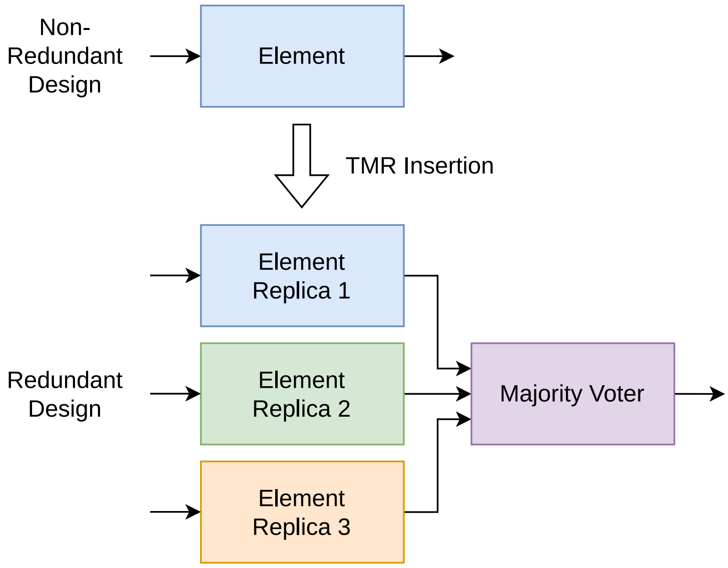

Figure 1: Diagram demonstrating how TMR is inserted into an abstract design

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):

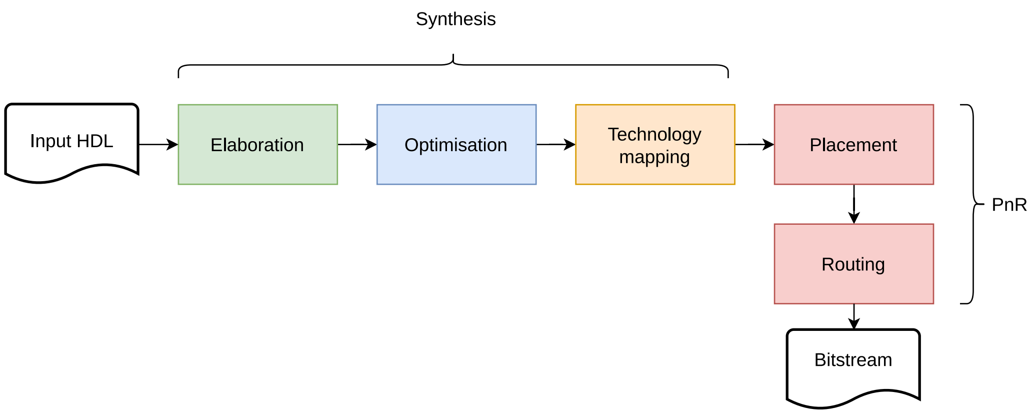

Figure 2: Simplified representation of a typical EDA synthesis flow

- **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.

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 combinatorial 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 [4] 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 [5], 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 mm^2 in a standard PowerPC 750, to 120 mm^2 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 [6]. 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 [7]. 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 [8], [9]. 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 [8] 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 [9], 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 [4].
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. [10]), 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 [9], 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. [11] (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 [12], and Benites' thesis [13], 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 combinatorial 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 combinatorial 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 [8] 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.

[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.

Xilinx, the largest designer of FPGAs, also has a netlist-level TMR software package known as TMRTool [14]. This implements a Xilinx proprietary algorithm known as 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, combinatorial 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 [10]. 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. [15] 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 [16]. 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 [15] 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 [17], 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 [18]. 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 [8], 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 [12], [13] discusses verification of the automated TMR process, and other authors [5], [15], [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’ [17] 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.

Chapter 3
Methodology

3.1. Concept

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 [8], the RTL annotation-driven approach of Kulis [17], and parts of the verification methodology of Benites [12] and Beltrame [22], to form the TaMaRa algorithm and verification methodology. Based on the dichotomy identified in Section 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 less concerns about optimisation removing the redundant TMR logic. However, as shown in , 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. **TODO whatever the solution for this is**

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 wants 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. As TaMaRa targets FPGAs from any vendor, and ASICs as well, a more general approach is necessary. 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:

```systemverilog
module mod(
    input logic a,
    (* tamara_error_sink *)
    output logic b
);
```

Listing 1: SystemVerilog snippet demonstrating the use of the `(* tamara_error_sink *)` annotation

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

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 2000 lines of C++20, and introduces one new command to Yosys: `tamara_tmr`.

3.2.1. Yosys background

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 [18]. 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 was used by Benites [12].

TaMaRa registers itself with Yosys by extending the `Yosys::Pass` interface, using a main `struct TamaraTMRPass` in `tamara_tmr_pass.cpp`.

TaMaRa operates at the netlist level, which in the context of Yosys means operating on RTL Intermediate Language (RTLIL) circuits. In the Yosys hierarchy, RTLIL sits between the frontend and backend: it is generated by a Verilog frontend, optimised, then transformed into an FPGA-specific netlist using a backend. 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].

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.

3.2.2. TaMaRa TMR algorithm implementation

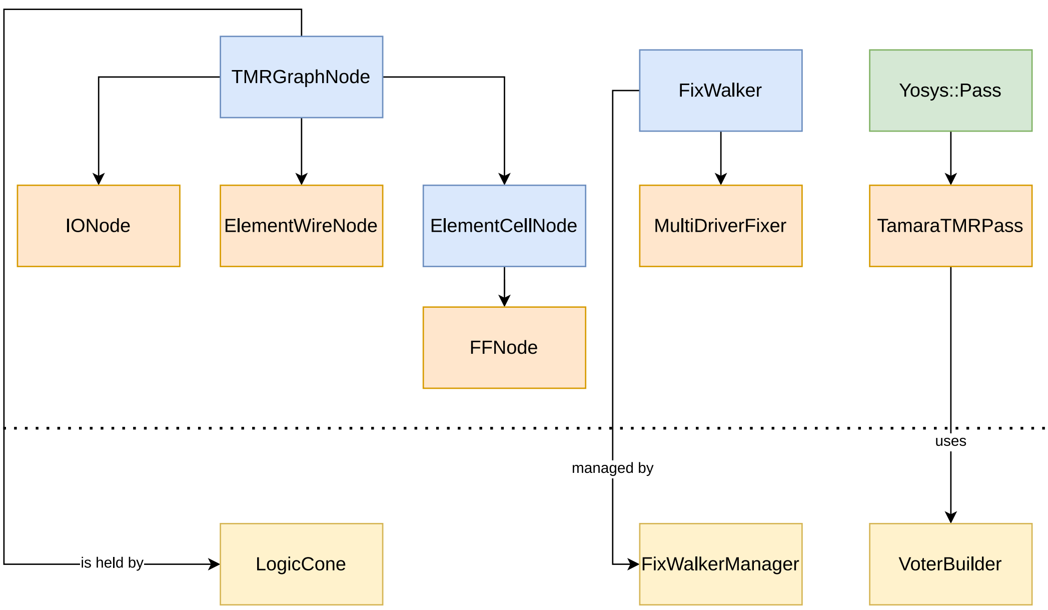

Figure 3: Class diagram of the TaMaRa codebase

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 4.

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 (`tamara::LogicCone`)
    2. Replicate all combinatorial RTLIL primitives inside the logic cone
    3. Insert the necessary voter(s) for each bit
    4. Wire up the newly formed netlist, including connected the voters and performing any necessary fixes (see `tamara::FixWalker`, `tamara::FixWalkerManager` and `tamara::MultiDriverFixer`)
3. With the initial search complete, compute any follow on/successor logic cones from the initial terminals
4. Repeat step 2 but for each successor logic cone

5. Continue until no more successors remain

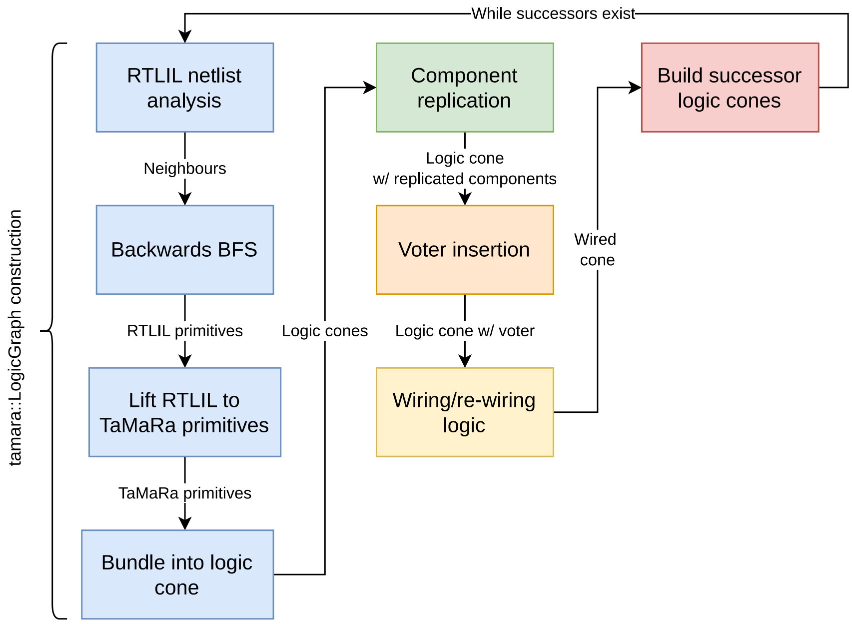

Figure 4: Logic flow of the TaMaRa TMR algorithm

TODO need to cover what a logic cone actually is - or do we cover that enough earlier?

One of the most important aspects of the TaMaRa algorithm is its ability to handle multi-bit wiring. In Yosys' RTLIL representation, an `RTLIL::Wire` instance may also be a bus, not just a single bit wire. This complicates matters, because the `tamara::VoterBuilder` class can only handle single bit voters. TaMaRa handles multi-bit wires through a mix of user commands and internal TaMaRa code. The user is instructed to run the Yosys commands `splitcells; splitnets;` beforehand as part of their synthesis script, which splits up internal multi-bit wires and cells into multiple single-bit wires and cells. This, however, still means that the output ports are multi-bit. To handle this, TaMaRa's `VoterBuilder` class has the ability to take an N-bit input signal, and split it into N individual voters. Then, once all the N voters have been generated, the `VoterBuilder` feeds the correct voter `out` bit to the output port. It is also capable of producing a correct error signal by OR'ing together all the voter error signals using a Yosys `$reduce_or` cell. Furthermore, the `VoterBuilder` is capable of OR'ing together multiple of these `$reduce_or` cells to handle multiple voters across multiple logic cones. This works by building an "OR chain", which is a chain of `$reduce_or` cells that are themselves OR'd together.

TODO Yosys 'show' result of VoterBuilder OR chain and $reduce_or

TODO talk about the wiring process in (more) detail

Once wiring is completed, however, there is still more work the algorithm needs to do. There are many wiring edge cases that are not handled correctly by the initial pass, and so the `tamara::FixWalker` and `tamara::FixWalkerManager` was designed as a modular method to “fix up” specific wiring edge cases. The `FixWalkerManager` analyses the RTLIL netlist, and runs a number of callbacks for each provided `FixWalker` to consider individual cells, wires and modules. One important `FixWalker` instance is the `MultiDriverFixer`. When the algorithm replicates wires (`ElementWireNode` instances, specifically), it causes some wire instances to have three separate, conflicting drivers, which is not legal in an RTLIL netlist.

Relatively speaking, the above description is only a minimal summary of the wiring logic of the TaMaRa algorithm. In reality, RTLIL wiring is highly complex and the logic to handle multi-bit wiring in all possible cases was hundreds of lines of code. One specific example that deserves attention is when an output signal is itself multi-bit, which requires an enormous procedure to detect available `RTLIL::SigBits` and route the wires accordingly, plus the relevant error handling code. This was a complex task that took a significant amount of overall development time of the 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. For example, Listing 2 demonstrates how logic cones are checked to conform with their specification (they may only start with an `IONode` or an `FFNode`).

```cpp
void LogicCone::verifyInputNodes() const {
    for (const auto &node : inputNodes) {
        if (dynamic_pointer_cast<IONode>(node) == nullptr && dynamic_pointer_cast<FFNode>(node) == nullptr) {
            log_error("TaMaRa internal error: Logic cone input node should be either IONode or FFNode, but "
                      "instead it was %s %s!\n",
                node->identify().c_str(), log_id(getNodeName(node)));
        }
    }
}
```

Listing 2: Example of TaMaRa internal error handling code

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 use in safety critical sectors like aerospace and defence, comprehensive verification and testing of the TaMaRa flow is extremely important in this thesis. We want to verify to a very high level of accuracy that TaMaRa both works by preventing SEUs to an acceptable standard, and also does not change the underlying behaviour of the circuits it processes.

3.3.1. Manual verification

The design and use of RTL testbenches has, and 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 combinatorial circuits
- Small RISC-V CPUs: picorv32, femtorv32, minimax, Browndeer Technologies' rv8
  - CPUs are highly representative of large Verilog projects, and include complex combinatorial 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. For example, one of the most important tests was `not_dff_tmr.sv`, a simple NOT-gate into a D-flip-flop, whose SystemVerilog code is shown in Listing 3.

```systemverilog
(* tamara_triplicate *)
module not_dff_tmr(
    input logic a,
    input logic clk,
    output logic o,
    (* tamara_error_sink *)
    output logic err
);

logic ff;

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

assign o = !ff;

`ifndef TAMARA
assign err = 0;
`endif

endmodule
```

Listing 3: SystemVerilog source code for `not_dff_tmr`, a key initial testbench

3.3.2. 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. **TODO Describe SAT in more detail** Via the Cook-Levin theorem, as proved by Karp [23], we know that SAT 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].

For TaMaRa specifically, formal verification is abstracted through the use of Yosys' `eqy`, `mcy` and `sby` (SymbiYosys) tools. `eqy` is used for formal equivalence checking between two circuits, and is responsible for partitioning the input circuit to a form suitable for equivalence checking. This is then sent on to `sby`, which in turn transforms the circuit into a suitable SMT proof for an SMT solver. TaMaRa was going to use the Bitwuzla [27] solver, but due to upstream issues with both Yosys and Bitwuzla, settled for using the industry standard Yices [30], which is quite fast. `mcy`, Yosys' mutation coverage tool, was originally written to verify the correctness of self-checking testbenches and verify the coverage of a project's testbenches. Essentially, it injects faults into a design and verifies that the self-checking testbench correctly flags these mutated designs as invalid. **TODO MCY and SBY**

The purpose of applying equivalence checking to the TaMaRa verification flow is to formally prove (for specific circuits, at least) that the tool holds up one of its key properties: that it does not change the underlying behaviour of the circuit during processing. We could also check this using testbenches, or for simple combinatorial circuits by comparing the truth table manually, but SMT-based formal equivalence checking supports all circuit types and is much more reliable. 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.

Mutation coverage is slightly more complex, but essentially allows us to prove for a particular circuit that TaMaRa actually corrects a number of different variations of simulated SEUs. Using a technique developed by Engelhardt [31], we use `mcy`'s fault injection capabilities to

3.3.3. RTL fuzzing techniques

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% branch coverage.

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 [32], which uses a tree formed by a 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 [33], 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, particularly for EDA tools. Herklotz [34] 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. Hence, part of the TaMaRa verification flow will involve using Verismith to generate small random Verilog designs, and running TaMaRa end-to-end on these designs. Initially, we will be looking for crashes, assert failures and memory errors using AddressSanitizer, but later we will also use Yosys' eqy tool to prove that the designs stay the same before and after TaMaRa runs. Using the GNU Parallel tool, this work can be trivially distributed across multiple cores, generating around running TaMaRa on 1000 designs in around 5 minutes on an AMD Ryzen 9 5950X workstation.

Chapter 4
Results

Chapter 5
Conclusion

Appendix A: Example appendix

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] 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.

[5] 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.

[6] 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/

[7] 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.

[8] 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.

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

[10] 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

[11] 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.

[12] 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.

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

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

[15] 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.

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

[17] 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.

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

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

[31] N. Engelhardt, “Using MCY for fault injection.” Accessed: Jan. 26, 2025. [Online]. Available: https://github.com/nakengelhardt/faultinjection_mcy

[32] 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

[33] 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.

[34] 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.